# ElasticBloC: A Massively Scalable Architecture For Blockchain Based Applications

Hadi Jibbawi
*Lebanese University*
Beirut, Lebanon
hadi.jibbawi@gmail.com

Yehia Taher
*Université de Versailles – Paris-Saclay*
Versailles, France
yehia.taher@uvsq.fr

Rafiqul Haque
*Intelligencia R&D*
Paris, France
Rafiqul.Haque@intelligencia.fr

Ali Jaber
*Lebanese University*
Beirut, Lebanon
ali.jaber@ul.edu.lb

***Abstract*—Blockchain is an emerging technology that would possibly disrupt the existing centralized financial systems lead to the rise to a new technology era for the financial sector. Additionally, different new use cases such as healthcare, identity management, etc. suggest that Blockchain has much wider applications. Blockchain is founded on distributed ledger technology that ensures trust through consensus between parties in a peer-to-peer network instead of the need to a third party or central authority. However, blockchain has several limitations such as scalability, latency, low throughput which are the main barriers for Blockchain being adopted by the industries. Of all, scalability is the most critical limitation of blockchain that needs an efficient and effective solution. In this paper, we aim to enhance the scalability of blockchain by designing and implementing a massively scalable architecture for private blockchain-based applications, called ElasticBloC. To evaluate our contribution, we conducted several experiments on ElasticBloC. The results showed that ElasticBloC is a high-performant architecture that scales massively.**

***Index Terms*—Blockchain, Performance, Scalability**

## I. Introduction

Over the last few years, Blockchain has drawn huge attention to industry experts, technology evangelists, and academic researchers due to its immense potentiality described in a large body of literature and other sources such as blogs, forums, etc. Many researchers and industry experts argued that it is a revolutionary technology like the Internet that will provide a highly efficient way to transact in a secure, immutable, transparent, and auditable manner. In 2016, it was one of the technologies that reached a peak in inflated expectation; since then the interests in blockchain have been soaring to different types of industries. In particular, the interest of Blockchain technology among the financial industry started to grow as it might be a potential one that would enable them to avoid financial debacles such as the Heartland Payment Systems data breach that had happened in 2008.

Security is an ever-growing concern in the financial industry. With the advent of digital financial information systems and rich transaction technologies, the operations have become faster and the operational activities have spanned largely; at the same time, the risk of breaching information has been increased enormously. Although, there are advanced technologies encryption technologies that enable cryptic transmission of financial data between financial actors (e.g., banks, insider intruder), security remain a problem because cryptographic algorithms are still weak to many attacks launched by the adversaries. Blockchain was deemed a major breakthrough technology that would prevent some unsolved security issues. Therefore, the huge adoption of Blockchain was forecasted by many such as Gartner within not only the financial industry but also the other industries.

Furthermore, trust is critical when it is related to communication between two or more parties. Historically and till now, trust is achieved mostly by the third party like banks or authorities, that holds our data. In other words, communicating parties rely on a common ledger which is held and managed by this third party. As a typical example, Clearing Households validates and manages the communication between trading parties. So Blockchain technology comes as a potential technology. Its disruptive aspect is that it eliminates the need for intermediaries while performing transactions [7]. Hence, it can empower groups of parties to agree on events without needing the third party, such as the promise of this new technology [8].

Our study revealed several limitations of Blockchain technologies, as mentioned in the earlier section. However, the major concerns of Blockchain technologies are two-fold: scalability and performance. Scalability is considered the critical drawback that stands against blockchain technology. In fact, it is the first limitation that must be addressed to make blockchain an acceptable technology. The reason is obvious. Consider a Walmart payment system that processes more 250 transactions every hour. Since Blockchain technology replicates blocks in different consensus servers hosted in different locations to increase trust and guarantee security against any odd modification, Blockchain must require a scalable infrastructure not only at the operational level but also at the physical level. Because blockchain technologies are founded on immutability principle which means that every block created cannot be updated or replaced by a new block. All new blocks will be appended only which will require physical scalability especially for retail companies like Walmart, banks, and high-end manufacturing companies.

