BigchainDB: A Scalable Blockchain Database

Trent McConaghy, Rodolphe Marques, Andreas Müller, Dimitri De Jonghe, T. Troy McConaghy, Greg McMullen, Ryan Henderson, Sylvain Bellemare, and Alberto Granzotto

June 8, 2016
ascribe GmbH, Berlin, Germany

This paper describes BigchainDB. BigchainDB fills a gap in the decentralization ecosystem: a decentralized database, at scale. It points to performance of 1 million writes per second throughput, storing petabytes of data, and sub-second latency. The BigchainDB design starts with a distributed database (DB), and through a set of innovations adds blockchain characteristics: decentralized control, immutability, and creation & movement of digital assets. BigchainDB inherits characteristics of modern distributed databases: linear scaling in throughput and capacity with the number of nodes, a full-featured NoSQL query language, efficient querying, and permissioning. Being built on an existing distributed DB, it also inherits enterprise-hardened code for most of its codebase. Scalable capacity means that legally binding contracts and certificates may be stored directly on the blockchain database. The permissioning system enables configurations ranging from private enterprise blockchain databases to open, public blockchain databases. BigchainDB is complementary to decentralized processing platforms like Ethereum, and decentralized file systems like InterPlanetary File System (IPFS). This paper describes technology perspectives that led to the BigchainDB design: traditional blockchains, distributed databases, and a case study of the domain name system (DNS). We introduce a concept called blockchain pipelining, which is key to scalability when adding blockchain-like characteristics to the distributed DB. We present a thorough description of BigchainDB, an analysis of latency, and preliminary experimental results. The paper concludes with a description of use cases.

This is no longer a living document. Significant changes made since June 8, 2016 are noted in an Addendum attached at the end.

1. Introduction

1.1. Towards a Decentralized Application Stack

The introduction of Bitcoin [1] has triggered a new wave of decentralization in computing. Bitcoin illustrated a novel set of benefits: decentralized control, where "no one" owns or controls the network; immutability, where written data is tamper-resistant ("forever"); and the ability to create & transfer assets on the network, without reliance on a central entity.

The initial excitement surrounding Bitcoin stemmed from its use as a token of value, for example as an alternative to government-issued currencies. As people learned more about the underlying blockchain technology, they extended the scope of the technology itself (e.g. smart contracts), as well as applications (e.g. intellectual property).

With this increase in scope, single monolithic "blockchain" technologies are being re-framed and refactored into building blocks at four levels of the stack:

1. Applications
2. Decentralized computing platforms ("blockchain platforms")
3. Decentralized processing ("smart contracts") and decentralized storage (file systems, databases), and decentralized communication
4. Cryptographic primitives, consensus protocols, and other algorithms

1.2. Blockchains and Databases

We can frame a traditional blockchain as a database (DB), in the sense that it provides a storage mechanism. If we measure the Bitcoin blockchain by traditional DB criteria, it's terrible: throughput is just a few transactions per second (tps), latency before a single confirmed write is 10 minutes, and capacity is a few dozen GB. Furthermore, adding nodes causes more problems: with a doubling of nodes, network traffic quadruples with no improvement in throughput, latency, or capacity. It also has essentially no querying abilities: a NoQL[1] database.

In contrast, a modern distributed DB can have throughput exceeding 1 million tps, capacity of petabytes and beyond, latency of a fraction of a second, and throughput and capacity that increases as nodes get added. Modern DBs also have rich abilities for insertion, queries, and access control in SQL or NoSQL flavors; in fact SQL is an international ANSI and ISO standard.

1.3. The Need for Scale

Decentralized technologies hold great promise to rewire modern financial systems, supply chains, creative industries, and even the Internet itself. But these ambitious goals need

[1]We are introducing the term NoQL to describe a database with essentially no query abilities. This term is not to be confused with the database company noql (`http://www.noql.com`).

scale: the storage technology needs throughput of up to millions of transactions per second (or higher), sub-second latency[2], and capacity of petabytes or more. These needs exceed the performance of the Bitcoin blockchain by many orders of magnitude.

1.4. BigchainDB : Blockchains Meet Big Data

This paper introduces BigchainDB, which is for database-style decentralized storage: a blockchain database. BigchainDB combines the key benefits of distributed DBs and traditional blockchains, with an emphasis on scale, as Table 1 summarizes.

Table 1: BigchainDB compared to traditional blockchains, and traditional distributed DBs

	Traditional Blockchain	Traditional Distributed DB	BigchainDB
High Throughput; increases with nodes↑	-	✓	✓
Low Latency	-	✓	✓
High Capacity; increases with nodes↑	-	✓	✓
Rich querying	-	✓	✓
Rich permissioning	-	✓	✓
Decentralized control	✓	-	✓
Immutability	✓	-	✓
Creation & movement of digital assets	✓	-	✓
Event chain structure	Merkle Tree	-	Hash Chain

We built BigchainDB on top of an enterprise-grade distributed DB, from which BigchainDB inherits high throughput, high capacity, low latency, a full-featured efficient NoSQL query language, and permissioning. Nodes can be added to increase throughput and capacity.

BigchainDB has the blockchain benefits of decentralized control, immutability, and creation & transfer of assets. The decentralized control is via a federation of nodes with voting permissions, that is, a super-peer P2P network [2]. The voting operates at a layer above the DB's built-in consensus. Immutability / tamper-resistance is achieved via several mechanisms: shard replication, reversion of disallowed updates or deletes, regular database backups, and cryptographic signing of all transactions, blocks & votes. Each vote on a block also includes the hash of a previous block (except for that block's votes). Any entity with asset-issuance permissions can issue an asset; an asset can only be acquired by new owners if they fulfill its cryptographic conditions. This means

[2]It takes light 140 ms to make one trip around the world, or 70 ms halfway around. Some financial applications need 30-100 ms latency, though due to speed-of-light constraints those necessarily need to be more locally constrained. Section 6 explores this in detail.

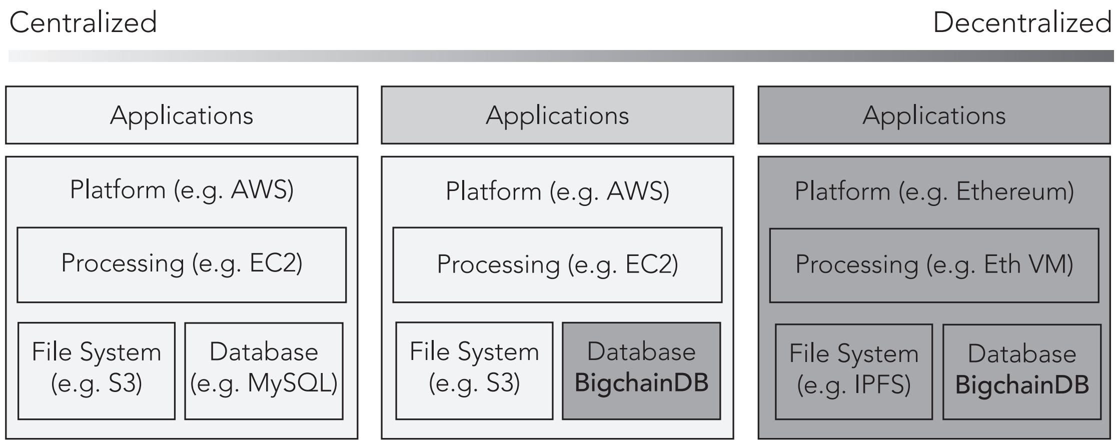

Figure 1: From a base context of a centralized cloud computing ecosystem (left), BigchainDB can be added as another database to gain some decentralization benefits (middle). It also fits into a full-blown decentralization ecosystem (right).

hackers or compromised system admins cannot arbitrarily change data, and there is no single-point-of-failure risk.

Scalable capacity means that legally binding contracts and certificates may be stored directly on the blockchain DB. The permissioning system enables configurations ranging from private enterprise blockchain DBs to open, public blockchain DBs. As we deploy BigchainDB, we are also deploying a public version.

1.5. BigchainDB in the Decentralization Ecosystem

Figure 1 illustrates how BigchainDB can be used in a fully decentralized setting, or as a mild extension from a traditional centralized computing context.

BigchainDB is complementary to decentralized processing / smart contracts (e.g. Ethereum VM [3][4] or Enigma [5][6]), decentralized file systems (e.g. IPFS [7]), and communication building blocks (e.g. email). It can be included in higher-level decentralized computing platforms (e.g. Eris/Tendermint [8][9]). It can be used side-by-side with identity protocols, financial asset protocols (e.g. Bitcoin [1]), intellectual property asset protocols (e.g. SPOOL [10]), and glue protocols (e.g. pegged sidechains [11], Interledger [12]). Scalability improvements to smart contracts blockchains will help fully decentralized applications to better exploit the scalability properties of BigchainDB.

BigchainDB works with more centralized computing systems as well. One use case is where decentralizing just storage brings the majority of benefit. Another use case is where scalability needs are greater than the capabilities of existing decentralized processing technologies; in this case BigchainDB provides a bridge to an eventual fully-decentralized system.

1.6. Contents

This paper first gives background on related building blocks, with an eye to scale:

- Section 2 - traditional blockchain scalability,
- Section 3 - distributed DBs, and

Then, this paper describes BigchainDB as follows:

- Section 4 - BigchainDB description,
- Section 5 - BigchainDB implementation, including capacity vs. nodes (Figure 9),
- Section 6 - BigchainDB latency analysis,
- Section 7 - private vs. public BigchainDBs in a permissioning context,
- Section 8 - BigchainDB benchmarks, including throughput vs. nodes (Figure 13),
- Section 9 - BigchainDB deployment, including use cases and timeline, and
- Section 10 - conclusion.

The appendices contain:

- Appendix A - a glossary, e.g. clarifying “distributed” vs. “decentralized”,
- Appendix B - blockchain scalability proposals,
- Appendix C - the Domain Name System (DNS), and
- Appendix D – further BigchainDB benchmarks.

2. Background: Traditional Blockchain Scalability

This section discusses how traditional blockchains perform with respect to scalability, with an emphasis on Bitcoin.

2.1. Technical Problem Description

One way to define a blockchain is *a distributed database (DB) that solves the “Strong Byzantine Generals” (SBG) problem* [13], the name given to a combination of the Byzantine Generals Problem and the Sybil Attack Problem. In the Byzantine Generals Problem [14], nodes need to agree on some value for a DB entry, under the constraint that the nodes may fail in arbitrary ways (including malicious behavior)[3]. The Sybil Attack Problem [17] arises when one or more nodes figure out how to get unfairly disproportionate influence in the process of agreeing on a value for an entry. It’s an “attack of the clones”—an army of seemingly independent voters actually working together to game the system.

[3]It has been noted that the Bitcoin blockchain falls short of solving the original Byzantine Generals Problem; it would be more accurate to say that it solves a relaxation of the problem [15, 16].

2.2. Bitcoin Scalability Issues

Bitcoin has scalability issues in terms of throughput, latency, capacity, and network bandwidth.

Throughput. The Bitcoin network processes just 1 transaction per second (tps) on average, with a theoretical maximum of 7 tps [18]. It could handle higher throughput if each block was bigger, though right now making blocks bigger would lead to size issues (see Capacity and network bandwidth, below). This throughput is unacceptably low when compared to the number of transactions processed by Visa (2,000 tps typical, 10,000 tps peak) [19], Twitter (5,000 tps typical, 15,000 tps peak), advertising networks (500,000 tps typical), trading networks, or email networks (global email volume is 183 billion emails/day or 2,100,000 tps [20]). An ideal global blockchain, or set of blockchains, would support all of these multiple high-throughput uses.

Latency. Each block on the Bitcoin blockchain takes 10 minutes to process. For sufficient security, it is better to wait for about an hour, giving more nodes time to confirm the transaction. By comparison, a transaction on the Visa network is approved in seconds at most. Many financial applications need latency of 30 to 100 ms.

Capacity and network bandwidth. The Bitcoin blockchain is about 70 GB (at the time of writing); it grew by 24 GB in 2015 [21]. It already takes nearly a day to download the entire blockchain. If throughput increased by a factor of 2,000, to Visa levels, the additional transactions would result in database growth of 3.9 GB/day or 1.42 PB/year. At 150,000 tps, the blockchain would grow by 214 PB/year (yes, petabytes). If throughput were 1M tps, it would completely overwhelm the bandwidth of any node's connection.

2.3. Technology Choices Affecting Scalability

The Bitcoin blockchain has taken some technology choices which hurt scaling:

1. **Consensus Algorithm: POW.** Bitcoin's mining reward actually incentivizes nodes to increase computational resource usage, without any additional improvements in throughput, latency, or capacity. A single confirmation from a node takes 10 minutes on average, so six confirmations take about an hour. In Bitcoin this is by design; Litecoin and other altcoins reduce the latency, but compromise security.

2. **Replication: Full.** That is, each node stores a copy of all the data; a "full node." This copy is typically kept on a single hard drive (or in memory). Ironically, this causes centralization: as amount of data grows, only those with the resources to hold all the data will be able to participate.

These characteristics prevent the Bitcoin blockchain from scaling up.

2.4. Blockchain Scalability Efforts

The Bitcoin / blockchain community has spent considerable effort on improving the performance of blockchains. Appendix B reviews various proposals in more detail.

Previous approaches shared something in common: they all started with a block chain design then tried to increase its performance. There's another way: start with a "big data" distributed database, then give it blockchain-like characteristics.

3. Background: Distributed Databases & Big Data

3.1. Introduction

We ask: does the world have any precedents for *distributed databases* at massive scale? The answer is yes. All large Internet companies, and many small ones, run "big data" distributed databases (DBs), including Facebook, Google, Amazon and Netflix.

Distributed DBs regularly store petabytes ($1,000,000$ GB) or more worth of content. In contrast, the Bitcoin blockchain currently stores 50 GB, the capacity of a modern thumb drive. Despite it's relatively-small data size, members of the Bitcoin community worry that it is getting too big. In fact, there are initiatives to prevent "blockchain bloat" caused by "dust" or "junk" transactions that "pollute" Bitcoin's 50 GB database [22].

Let's look at it another way: perhaps distributed DB technology has lessons for blockchain DB design.

Let's explore distributed DB scalability further.

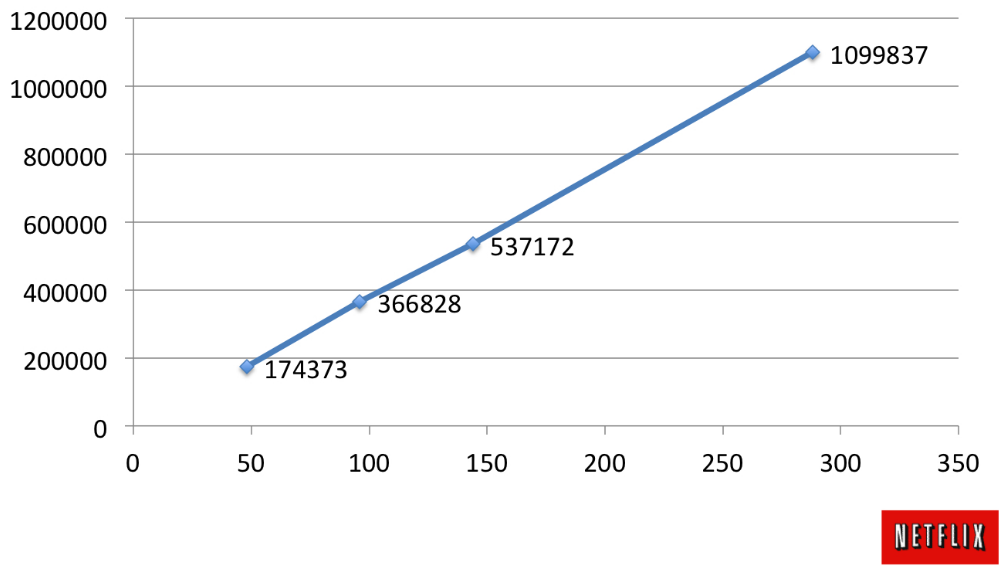

Figure 2: Netflix experimental data on throughput of its Cassandra database (Client writes/s by node count - Replication Factor=3). The x-axis is number of nodes; the y-axis is writes per second. From [23].

Figure 2 illustrates the throughput properties of Cassandra, a distributed DB technology used by Netflix. At the bottom left of the plot, we see that 50 distributed Cassandra nodes could handle $174,000$ writes per second. Increasing to 300 nodes allowed for 1.1 million writes per second [23]. A follow-up study three years later showed a throughput of 1 million writes per second with just a few dozen nodes [24]. To emphasize: the throughput of this DB increased as the number of nodes increased. The scaling was linear.

Each node also stores data. Critically, a node only stores a subset of all data, that is, it has partial replication. In the Netflix example [24], each piece of data has three copies in the system, i.e. a replication factor of three. Partial replication enables an increase in the number of nodes to increase storage capacity. Most modern distributed DBs have a linear increase in capacity with the number of nodes, an excellent property. Additionally, as the number of nodes increases, Cassandra's latency and network usage does not worsen. Cassandra can be distributed at scale not only throughout a region, but around the globe. Contrast this to the Bitcoin blockchain, where capacity does not change as the number of nodes increases.

The scalability properties of distributed DBs like Cassandra make an excellent reference target.

3.2. Consensus Algorithms in Distributed Databases

3.2.1. Introduction

As mentioned above, Cassandra keeps only some of the data in each node. Each bit of data is replicated on several nodes. The nodes responsible for replicating a bit of data use a consensus algorithm to ensure they agree on what to store. Cassandra uses the Paxos consensus algorithm; the relevant nodes will reach agreement even if some of them are unresponsive.

The Paxos consensus algorithm is one of many algorithms designed to solve the *consensus problem* in unreliable distributed systems. Loosely speaking, the consensus problem is the problem of figuring out how to get a bunch of isolated computing processes to agree on something, when some of them may be faulty, and they can only communicate by two-party messages. A solution takes the form of a consensus algorithm/protocol used by all of the non-faulty processes.

3.2.2. Byzantine Fault Tolerance

One of the first precise statements of the consensus problem was in a 1980 paper by Pease, Shostak and Lamport [25, 26]. That paper allowed the faulty processes to have arbitrary faults; for example, they could lie, collude, selectively participate, or pretend to be crashed. Such arbitrary faults are also known as *Byzantine faults*, after a 1982 follow-up paper on the same problem [14] which called it the "Byzantine Generals Problem." A consensus algorithm which enables a distributed system to come to consensus despite Byzantine faults is said to be *Byzantine fault tolerant* (BFT).

The 1980 paper had several nice results: a proof that, given f Byzantine faulty processes, at least $3f + 1$ processes are needed, an example (inefficient) solution with $3f + 1$ processes, and a proof that there is *no* solution with less than $3f + 1$ processes. It also considered what's possible if message authentication is used (i.e. if a process changes a message before relaying it to another process, then the change can be detected). In that case, $2f + 1$ processes suffice.

3.2.3. (Benign) Fault Tolerance

One of the most common ways for a process to be faulty is for it to be unresponsive. That can happen, for example, if a hard drive fails or a CPU overheats. Such faults are known as *benign faults* or *fail-stop faults*. A consensus algorithm which enables a distributed system to come to consensus despite benign faults is said to be *fault tolerant* (FT). (It would be more precise to say "benign-fault tolerant," but it's not up to us.) In general, fault-tolerant consensus algorithms require at least $2f + 1$ processes to be able to tolerate up to f faulty processes.