In the existing blockchain protocols such as Ethereum, Bitcoin, Ripple, and Tendermint, each participating node should process every transaction in the network. In this case, nodes need more storage, bandwidth, and computation power as the blockchain expands. Indeed, this technology will lose its decentralization, because the blockchain will reach a limit that only specific nodes can process a block [19]. According to Vitalik, with the current design of blockchain technology, scalability cannot be achieved because it focuses on decentralization and security, not scalability. While a decentralization consensus mechanism offers some critical benefits, such as fault tolerance, a strong guarantee of security, political neutrality, and authenticity, it comes at the cost of scalability. Existing solutions, some as to be mentioned in Section 2, vary in their aspects. Some scale to a limit, another downgrade the performance [22], etc.

Realizing the significance of a massively scalable infrastructure of Blockchain technology which cannot be addressed by existing solutions, in this paper we developed a scalable Blockchain technology in which scalability is strongly correlated with the performance that has an impact on blockchain efficiency.

The remainder of this paper is organized as follows. Section 2 discusses works related to the core issue of this paper. We explained our solution ElasticBloc in section 3. We reported our experiments with ElasticBloc and discussed the results in Section 4. We present a conclusion in Section 5.

## II. Related Works

This research revolves around the scalability issue of blockchain technology. In this section, we described a review of works related to this issue. Ehmke et al. [10] proposed a solution based on the idea of Ethereum to keep the state of the system explicitly in the current block but further pursues this by including the relevant part of the current system state in new transactions as well. This enables other participants to validate incoming transactions without having to download the whole blockchain initially. The scalability in the proposed solution is the logical level that is, the authors' developed techniques extending Merkle Patricia Tree [4]. In [9], Dorri et al. proposed a tiered Lightweight Scalable Blockchain (LSB) that is optimized for IoT requirements. The authors explored LSB in a smart home setting as a representative example of broader IoT applications. LSB achieves decentralization by forming an overlay network where high resource devices jointly manage a public Blockchain (BC) that ensures end-to-end privacy and security. The overlay is organized as distinct clusters to reduce overheads and the cluster heads are responsible for managing the public BC. LSB incorporates several optimizations which include algorithms for lightweight consensus, distributed trust and throughput management. To ensure scalability, the overlay nodes are organized as clusters and only the cluster heads (CH) are responsible for managing the public BC. Technically speaking, this is a conventional approach to gain scalability which is very limited. However, it is not possible to gain massive scalability because it is not supported by underlying system-level technologies such as file system.

Zamani et al. [26] developed a solution called RapidChain sharding-based public blockchain protocol that is resilient to Byzantine faults from up to a 1/3 fraction of its participants and achieves complete sharding of the communication, computation, and storage overhead of processing transactions without assuming any trusted setup. RapidChain employs an optimal intra-committee consensus algorithm that can achieve very high throughputs via block pipelining, a novel gossiping protocol for large blocks, and a provably-secure reconfiguration mechanism to ensure robustness. Using an efficient cross-shard transaction verification technique, the proposed protocol avoids gossiping transactions to the entire network. The empirical evaluations suggest that RapidChain can process (and confirm) more than 7,300 tx/sec with an expected confirmation latency of roughly 8.7 seconds in a network of 4,000 nodes with an overwhelming time-to-failure of more than 4,500 years. RapidChain is focused more on performance. A limited effort was put on scalability; not to mention that the scalability is logical like the one proposed in [9]. This does not address the massive scalability limitation.

Eyal et al. [14] proposed a protocol called Bitcoin-NG that is founded on several novel metrics of interest in quantifying the security and efficiency of Bitcoin-like blockchain protocols. We implement Bitcoin-NG and perform large-scale experiments at 15% the size of the operational Bitcoin system, using unchanged clients of both protocols. These experiments demonstrate that Bitcoin-NG scales optimally, with bandwidth limited only by the capacity of the individual nodes and latency limited only by the propagation time of the network. The scalability yet again is achieved at logical, not at the physical level. Zhang and Jacobsen proposed DCS properties (Decentralization, Consistency, and Scalability) as an analogy to the CAP theorem. The authors provided a general structure of the blockchain platform which decomposes the distributed ledger into six layers: Application, Modeling, Contract, System, Data, and Network. Finally, we classify research angles across three dimensions: DCS properties impacted, targeted applications, and related layers. The proposed solution is yet again limited in terms of scalability. Guo et al. [13] proposed a solution that relies on two key techniques: a fair contract partition algorithm leveraging integer linear programming to partition a set of smart contracts into multiple subsets, and a random assignment protocol assigning subsets randomly to a subgroup of users. This is a logical model for gaining scalability as all the other authors proposed.