3.2.4. Paxos

The best-known fault-tolerant consensus algorithm is Paxos; it was first published by Lamport in 1998 [27]. Since then, many variations have been developed (e.g. "Fast Paxos" [28]) so there is now a whole family of Paxos algorithms, including some that are BFT. [29]

Mike Burrows of Google (co-inventor of Google's Chubby, BigTable, and Dapper) has said, "There is only one consensus protocol, and that's Paxos," [30] and "all working protocols for asynchronous consensus we have so far encountered have Paxos at their core." [31] Henry Robinson of Cloudera has said, "all other approaches are just broken versions of Paxos" and "it's clear that a good consensus protocol is surprisingly hard to find." [30]

Paxos and its lineage are used at Google, IBM, Microsoft, OpenReplica, VMWare, XtreemFS, Heroku, Ceph, Clustrix, Neo4j, and many more. [32]

Paxos is notoriously difficult to understand and risky to implement. To address this, Raft [33] was designed specifically for ease of understanding, and therefore has lower implementation risk. Raft has a BFT variant named Tangaroa. [34]

3.2.5. The FLP Result

An *asynchronous process* is a process which can't promise it will get back to you with a result within some time limit. It's common to model processes as being asynchronous, especially for large systems spread all over the globe (such as the World Wide Web). An *asynchronous consensus protocol* is one that works with asynchronous processes.

In 1985, Fischer, Lynch and Paterson (FLP) published a surprising result: "no completely asynchronous consensus protocol can tolerate even a single unannounced process death [i.e. benign fault]." [35] If that's the case, then it seems there's little hope for tolerating more faults (or other kinds of faults)! Practical consensus algorithms can get around the "FLP result" by assuming some kind of synchrony (e.g. "partial synchrony" or "weak synchrony"), or by allowing some form of probablistic consensus (e.g. with the probability of consensus approaching 1 over time).

Bitcoin's consensus algorithm does the latter: one can never be sure that the Bitcoin network has come to consensus about a block being in the final blockchain: there's always the possibility that block might be in a side branch. All one can do is estimate the probability that a block is in the final blockchain.

3.2.6. Practical BFT Consensus Algorithms

The early BFT consensus algorithms were either slow & expensive, or intended for synchronous systems [36, 37, 38, 39]. That all changed in 1999, when Castro and Liskov published their paper titled "Practical Byzantine Fault Tolerance" (PBFT) [40, 41]. As the title suggests, it described a more practical (usable) BFT consensus algorithm, and kicked off a flurry of research into practical BFT consensus algorithms. That research continues today. Aardvark [42], RBFT [43] and Stellar [44] are examples of algorithms aimed at improving speed and reliability.

3.3. Replication Factor & Blockchain "Full Nodes"

A modern distributed DB is designed to appear like a single monolithic DB, but under the hood it distributes storage across a network holding many cheap storage devices. Each data record is stored redundantly on multiple drives, so if a drive fails the data can still be easily recovered. If only one disk fails at a time, there only needs to be one backup drive for that data. The risk can be made arbitrarily small, based on assumptions of how many disks might fail at once. Modern distributed DBs typically have three backups per data object, i.e. a replication factor of 3 [45].

In contrast, Bitcoin has about $6,500$ full nodes [46]—a replication factor of $6,500$. The chance of all nodes going down at once in any given hour (assuming complete independence) is $(1/8760)^{6500}$, or 10^{-25626}. The chance of all nodes going down would occur once every $3,000$ billion years. To say this is overkill is to put it mildly.

Of course, hardware failure is not the only reason for lost data. Attacks against the nodes of the network have a much higher probability of destroying data. A well-targeted attack to two or three mining pools could remove 50% of the computing power from the current Bitcoin network, making the network unusable until the next adjustment to POW complexity, which happens about every two weeks.

3.4. Strengths and Weaknesses

Let's review the strengths and weaknesses of DBs that use distributed consensus algorithms such as Paxos.

Strengths. As discussed above, Paxos is a field-proven consensus algorithm that tolerates benign faults (and extensions for Byzantine tolerance have been developed). It is used by "big data" distributed DBs with the well-documented ability to handle high throughput, low latency, high capacity, efficient network utilization, and any shape of data, including table-like SQL interfaces, object structures of NoSQL DBs, and graph

DBs, and they handle replication in a sane fashion. Raft, a Paxos derivative, makes distributed consensus systems easier to design and deploy.

Weaknesses. While their technical attributes and performance are impressive, traditional "big data" distributed DBs are not perfect: they are centralized. They are deployed by a single authority with central control, rather than decentralized control as in blockchains. This creates a number of failings. Centralized DBs are:

- **Controlled by a single admin user** so that if the admin user (or account) gets compromised, then the entire database might become compromised.
- **Mutable.** A hacker could change a 5-year-old record without anyone noticing (assuming no additional safeguards in place). For example, this would have prevented police from doctoring evidence in the India exam scandal [47]. In blockchains, tampering with past transactons usually quite difficult. Even if someone does manage to change a past transaction, the change is easy to spot, because the hash of its block get stored in the next block; an auditor would detect a hash mismatch.
- **Not usable by participants with divergent interests** in situations where they do not want to cede control to a single administrator. For example, the risk of losing control of the management of information is one reason that copyright rightsholders in the music industry do not share a single DB.
- **Not designed to stop Sybil attacks**, where one errant node can swamp all the votes in the system.
- **Traditionally without support for the creation and transfer of digital assets** where only the owner of the digital asset, not the administrator of the DB, can transfer the asset.
- **Not typically open to the public** to read, let alone write. Public openness is important for public utilities. A notable exception is Wikidata [48].

3.5. Fault Tolerance in the BigchainDB System

Simultaneously preserving the scalability and trustless decentralization of both large-scale databases and decentralized blockchains is the main objective of the BigchainDB system. The following were considered when designing BigchainDB's security measures:

- **Benign faults**: In the BigchainDB setup, nodes communicate through a database which uses a fault-tolerant consensus protocol such as Raft or Paxos. Hence we can assume that if there are $2f + 1$ nodes, f benign-faulty nodes can be tolerated (at any point in time) and each node sees the same order of writes to the database.
- **Byzantine faults**: In order to operate in a trustless network, BigchainDB incorporates measures against malicious or unpredictable behavior of nodes in the

system. These include mechanisms for voting upon transaction and block validation. Efforts to achieve full Byzantine tolerance are on the roadmap and will be tested with regular security audits.

- **Sybil Attack**: Deploying BigchainDB in a federation with a high barrier of entry based on trust and reputation discourages the participants from performing an attack of the clones. The DNS system, for example, is living proof of an Internet-scale distributed federation. Appendix C describes how the DNS has successfully run a decentralized Internet-scale database for decades.

4. BigchainDB Description

4.1. Principles

Rather than trying to scale up blockchain technology, BigchainDB starts with a "big data" distributed database, and adds blockchain characteristics. It avoids the technology choices that plague Bitcoin, such as full replication.

We built BigchainDB on top of an enterprise-grade distributed DB, from which BigchainDB inherits high throughput, high capacity, a full-featured NoSQL query language, efficient querying, and permissioning. Nodes can be added to increase throughput and capacity.

Since the big data DB has its own built-in consensus algorithm to tolerate benign faults, we exploit that solution directly. We "get out of the way" of the algorithm to let it decide which transactions to write, and what the block order is. We disallow private, peer-to-peer communication between the nodes except via the DB's built-in communication, for great savings in complexity and for reduced security risk[4]. This means that malicious nodes cannot transmit one message to part of the network and different message to other part of the network. Everytime a node "speaks," all the others can listen.

4.2. High-Level Description

We focused on adding the following blockchain features to the DB:

1. **Decentralized control**, where "no one" owns or controls a network;
2. **Immutability**, where written data is tamper-resistant ("forever"); and
3. **The ability to create & transfer assets** on the network, without reliance on a central entity.

Decentralized control is achieved via a DNS-like federation of nodes with voting permissions. Other nodes can connect to read and propose transactions; this makes it a super-peer P2P network [2]. The voting operates at a layer above the DB's built-in consensus. Quorum is a majority of votes. For speed, each block is written before a

[4]Though we must vigilantly exercise restraint in design, as intuition is to just get the nodes talking directly!

quorum of nodes validates and votes on it. Chainification actually happens at voting time. Every block has an id equal to the hash of its transactions, timestamp, voters list and public key of its creator-node. It also has a cryptographic signature and a list of votes. A block doesn't include the hash (id) of the previous block when it first gets written. Instead, votes get appended to the block over time, and each vote has a "previous block" attribute equal to the hash (id) of the block coming before it. Immutability / tamper-resistance is achieved via several mechanisms: shard replication, reversion of disallowed updates or deletes, regular database backups, and cryptographic signing of all transactions, blocks & votes. Any entity with asset-issuance permissions can issue an asset; an asset can only be acquired by new owners if they fulfill its cryptographic conditions. This means hackers or compromised system admins cannot arbitrarily change data, and there is no single-point-of-failure risk.

4.3. Architecture

Figure 3 illustrates the architecture of the BigchainDB system. The BigchainDB system presents its API to clients as if it is a single blockchain database. Under the hood, there are actually two distributed databases[5], **S** (transaction set or "backlog") and **C** (block chain), connected by the BigchainDB Consensus Algorithm (BCA). The BCA runs on each signing node. Non-signing clients may connect to BigchainDB; depending on permissions they may be able to read, issue assets, transfer assets, and more; section 7 explores this more.

Each of the distributed DBs, **S** and **C**, is an off-the-shelf big data DB. We do not interfere with the internal workings of each DB; in this way, we get to leverage the scalability properties of the DBs, in addition to features like revision control and benefits like battle-tested code. Each DB is running its own internal Paxos-like consensus algorithm for consistency among the drives.

The first DB holds the "backlog" transactions—an unordered set of transactions **S**. When a transaction comes in, it gets validated by the receiving node and if it's valid (according to that node), then it gets stored in **S**. (Identical transactions arriving later will be rejected.) The receiving node also randomly assigns the transaction to one of the other nodes.

There are N signing nodes. $\mathbf{S}_k = \{\mathbf{t}_{k,1}, \mathbf{t}_{k,2}, \dots\}$ is the set of transactions assigned to node **k**.

Node **k** running the BigchainDB Consensus Algorithm (BCA) processes transactions from **S** as follows: It moves transactions from the unordered set $\mathbf{S}_k$ into an ordered list, creates a block for the transactions, and puts the block into the second database **C**. **C** is an ordered list of blocks where each block has reference to a parent block and its data, that is, a blockchain.

A signing node can vote on whether it considers a block *valid* or *invalid*. To decide, the signing node checks the validity of every transaction in the block, and if it finds an

[5]This can be implemented as two databases, or as two tables in the same database. While there is no practical difference, for the sake of clarity we describe it as two separate databases.

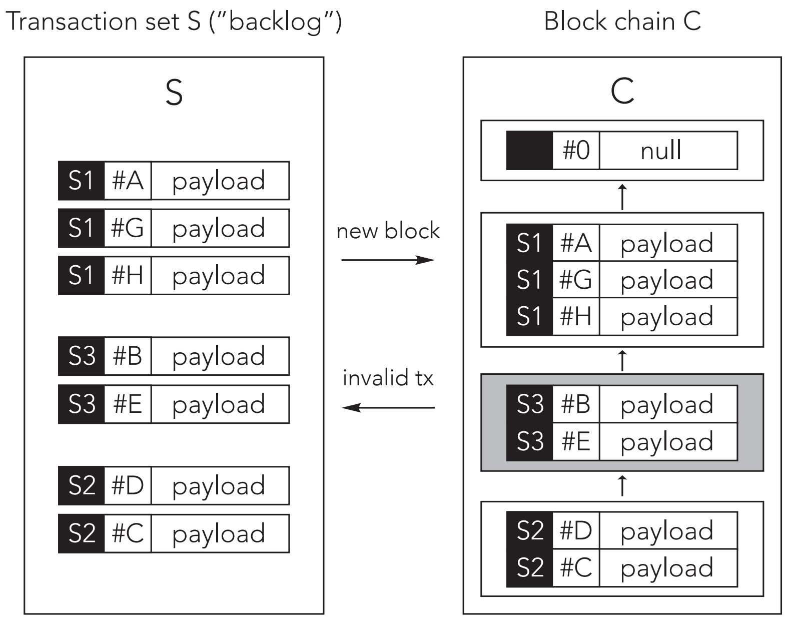

Figure 3: Architecture of BigchainDB system. There are two big data distributed databases: a Transaction Set **S** (left) to take in and assign incoming transactions, and a Blockchain **C** (right) holding ordered transactions that are "etched into stone". The signing nodes running the BigchainDB Consensus Algorithm update **S**, **C**, and the transactions (txs) between them.

invalid transaction, then the signing node votes that the block is *invalid*. If the signing node finds no invalid transactions, then it votes that the block is *valid*.

Each block starts out as *undecided*, with no votes from signing nodes. Once there is majority of positive (*valid*) votes for a block, or a majority of negative (*invalid*) votes, the block goes from *undecided* to *decided_valid* or *decided_invalid*, respectively, and voting on the block stops. Once it is decided, it can be treated as "etched into stone." This process is similar to the idea of multiple confirmations in Bitcoin blockchain.

A block **B** in the blockchain has an ID, timestamp, the actual transactions, and vote information. Section 4.5 describes block, transaction, and voting models precisely.

4.4. Behavioral Description

This section examines the flow of transactions from a client to a given server node. Each server node has its own view of the transaction backlog **S**, and the chain **C**.

Figure 4 and subsequent figures illustrate the high-level architecture where each card is a physical machine. The client machines are on the left[6]. Clients are connected to the BigchainDB server node(s) (voting node), shown on the right. Any client may send transactions to any BigchainDB server node.

[6]In some images, we truncate the illustration of the client, for brevity.

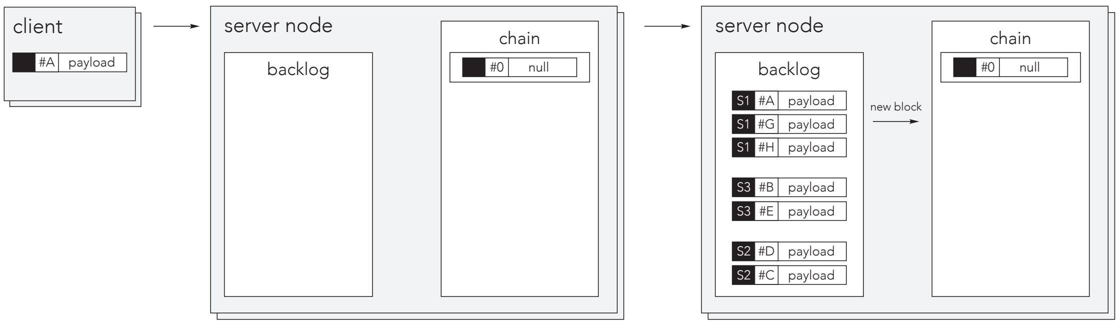

Figure 4: *Left:* The backlog **S** starts empty and the chain **C** starts with only a genesis block. *Right:* Clients have inserted transactions into backlog **S** and assigned to nodes 1, 3, and 2.

In Figure 4 left, one client has a transaction with ID #A, and a payload. BigchainDB's backlog **S** is empty; and the chain **C** is empty except for a genesis block with a null transaction. Other clients also have transactions that they transmit to server nodes.

When a client submits a transaction, the receiving node assigns it to one of the federation nodes, possibly itself, and stores it in the backlog **S**. Figure 4 right illustrates an example state. We see that node 1 is assigned three transactions, having IDs of #A, #G, and #H. Node 3 is assigned transactions with IDs #B and #E. Node 2 is assigned transactions #D and #C. Nothing has been stored on the chain **C** yet (besides the genesis block).

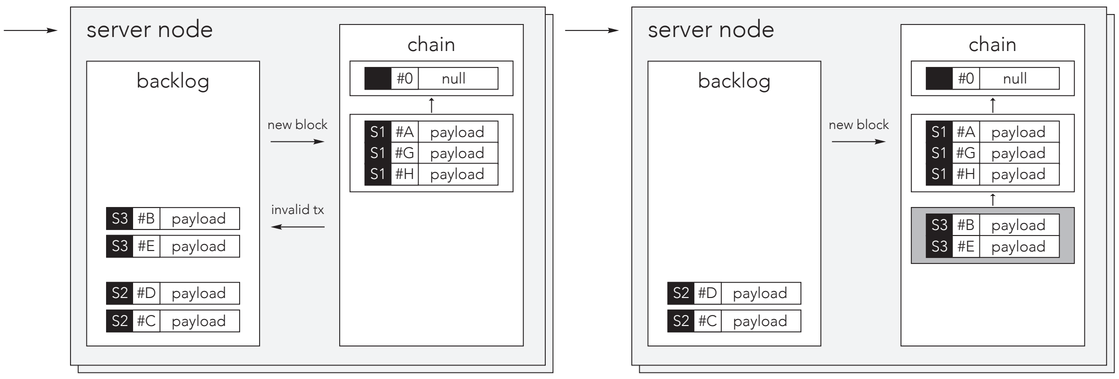

Figure 5: *Left:* Node 1 has moved its assigned transactions from backlog **S** to chain **C**. *Right:* Node 3 has processed its assigned transactions too.

Figure 5 left shows a state where Node 1 has processed all the transactions assigned to it. It has taken the transactions #A, #G, and #H from the backlog **S**, created a block to hold them, then written the block onto the chain **C**. The block points to **C**'s previous block.

Figure 5 right shows where Node 3 has processed all of its assigned transactions too, and therefore written them as a block in chain **C**.

When a block is first written to **C**, it starts off as *undecided*. Each server node may vote positively (for) or negatively (against) a block. A block should only be voted positively if all previous blocks are not *undecided*, and all transactions in the block itself are valid. As soon as there is a majority of positive votes for a block, or a majority of negative votes, the block goes from *undecided* to *decided_valid* or *decided_invalid*, respectively.

In this example, the block created by Node 1 gets voted on, and becomes *decided_valid*. Then, the block from node 3 gets voted on, and becomes *decided_invalid*. In Figure 5 right, we depict the distinction as a clear background for *decided_valid*, versus a shaded background for *decided_invalid*.)

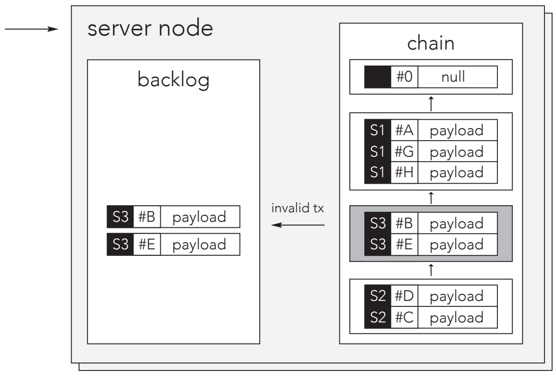

Figure 6: Transactions from an invalid block (on right, shaded) get re-inserted into backlog **S** for re-consideration.

While the overall block was considered invalid, some of the transactions in the invalid block may have actually been valid, and so BigchainDB gives them another chance. Figure 6 illustrates how: transactions #B and #E get re-inserted into backlog **S** for new consideration. Figure 6 also shows how BigchainDB approaches storage of invalid blocks. There's a block in the chain **C** that is invalid. However, BigchainDB doesn't remove the block; there's no need, as the block is already marked invalid, disk space is not a problem, and it's faster and simpler to keep all blocks there. Similarly, voting doesn't stop after a block becomes decided, because it's faster for each node to simply vote than the extra step to check whether voting is necessary.