To sum up, the research on Blockchain technology is still limited. Most of the research focuses on the performance of blockchain applications. The solutions concerning scalability proposed in the literature by far deals with logical scalability such as partitioning/sharding blocks. However, it is not adequate to gain massive scalability which needs physical level scalability that is achieved through system-level technologies such as distributed file systems.

## III. ELASTICBLOC – THE MASSIVELY SCALABLE BLOCKCHAIN SOLUTION

This section provides a detailed description of the core contribution of this paper. It begins with an overview of ElasticBloC, then a description of the high-level architecture of ElasticBloC is provided followed by a presentation of the solution workflow. The functionalities of ElasticBloC are briefly explained and finally, I explained the implementation of ElasticBloC.

### A. A Brief Overview of ElasticBloC

ElasticBloC is a solution for developing blockchain applications. It is a generic solution that aims to support building all types of blockchain-based applications such as asset management, smart contract or notarization in a scalable manner. The users can deploy these applications on ElasticBloC which perform the internal blockchain functions such as generating blocks, adding blocks in the storage, retrieving the blocks, etc. ElasticBloC guarantees scalability at the physical level for blockchain applications.

ElasticBloC relies on a cluster computing paradigm that underpins developing an ecosystem consisting of a massive number of physical nodes (servers). As mentioned earlier, the focus of the solution proposed in this paper is to achieve the scalability of the physical layer instead of the logical layer. The scalability at the logical layer can be achieved in many ways such as the logical partition of blocks and then store the partition in different nodes within the cluster which consists of the limited number of nodes. However, scalability at the physical level needs extensible architecture. ElasticBloC architecture is extensible which enables users to add physical nodes on the fly or offline to enhance the capability to store any number of blocks generated by transaction applications. In order to gain easy extensibility, it reuses an existing distributed file system that simplifies adding new nodes. This file system supports commodity hardware; therefore, building a large cluster using ElasticBloC is cost-effective.

Performance is another issue dealt with by ElasticBloC. The file system adopted in blockchain support extreme parallelism as it underlies the MapReduce functional programming model used by the applications to read and write blocks. Furthermore, ElasticBloC adopted a technology that avoids computationally expensive proof of work [48]. Proof-of-work based consensus protocols are also slow, requiring up to an hour to reasonably confirm a payment to prevent double-spending. ElasticBloC used technology that relies on Byzantine Consensus Protocol which reduces the computational cost significantly. The core of this protocol Byzantine Fault Tolerance [5]. The extreme parallelism ensued by the file system and BFT based consensus protocol makes ElasticBloC high-performant.

### B. Architecture of ElasticBloC

ElasticBloC is composed of several components. Fig. 1 depicts the architecture of ElasticBloC. The components are briefly explained in the following:

- **Transaction Gateway:** The transaction gateway is a connection component that enables to discover and connect with blockchain endpoint. It is also a channel through which users launch a transaction request.
- **Request Receiver:** It is an upfront server that receives the HTTP requests.
- **Communication Interface:** It is a standard interface for communication with Python applications. Since ElasticBloC is developed using Python programming language, this component is critical in communicating with Python applications
- **Operation Synthesizer:** It handles various operations including scaling up complex applications, object-relational mapping, validation of a request, authentication checking, and upload handling, etc.
- **Blockchain Engine:** It is one of the key components that perform a multitude of tasks including handling events, data modeling, and operation orchestration.

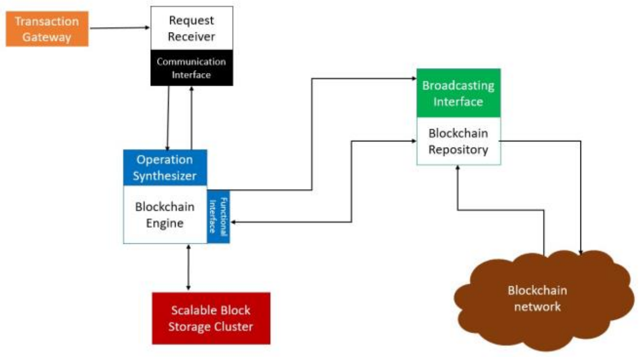


Fig. 1. ElasticBloC Architecture

- **Functional Interface:** It is another important component of ElasticBloC. It allows for Byzantine Fault Tolerant replication of applications written in any programming language. The consensus engine communicates with the application via a socket protocol that satisfies the functional interface. This interface consists of 3 primary message types that get delivered from the core to the application. The application replies with corresponding response messages. It consists of three message types:
  - **DeliverTx Messages:** Each transaction in the blockchain is delivered with this message.