Figure 7 emphasizes how multiple machines are configured. Figure 7 left shows that more than one client may talk to a given node. Figure 7 right shows that there is more than one node, though each node has a view into the backlog **S** and the chain **C**.

4.5. Data Models

4.5.1. Transaction Model

Transactions are the most basic kind of record stored by BigchainDB. There are two kinds: creation transactions and transfer transactions. A creation transaction initiates

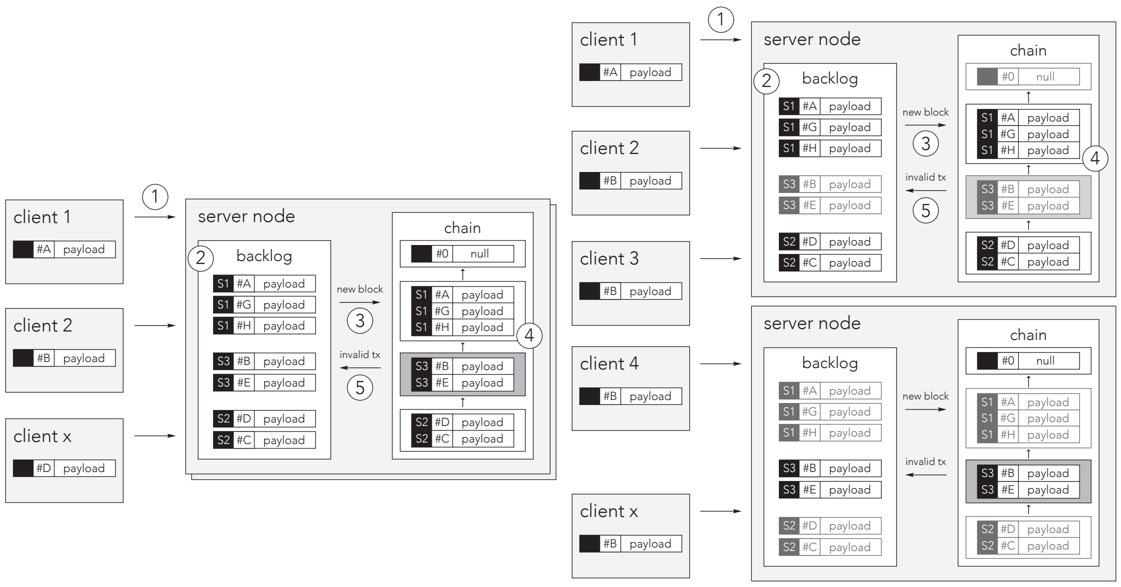

Figure 7: *Left:* More than one client may talk to a given node. *Right:* there are multiple nodes. Typically, a client connects to just one node (an arbitrarily picked one).

the records of an asset[7] in BigchainDB, including some description of what it is, a list of its initial owners, and the conditions that must be fulfilled by anyone wishing to transfer it. A transfer transaction transfers ownership of the asset to new owners, and assigns new spending conditions.

A transaction is represented by a JSON document with the following structure:

```
{
    "id": "<hash of transaction, excluding signatures>",
    "version": "<version number of the transaction model>",
    "transaction": {
        "fulfillments": ["<list of fulfillments>"],
        "conditions": ["<list of conditions>"],
        "operation": "<string>",
        "timestamp": "<timestamp from client>",
        "data": {
            "hash": "<hash of payload>",
            "payload": "<any JSON document>"
        }
    }
}
```

where:

- `id`: The hash of everything inside the serialized `transaction` body (see below), with one wrinkle: for each fulfillment in `fulfillments`, `fulfillment` is set to `null`. The `id` is also the database primary key.

[7]While we use the word “asset,” BigchainDB can be used to record information about things more general than assets.

- `version`: Version number of the transaction model, so that software can support different transaction models.
- `fulfillments`: List of fulfillments. Each fulfillment contains a pointer to an unspent asset and a crypto-fulfillment that satisfies a spending condition set on the unspent asset. A fulfillment is usually a signature proving the ownership of the asset.
- `conditions`: List of conditions. Each condition is a crypto-condition that needs to be fulfilled by by a transfer transaction in order to transfer ownership to new owners.
- `operation`: String representation of the operation being performed (currently either "CREATE" or "TRANSFER"). It determines how the transaction should be validated.
- `timestamp`: Time of creation of the transaction in UTC. It's provided by the client.
- `hash`: The hash of the serialized `payload`.
- `payload`: Can be any JSON document. It may be empty in the case of a transfer transaction.

Full explanations of transactions, conditions and fulfillments are beyond the scope of this paper; see the BigchainDB Documentation [49] and the Interledger Protocol [12] for additional details.

4.5.2. Block Model

A block is represented by a JSON document with the following structure:

```
{
    "id": "<hash of block>",
    "block": {
        "timestamp": "<block-creation timestamp>",
        "transactions": ["<list of transactions>"],
        "node_pubkey": "<public key of the node creating the block>",
        "voters": ["<list of federation nodes public keys>"]
    },
    "signature": "<signature of block>",
    "votes": ["<list of votes>"]
}
```

- `id`: The hash of the serialized `block`. This is also a database primary key; that's how we ensure that all blocks are unique.
- `block`:
  - `timestamp`: Timestamp when the block was created. It's provided by the node that created the block.

  - `transactions`: A list of the transactions included in the block.
  - `node_pubkey`: The public key of the node that created the block.
  - `voters`: A list of public keys of federation nodes. Since the size of the federation may change over time, this will tell us how many nodes existed in the federation when the block was created, so that at a later point in time we can check that the block received the correct number of votes.

- `signature`: Signature of the block by the node that created the block. (To create the signature, the node serializes the block contents and signs that with its private key.)
- `votes`: Initially an empty list. New votes are appended as they come in from the nodes.

4.5.3. Vote Model

Each node must generate a vote for each block, to be appended to that block's `votes` list. A vote has the following structure:

```
{
    "node_pubkey": "<the public key of the voting node>",
    "vote": {
        "voting_for_block": "<id of the block the node is voting for>",
        "previous_block": "<id of the block previous to this one>",
        "is_block_valid": "<true|false>",
        "invalid_reason": "<None|DOUBLE_SPEND|TRANSACTIONS_HASH_MISMATCH|
    NODES_PUBKEYS_MISMATCH",
        "timestamp": "<timestamp of the voting action>"
    },
    "signature": "<signature of vote>"
}
```

4.6. Block Validity and Blockchain Pipelining

Figure 8 shows an example of a blockchain $\mathbf{C}$. Block $\mathbf{B}_1$ is the genesis block with a null transaction.

Blocks are written to $\mathbf{C}$ in an order decided by the underlying DB. This means that when a signing node inserts a block into $\mathbf{C}$, it cannot provide a vote at the same time (because a vote includes a reference to the previous block, but the previous block isn't known yet). Only after the write is fully-committed does the block order become clear.

Nodes vote on blocks after order becomes clear. When a block is created, it starts off *undecided*. As soon as there is majority of positive votes for a block, or a majority of negative votes, the block goes from *undecided* to *decided_valid* or *decided_invalid*, respectively.

Note that, unlike a typical block chain, the BigchainDB block model doesn't have a reference to the previous block. Instead, it has a list of votes, and each vote has a reference to the previous block (i.e. the block that the voting node considered the

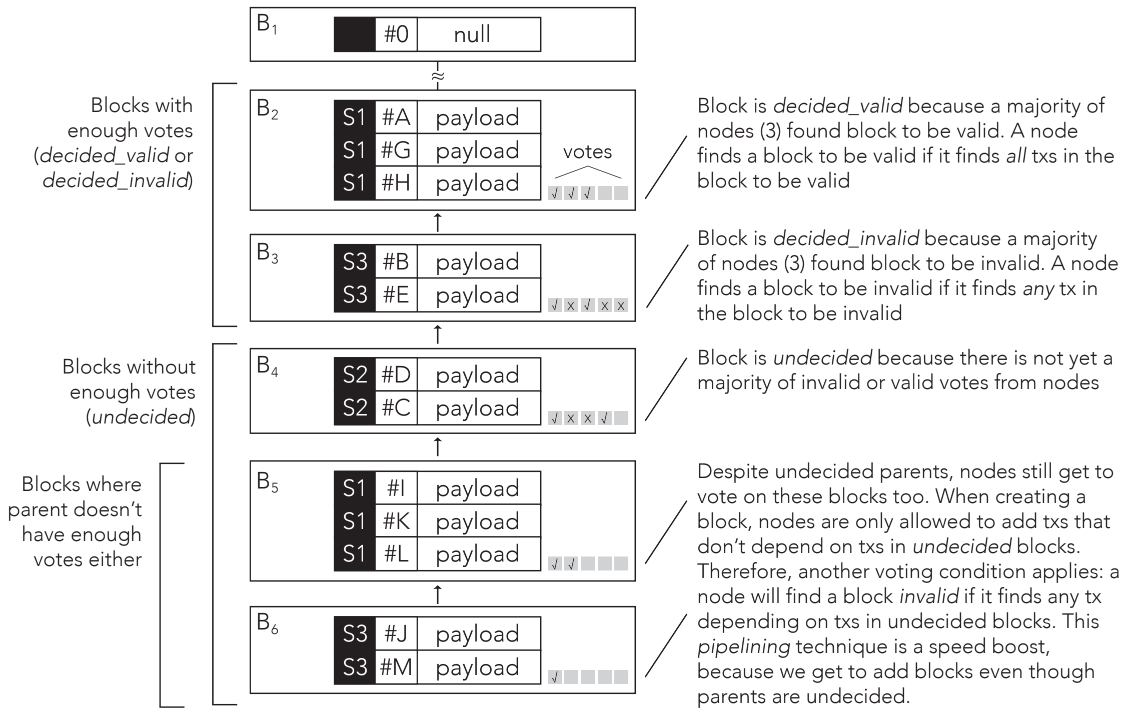
B1
#0
null
B2
S1 #A payload
S1 #G payload
S1 #H payload
votes
√ √ √
B3
S3 #B payload
S3 #E payload
√ x √ x x
B4
S2 #D payload
S2 #C payload
√ x x √
B5
S1 #I payload
S1 #K payload
S1 #L payload
√ √
B6
S3 #J payload
S3 #M payload
√
Blocks with enough votes (*decided_valid* or *decided_invalid*)
Blocks without enough votes (*undecided*)
Blocks where parent doesn't have enough votes either
Block is *decided_valid* because a majority of nodes (3) found block to be valid. A node finds a block to be valid if it finds *all* txs in the block to be valid
Block is *decided_invalid* because a majority of nodes (3) found block to be invalid. A node finds a block to be invalid if it finds *any* tx in the block to be invalid
Block is *undecided* because there is not yet a majority of invalid or valid votes from nodes
Despite undecided parents, nodes still get to vote on these blocks too. When creating a block, nodes are only allowed to add txs that don't depend on txs in *undecided* blocks. Therefore, another voting condition applies: a node will find a block *invalid* if it finds any tx depending on txs in undecided blocks. This *pipelining* technique is a speed boost, because we get to add blocks even though parents are undecided.

Figure 8: Pipelining in the BigchainDB blockchain **C**. Votes accumulate on each block. Blocks can continue to accumulate on the blockchain, even though their parents, grandparents, etc. may be undecided. The key is that when adding a new block, we can include transactions that do not depend on transactions in undecided blocks.

previous block, based on its view of the changefeed). We say that "chainification happens at *voting* time."

Normally, all votes will reference the same previous block, but it's possible that different votes may claim that different blocks are the previous block. This can happen, for example, if a node goes down. When it comes back up, there's no reliable way to recover the order of the new blocks it missed while down. What should it put as the id/hash of the previous block? We will experiment with various options (e.g. an id/hash value of "null" or the same id/hash as the majority of other votes).

If a node goes down, it won't accumulate a long list of new blocks that it must vote on when it comes back up. That's because the list of expected voters (nodes) for a block is set when that block is created. If a node is down when a block is created, then it won't be put on the list of expected voters for that block.

Block $\mathbf{B}_2$ has received three votes of five possible. In this example, all three votes are positive. Since the majority of nodes voted that the block is *valid*, the block is considered *decided_valid*.

Block $\mathbf{B}_3$ has received five votes of five possible. There was a positive vote, then negative, then positive, then two more negative votes. Since the majority of nodes voted that the block is *invalid*, the block is considered *decided_invalid*. This block can stay in the chain because all the votes show that it is invalid. It will be ignored when validating future transactions. By keeping the block in place, we can quickly progress the chain to child blocks.

Block $\mathbf{B}_4$ is *undecided* because it does not yet have a majority of invalid or valid votes from nodes. Voting on $\mathbf{B}_4$ continues.

It is crucial that despite $\mathbf{B}_4$ being *undecided*, it still has a child block $\mathbf{B}_5$. This is possible because the DB's built-in consensus algorithm determines the order of the blocks, and we have logically separated writing blocks from voting. "Forking" is not a risk as it is not even in the vocabulary of the DB, let alone supported in code. The reason is that every node is working on the same blockchain (instead of every node working on their own replica of the blockchain which may be different from the other nodes) and every node communicates through the database which is basically an open broadcast channel (instead of communicating individually with each node). Because of this, any node can try to add a block, but only one becomes the child to $\mathbf{B}_4$; the rest follow according to the built-in consensus algorithm. It is a single railroad track where the location of the next plank is based on previous planks. We do not have to stop at adding a single plank after an undecided block—we can keep aggressively laying track, such as block $\mathbf{B}_6$ in the figure.

When there are undecided parent blocks, we need to do one more thing to prevent double-spending: any transaction put into a new block must not depend on transactions in an undecided block. For example, inputs of a new transaction must not be in inputs of any undecided blocks. This is enforced in two ways: when creating new blocks on undecided blocks, such double-spend transactions are not allowed, and when voting, any block containing such transactions is voted invalid.

We call this "blockchain pipelining" because this behavior is reminiscent of pipelining in microprocessors. There, the microprocessor starts executing several possible instructions at once. Once the microprocessor has worked out the proper order for the instructions, it collates the results as output and ignores useless results. As with microprocessor pipelining, blockchain pipelining gives significant speed benefits.

4.7. BigchainDB Consensus Algorithm (BCA)

The BigChainDB Consensus Algorithm (BCA) is a state machine that runs on each "signing" node (server). This section outlines the BCA using Python-like pseudocode.[8]

4.7.1. Main Loop

Before starting the `mainLoop()` on each signing node, the databases **S** and **C** must be created and initialized. One of the initialization steps is to write a genesis block to **C**.

Listing 1 has high-level pseudocode for the BCA. It shows the `mainLoop()` running on signing node **k**. Every signing node runs the same `mainLoop()`.

[8] The actual code will be open source, so if you're curious about implementation details, you can read that.

Line 4 emphasizes that there is equal access by all the nodes to the databases **S** and **C**. The BCA operates by moving data from transaction set **S** to blockchain **C**, and occasionally in the other direction as well.

Listing 1: BigchainDB Consensus Algorithm. This algorithm runs on every signing node.

```
1 def mainLoop(): # Pseudocode for signing node k
2   # Assume S and C exist and are initialized,
3   # and C contains a genesis block.
4   global S, C # tx set and blockchain globally visible
5   while True:
6     S = assignTransactions(S, k)
7     Sk, C = addBlock(Sk, C, k)
8     C = voteOnBlocks(C, k)
9
```

Line 5 is the start of the main loop. All remaining pseudocode is part of this loop, which runs continuously until the node is shut down.

Line 6 accepts transactions into **S** and assigns them to nodes, line 7 moves unordered transactions from **S** into ordered, grouped-by-block transactions in **C**, and line 8 is where the node votes on undecided blocks.

Listing 2: Parallel version of BigchainDB Consensus Algorithm.

```
def mainLoopParallel():
  start => 1 assignTransactionLoop() processes
  start => 1 addBlockLoop() processes
  start => 1 voteLoop() processes

def assignTransactionLoop():
  while True:
    S = assignTransactions(S, k)

def addBlockLoop():
  while True:
    Sk, C = addBlock(Sk, C, k)

def voteLoop():
  while True:
    C = voteOnBlocks(C, k)
```

The pseudocode of Listing 1 is written as if there is a single process, but each major step can actually be a separate, independent process. In fact, there may be multiple processes doing each step; this helps performance tremendously. Listing 2 shows the pseudocode.

4.7.2. Assigning Transactions

Listing 3 algorithms are for assigning transactions, as follows:

Listing 3 `assignTransactions()` is the main routine that groups the two major steps: accepting and assigning incoming transactions (line 2), and reassigning old transactions (line 3).

Listing 3 `assignNewTransactions`() shows how a node accepts an incoming transaction from a client and assigns it to another node. The receiving node first checks if the transaction is valid. If it's invalid, an error message is sent back to the client. If it's valid (according to the receiving node), it gets randomly assigned to one of the other nodes. We assign transactions to a node rather than allowing nodes to grab transactions, because assignment greatly reduces double-spend detections in the block chain building side, and therefore helps throughput. We considered assigning nodes deterministically, for example based on the hash of the transaction. However, that would be problematic if a malicious node repeatedly inserted a bad transaction into **C**, then when it got kicked back to **S**, the malicious node got the same transaction again. Instead, we assign the node randomly with equal probability to each node, except the current node **k** in order to avoid a duplicate vote.

In the algorithm, line 7 accepts transactions and loops through them; line 8 checks the validity of the transaction; line 9 chooses which node to assign the transaction to, with uniform probability; line 11 records the assign time (see the next algorithm for why); and line 12 actually assigns the transaction to the chosen node.

Listing 3: Routines for accepting and assigning transactions.

```
1  def assignTransactions(S, k):
2    S = assignNewTransactions(S, k)
3    S = reassignOldTransactions(S, k)
4    return S
5
6  def assignNewTransactions(S, k):
7    for each new tx, t from outside:
8      if t is valid:  # defined later
9        i ∼ U({0, 1, ..., k-1, k+1, ..., N-1})
10       # i is chosen randomly from all nodes but this one (k)
11       t.assign_time = time()
12       Si = Si ∪ t
13     else:
14       # inform the sending-client why t is not valid
15   return S
16
17 def reassignOldTransactions(S, k):
18   for Sj in {S1, S2, ...}:
19     for each tx, t, in Sj:
20       if (time() - t.assign_time) > old_age_thr:
21         i ∼ U({0, 1, ..., k-1, k+1, ..., N-1})
22         t.assign_time = time()
23         Si = Si ∪ t
24         Sj = Sj - t
25   return S
26
```

Listing 3 `reassignOldTransactions`() re-assigns transactions that are too old. Transactions can get old if a node goes down, is running slowly, is acting maliciously, or is not performing its duties more generally. This routine ensures transactions assigned to a node don't get stuck in limbo, by re-assigning old-enough transactions to different nodes. It loops through all assigned transactions (lines 18-19), and if a transaction is old enough (line 20) a new node is randomly chosen for it (line 21), a new assign time is set (line 22), and the transaction is re-assigned from the old node (node **j**) to the new node (node **i**). For this routine to work, it also needs the unassigned-transactions to have an assign time, which is done in `assignNewTransactions`() (line 11)).

4.7.3. Adding and Voting on Blocks

Listing 4 `addBlock`() creates and adds a (non-genesis) block to **C**, and ends with a set of transactions to postpone

Listing 4: Routine for adding normal blocks.

```
1  def addBlock(Sk, C, k):
2    $T_{postpone}$ = {}
3    $B_{new}$ = ∅
4    $B_{tail}$ = most recent block in C
5    $T_{new}$ = []
6    for t in Sk:
7      if dependsOnUndecidedBlock(t, $B_{tail}$):
8        $T_{postpone}$ = $T_{postpone}$ ∪ t
9      elif transactionValid(t, $B_{tail}$):
10       $T_{new}$.append(t)
11   id = sha3 hash of {$T_{new}$, other data wrt Sec. 5.5}
12   votes = []
13   $B_{new}$ = Block(id, $T_{new}$, votes, other data wrt Sec. 5.5)
14   add $B_{new}$ to C # Consensus algorithm will determine order
15   Sk = ∅
16   Sk = Sk ∪ $T_{postpone}$
17   return Sk, C
18
```

Lines 2 – 3 initialize the routine's main variables – the block to add $\mathbf{B}_{\texttt{new}}$, and the transactions to postpone adding until later $\mathbf{T}_{\texttt{postpone}}$.

Lines 4 – 17 creates a block and adds it to **C**, in an order determined by **C**'s consensus algorithm. Line 4 updates its pointer $\mathbf{B}_{\texttt{tail}}$ to the most recent block in **C**. It is important to grab $\mathbf{B}_{\texttt{tail}}$ here rather than computing it on-the-fly, in case new blocks are added while the rest of the routine is running. Line 5 initializes the ordered list of transactions to be added to the block, and lines 7 – 10 add them one at a time. If a transaction **t** depends on an *undecided* block (risking double-spend) it will be postponed to another block by being added to $\mathbf{T}_{\texttt{postpone}}$ (lines 7 – 8). Otherwise, if it is considered valid, then it is added to $\mathbf{T}_{\texttt{new}}$ (lines 9 – 10). Otherwise, it will be discarded. Lines 11 – 14 create the block and add it to **C**.

Listing 4 lines 15 – 16 occur once the block has been successfully added. With new transactions now in **C**, those transactions can be removed from **S**, as line 15 does by clearing $\mathbf{S}_k$. Line 16 reconciles by adding back any postponed transactions, for example any transactions that risked being double-spends due to being added after an *undecided* block.

Listing 5: Routine for voting on blocks.

```
1  def voteOnBlocks(C, k):
2    B = oldest block in C that node k hasnt voted on
3    while B:
4      vote = transactionsValid(B)
5      B.V[k] = vote
6      if B is decided and invalid: copy txs from B back into S
7      B = (child block of B) or ∅
8    return C
9
```

Listing 5: Routine for voting on blocks `voteOnBlocks`() is the routine for node **k** to vote on blocks that it hasn't yet voted on.

Note that a node actually votes on blocks that may have already been decided, because it's faster to vote than to first query whether the block is decided. Lines 3 − 8 iterate from the oldest block that node **k** hasn't voted on (found in line 2) to the newest block (when temporary variable goes to ∅ in line 7). For each block, line 4 computes a Boolean of whether all transactions in the block **B** are valid, and line 5 stores that in **B**'s votes variable **B**.**V**, signed by node **k**. Line 6 gives potentially valid transactions another chance.

4.7.4. Transaction Validity

Listing 6: Routines for transaction validity.

```
def transactionsValid(T, Bi):
  # are all txs valid?
  for t in T:
    if not transactionValid(t, Bi):
      return False
  return True

def transactionValid(t, Bi):
  # Is tx valid in all blocks up to and including Bi?
  # (Ignore Bi+1, Bi+2, ...)
  if t is ill-formed, commits double-spend, etc.
    return False
  if dependsOnUndecidedBlock(t, Bi)
    return False
  return True

def dependsOnUndecidedBlock(t, Bi):
  # returns True if any of the inputs of t are in a block
  # that is not voted enough (enough x's or √'s)
  # in [B0, B1, ... , Bi]. Ignores [Bi+1, Bi+2, ...]

```

Listing 6 outlines the routines for determining the validity of transactions.

`transactionsValid`() is the top-level routine to simply loop through all the transactions supplied in the transaction list **T** (lines `3` − `6`), and if any transaction is found to be invalid (line `4`) the routine returns False.

`transactionValid`() measures whether a transaction is valid, based on traditional blockchain validity measures (ill-formed, double-spend, etc.) in lines `11` − `12` and also based on whether it depends on an undecided block (lines `13` − `14`).

`dependsOnUndecidedBlock`() clarifies what it means to depend on an undecided block.

4.8. Consensus Algorithm Checklist

As we were designing the BCA, we addressed some concerns described below.

Block construction order. When a node finalizes the creation of a block, that block must not allow any more transactions to be added to it. This is to ensure that blocks created after the block can check that their transactions don't double-spend assets spent by previous blocks' transactions. This wouldn't be possible if a block could get more transactions added to it after block finalization.

Hashing votes. Is there transaction malleability because votes are not hashed? This may look like a problem, because a block's hash can be propagated to its child block before all its votes are in. A preliminary answer would be to have a second chain of hashes that actually includes the votes. But the solution can be simpler than that: a hash without votes is fine because the votes are digitally signed by the signing nodes, and therefore not malleable.

Dropping transactions. If a node goes down, what happens to the transactions assigned to it? Do those transactions get dropped? In our initial design, the answer was mostly no, because all transactions are stored in **S** until they have been committed to a block. However, if a node went down or, more generally misbehaved, transactions assigned to that node might not be handled. To address this, we added a way to re-assign transactions if the previous node assignment got stale: algorithm `reassignOldTransactions`() in Listing 3.

Denial of service. Are there any transactions that can be repeatedly called by aggressor clients or a malicious server node, which tie up the network? To our knowledge, this is not an issue any more than with a traditional web service.

Client transaction order. We must ensure that transactions sent from the same client in a particular order are processed in that order—or at least with a bias to that order. When a client sends two transactions to the same node, that receiving node could change their order before writing them to the backlog. That wrong ordering would probably be preserved by the RethinkDB changelog, so all other nodes would see the same wrong ordering. At the time of writing, we were considering several possible solutions, including using multi-node consensus on client-provided timestamps.

Database built-in communication vulnerability. The nodes communicate using the big data DB's own built-in consensus algorithm like Paxos to tolerate benign failures.

Is this a vulnerability? The answer is that many nodes would have to be affected for it to have any major consequences.

Double spends. Are there any ways to double-spend? This is a useful question to keep asking at all stages of development. In this regard BigchainDB does exactly the same as the Bitcoin network. All past transactions are checked to make sure that input was not already spent. This can be fast for BigchainDB because it can use an optimized query of the underlying DB.

Malicious behavior. Questions: How does BigchainDB detect that a node has bad (Byzantine) behavior? Does it discourage bad behavior? How? Answers: Overall, it's a simpler problem because of the federation model. Bad behavior can be detected when a node's vote on a block is different than the majority. There are many possible ways to discourage bad behavior, from manual punishment decided by the federation, to needing to post a security deposit (bond) and automatically losing it upon bad behavior.

Admin becoming god. Does the system administrator have any powers that allow them to play "god", and thus constitute a single point of failure? We were careful to limit the power of the system administrator to even less than a voting node. So to our knowledge, the system administrator cannot play god because all write transactions (including updating software) need to be voted on by the federation.

Offline nodes. Q: What happens if a voting node goes offline? If many go offline? A: One or a few offline nodes is fine, as a quorum (the number of nodes needed to decide a block) is still online. If there are many offline nodes, then a block could get stuck in an undecided state. (Transactions depending on transactions in the undecided block would also get stuck in the backlog.) Our solution is to wait until enough nodes come back online to vote on and decide the block. It is a reasonable solution, because all consensus algorithms require some minumum proportion of nodes to be online to work. An alternative solution would be to limit the amount of time that a block can be undecided before being marked *decided_invalid*, so that all its transactions can be copied back to the backlog for reconsideration.

Chaining blocks rather than transactions. Q: Why do we chain together blocks, rather than chaining together transactions? A: There are several reasons. First, it's easier to write 1000 blocks per second (each containing up to 1000 transactions) than it is to write 1 million transactions per second (to the blockchain database). Second, each voting node only has to append a vote (data structure) to each block, rather than to each transaction, saving a lot of storage space. (If a voting node votes yes for a block, then we can conclude that it found all the contained transactions to be valid.) Lastly, when constructing a vote, the signing node must compute a cryptographic signature. That takes time. We save time by doing that only once per block (rather than per transaction).

4.9. Transaction Validity, Incentivization, and Native Assets

There are many things to check when determining if a transaction is valid. Signatures must be valid. Certain fields must be present (and no others). Various values must have the correct syntax. If the transaction is to create or register a new asset, then the same asset must not already exist. If the transaction is to transfer an asset, then the asset must exist, the transfer transaction must be requested by the current owner (who must sign it with their private key), not by a previous owner and not by a non-owner. "You can only spend what you have."

Every voting node checks the validity of every transaction (so it can decide how to vote on the transaction's block). Recall that BigchainDB consensus is federation-based. A node gets to vote on a transaction based on whether it has been given a voting node role. Contrast this to a POW model, where the probability of a node voting is proportional to its hash power, which assuming all miners have state-of-the-art hardware is equivalent to electricity spent; or to POS where probability of a node voting is proportional to how much money it has.

Traditionally, blockchains have held two types of assets. "Native assets," like Bitcoins or Litecoins, are built into the core protocol. The consensus uses these assets to measure transaction validity and to reward voting by native-asset transaction fees and mining rewards. Second are non-native "overlay assets" in overlay protocols sitting above the core protocol (e.g. SPOOL [10]). However, this traditional approach to native assets and reward has weaknesses:

- **Overlay Asset Double-Spend.** Traditional blockchains' consensus models do not account for overlay assets. There is nothing at the core protocol level to prevent a double-spend of an overlay asset[9].
- **Native Asset Friction to Network Participation.** Traditional blockchain voting nodes need to get paid in the native asset, so any new participants in the network must acquire the native asset, typically on an exchange, before being able to conduct a transaction. Acquiring the native asset is especially difficult on newer blockchains with native assets that are not yet available on many exchanges. This is a high barrier to entry when compared to traditional web services, where any new participant can conduct a transaction by paying in a standard currency like U.S. dollars with a standard payment method like a credit card.

BigchainDB overcomes these issues as follows (and as shown in Table 2):

- **Native consensus voting on every asset.** Every transaction keeps track of which asset it is operating on, chaining back to the transaction that issued the

[9]Ethereum is unlike traditional blockchains in this regard. According to Christian Lundkvist, "If a non-native asset has well-reviewed and tested rules (such as the Standard Token Contract) then the core protocol makes sure that the contract is executing correctly, which does enforce things like protection against double spending of tokens/non-native assets. Furthermore, in upcoming hard forks Ether will be implemented as a token on par with other tokens, i.e. using smart contract logic."

asset. Every asset is “native” in the sense that it’s used to measure transaction validity. This overcomes the issue of “asset overlay double-spend.”

- **Low friction to network participation.** Like a traditional web service, the network operator sets the terms on how to join the network and conduct transactions—but in this case the network operator is the collective will of the voting nodes. The voting nodes also determine how users pay for transactions.

Table 2: Native Assets Validity & Incentivization

	Traditional Blockchain		BigchainDB	
	Native Asset	Overlay Assets	Native Asset	Overlay Assets
Asset Types	Y (one)	Y (multiple)	Y (multiple)	N
Validated by Blockchain Consensus	Y	N	Y	N/A
Incentivization Via	Native Asset (e.g. Mining Reward)		External Fees (e.g. Transactions, Storage)	
Payment for Transactions	Obtain Native Assets (e.g. Via an Exchange)		Fiat Currency (e.g. Traditional channels)	

4.10. Incentivization & Security

In POW and POS blockchains, the network, incentive model, and security of the network are inextricably linked. Security is intrinsic to the system. But as discussed in section 2 and appendix B, both POW and POS have scalability challenges. Though it’s a double-edged sword: when incentives are intrinsic to the system too, there is motivation to game the system. An example is the emergence of mining pools to benefit the most from Bitcoin’s built-in incentive (mining rewards).

In a federation like BigchainDB, the security of each node and aggregate security over the entire network are extrinsic. This means the rules for confidentiality, availability, and integrity are outside the core network design [50]. For instance, if all nodes have weak rules for security, the network will be breached. By contrast, if a minimum fraction of the network nodes have reasonable security standards, the network as a whole can withstand attacks. Extrinsic incentives can have benefits: in a private deployment, the network participants are motivated simply by the benefits of being part of the network (e.g. lower costs, lower fraud, new functionality). Extrinsic incentives can work in a public deployment too, for similar reasons: the voting nodes may have their own reasons for an open, public database to survive, for example a mandate as a nonprofit, and this makes the interests aligned (see also section 7.5).

5. BigchainDB Implementation Details

5.1. Choice of Distributed DB

The BigchainDB design is flexible enough to have been built on top of a wide variety of existing distributed DBs. Of course, we had to choose a first one to build on. To select which, we first did benchmarking, then added additional criteria.

There are > 100 DBs to choose from, listed for example at [51] and [52]. This was our shortlist: Cassandra [53], HBase [54], Redis [55], Riak [56], MongoDB [57], RethinkDB [58], and ElasticSearch [59]. Each of these DBs uses Paxos or a Paxos descendant such as Raft [33].

First, we did preliminary performance investigation of the DBs in our shortlist: Each had $15 - 105$ writes/s per thread, $290 - 1000$ serial reads/s per thread, and $80 - 400$ reads/s per thread.

While there was some variation in performance among the DBs, the key thing to notice is that performance is per thread: performance improves as the number of threads increases. This is different than traditional blockchain technologies, where performance stays flat or worsens.

Given that all DBs tested had good scalability properties, we realized that other criteria were even more important. In particular:

1. **Consistency.** Distributed DBs must make a trade-off between performance and consistency (in the CAP theorem [60] sense, not ACID sense [61]). For a blockchain, consistency means trustworthy ordering of transactions, so we prefer DBs with strong consistency guarantees.

2. **Automatic Change Notifications.** One way for a node to find out if a change has happened in a DB is to ask it on a regular basis (i.e. polling), but that's not as efficient as having the DB automatically notify the node of changes. We wanted a DB with automatic change notifications as a standard feature.

   Automatic change notifications bring another benefit: they improve tamper-resistance (beyond what a chain of hashes offers). If a hacker somehow manages to delete or update a record in the data store, the hashes change (like any blockchain). In addition, a datastore with automatic change notifications would notify all the nodes, which can then immediately revert the change and restore the hash integrity.

Of the options considered, we found that RethinkDB met our needs best. It has strong consistency guarantees [62] and it offers automatic change notifications ("changefeeds") as a standard feature [63]. Therefore, we built the first version of BigchainDB on top of RethinkDB.

RethinkDB is a JSON (NoSQL) database with a flexible query language [64]. It is optimized for scalable realtime feeds, which is useful for collaborative apps, streaming analytics, multiplayer games, realtime marketplaces, and connected devices / IoT[10]. It is written in C++, is open source, and has a vibrant development community [65].

[10] IoT = Internet of Things

In the future, we envision a variety of distributed databases being “blockchain-ified” according to the approach of this paper. Every relational database, document store and graph store might someday have a blockchain version.

5.2. BigchainDB Capacity

Each node in the RethinkDB cluster adds its own storage capacity to the total database capacity.

For example, if each RethinkDB node were run on a d2.8xlarge instance on Amazon Web Services (AWS), then each of those instances could contribute its (24×2000) GB = 48000 GB of storage to the database. 32 nodes would have $32 \times 48000 = 1536000$ GB total capacity, i.e. more than a petabyte. (This calculation assumes no replication. A replication factor of R would decrease the total capacity by a factor of R.)

For quick reference, Figure 9 shows how total capacity depends on the number of nodes.

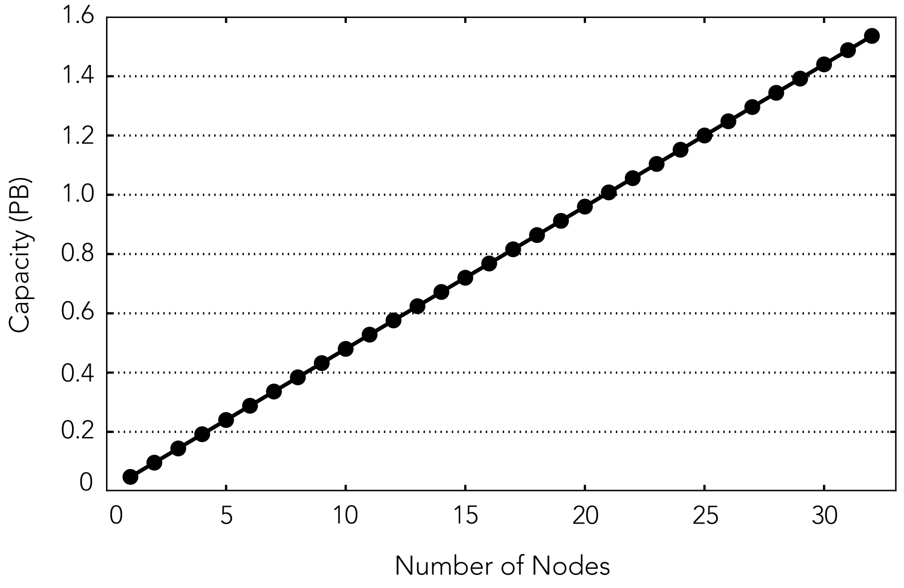

Figure 9: BigchainDB Capacity versus Number of Nodes. Each node adds another 48000 GB to the total storage capacity.

5.3. Serialization

Before we can hash or sign a JSON message (e.g. a transaction payload), we must convert it to a string in a standard way (i.e. with the same result, regardless of the programming language or computer architecture used). That is, we must serialize the JSON message in a standard way. Fortunately, there *is* a standard: RFC 7159 [66].

We do JSON serialization using the `dumps()` function in `python-rapidjson`[11], a

[11]`https://github.com/kenrobbins/python-rapidjson`

Python wrapper for `rapidjson`[12] (a fast and RFC 7159-compliant JSON parser/generator written in C++). Here's how we call it:

```
import rapidjson
rapidjson.dumps(data,
                skipkeys=False,
                ensure_ascii=False,
                sort_keys=True)
```

Here's what the parameters mean:

- `data` is the JSON message, stored in a Python dict
- `skipkeys` = `False`: Ensures that all keys are strings
- `ensure_ascii` = `False`: The RFC recommends UTF-8 encoding for maximum interoperability. By setting `ensure_ascii` to `False` we allow Unicode characters and force the encoding to UTF-8
- `sort_keys` = `True`: The output is sorted by keys

5.4. Cryptography

This section outlines the cryptographic algorithms used by BigchainDB.

5.4.1. Cryptographic Hashes

All hashes are calculated using the SHA3-256 algorithm. We store the hex-encoded hash in BigchainDB. Here is a Python implementation example, using `pysha3`[13]:

```
import hashlib
# monkey patch hashlib with sha3 functions
import sha3
data = "message"
tx_hash = hashlib.sha3_256(data).hexdigest()
```

5.4.2. Keys and Signatures

We use the Ed25519 public-key signature system [67] for generating public/private key pairs (also called verifying/signing keys). Ed25519 is an instance of the Edwards-curve Digital Signature Algorithm (EdDSA). As of April 2016, EdDSA was in "Internet-Draft" status with the IETF but was already widely used [68, 69].

BigchainDB uses the the `ed25519` Python package[14], overloaded by the cryptoconditions library[15].

All keys are represented using base58 encoding by default.