- **CheckTx Messages:** The CheckTx message is similar to DeliverTx, but it is only for validating transactions.
- **Commit Messages:** The Commit message is used to compute a cryptographic commitment to the current application state, to be placed into the next block header.

- **Broadcasting Interface:** It receives HTTP post request from blockchain engine and broadcast it blockchain repository.
- **Blockchain Repository:** It securely and consistently replicates an application on many nodes. It works even if up to 1/3 of machines fail in arbitrary ways [1]. Every machine that is not faulty sees the same transaction log and computes the same state. Secure and consistent replication is a fundamental problem in distributed systems; it plays a critical role in the fault tolerance of a broad range of applications, from currencies, to elections, to infrastructure orchestration, and beyond.
  The ability to tolerate machines failing in arbitrary ways, including becoming malicious, is known as Byzantine fault tolerance (BFT) [5].
- **Blockchain Database Network:** It is a network of four or more nodes.
- **Scalable Block Storage Cluster:** This is the most important component that enhances physical infrastructure to a massive number of nodes. It enormously increases the capability of storing blocks as records. It is founded on column-oriented databases that are supported by a distributed file system that can support building a blockchain lake consisting of thousands of nodes. The cluster consists of one or more master nodes and hundreds of data nodes that essentially store the blocks. It is highly faulted tolerant because each block is replicated into three nodes (can be more depending on users' preference). If any node is not functioning two other nodes are available.
  In fact, it is not only a storage cluster, but the column-oriented database also enables querying and managing blocks.

### C. *ElasticBloC Operational Methods*

ElasticBloC enables us to perform different blockchain operations using various methods that are described into two categories: BigchainDB methods that are provided by BigchainDB server and HBase connection methods for establishing a connection with BigchainDB and performing various operations. I implemented all HBase connection methods within the scope of this paper. These methods are presented in the following subsections.

a) BigchainDB Methods
ElasticBloC provides a library that allows the client to perform ElasticBloC functionalities. This library is called "bigchaindb_driver". The table below lists the major methods that a client can use to transact or operate in ElasticBloC.

TABLE I
BIGCHAINDB DRIVER MAIN METHODS

| Class | Method | Parameters | Description |
|---|---|---|---|
| **BigchainDB** | BigchainDB | *nodes | Creates an instance of bigchaindb_ driver which is able to create, sign, and send transactions to several nodes |
| **BigchainDB** | api_info | Headers | Retrieves the HTTP API details provided by the BigchainDB server |
| **BigchainDB** | Info | Headers | Retrieves information of the node connected to via the root endpoint, such as sever version and overview of all the endpoints |
| **Transactions Endpoint** | Fulfill | transaction, private_keys | Fulfills the given transaction |
| **Transactions Endpoint** | Get | *, asset_id, operation, headers | Retrieves a list of transactions that have the specified asset |
| **Transactions Endpoint** | Prepare | *, operation (CREATE or TRANSFER), signers, recipients, assets, metadata, inputs | Prepares a transaction payload, ready to be fulfilled |
| **Transactions Endpoint** | Retrieve | transaction_id, headers | Retrieves the transaction of given id |
| **Transactions Endpoint** | Send | transaction, mode, headers | Sends a transaction to the first specified nodes |
| **Outputs Endpoint** | Get | public_key, spent, headers | Retrieves transaction outputs by the public key |
| **Assets Endpoint** | Get | *, search, limit, headers | Retrieves the assets that match the search text |
| **Crypto** | Generate_keypair | None | Generates a cryptographic key pair |

The above table represents the interface of functions that a client can use to communicate with ElasticBloC. This facilitates dealing with such modular architecture. The reason behind this facilitation is that once a client transacts or operates via these functions, the rest of the flow is automated i.e. the operation flows through the required components automatically. So, the client communicates with one component.
BigchainDB driver calls in its method's implementation the methods of the BigchainDB-HBase connector, that will be explained in the successive section, to perform any operation that accesses HBase, such as retrieving data

(transactions, assets, metadata, etc.), checking the pre-existence of a newly submitted transaction, or storing of committed block with its details.

b) BigchainDB-HBase Connector
BigchainDB-HBase connector is considered the core of our contribution. In this connector, I implemented a group of methods that BigchainDB can expose in order to connect and operate with HBase as a backend database. The following table describes the methods implemented with the connector.
Hence the preceding methods represent the interface of the connector that integrates BigchainDB sever with HBase.

### D. *Solution Workflow*

ElasticBloC has a unique general workflow. In fact, this workflow differs in its small parts according to the submitted transaction mode or the nature of the desired operation. The diagram below shows the general workflow of ElasticBloC. Once the client has a valid transaction i.e. the transaction conforms to the BigchainDB Transactions Specification, he submits it to one or more ElasticBloC nodes through the BigchainDB HTTP API [17]. In particular, it embeds the transaction in an HTTP request and specifies one of the predefined ends points to send through. These endpoints are:

- POST /API/v1/transactions
- POST /API/v1/transactions?mode=async
- POST /API/v1/transactions?mode=sync
- POST /API/v1/transactions?mode=commit

After that, the HTTP request holding the transaction arrives at the BigchainDB node at the Gunicorn [16] web server in that node. Then Gunicorn forwards the request towards the BigchainDB server using its exposed Web Server Gateway Interface (WSGI). The request reaches the BigchainDB server through the Flask web application development framework which simplifies working with WSGI/Gunicorn. The BigchainDB server uses a Python method to check the transaction's validity. If the transaction is not valid, then the HTTP response status code is 400 which means error. Otherwise, it is put into a new JSON string and sent to the local Tendermint instance via Tendermint Broadcast API.

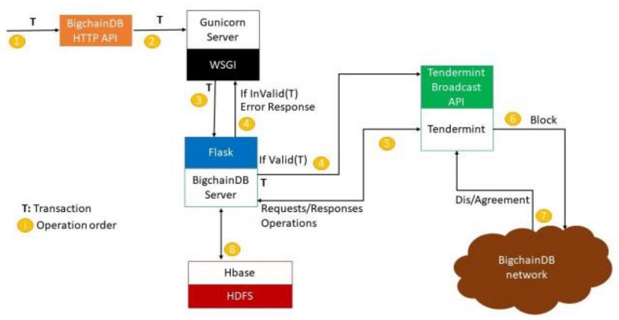


Fig. 2. ElasticBloC Workflow

TABLE II
BIGCHAINDB-HBASE CONNECTOR FUNCTIONALITIES METHODS

| Method | Parameters | Description |
|---|---|---|
| connect | backend, host, port, name, connection_timeout | Creates a new connection to the backend database (HBase) |
| create_tables | connection, dbname | Creates tables in HBase to be used by BigchainDB |
| delete_tables | connection, dbname | Deletes the created tables in HBase |
| store_transaction | connection, signed_transaction | Stores a transaction in Transactions table |
| store_transactions | connection, signed_transactions | Stores a list of transactions in Transaction table |
| get_transaction | connection, transaction_id | Gets a transaction from Transactions table |
| get_transactions | connection, transaction_ids | Gets a list of transactions from Transactions table |
| store_metadatas | connection, metadata | Stores metadata in Metadata table |
| get_metadata | connection, transaction_ids | Gets metadata from Metadata table |
| store_asset | connection, asset | Stores asset in Assets table |
| store_assets | connection, assets | Stores a list of assets in Assets table |
| get_asset | connection, asset_id | Gets an asset from Assets table |
| get_assets | connection, asset_ids | Gets a list of assets from Assets table |
| store_block | connection, block | Stores a block in Blocks table |
| get_block | connection, block_id | Gets a block from Blocks table |
| get_spent | connection, transaction_id, output_index | Check if a transaction_id was already used as an input. A transaction can be used as an input for another transaction. Bigchain needs to make sure that a given txid is only used once Gets the spending transaction |
| get_latest_block | Connection | Gets the latest committed block |
| get_txids_filtered | connection, asset_id, operation | Gets all transactions for a particular asset id and optional operation |
| get_owned_ids | connection, owner | Gets a list of transactions ids we can use which has inputs |
| get_spending_transactions | connection, inputs | Gets transactions which spend given inputs |
| get_block_with_transaction | connection, transaction_id | Gets block holding a specific transaction |
| delete_transaction | connection, transaction_id | Deletes a transaction from database and its relevant asset and metadata |
| delete_transactions | connection, transaction_ids | Deletes transactions from database and their relevant assets and metadata |
| delete_latest_block | connection | Delete the latest commited block |
| store_unspent_outputs | connection, unspent_outputs | Stores unspent outputs in utxos table |
| get_unspent_outputs | connection, *, query | Gets unspent outputs |
| delete_unspent_outputs | connection, unspent_outputs | Deletes unspent outputs from utxos table |
| store_pre_commit_state | Connection, state | Stores pre_commit state |
| get_pre_commit_state | Connection, commit_id | Gets pre_commit state of a commit id |