[12]`https://github.com/miloyip/rapidjson`
[13]`https://bitbucket.org/tiran/pykeccak`
[14]`https://github.com/warner/python-ed25519`
[15]`https://github.com/bigchaindb/cryptoconditions`

6. BigchainDB Transaction Latency

A key question is how long it takes for a transaction to get "etched in stone" (i.e. into a block that is *decided_valid*). To begin answering that question, we can trace the life of a transaction **t**, from the time a client sends it to the time the client gets a confirmation that **t** is in a *decided_valid* block. Figure 10 and Figure 11 illustrate the life of a transaction.

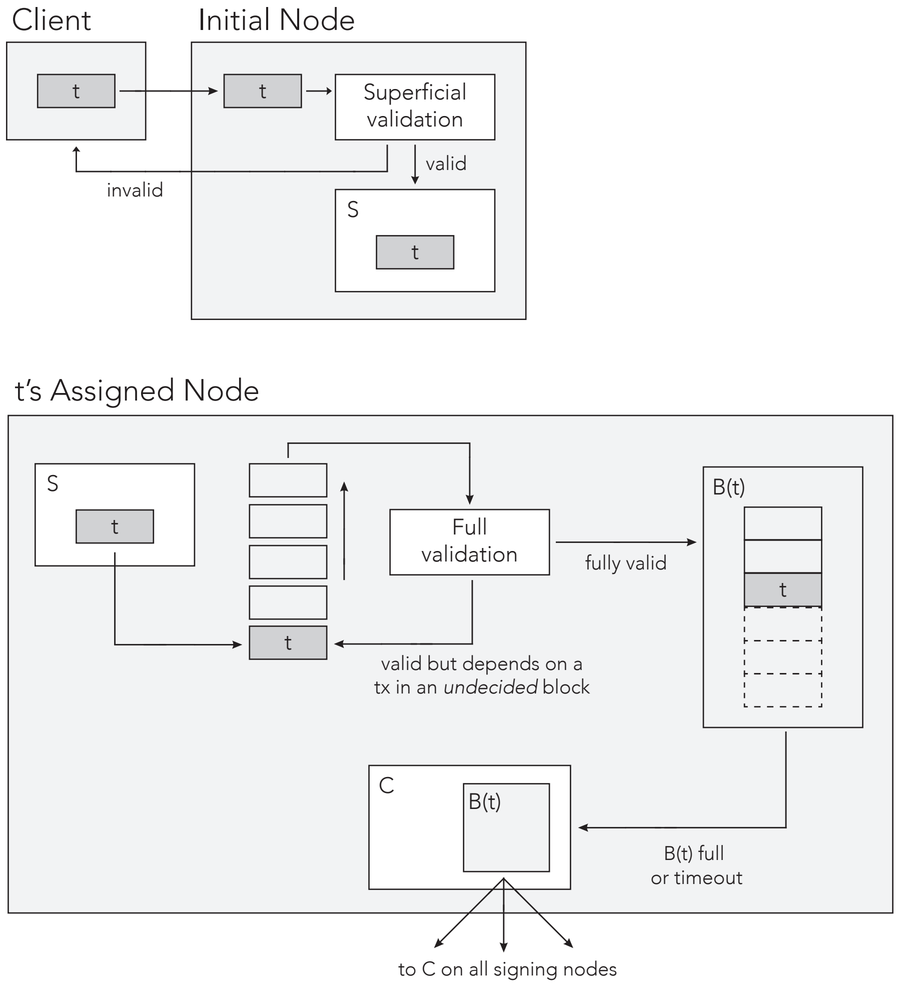

Figure 10: Life of a Transaction, Part 1/2

The time interval required for each step will vary. It can depend on how busy a node is, how busy the cluster is, network latency, and other factors. Nevertheless, we can still identify each step in the life of a transaction, to determine the main sources of latency.

Generally speaking, the client will send their transaction **t** over the Internet to a BigchainDB node. The transmission time $t_{\texttt{in}}$ depends on how far the client is from the BigchainDB node, but it will typically range from tens to hundreds of milliseconds (ms). Once **t** is in a *decided_valid* block, a BigchainDB node can send a success notification to the client. The transmission time $t_{\texttt{out}}$ will be approximately the same as $t_{\texttt{in}}$. Figure 12

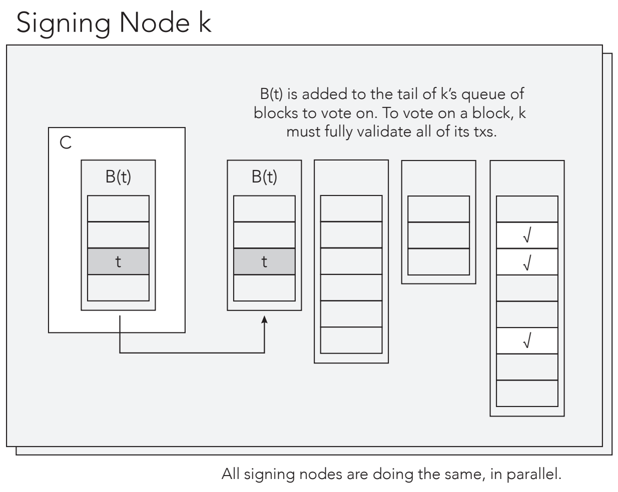
Signing Node k
B(t) is added to the tail of k's queue of blocks to vote on. To vote on a block, k must fully validate all of its txs.
C
B(t)
t
B(t)
t
√
√
√
All signing nodes are doing the same, in parallel.

Figure 11: Life of a Transaction, Part 2/2

illustrates $t_{\texttt{in}}$ and $t_{\texttt{out}}$.

We can write the total latency as:

$$t_{\texttt{total}} = t_{\texttt{in}} + t_{\texttt{internal}} + t_{\texttt{out}} \tag{1}$$

where $t_{\texttt{internal}}$ is internal latency: the latency contributed by the Bigchain DB cluster itself. $t_{\texttt{in}}$ and $t_{\texttt{out}}$ depend on the client, but $t_{\texttt{internal}}$ will be independent of the client (as a first approximation). The remainder of this section is focused on developing an estimate for $t_{\texttt{internal}}$.

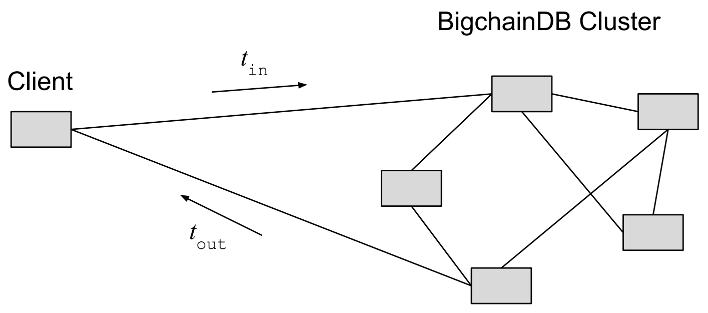

Figure 12: Transmission latencies between the client and the BigchainDB cluster.

Let's start with some notation. There are many sources of latency within the BigchainDB cluster, but a key one is the time it takes information to travel from one node to another node. Let's call the typical one-hop node-to-node latency $t_{\texttt{hop}}$. The duration of $t_{\texttt{hop}}$ depends a lot on how the nodes are distributed. If the nodes are all in one data center, then $t_{\texttt{hop}}$ might be less than 1 ms. If the nodes are distributed globally, then $t_{\texttt{hop}}$ might be 150 ms.

Another key source of latency is query latency $t_{\texttt{q}}$. If a node queries the underlying (distributed) database, it might happen that the node itself already has all the information it needs to determine the result. That is probably unusual, so we neglect that possibility. More typically, the required information is on one or more other nodes. Getting that information requires at least two internal network hops: one to send the query out, and one to get information back. For that case, we can write:

$$t_{\texttt{q}} \geq (2 \cdot t_{\texttt{hop}}) + t_{\texttt{qp}} \tag{2}$$

where $t_{\texttt{qp}}$ is the query processing time.

If all nodes in the cluster are in one data center, then $t_{\texttt{hop}}$ and $t_{\texttt{qp}}$ might be similar in duration, so we may not be able to neglect $t_{\texttt{qp}}$ relative to $t_{\texttt{hop}}$.

Let's return to figuring out a back-of-the-envelope estimate for $t_{\texttt{internal}}$. In general, it could be quite large, because a transaction might bounce back and forth between the backlog $\mathbf{S}$ and the bigchain $\mathbf{C}$ before it finally ends up in a *decided_verified* block. What we *can* do is determine an approximate *minimum* $t_{\texttt{internal}}$ (i.e. a lower bound).

When $\mathbf{t}$ arrives at a BigchainDB node, the node does a superficial validation of $\mathbf{t}$ (i.e. not checking if it depends on a transaction in an undecided block). That requires at least one query (e.g. to check if $\mathbf{t}$ does a double-spend), so the time required is at least $t_{\texttt{q}}$. (If $\mathbf{t}$ is invalid, then the client can be notified and that's the end of the story for $\mathbf{t}$.)

If $\mathbf{t}$ is valid, then the BigchainDB node assigns $\mathbf{t}$ to a randomly-choosen node. It then writes $\mathbf{t}$ to the backlog ($\mathbf{S}$). The underlying distributed database will notify all the other nodes about the change to $\mathbf{S}$ (i.e. that there is a new transaction), along with the contents of $\mathbf{t}$. It takes at least $t_{\texttt{hop}}$ time for $\mathbf{t}$ to propagate over the internal BigchainDB network.

$\mathbf{t}$ then enters the tail of a queue on the assigned node, where it waits for the assigned node to check it for validity (including whether $\mathbf{t}$ depends on a transaction in an *undecided* block). In general, there may be several transactions ahead of $\mathbf{t}$ in that queue. The assigned node must check each of those transactions first; each check requires at least one query, so at least $t_{\texttt{q}}$ time is needed to check each transaction ahead of $\mathbf{t}$. In the best case, there are no transactions ahead of $\mathbf{t}$ in the assigned node's queue, so the waiting time is zero.

Once $\mathbf{t}$ gets its turn at being considered, the assigned node must check to see if $\mathbf{t}$ is valid (including whether $\mathbf{t}$ depends on a transaction in an *undecided* block). That takes at least $t_{\texttt{q}}$ time. If $\mathbf{t}$ *does* depend on a transaction in an *undecided* block, then it must go back to waiting for consideration for inclusion in a block (i.e. back to the tail of the assigned node's queue).

Suppose $\mathbf{t}$ is okayed for inclusion in a block. Let's call that block $\mathbf{B(t)}$. $\mathbf{t}$ must wait for $\mathbf{B(t)}$ to accumulate 1000 transactions (or whatever value the BigchainDB operator sets), or for a timeout to occur (e.g. five seconds since the last transaction was added to the block). The timeout is to ensure that a block doesn't wait forever for new transactions. When there are lots of new transactions coming in, the time $\mathbf{t}$ spends waiting for $\mathbf{B(t)}$ to fill up will typically be negligible compared to $t_{\texttt{hop}}$, so we can ignore it.

The assigned node then writes $\mathbf{B}(\mathbf{t})$ to $\mathbf{C}$. It takes time for $\mathbf{B}(\mathbf{t})$ to propagate to all other nodes in the cluster: at least $t_{\texttt{hop}}$.

Each signing node will be notified about the new block $\mathbf{B}(\mathbf{t})$, including its contents. Signing node $\mathbf{k}$ will add the newly-arrived block to the tail of its queue of blocks to vote on. $\mathbf{k}$'s local copy of $\mathbf{B}(\mathbf{t})$ will wait for $\mathbf{k}$ to vote on all other blocks ahead of $\mathbf{B}(\mathbf{t})$ in $\mathbf{k}$'s queue. In the best case, there are no nodes ahead of $\mathbf{B}(\mathbf{t})$ in $\mathbf{k}$'s queue, so the waiting time is zero.

How long does it take for a node to vote on one block? If there are 1000 transactions in the block, then the node may have to check the validity of all 1000 transactions. (It doesn't always have to check all the transactions: if it finds an invalid one, it can stop before checking any more.) Once the validity checks are done, the node must compose a vote (data structure) and calculate its digital signature, but the time to do that is negligible compared to the time needed to check validity.

The node doesn't have to check the validity of each transaction one at a time. It can check many transactions in parallel at the same time, depending on how many processes are available to do validity-checking. In principle, there may be sufficient processors available to check all transactions for validity in parallel at once. Therefore, in the best case, the time to vote on one block will be approximately the same as the time to check one transaction for validity: $t_{\texttt{q}}$.

Once $\mathbf{B}(\mathbf{t})$ gets to the head of $\mathbf{k}$'s queue, $\mathbf{B}(\mathbf{t})$ might already be decided, but $\mathbf{k}$ votes on it regardless (i.e. $\mathbf{k}$ doesn't spend time checking if $\mathbf{B}(\mathbf{t})$ is already decided). As explained above, voting on $\mathbf{B}(\mathbf{t})$ takes at least $t_{\texttt{q}}$ time.

Once $\mathbf{B}(\mathbf{t})$ has gotten votes from a majority of the signing nodes, it becomes either *decided_valid* or *decided_invalid*. (The list of nodes which can vote on $\mathbf{B}(\mathbf{t})$ is set when $\mathbf{B}(\mathbf{t})$ is created, and doesn't change if nodes are added or removed from the cluster.) The deciding vote takes time $t_{\texttt{hop}}$ to propagate to all the other nodes in the cluster.

If $\mathbf{B}(\mathbf{t})$ is *decided_invalid* then the transactions inside $\mathbf{B}(\mathbf{t})$ (including $\mathbf{t}$) get sent back to $\mathbf{S}$ for reconsideration in a future block.

If $\mathbf{B}(\mathbf{t})$ is *decided_valid*, then $\mathbf{t}$ is "etched in stone" and a success notification message can be sent to the client.

We can now estimate a minumum $t_{\texttt{internal}}$ by adding up all the times outlined in the preceding paragraphs:

$$t_{\texttt{internal}} \geq 3 \cdot t_{\texttt{q}} + 3 \cdot t_{\texttt{hop}} \tag{3}$$

Then, using Eq. (2):

$$t_{\texttt{internal}} \geq 9 \cdot t_{\texttt{hop}} + 3 \cdot t_{\texttt{qp}} \tag{4}$$

If the cluster nodes are widely-distributed, then $t_{\texttt{hop}}$ is much larger than $t_{\texttt{qp}}$ and:

$$t_{\texttt{internal}} \geq 9 \cdot t_{\texttt{hop}} \tag{5}$$

As a rule of thumb for widely-distributed clusters, *the minimum internal latency is about an order of magnitude larger than the one-hop node-to-node latency.* (Remember that $t_{\texttt{internal}}$ ignores client-to-BigchainDB network latency.)

There are a few general cases, depending on how the BigchainDB nodes are distributed. Table 3 summarizes.

Table 3: Latency based on geographic distribution of the cluster

How nodes are distributed	One-hop node-to-node latency in the cluster ($t_{\texttt{hop}}$)	Minimum internal transaction latency in the cluster (minimum $t_{\texttt{internal}}$)
In one data center	≈ 0.25 ms	≈ 2.25 ms $+ 3 \cdot t_{\texttt{qp}}$
In one region (e.g. America)	≈ 70 ms	≈ 630 ms
Spread globally	≈ 150 ms	≈ 1350 ms

There are a couple of caveats to keep in mind when reading Table 3: 1) The minimum internal latency estimates in Table 3 are order-of-magnitude approximations. They should only be interpreted as *guidelines* for what to expect. 2) In a data center, the query latency $t_{\texttt{qp}}$ may be similar in magnitude to $t_{\texttt{hop}}$, so to get a better estimate of the minimum internal latency, one needs an estimate of $t_{\texttt{qp}}$.

7. Private vs. Public BigchainDB, and Authentication

7.1. Introduction

The way that BigchainDB is designed, permissioning sits at a layer above the core of the design. However, we have already seen many questions about "private vs. public" versions of BigchainDB, privacy, and authentication. In our view, a rich permissioning framework is the technology foundation. This section explores permissions, roles, private BigchainDBs, and privacy. It then has an extended section on a public BigchainDB, which we believe is tremendously important. It finally discusses authentication and the role of certificate-granting authorities.

7.2. Permissions, Identities, and Roles

Permissions are rules about what a user can do with a piece of data. Permissions are used in all kinds of computing environments, from shared file systems like Dropbox and Google Drive, to local file systems in Windows, iOS, and Linux, to distributed DBs. We should expect blockchain DBs to have rich permissioning systems.

Permissioning ideas from these other systems can inform our design. In Unix, each file or directory has three identity roles (owning user, owning group, others) and three types of permissions for each role (read, write, execute), for a total of nine permission values. For example, the permission values "`rwxr--r---`" means that the owning user can read, write, and execute (`rwx`); the owning group can read but not write or execute (`r--`), and others have no permissions (`---`).

A BigchainDB database instance is characterized by which identities have which permissions. Table 4 and Table 5 gives examples of permissions on a private and public

BigchainDB, respectively. This is loosely comparable to a corporation's internal intranet and the public Internet. We will elaborate on these shortly.

Table 4: Example Permissioning / Roles in an Enterprise BigchainDB Instance

Action	Requires vote	Voting Node	Sys Admin	Issuer	Trader	Broker	Authenticator	Auditor	Core vs Overlay
Vote on Admin & Asset Actions		Y							**Core**
Admin actions									
Update Role or Permissions	Y	Y	Y						**Core**
Add/Remove Voting Node	Y	Y	Y[16]						**Core**
Update software	Y	Y	Y						**Core**
Asset actions									
Issue Asset	Y			Y					**Core**
Transfer Asset	Y			O	O	P			**Core**
Receive Asset	Y			Y	Y				**Core**
Grant Read Access on Asset	Y			O	O	P	P		**Core**
Consign Asset	Y			O	O				**Overlay**
Receive Asset Consignment	Y			Y	Y	Y			**Overlay**
Add Asset Information	Y			O	O	P			**Overlay**
Add Authentication Information	Y			O	O		P		**Overlay**
Create Certificate of Authenticity	N			O	O	P			**Overlay**
Read actions									
Read Asset Information	N	Y	Y	O	Y	P	P	P	**Overlay**
Read Certificate of Authenticity	N	Y	Y	O	Y	P	P	P	**Overlay**

An **identity**, which signifies the holder of a unique private key, can be granted a **permission** for each transaction type. Permissions, as reflected on the tables, can be as follows: "**Y**" means the identity can perform a transaction; "**O**" means the identity can perform a transaction if the identity is the owner of the asset, which is indicated by holding the private key to that asset; and "**P**" means can perform a transaction, after the owner of the asset has given permission to the identity. Most transactions need

[16]Action is permitted only during the network initiatization process. Once a network is live, the sys admin can no longer act unilaterally.

to be voted as approved or not approved by voting nodes, with the exception of read operations.

A role is a group of individual permissions. Roles facilitate permission assignment and help clarify the rights given to users in their identity and with their permissions. Roles can be custom-created depending on the context. An identity may hold multiple roles, where the identity's permissions are the sum of the identity's role permissions and any other permissions that have been granted to it.

The core BigchainDB protocol includes as few actions or transaction types as possible, in order to maximize backwards-compatibility and minimize complexity. Overlay protocols can add new features, such as SPOOL [10] for unique asset ownership, which adds actions like consignment and authentication to property transactions. Table 4 and Table 5 each have core protocol actions, as well as some overlay protocol actions from SPOOL.

7.3. Private BigchainDBs

A private BigchainDB could be set up amongst a group of interested parties to facilitate or verify transactions between them in a wide variety of contexts, such as exchanging of securities, improving supply chain transparency, or managing the disbursement of royalties. For instance, the music industry could choose to form a trust network including record labels, musicians, collecting societies, record stores, streaming services, and support providers such as lawyers and communications agencies. A consortium of parties operating a private BigchainDB comprise an "enterprise trust network" (ETN).

Table 4 illustrates permissioning of a sample private BigchainDB instance for use in an ETN.

The first column shows actions allowed; the second column shows whether the action needs voting; columns 3-9 are roles; and the final column indicates whether the action is part of the core BigchainDB protocol.

An identity holding the "Voting Node" role (column 3) can vote on asset actions. No other role is able to do so.

Voting Nodes can update permissions for any other identity (row 2). Changing permissions requires voting consensus from other nodes. This is how new identities entering the system are assigned permissions or roles, and also how Voting Nodes are added or removed.

A Voting Node may propose an update to the Voting Node software (row 4). Voting Nodes will only update once they reach consensus. Voting Nodes also must be able to read an asset (rows 14-15) in order to be able to vote.

Like a Voting Node, an identity with "Sys Admin" role can propose to update permissions or update voting node software. This role is helpful because voting nodes may not be up-to-date technically when the Sys Admin is. Crucially, the Sys Admin cannot unilaterally update the software; rather, it can only propose software and ensure that the voting nodes hit consensus about updating. The Sys Admin must also be able to read an asset (rows 14-15), in order to debug issues that may arise with the software.

The main job of the “Issuer” role is to issue assets (row 5). But it can also do everything that a Trader role can do.

The “Trader” role conducts trades of assets, has others conduct trades on its behalf, and lines up reading and authentication of assets. It can transfer ownership to an identity, though only if it is the owner of the asset as indicated by the “O” (row 6); or be on the receiving end of an asset transfer (row 7). Similarly, if it is the owner then it can consign an asset to have another identity transact with the asset on its behalf (row 9); or be on the receiving end as consignee (row 10). By default, read permissions are off, but a Trader can allow others to read the asset info (row 10 grants permission; row 15 read). The Trader can also add arbitrary data or files to an asset (row 11).

A “Broker / Consignee” role (column 7) gets a subset of the Trader’s permissions - only what is needed to be able to sell the work on the owner’s behalf.

We describe the “Authenticator” role (column 8) further in section 7.6.

For simplicity of presentation, some details have been omitted compared to the actual implementation. For example, usually a Consignee has to accept a consignment request.

7.4. Privacy

Q: Bank A doesn’t want Bank B to see their transactions. But if they’re both voting nodes, can Bank B see Bank A’s transactions?
A: This is not really related to BigchainDB; it doesn’t care about the content of the transaction. For the transaction to be valid, it just needs to have a current unspent input and a correct signature.

Q: But if every voting node knows the identity of the nodes in a transaction, and can see the amount transacted in order to validate the transaction, than isn’t that a loss of privacy?
A: The nodes in a transaction are just public keys. The way that the mapping between a public key and an identity is done should be taken care of by Bank A and Bank B. If they want to hide the amount that is being transacted, they can do that; BigchainDB doesn’t care.

Let’s say that Bank A creates an input “A” and gives it to “PUBKA”, and inside the data field it says that this is an input for “B”, and “B” is just a serial number. BigchainDB makes sure that input “A” can only be spent once and that only “PUBKA” can spend it. There are no amounts involved.

7.5. A Public BigchainDB

7.5.1. Introduction

A BigchainDB can be configured to be more public, with permissioning such that anyone can issue assets, trade assets, read assets, and authenticate. We are taking steps towards a first public BigchainDB[17].

[17]We envision that ultimately there will be many public BigchainDBs. More than one is not a big problem, as there is no native token built into the DB. Universal resource indicators (URIs) will

7.5.2. Motivations

Decentralization technology has potential to enable a new phase for the Internet that is open and democratic but also easy to use and trust [70][71][72]. It is intrinsically democratic, or at the very least disintermediating. It is also trustworthy: cryptography enables the conduct of secure and reliable transactions with strangers without needing to trust them, and without needing a brand as proxy.

The discourse is around benefits in both the public and private sector. In the public sector, the most obvious benefit is in the future shape of the Internet and especially the World Wide Web [73]. These technologies have fundamentally reshaped society over the past two decades. The 90s Web started out open, free-spirited, and democratic. In the past 15 years, power has consolidated across social media platforms and the cloud. People around the world have come to trust and rely on these services, which offer a reliability and ease of use that did not exist in the early Internet. However, these services are massively centralized, resulting in both strict control by the central bodies and vulnerability to hacking by criminals and nation-states.

Decentralization promises a large positive impact on society. An antidote to these centralized services and concentration of power is to re-imagine and re-create our Internet via decentralized networks, with the goal of giving people control over their data and assets and redistributing power across the network.

7.5.3. Public BigchainDB Roles

Table 5 describes permissioning of a public BigchainDB instance. Here, BigchainDB is configured such that each User (column 6) can do anything, except for sensitive roles such as voting, administration, and authentication. Critically, Users can issue any asset (column 6, row 5) and read all assets (column 6, row 14); this is one of the defining features of an open blockchain.

7.5.4. Public BigchainDB Federation Caretakers

At the core of a public BigchainDB are the "Caretakers": organizations with an identity that has a "Voting Node" role. An identity with that role can vote to approve or reject transactions, and can vote whether to assign the "Voting Node" role to another identity. (Note: an organization can have more than one identity, so a Caretaker could have two or more identities with a "Voting Node" role.)

To start, the public BigchainDB will have five identities with the Voting Node role: three held by ascribe and two held by other organizations chosen by ascribe. That is, the public BigchainDB will start with three Caretakers: ascribe and two others. From there, additional Caretakers will be selected and added to the federation by existing Caretakers. Caretakers will have divergent interests to avoid collusion, but must have one thing in common: they must have the interests of the Internet at heart. In choosing Caretakers, there will be a preference to organizations that are non-profit or building

make them easy to distinguish.

Table 5: Example Permissioning / Roles in an Public BigchainDB

Action	Requires vote	Voting Node	Sys Admin	Issuer	User	Authenticator
Vote on Admin & Asset Actions		Y				**Core**
Admin actions						
Update Role or Permissions	Y	Y	Y			**Core**
Add/Remove Voting Node	Y	Y	Y[16]			**Core**
Update software	Y	Y	Y			**Core**
Asset actions						
Issue Asset	Y			Y		**Core**
Transfer Asset	Y			O		**Core**
Receive Asset	Y			Y		**Core**
Grant Read Access on Asset	Y			N/A	N/A	**Core**
Consign Asset	Y			O		**Overlay**
Receive Asset Consignment	Y			Y		**Overlay**
Add Asset Information	Y			O		**Overlay**
Add Authentication Information	Y			O	P	**Overlay**
Create Certificate of Authenticity	N			O		**Overlay**
Read actions						
Read Asset Information	N	Y	Y	O	P	**Overlay**
Read Certificate of Authenticity	N	Y	Y	O	P	**Overlay**

foundational technology for a decentralized Internet; and for diversity in terms of region, language, and specific mandate.

The right organizational structure will be critical to the success of a public BigchainDB. Governance issues have plagued the Bitcoin blockchain [74]. We can take these as lessons in the design of a public BigchainDB. We are consulting with lawyers, developers, academics, activists, and potential Caretakers to develop a strong, stable system that is transparent enough to be relied on and flexible enough to meet the needs of the network.

Ultimately, the public BigchainDB will operate entirely independently under its own legal entity. It will choose its own Caretakers and set its own rules—but it will always work toward the long-term goal of a free, open, and decentralized Internet.

We are gathering a consortium of potential public BigchainDB nodes. If you think your organization fits as one of these nodes, or have suggestions for nodes, please contact the authors.

7.6. BigchainDB Authentication of Assets

The "Authenticator" role gives a formal place for authentication and certificate-granting authorities. Examples may include a credit rating agency, an art expert certifying the authenticity of a painting, a university issuing a degree, a governmental body issuing a permit, or a notary stamping a document.

While the BigchainDB can function completely without the Authenticator role, in a decentralized network where anyone can issue assets, it is clear that third parties will step in to provide an extra layer of trust for asset buyers.

These third parties would do all the things trusted third parties do today—act as an escrow agent, place a stamp or seal of approval, issue a certificate, or rate the quality or reputation of the asset issuer.

For authentication to be issued, a Trader enables an identity to have read and authentication permission on an asset (Table 4, row 8), then the Authenticator reviews all relevant information about the asset, and issues a report as a transaction (Table 4, row 12).

The owner of the asset may then create a cryptographic Certificate of Authenticity (COA), a digital document that includes all the digitally signed authentication reports from the various authenticators. The COA is digitally signed as well, so even if printed out, tampering can be detected. The COA can then be used by the seller as a pointer to authenticity, to show that the asset is not fraudulent.

A public BigchainDB should not be prescriptive—it should be open to new sources of authority. The issuance of certificates does not have to be limited to traditional authorities. BigchainDB allows flexibility in this respect, remaining open to the many possible approaches that will undoubtedly be created by users. BigchainDB therefore limits its role to providing a mechanism for reliable gathering and aggregation of signed data.

We imagine a rich community of authorities signing assets. For example, point-of-creation software or hardware vendors could certify that a particular digital creation was created by their software or hardware at a given point in time. Individuals could leave certified reviews for movies, restaurants, consumer goods, or even reputational reviews for other individuals. Other examples could emerge from ideas in prediction markets [75], the issuance of securities, and the rating of those securities.

In a model where anyone can issue an authoritative statement about someone or something, the reputation of the Authenticator will be critically important. How do you know whose certificates you can trust? We anticipate the development of social media reputation systems that go beyond Facebook Friends and Likes. BigchainDB enables the widespread implementation of new reputation systems such as the Backfeed protocol for management of distributed collaboration [76], or systems drawing inspiration from fictional reputation economies described by Cory Doctorow (Whuffie) [77], Daniel Suarez (Levels) [78], and others.

8. Experimental Results

8.1. Goals

BigchainDB's algorithms are designed to "get out of the way" of the underlying database, so we expect the main limiter on performance to be how that underlying database interacts with the physical compute resources (e.g. write speed and I/O among nodes). Because of that, and because BigchainDB is built on top of RethinkDB, we began with experiments to test the scalability properties of RethinkDB.

Full benchmarks on the performance of BigchainDB will appear in the near future.

8.2. Experiments on Throughput

Appendix D.1 describes the details of the experimental setups.

In one experiment, we increased the number of nodes every ten seconds, up to 32 nodes. We used RethinkDB's System Statistics Table to record the write throughput over time.

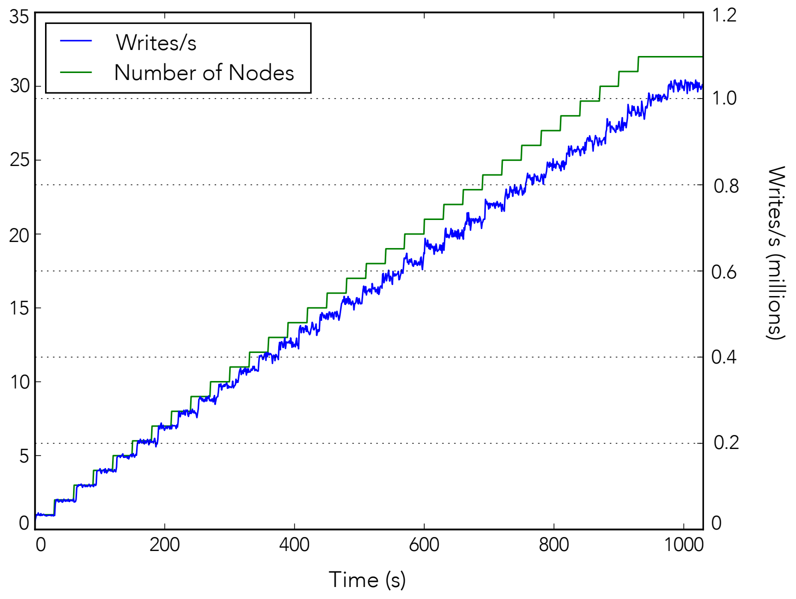

Figure 13: Time-series plot, where when we increased the number of nodes, the throughput increased proportionately.

Figure 13 shows how write throughput increased every time a node was added. When the number of nodes reached 32, the write throughput was just over 1 million transactions per second (i.e. 1000 blocks written per second, with 1000 valid transactions per block).[18]

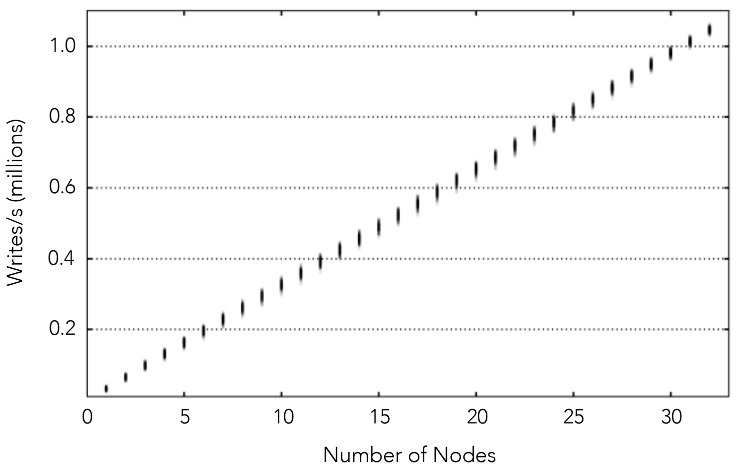

Figure 14: Write performance versus number of nodes. There is linear scaling in write performance with the number of nodes.

Figure 14 shows data from the same experiment, except it shows how write throughput was affected by the number of nodes (rather than time). The plot is both boring and exciting: it shows how write throughput increases linearly with the number of nodes.

8.3. Other Experiments

Appendix D contains descriptions and results of further experiments.

9. BigchainDB Deployment

9.1. BigchainDB Use Cases

Many BigchainDB use cases are like traditional blockchain use cases, except focused on situations where higher throughput, lower latency, or more storage capacity is necessary; where powerful querying or permissioning are helpful; or for simplicity of deployment since it feels like using a NoSQL database. For example, BigchainDB can handle the throughput of high-volume payment processors, and directly store contracts receipts, or other related documents on the DB alongside the actual transaction. Another example is "Bitcoin 2.0" applications, to keep transaction costs reasonable as the application scales[19].

[18] In Figure 13, the y-axis label of "Writes/s" should be interpreted to mean "Effective transaction-writes per second". The same is true of Figure 14.

[19] As of February 2016, the fee required to ensure that a Bitcoin transaction goes through is about \$0.10 USD. There is no predefined fee for Bitcoin transactions; it's determined by market forces. To estimate the fee, we can look at the fees that were paid in the last X blocks, and use that information

Some BigchainDB use cases are also like traditional distributed DB use cases, except focused where blockchain characteristics can benefit: decentralization, immutability, and the ability to create and transfer digital assets.

BigchainDB use cases include:

- **Tracking intellectual property assets** along the licensing chain. BigchainDB can reduce licensing friction in channels connecting creators to audiences, and gives perfect provenance to digital artifacts. A typical music service has 38 million songs—BigchainDB could store this information in a heartbeat, along with licensing information about each song and information about use by subscribers. In another example, consider a medium-sized photo marketplace running $100,000$ transactions a day; to put this on Bitcoin would cost $\$10,000$ per day and tie up the Bitcoin network.

- **Receipts, and certification.** BigchainDB reduces legal friction by providing irrefutable evidence of an electronic action. And, BigchainDB is big enough that supporting information like receipts and certificates of authenticity (COAs) can be stored *directly* on it, rather than linking to the document or storing a hash.

- **Legally-binding contracts** can be stored directly on the BigchainDB next to the transaction, in a format that is readable by humans and computers [81].

- **Creation and real-time movement of high-volume financial assets.** Only the owner of the asset can move the asset, rather than the network administrator like in previous database systems. This capability reduces costs, minimizes transaction latency, and enables new applications.

- **Tracking high-volume physical assets** along whole supply chain. BigchainDB can help reduce fraud, providing massive cost savings. Every RFID tag in existence could be entered on a BigchainDB.

- **Smart contracts** (decentralized processing), where the application must be fully decentralized and database functionality is crucial.

- **Data science** applications where the BigchainDB captures massive data streams, and data scientists can easily run queries against BigchainDB in their data mining and analytics tasks.

9.2. Transaction Encryption

Normally, transactions stored in BigchainDB aren't encrypted, but users can encrypt the payload if they want, using the encryption technology of their choice. (The payload

to calculate a fee that will give a high probability that a transaction will go into the next block. Usually a higher fee will give a higher priority to a transaction, but in the end, it's the miners that decide whether they want to include the transaction in the block or not. Usually the fees are around 15k-20k satoshi per 1000 bytes, so that would be the average fee to pay. Useful resources are blockchain.info [79], Tradeblock [80] (in the charts, select fee/size), and the Bitcoin wiki page on transaction fees [46].

of a transaction can be any valid JSON, up to some maximum size as explained below.) Other aspects of the transaction, such as the current owner's public key and the new owner's public key, aren't encrypted and can't be encrypted.

9.3. BigchainDB Limitations

Because BigchainDB is built on top of an existing "big data" database, it inherits many of the limitations of that database.

The first version of BigchainDB is built on top of RethinkDB, so it inherits some of RethinkDB's limitations, including a maximum of 32 shards per table (increasing to 64). While there's no hard limit on the document (transaction) size in RethinkDB, there is a recommended limit of 16MB for memory performance reasons. Large files can be stored elsewhere; one would store only the file location or hash (or both) in the transaction payload.

9.4. BigchainDB Products & Services

We envision the following products and services surrounding BigchainDB:

1. **BigchainDB: a blockchain database** with high throughput, high capacity, low latency, rich query support, and permissioning.
   - For large enterprises and industry consortia creating new private trust networks, to take advantage of blockchain capabilities at scale or to augment their existing blockchain deployments with querying and other database functionality
   - BigchainDB will be available in an out-of-the-box version that can be deployed just like any other DB, or customized versions (via services, or customized directly by the user).
   - BigchainDB will include interfaces such as a REST API, language-specific bindings (e.g. for Python), RPC (like bitcoind), and command line. Below that will be an out-of-the-box core protocol, out-of-the-box asset overlay protocol [10], and customizable overlay protocols.
   - BigchainDB will support legally binding contracts, which are generated automatically and stored directly, in a format readable by both humans and computers [81]. There will be out-of-box contracts for out-of-the-box protocols, and customizable contracts for customizable protocols.
   - BigchainDB will offer cryptographic COAs, which can be generated automatically and stored directly on the BigchainDB. There will be out-of-box and customizable versions.
   - BigchainDB is built on a large, open-source pre-existing database codebase that has been hardened on enterprise usage over many years. New code will be security-audited and open source.

2. **BigchainDB as a Service**, using a public BigchainDB instance, or a private BigchainDB with more flexible permissioning.
    - For developers who want the benefits of blockchain databases without the hassle of setting up private networks.
    - For cloud providers and blockchain platform providers who want scalable blockchain database as part of their service.
    - For "Bitcoin 2.0" companies looking to keep transaction costs reasonable as they go to scale
    - Main interfaces will be a REST API directly, REST API through cloud providers, and language-specific bindings (e.g. Python).
    - With the features of the BigchainDB listed above.
3. **"Blockchain-ify your database"** service, to help others bring blockchain properties to other distributed DBs. Think MySqlChain, CassandraChain, and Neo4jChain.
    - For DB vendors looking to extend their DB towards blockchain applications.
    - For DB developers who want to play with blockchain technology.