Now, the operations between the local Tendermint instance and BigchainDB are established by the Application Blockchain Interface (ABCI) which is an integral part of Tendermint and implemented also at the BigchainDB server side. In this case, Tendermint uses the broadcast endpoint which is relevant to the initial BigchainDB's request chosen. For example, if a client sent a transaction through /API/v1/transactions?mode=commit endpoint, Tendermint uses /broadcast_tx_commit endpoint respectively. Tendermint stores the initial validated transactions in its own mempool (memory pool). When it decides to create a block, Tendermint sends the creation request to BigchainDB by exposing a specific ABCI method. Then, it starts to send initial validated transactions that needed to be grouped in the desired block also using another ABCI method to BigchainDB which rechecks the validity of the transaction before it is added to the block.

The proposed block is then broadcasted to the network by Tendermint. Then it makes sure that all the nodes agree on this block in a Byzantine fault tolerance way. When the network agrees on a new block, Tendermint appends the new block to the blockchain in its local LevelDB, and the BigchainDB server receives a commit message enforcing it to write the new block and the including transactions, assets, and metadata in a separate way to the HBase repository. HBase then writes these data into the Hadoop Distributed File System underlying it. The same process is done at each node in the ElasticBloC network.

### *E. Implementation of ElasticBloC*

As mentioned earlier, my goal is to use existing technologies for building ElasticBloC. The reason is two-fold: avoiding reinventing the same technology that already exists and it would be impractically ambitious to develop a complex architecture like ElasticBloC. I developed ElasticBloC using the most advanced technologies. I briefly described the main technologies in the following:

a) Tendermint

Tendermint is a secure state-machine replication algorithm in the blockchain paradigm. It provides a form of BFT-ABC (Atomic Broadcast) that is furthermore accountable - if safety is violated, it is always possible to verify who acted maliciously [3].

Tendermint begins with a set of validators, identified by their public key, where each validator is responsible for maintaining a full copy of the replicated state, and for proposing new blocks (batches of transactions), and voting on them [20]. Each block is assigned an incrementing index, or height, such that a valid blockchain has only one valid block at each height. At each height, validators take turns proposing new blocks in rounds, such that for any given round there is at most one valid proposer. It may take multiple rounds to commit a block at a given height due to the asynchrony of the network, and the network may halt altogether if one-third or more of the validators are offline or partitioned [3]. Validators engage in two phases of voting on a proposed block before it is committed, and follow a simple locking mechanism which prevents any malicious coalition of less than one-third of the validators from compromising safety [2].

In this, the broadcast interface and the Blockchain repository (See in Fig. 1) are implemented using Tendermint.

b) BigchainDB

The Blockchain Engine of ElasticBloC is implemented using 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 the scale [21]. BigchainDB on top of an enterprise-grade distributed DB, from which BigchainDB inherits high throughput, high capacity, low latency, a full-featured NoSQL query language, and permissioning. Nodes can be added to increase throughput and capacity.

c) HBase

The scalable block storage cluster (See Fig. 1) is implemented using HBase technology. Although BigchainDB aims at increasing scalability, yet massive scalability could not be achieved using BigchainDB.