9.5. Timeline

Like many, we have known about Bitcoin and blockchain scalability issues for years. Here's the timeline of how BigchainDB took shape:

- Oct 2014 – Gave our first public talk on big data and blockchains [82]
- Apr 2015 – Preliminary investigations; paused the project to focus on our IP business
- Sep 2015 – Re-initiated the project; detailed design; building and optimizing
- Dec 2015 – Benchmark results of $100,000$ transactions/s
- Dec 2015 – Alpha version of the software integrated into an enterprise customer prototype
- Dec 2015 – Initial drafts of the whitepaper shared with some reviewers
- Jan 2016 – Benchmark results of $1,000,000$ transactions/s
- Feb 10, 2016 – BigchainDB was publicly announced
- Feb 10, 2016 – The first public draft of the whitepaper was released
- Feb 10, 2016 – Version 0.1.0 of the software was released open-source on GitHub[20]. The software was not recommended for external use yet, but development was in the open.

[20] `http://github.com/bigchaindb/bigchaindb`. Core BigchainDB software is licensed under an Affero GNU Public License version 3 (AGPLv3). The BigchainDB drivers supported by ascribe GmbH are licensed under an Apache License (version 2.0).

- Apr 26, 2016 – Version 0.2.0 released

The current BigchainDB Roadmap can be found in the `bigchaindb/org` repository on GitHub[21].

10. Conclusion

This paper has introduced BigchainDB. BigchainDB fills a gap in the decentralization ecosystem: a decentralized database, at scale. BigchainDB performance points to 1 million writes per second, sub-second latency, and petabyte capacity. It has easy-to-use and efficient querying. It features a rich permissioning system that supports public and private blockchains. It is complementary to decentralized processing technologies (smart contracts) and decentralized file systems, and can be a building block within blockchain platforms.

11. Acknowledgments

We would like to thank the following people for their reviews, suggested edits and helpful advice: David Holtzman, Jim Rutt, Evan Schwartz, Ian Grigg, Benjamin Bollen, Carsten Stocker, Adi Ben-Ari, Jae Kwon, Juan Benet and Bruce Pon. A big spark to this project came from conversations with Jim Rutt going back to early ascribe days, about the need for scale and how to reconcile it with latency and voting. We are grateful for all advice we have received, from all the amazing folks involved. If you have feedback, please email trent@bigchaindb.com.

Appendices

A. Glossary

SQL DB – a database that stores data in table format and supports the Structured Query Language (SQL); a relational database. Example: MySQL.

Ledger – a database that stores data in a table format, where entries are economic transactions

NoSQL DB – a database that stores data in a non-table format, such as key-value store or graph store. Example: RethinkDB.

NoQL DB – a database without any query language to speak of. Obviously, this hinders data management. Example: Bitcoin blockchain.

[21]`https://github.com/bigchaindb/org/blob/master/ROADMAP.md`

Distributed DB – a database that distributes data among more than one node in a network[22]. Example: RethinkDB.

Fully replicated DB – a distributed DB where every node holds all the data.

Partially replicated DB – a distributed DB where every node holds a fraction of the data.

Decentralized DB – a DB where no single node owns or controls the network.

Immutable DB – a DB where storage on the network is tamper-resistant.

Blockchain DB – a distributed, decentralized, immutable DB, that also has ability for creation & transfer of assets without reliance on a central entity.

Bitcoin blockchain – a specific NoQL, fully-replicated, blockchain DB.

BigchainDB – a specific NoSQL[23], partially-replicated, blockchain DB.

B. Blockchain Scalability Proposals

Here, we review some proposals to solve the Strong Byzantine Generals' (SBG) problem while scaling the blockchain, and to allow blockchain-like behavior at greater scales. This list is not intended to be exhaustive.

B.1. Base Consensus Approaches

Consensus refers to the way nodes on a blockchain network approve or reject new transactions. These approaches differ in (a) how a node becomes a voting node, and (b) how each node's voting weight is set. These choices can impact the blockchain's performance.

Proof of Work (POW). POW is the baseline approach used by the Bitcoin blockchain. There is no restriction on who can enter the network as a voter. A node is chosen at random, proportional to the processing ability it brings to the network, according to a mathematical puzzle — its "hash rate". Work [83] may be SHA256 hashing (the algorithm used by Bitcoin), scrypt hashing (used by Litecoin), or something else.

POW has a natural tendency towards centralization. It is a contest to garner the most hashing power. Power is currently held by a handful of mining pools.

Proof of Stake (POS) [84]. In the POS model, there is no restriction on who can enter the network. To validate a block, a node is chosen at random, proportionally to how much "stake" it has. "Stake" is a function of the amount of coin held, and sometimes of "coin age, a measurement of how many days have passed since last time the coin voted.

POS promises lower latency and does not have the extreme computational requirements of POW.

[22]Sometimes there is confusion, and "distributed" is used when the actual meaning is that of "decentralized", most notably with the term "distributed ledger".

[23]There can be SQL support to via wrapping the NoSQL functionality or using the BigchainDB design on a distributed SQL DB

However, over the last couple years, POS proposals have evolved as issues are identified (e.g. "nothing at stake," "rich get richer," and "long range attacks") and fixes proposed. These fixes have resulted in POS protocols becoming increasing complex. Complex systems generally have more vulnerabilities, compromising security.

Federation. A federated blockchain is composed of a number of nodes operating under rules set for or by the group. Each member of a federation typically has an equal vote, and each federation has its own rules about who can join as voting node. Typically, the majority or 2/3 of the voting nodes need to agree, for a transaction or block to be accepted ("quorum").

Federations may have of any number of voting nodes. More nodes mean higher latency, and less nodes means the federation is not as decentralized as many would like. Besides voting nodes, other nodes may have permission to issue assets, transfer assets, read, and so on (super-peer P2P network).

Membership rules for voting nodes can vary widely between models. In the (pre-acquisition) Hyperledger model, the requirement was having a TLD and SSL certificate [85]. In the original Stellar model, membership was based on a social network, until it was forked [86]. In Tendermint [87], Slasher [88, 89], and Casper [90], anyone could join by posting a fixed bond as a security deposit, which would be lost if the voting node were to act maliciously[24].

Membership rules can directly affect the size of the federation. For example, in Tendermint, the lower the security deposit (bond), the more voting nodes there are likely to be.

Federations imply that to be a voting node, one must reveal their identity. This means they are not as suitable where censorship resistance is a key design spec. This is different than POW and POS.

B.2. Consensus Mashups

The base consensus approaches described above can be creatively combined.

Hierarchy of POW—Centralized. Big Bitcoin exchanges operate their own internal DBs of transactions, then synchronize a summary of transactions with the Bitcoin blockchain periodically. This is similar to how stock exchange "dark pools" operate—financial institutions make trades outside the public stock exchange, and periodically synchronize with the public exchange.

Hierarchy of Small Federation—Big Federation. An example is AI Coin [91]. The top level has 5 power nodes with greater influence, and the bottom level has 50 nodes with less influence.

Hierarchy of POW—Federation. An example is Factom [92]. The bottom level is a document store; then document hashes are grouped together in higher and higher

[24]These are arguably proof of stake. It depends whether one's definition of "proof of stake" means "has a stake, anywhere" versus "degree of voting power is a function of amount at stake".

levels, Merkle tree style; and the top level the Merkle tree root is stored on the Bitcoin blockchain.

POW then POS. An example is the Ethereum rollout plan. The Ethereum team realized that if only a few coins were in circulation in a POS model, it would be easy for a bad actor to dominate by buying all the coins, and that they needed more time to develop an efficient yet trustworthy POS algorithm. Therefore, Ethereum started with POW mining to build the network and get coins into wider circulation, and plans to switch once there are sufficient coins and the POS approach is ready.

X then Centralized then X'. This model is applied when the consensus algorithm being used gets broken. Voting is temporarily handled by the project's managing entity until a fixed version of the algorithm is developed and released. This happened with Stellar. Stellar started as a federation but the project was split in a fork [86]. Stellar ran on a single server in early 2015 while a new consensus protocol [44] was developed and released in April 2015 [93]. The new version is like a federation, but each node chooses which other nodes to trust for validation [44]. Another example of this model is Peercoin, one of the first POS variants. After a fork in early 2015, the developer had to sign transactions until a fix was released [94].

B.3. Engineering Optimizations

This section reviews some of the possible steps to improve the efficiency and throughput of existing blockchain models.

Shrink Problem Scope. This range of optimizations aims to make the blockchain itself smaller. One trick to minimize the size of a blockchain is to record only unspent outputs. This works if the history of transactions is not important, but in many blockchain applications, from art provenance to supply chain tracking, history is crucial. Another trick, called Simple Payment Verification (SPV), is to store only block headers rather than the full block. It allows a node to check if a given transaction is in the block without actually holding the transactions. Mobile devices typically use Bitcoin SPV wallets. Cryptonite is an example that combines several of these tricks [95]. These optimizations makes it easier for nodes to participate in the network, but ultimately does not solve the core consensus problem.

Different POW hashing algorithm. This kind of optimization seeks to make the hashing work performed by the network more efficient. Litecoin is one of several models using scrypt hashing instead of Bitcoin's SHA256 hashing, requiring about 2/3 less computational effort than SHA256. This efficiency gain does not improve scalability, because it still creates a hash power arms race between miners.

Compression. Data on a blockchain has a particular structure, so it is not out of the question that the right compression algorithm could reduce size by one or more orders of magnitude. This is a nice trick without much compromise for a simple transaction ledger. Compression typically hinders the ability to efficiently query a database.

Better BFT Algorithm. The first solution to the Byzantine Generals problem was published in 1980 [25], and since that time many proposals have been published at distributed computing conferences and other venues. Modern examples include Aardvark [42] and Redundant Byzantine Fault Tolerance (RBFT) [43]. These proposals are certainly useful, but in on their own do not address the need for Sybil tolerance (attack of the clones problem).

Multiple Independent Chains. Here, the idea is to have multiple blockchains, with each chain focusing on a particular set of users or use cases and implementing a model best suited to those use cases. The countless centralized DBs in active use operate on this principle right now; each has a specific use case. We should actually expect this to happen similarly with blockchains, especially privately deployed ones but also for public ones. It is the blockchain version of the Internet's Rule 34: "If it exists there is blockchain of it."

For public examples, you could use Ethereum if you want decentralized processing, Primecoin if you want POW to be slightly more helpful to the world, and Dogecoin if you want much cute, very meme. For private examples, organizations and consortiums will simply deploy blockchains according to their specific needs, just as they currently deploy DBs and other compute infrastructure.

A challenge lies in security: if the computational power in a POW blockchain or coin value in a POS blockchain is too low, they can be overwhelmed by malicious actors. However, in a federation model, this could be workable, assuming that an individual blockchain can meet the specific use case's performance goals, in particular throughput and latency.

Multiple Independent Chains with Shared Resources for Security. Pegged sidechains are the most famous example, where mining among chains has the effect of being merged [83]. SuperNET [96] and Ethereum's hypercubes and multichain proposals [97] fit in this category. However, if the goal is simply to get a DB to run at scale, breaking the DB into many heterogeneous sub-chains adds cognitive and engineering complexity and introduces risk.

. . . **and more**. The models described above are just a sampling. There continue to be innovations (and controversy [98, 74]). For example, a proposed change to the Bitcoin blockchain called Bitcoin-NG [99] aims to reduce the time to first confirmation while minimizing all other changes to the Bitcoin blockchain design. The Bitcoin roadmap [100, 101] contains many other ideas, most notably segregated witness [102].

C. Case Study: DNS as a Decentralized Internet-scale Database

C.1. Introduction

In the previous section, we reviewed big data" distributed databases (DBs), highlighting their Internet-level scalability properties and solid foundation in consensus via Paxos.

We also highlighted the core weakness: centralized control where a trusted third party is always holding the keys.

We are left with the question: Are there any precedents for distributed DBs going not only to Internet scale, but in a decentralized and trusted fashion?

There is one DB that not only operates at Internet scale, but also has decentralized control, and is crucial to the functioning of the Internet as we know it: the Domain Name System (DNS).

C.2. History of DNS

By the early 1980s, the rapid growth of the Internet made managing numeric domains a major bookkeeping headache [103]. To address this, in 1983 Jon Postel proposed the DNS. The DNS was originally implemented as a centralized DB, operated by the U.S. government. In 1993 the U.S. government handed control to Network Solutions Inc. (NSI), a private corporation.

NSI faced a dual challenge. It had to make the DNS function effectively, but in a way that took power away from any single major stakeholder, including the U.S. government and even NSI itself. David Holtzman, Chief Technology Officer of NSI, architected a solution: a federation of nodes spread around the globe, where each node's interests were as orthogonal as possible to the interests of all the other nodes, in order to prevent collusion [103]. Holtzman deployed this DB while NSI's Chief Executive Officer, Jim Rutt, worked vigorously to hold off the objections of the U.S. Commerce Department and U.S. Department of Defence, which had hoped to maintain control [104]. In the late 90s, NSI handed off DNS oversight to the Internet Corporation for Assigned Names and Numbers (ICANN), a new non-governmental, non-national organization [103].

At its core, DNS is simply a mapping from a domain name (e.g. amazon.com) to a number (e.g. 54.93.255.255). People trust the DNS because no one really controls it; it's administered by ICANN.

The DNS was architected to evolve and extend over time. For example, the original design did not include sufficient security measures, so the DNS Security Extensions (DNSSEC) were added to bring security while maintaining backwards compatibility [105].

It is hard to imagine something more Internet scale than the database underpinning the Internet's domain name system. The decentralized DNS successfully deployed at Internet scale, both in terms of technology and governance. ICANN has not always been popular, but it has lasted and held the Internet together through its explosive growth, and survived heavy pressure from governments, corporations, and hackers.

Domain names have digital scarcity via a public ledger that requires little extra trust by the user. There can be only one amazon.com in the DNS model. But DNS is a consensual arrangement. Anyone could create an alternative registry that could work in a similar manner, assigning amazon.com to someone else. The alternative registry would be near useless, however, because there is a critical mass of users that have already voted on which domain system will be in use, with their network devices by choosing what name server to use, and with their wallets by purchasing and using domain names within the existing DNS system.

C.3. Strengths and Weaknesses

Weaknesses. The DNS does not address the challenge of large scale data storage, or for the blockchain characteristics of immutability or creation & transfer of assets. But, it didn't aim to.

Strengths. The DNS shows that decentralized control, in the form of federations, can work at Internet scale. It also demonstrates that it is crucial to get the right federation, with the right rules.

D. Other Experiments

D.1. Experimental Setup

To test the writing performance, we created a process that inserts a block in the database in an infinite loop.

The block is a valid block with small transactions. In our case, we used valid transactions without any payload. An entire block occupies about 900KB.

```
while True:
  r.table(table).insert(r.json(BLOCK_SERIALIZED), durability='soft').
 run(conn)
```

In `hard` durability mode, writes are committed to disk before acknowledgments are sent; in `soft` mode, writes are acknowledged immediately after being stored in memory.

This means that the insert will block until RethinkDB acknowledges that the data was cached. In each server we can start multiple processes.

Write Units. Let's define 1 *write unit* as being 1 process. For example, in a 32 node cluster, with each node running 2 processes, we would have 64 write units. This will make it easier to compare different tests.

Sharding in distributed datastores means partitioning a table so that the data can be evenly distributed between all nodes in the cluster. In most distributed datastores, there is a maximum number of shards per table. For RethinkDB, that limit is 32 shards per table.

In RethinkDB, a shard is also called a primary replica, since by default the replication factor is 1. Increasing the replication factor produces secondary replicas that are used for data redundancy. If a node holding a primary replica goes down, another node holding a secondary replica of the same data can step up and become the primary replica.

Compute Resources. For these tests we are using 32-core AWS EC2 instances with SSD storage and 10Gbps network connections (`c3.8xlarge`). For the tests, we used either 32- or 64-node clusters all running in the same AWS region.

D.2. Experiments on Throughput

The experimental setup is like the setup described in section D.1.

D.2.1. Experiment 1

Settings:

- **Number of nodes:** 32
- **Number of processes:** 2 processes per node
- **Write units:** $32 \times 2 = 64$ write units

Results:

- **Output:** stable 1K writes per second

This was the most successful experiment. We were able to reach a stable output of 1K blocks per second. The load on the machines is stable and the IO is at an average of $50 - 60\%$.

Other tests have shown that increasing the number write units per machine can lead to a stable performance up to 1.5K writes per second but the load on the nodes would increase until the node would eventually fail. This means that we are able to handle bursts for a short amount of time ($10 - 20$ min).

This test can be used has a baseline for the future in where 64 writes equal 1K transactions per second. Or, that each write unit produces an output of $(1000/64) \approx 64$ writes per second.

D.2.2. Experiment 2

Settings:

- **Number of nodes:** 32
- **Number of processes:**
  - 16 nodes running 2 processes
  - 16 nodes running 3 processes
- **Write units:** $16 \times 3 + 16 \times 2 = 80$ write units
- **Expected output:** 1.25K writes per second

Results:

- **Output:** stable 1.2K writes per second

Increasing a bit the number of write units shows an increase in output close to the expected value but in this case the IO around 90% close to the limit that the machine can handle.

D.2.3. Experiment 3

Settings:

- **Number of nodes:** 32
- **Number of processes:**
  - 16 nodes running 2 processes
  - 16 nodes running 4 processes
- **Write units:** $16 \times 4 + 16 \times 2 = 96$ write units
- **Expected output:** 1.5K writes per second

Results:

- **Output:** stable 1.4K writes per second

This test produces results similar to previous one. The reason why we don't quite reach the expected output may be because RethinkDB needs time to cache results and at some point increasing the number of write units will not result in an higher output. Another problem is that as the RethinkDB cache fills (because the RethinkDB is not able to flush the data to disk fast enough due to IO limitations) the performance will decrease because the processes will take more time inserting blocks.

D.2.4. Experiment 4

Settings:

- **Number of nodes:** 64
- **Number of processes:** 1 process per node
- **Write units:** $64 \times 1 = 64$ write units
- **Expected output:** 1K writes per second

Results:

- **Output:** stable 1K writes per second

In this case, we are increasing the number of nodes in the cluster by $2\times$. This won't have an impact in the write performance because the maximum amount of shards per table in RethinkDB is 32 (RethinkDB will probably increase this limit in the future). What this provides is more CPU power (and storage for replicas, more about replication in the next section). We just halved the amount write units per node maintaining the same output. The IO in the nodes holding the primary replica is the same has Experiment D.2.1.

D.2.5. Experiment 5

Settings:

- **Number of nodes:** 64
- **Number of processes:** 2 processes per node
- **Write units:** $64 \times 2 = 128$ write units
- **Expected output:** 2000 writes per second

Results:

- **Output:** unstable 2K (peak) writes per second

In this case, we are doubling the amount of write units. We are able to reach the expected output, but the output performance is unstable due to the fact that we reached the IO limit on the machines.

D.2.6. Experiment 6

Settings:

- **Number of nodes:** 64
- **Number of processes:**
  - 32 nodes running 1 process
  - 32 nodes running 2 processes
- **Write units:** $32 \times 2 + 32 \times 1 = 96$ write units
- **Expected output:** 1.5K writes per second

Results:

- **Output:** stable 1.5K writes per second

This test is similar to Experiment D.2.3. The only difference is that now the write units are distributed between 64 nodes meaning that each node is writing to its local cache and we don't overload the cache of the nodes like we did with Experiment D.2.3. This is another advantage of adding more nodes beyond 32.

D.3. Experiments on Replication

Replication is used for data redundancy. In RethinkDB we are able to specify the number of shards and replicas per table. Data in secondary replicas is no directly used, it's just a mirror of a primary replica and used in case the node holding the primary replica fails.

RethinkDB does a good job trying to distribute data evenly between nodes. We ran some tests to check this.

By increasing the number of replicas we also increase the number of writes in the cluster. For a replication factor of 2 we double the amount of writes on the cluster, with a replication factor of 3 we triple the amount of writes and so on.

With 64 nodes and since we can only have 32 shards we have 32 nodes holding shards (primary replicas)

With a replication factor of 2, we will have 64 replicas (32 primary replicas and 32 secondary replicas). Since we already have 32 nodes holding the 32 shards/primary replicas RethinkDB uses the other 32 nodes to hold the secondary replicas. So in a 64 node cluster with 32 shards and a replication factor of 2, 32 nodes will be holding the primary replicas and the other 32 nodes will be holding the secondary replicas.

If we run Experiment D.2.4 again with this setup, except now with a replication factor of 2, we get twice the amount of writes. A nice result is that the IO in the nodes holding the primary replicas does not increase when compared to Experiment D.2.4 because all of the excess writing is now being done the 32 nodes holding the secondary replicas.

Also regarding replication: if I have a 64 node cluster and create a table with 32 shards, 32 nodes will be holding primary replicas and the other nodes do not hold any data. If I create another table with 32 shards RethinkDB will create the shards in the nodes that where not holding any data, evenly distributing the data.

References

[1] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. `https://bitcoin.org/bitcoin.pdf`, 2009.

[2] M Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems, 3rd Edition.* Springer Science & Business Media, 2011.

[3] Ethereum. `https://ethereum.org/`.

[4] Vitalik Buterin. Ethereum White Paper: A Next Generation Smart Contract & Decentralized Application Platform. `http://blog.lavoiedubitcoin.info/public/Bibliotheque/EthereumWhitePaper.pdf`.

[5] Enigma. `http://enigma.media.mit.edu/`.

[6] Guy Zyskind, Oz Nathan, and Alex Pentland. Enigma: Decentralized Computation Platform with Guaranteed Privacy. 2015. `http://enigma.media.mit.edu/enigma_full.pdf`.

[7] J. Benet. IPFS – Content Addressed, Versioned, P2P File System. `http://static.benet.ai/t/ipfs.pdf`, 2014.

[8] Eris Industries. `http://erisindustries.com/`.

[9] Tendermint. `http://www.tendermint.com`.

[10] Dimitri De Jonghe and Trent McConaghy. SPOOL Protocol. `https://github.com/ascribe/spool`.

[11] A. Back. Enabling blockchain innovations with pegged sidechains. Technical report, October 2010. http://www.blockstream.com/sidechains.pdf.

[12] S. Thomas and Schwartz E. A Protocol for Interledger Payments. https://interledger.org/interledger.pdf, 2015.

[13] P. Koshy. Bitcoin and the Byzantine Generals Problem – a Crusade is needed? A Revolution? http://financialcryptography.com/mt/archives/001522.html, November 2014.

[14] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):382–401, July 1982. http://research.microsoft.com/en-us/um/people/lamport/pubs/byz.pdf.

[15] Juan Garay, Aggelos Kiayias, and Nikos Leonardos. The bitcoin backbone protocol: Analysis and applications. In *Advances in Cryptology-EUROCRYPT 2015*, pages 281–310. Springer, 2015.

[16] Loi Luu, Viswesh Narayanan, Kunal Baweja, Chaodong Zheng, Seth Gilbert, and Prateek Saxena. Scp: A computationally-scalable byzantine consensus protocol for blockchains. 2015.

[17] John R Douceur. The Sybil Attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002. http://research.microsoft.com/pubs/74220/IPTPS2002.pdf.

[18] Bitcoin Wiki. Scalability. https://en.bitcoin.it/wiki/Scalability, 2015.

[19] M. Trillo. Stress Test Prepares VisaNet for the Most Wonderful Time of the Year. http://www.visa.com/blogarchives/us/2013/10/10/stress-test-prepares-visanet-for-the-most-wonderful-time-of-the-year/index.html, October 2013.

[20] Sourabh. How Much Email Do We Use Daily? 182.9 Billion Emails Sent/Received Per Day Worldwide. http://sourcedigit.com/4233-much-email-use-daily-182-9-billion-emails-sentreceived-per-day-worldwide/, February 2014.

[21] Blockchain.info. Blockchain size. https://blockchain.info/charts/blocks-size, December, 30th 2015.

[22] A. Wagner. Ensuring Network Scalability: How to Fight Blockchain Bloat. https://bitcoinmagazine.com/articles/how-to-ensure-network-scalibility-fighting-blockchain-bloat-1415304056, November 2014.

[23] Adrian Cockcroft and Denis Sheahan. Benchmarking Cassandra Scalability on AWS – Over a million writes per second. 2011. http://techblog.netflix.com/2011/11/benchmarking-cassandra-scalability-on.html.

[24] C. Kalantzis. Revisiting 1 Million Writes per second. http://techblog.netflix.com/2014/07/revisiting-1-million-writes-per-second.html.

[25] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, April 1980. http://research.microsoft.com/en-US/um/people/Lamport/pubs/reaching.pdf.

[26] Leslie Lamport. My Writings. http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html.

[27] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998. http://research.microsoft.com/en-us/um/people/lamport/pubs/lamport-paxos.pdf.

[28] Leslie Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006. http://research.microsoft.com/pubs/64624/tr-2005-112.pdf.

[29] Leslie Lamport. Byzantizing paxos by refinement. In *Distributed Computing*, pages 211–224. Springer, 2011. http://research.microsoft.com/en-us/um/people/lamport/tla/byzsimple.pdf.

[30] H. Robinson. Consensus Protocols: Paxos. http://the-paper-trail.org/blog/consensus-protocols-paxos/, February 2009.

[31] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *OSDI'06, Proceedings of the 7th symposium on Operating systems design and implementation, Seattle, WA*, pages 335–350. USENIX Association, November 2006. http://static.googleusercontent.com/media/research.google.com/en//archive/chubby-osdi06.pdf.

[32] Wikipedia. Paxos (Computer Science). http://en.wikipedia.org/wiki/Paxos_(computer_science).

[33] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014. https://ramcloud.stanford.edu/raft.pdf.

[34] C. Copeland and H. Zhong. Tangaroa: a Byzantine Fault Tolerant Raft. August 2014. http://www.scs.stanford.edu/14au-cs244b/labs/projects/copeland_zhong.pdf.

[35] Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985. https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf.

[36] Wikipedia. Byzantine fault tolerance. https://en.wikipedia.org/wiki/Byzantine_fault_tolerance.

[37] Michael Paulitsch, Jennifer Morris, Brendan Hall, Kevin Driscoll, Elizabeth Latronico, and Philip Koopman. Coverage and the Use of Cyclic Redundancy Codes in Ultra-Dependable System. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 346–355. IEEE, 2005.

[38] Albert L Hopkins Jr, Jaynarayan H Lala, and T Basil Smith III. The Evolution of Fault Tolerant Computing at the Charles Stark Draper Laboratory 1955–85. In *The Evolution of fault-tolerant computing*, pages 121–140. Springer, 1987.

[39] Kevin Driscoll, G Papadopoulis, S Nelson, G Hartmann, and G Ramohalli. Multi-microprocessor flight control system. In *Proceedings of the IEEE/AIAA 5th Digital Avionics Systems Conference, Institute of Electrical and Electronic Engineers, New York, NY*, 1983.

[40] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, volume 99, pages 173–186, February 1999.

[41] M. Castro. *Practical Byzantine Fault Tolerance.* PhD thesis, MIT, 2001. http://research.microsoft.com/en-us/um/people/mcastro/publications/thesis.pdf.

[42] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *NSDI*, volume 9, pages 153–168, November 2009. https://www.usenix.org/legacy/event/nsdi09/tech/full_papers/clement/clement.pdf.

[43] Pierre-Louis Aublin, Sonia Ben Mokhtar, and Vivien Quéma. RBFT: Redundant Byzantine Fault Tolerance. In *Distributed Computing Systems (ICDCS), 2013 IEEE 33rd International Conference on*, pages 297–306. IEEE, 2013. http://www.computer.org/csdl/proceedings/icdcs/2013/5000/00/5000a297.pdf.

[44] D. Mazieres. The Stellar Consensus Protocol: A Federated Model for Internet-Level Consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf, December 2015. draft of Nov 17, 2015, retrieved Dec 30, 2015.

[45] Wikipedia. RAID. http://en.wikipedia.org/wiki/RAID.

[46] Bitcoin Wiki. Transaction Fees. https://en.bitcoin.it/wiki/Transaction_fees, 2015.

[47] A. Sethy. The mystery of India's deadly exam scam. http://www.theguardian.com/world/2015/dec/17/the-mystery-of-indias-deadly-exam-scam, December 2015.

[48] WikiData. https://www.wikidata.org.

[49] BigchainDB Documentation. http://bigchaindb.readthedocs.io/en/latest/index.html.

[50] M. Gault. The CIA Secret to Cybersecurity that No One Seems to Get. http://www.wired.com/2015/12/the-cia-secret-to-cybersecurity-that-no-one-seems-to-get, December 2015.

[51] NoSQL Database. NoSQL: Your Ultimate Guide to the Non-Relational Universe. http://www.nosql-database.org.

[52] Toad World. Toad for Cloud Databases Community. http://www.toadworld.com/products/toad-for-cloud-databases/w/wiki/308.survey-distributed-databases, 2015.

[53] The Apache Cassandra Project. https://cassandra.apache.org.

[54] Apache HBase. https://hbase.apache.org.

[55] Redis. https://www.redis.io.

[56] Basho. Riak. https://docs.basho.com/riak.

[57] MongoDB. https://www.mongodb.org.

[58] RethinkDB. https://www.rethinkdb.com.

[59] ElasticSearch. https://www.elastic.co/products/elasticsearch.

[60] Wikipedia. CAP Theorem. https://en.wikipedia.org/wiki/CAP_theorem.

[61] Wikipedia. ACID. https://en.wikipedia.org/wiki/ACID.

[62] RethinkDB Consistency Guarantees. https://rethinkdb.com/docs/consistency/.

[63] RethinkDB Changefeeds. https://rethinkdb.com/docs/changefeeds.

[64] RethinkDB Frequently Asked Questions. https://www.rethinkdb.com/faq/.

[65] GitHub. rethinkdb/rethinkdb. https://github.com/rethinkdb/rethinkdb.

[66] T. Bray. The javascript object notation (json) data interchange format. RFC 7159, RFC Editor, March 2014. http://www.rfc-editor.org/rfc/rfc7159.txt.

[67] Ed25519: high-speed high-security signatures. https://ed25519.cr.yp.to/.

[68] Simon Josefsson and Ilari Liusvaara. Edwards-curve digital signature algorithm (eddsa). Internet-Draft draft-irtf-cfrg-eddsa-05, IETF Secretariat, March 2016. http://www.ietf.org/internet-drafts/draft-irtf-cfrg-eddsa-05.txt.

[69] Thins that use Ed25519. https://ianix.com/pub/ed25519-deployment.html.

[70] Melanie Swan. *Blockchain: Blueprint for a New Economy.* " O'Reilly Media, Inc.", 2015. http://shop.oreilly.com/product/0636920037040.do.

[71] M. Andreesen. Why Bitcoin Matters. http://dealbook.nytimes.com/2014/01/21/why-bitcoin-matters, January 2014. New York Times.

[72] J. Monegro. The Blockchain Application Stack. http://joel.mn/post/103546215249/the-blockchain-application-stack, November 2014. Joel Monegro Blog.

[73] T. Berners-Lee. Information Management: A Proposal. http://www.w3.org/History/1989/proposal.html, 1989. World Wide Web Consortium.

[74] N. Popper. A Bitcoin Believer's Crisis of Faith. http://www.nytimes.com/2016/01/17/business/dealbook/the-bitcoin-believer-who-gave-up.html?_r=0, January 2016.

[75] Robin Hanson. Information Prizes – Patronizing Basic Research, Finding Consensus. In *Western Economics Association meeting, Lake Tahoe*, June 2013. http://mason.gmu.edu/~rhanson/ideafutures.html.

[76] Backfeed. http://backfeed.cc.

[77] Cory Doctorow. *Down and Out in the Magic Kingdom.* Macmillan, February 2003. http://www.amazon.com/Down-Magic-Kingdom-Cory-Doctorow/dp/076530953X.

[78] D. Suarez. Freedom (TM). http://www.amazon.com/Freedom-TM-Daniel-Suarez/dp/0525951571, January 2010.

[79] Blockchain.info. Total transaction fees. https://server2.blockchain.info/charts/transaction-fees, December, 30th 2015.

[80] Tradeblock. Recent Blocks. https://tradeblock.com/bitcoin/.

[81] Ian Grigg. The Ricardian Contract. In *Electronic Contracting, 2004. Proceedings. First IEEE International Workshop on*, pages 25–31. IEEE, 2004. http://iang.org/papers/ricardian_contract.html.

[82] Trent McConaghy. Blockchain, Throughput, and Big Data. http://trent.st/content/2014-10-28%20mcconaghy%20-%20blockchain%20big%20data.pdf, October 2014. Berlin, Germany.

[83] A. Back. Hashcash - a denial of service counter-measure. Technical report, August 2002. technical report.

[84] Bitcoin Wiki. Proof of Stake. https://en.bitcoin.it/wiki/Proof_of_Stake, 2015.

[85] Bitsmith. Dan O'Prey talks Hyperledger. http://www.thecoinsman.com/2014/08/decentralization/dan-oprey-talks-hyperledger/, August 2014.

[86] J. Kim. Safety, liveness and fault tolerance — the consensus choices. https://www.stellar.org/blog/safety_liveness_and_fault_tolerance_consensus_choice/, December 2014.

[87] J. Kwon. Tendermint: Consensus without Mining. http://tendermint.com/docs/tendermint.pdf, fall 2014.

[88] Vitalik Buterin. Slasher: A Punitive Proof-of-Stake Algorithm. https://blog.ethereum.org/2014/01/15/slasher-a-punitive-proof-of-stake-algorithm/, January, 15th 2014.

[89] Vitalik Buterin. Slasher Ghost, and other Developments in Proof of Stake. https://blog.ethereum.org/2014/10/03/slasher-ghost-developments-proof-stake/, October, 3th 2014.

[90] V. Zamfir. Introducing Casper 'the Friendly Ghost'. https://blog.ethereum.org/2015/08/01/introducing-casper-friendly-ghost/, August 2015.

[91] AI Coin. http://www.ai-coin.org.

[92] Factom. http://factom.org/.

[93] J. Kim. Stellar Consensus Protocol: Proof and Code. https://www.stellar.org/blog/stellar-consensus-protocol-proof-code/, April 2015.

[94] Wikipedia. Peercoin. http://en.wikipedia.org/wiki/Peercoin.

[95] Cryptonite. http://cryptonite.info/.

[96] J.S. Galt. JL777's vision of the Supernet. https://bitcoinmagazine.com/18167/what-is-the-supernet-jl777s-vision/, November 2014.

[97] Vitalik Buterin. Scalability, Part 2: Hypercubes. https://blog.ethereum.org/2014/10/21/scalability-part-2-hypercubes/, October, 21st 2014.

[98] Bitcoin Wiki. Block size limit controversy. https://en.bitcoin.it/wiki/Block_size_limit_controversy.

[99] Ittay Eyal, Adem Efe Gencer, Emin Gun Sirer, and Robbert van Renesse. Bitcoin-NG: A Scalable Blockchain Protocol. In *NSDI*, 2016. http://diyhpl.us/~bryan/papers2/bitcoin/Bitcoin-NG:%20A%20scalable%20blockchain%20protocol.pdf.

[100] Bitcoin Core. Bitcoin Capacity Increases FAQ. https://bitcoincore.org/en/2015/12/23/capacity-increases-faq/, December 2015.

[101] G. Maxwell. Capacity increases for the Bitcoin system. https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2015-December/011865.html, December 2015.

[102] P. Wuille. Segregated witness and its impact on scalability. http://diyhpl.us/wiki/transcripts/scalingbitcoin/hong-kong/segregated-witness-and-its-impact-on-scalability/.

[103] Berkman Center for Internet and Society. Brief History of the Domain Name System. http://cyber.law.harvard.edu/icann/pressingissues2000/briefingbook/dnshistory.html, 2000. Harvard.

[104] J. Schwartz. Internet 'Bad Boy' Takes on a New Challenge. http://www.nytimes.com/2001/04/23/business/technology-Internet-bad-boy-takes-on-a-new-challenge.html, April 2001.

[105] ICANN. DNSSEC – What Is It and Why Is It Important? https://www.icann.org/resources/pages/dnssec-qaa-2014-01-29-en, January 2014.

Addendum to the BigchainDB Whitepaper

BigchainDB GmbH, Berlin, Germany

December 18, 2016

This addendum summarizes significant changes since the BigchainDB whitepaper was last updated. The online BigchainDB Documentation is kept up-to-date.

- There are more details about how BigchainDB achieves decentralization and immutability / tamper-resistance in the BigchainDB Documentation.
- **Sections 4 and 6.** The whitepaper described **S** and **C** as being two separate databases, but the actual implementation has them as three separate tables (in one database). **S** became the backlog table (of transactions). **C** became two append-only tables, one for blocks and one for votes. To understand why, see the discussion on Issue #368 and related issues on GitHub.
- **Section 4.3, Section 6 and Figure 10.** Transactions are *not* validated before being written to the backlog table (**S** in the whitepaper).
- **Section 4.5.** The data structures of transactions, blocks and votes have changed and will probably change some more. Their current schemas can be found in the BigchainDB Documentation. Each node makes one vote for each block. Each vote contains the id (hash) of the block being voted on, and the id (hash) of the previous block *as determined by the voting node*. Another node might consider a different block to be the previous block. In principle, each node records a different order of blocks (in its votes). This is okay because the check to see if a transaction is a double-spending attempt doesn't depend on an agreed-upon block ordering.
- (This isn't a change; it's more of an interesting realization.) If you pick a random block, its hash is stored in some votes, but the information in those votes never gets included in anything else (blocks or votes). Therefore there is no hash chain or Merkle chain of blocks. Interestingly, every CREATE transaction begins a Merkle directed acyclic graph (DAG) of transactions, because all TRANSFER transactions contain the hashes of previous transactions.
- **Section 5.1** By January 2017, one will be able to choose RethinkDB or MongoDB as the backend database. Both will be supported. In the future, even more databases may be supported. MongoDB was chosen as the second one to support because it's very similar to RethinkDB: it's document-oriented, has strong consistency guarantees, and has the same open source license (AGPL v3). Also, it's possible to get something like RethinkDB's changefeed with MongoDB.

- **Section 5.2.** A BigchainDB node can have arbitrarily-large storage capacity (e.g. in a RAID array). Other factors limit the maximum storage capacity of a cluster (e.g. available RAM in the case of RethinkDB).
- **Section 5.4** There have been some changes in the details of how cryptographic hashes and signatures are calculated. For the latest details, see the documentation page about cryptographic calculations.