Therefore, I implemented the scalable block storage of using HBase. HBase [15] is modeled on Google's BigTable database [6]. HBase provides a distributed, fault-tolerant scalable database, built on top of the HDFS file system [24], with random real-time read/write access to data. Each HBase table is stored as a multidimensional sparse map, with rows and columns, each cell having a timestamp [6]. A cell value at a given row and column is uniquely identified by:

**(Table, Row, Column-Family: Column, Timestamp) $\Longrightarrow$**
**Value**

HBase has its own Java client API, and tables in HBase can be used both as an input source and as an output target for MapReduce jobs through Table Input and Table Output Format. There is no HBase single point of failure. HBase uses Zookeeper [27], another Hadoop subproject, for the management of partial failures.

The HBase connector which is the primary contribution of this paper was implemented using Python. The first step of building the connector was to indicate the tables that are needed to store the architecture data (blocks, transactions, assets . . . ). The second step was to write a file that opens a connection to Hbase, based on the connection parameters and values given by the BigchainDB configuration file, and return an instance of this connection.

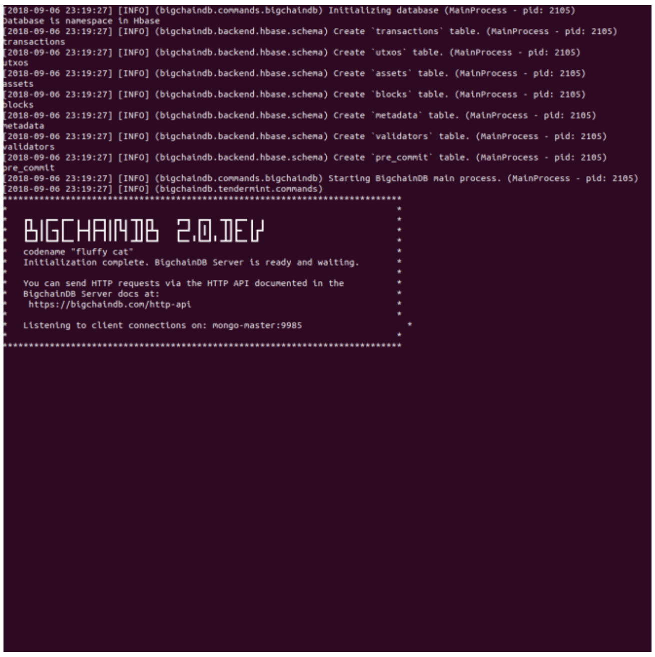
In the next, the schema file was written. The schema file defines and creates the database schema at HBase once BigchainDB is initialized. After that, the required querying methods were implemented. Some of these methods are for retrieving data, others for storing, updating or deleting data. In addition to the above, some web application development tools have been used in developing.

## *F. Experiments & Results*

This section describes some experiments that we conducted on ElasticBloC and discusses its results. The goal of the experiments is to evaluate two characteristics of ElasticBloC: scalability and performance. I conducted the following experiments are: Initial loading experiment, functionality experiment, and its result, scalability experiment and its result, and the ElasticBloC performance evaluation.

a) Initial Loading Experiment

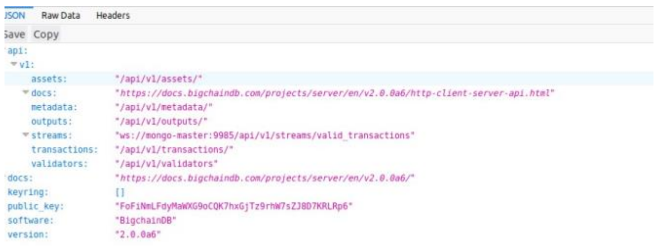
- Purpose:
  Testing the start-up running and initializing of the whole architecture.
- Requirements:
  The required steps are to run the ElasticBloC components and check the connectivity between these components. The following summarizes these steps:
  - Run the Hadoop cluster and HBase.
  - Run the BigchainDB server.
  - Run Tendermint instance.
- Results:
  The architecture components run successfully and the connection between the components is established. Once established, BigchainDB executed the schema file in the implemented connector and created the needed tables in HBase. The following screenshots represent some results of the successful initial start-up.



```
[2018-09-06 23:19:27] [INFO] (bigchaindb.commands.bigchaindb) Initializing database (MainProcess - pid: 2105)
Database is namespace in Hbase
[2018-09-06 23:19:27] [INFO] (bigchaindb.backend.hbase.schema) Create `transactions` table. (MainProcess - pid: 2105)
transactions
[2018-09-06 23:19:27] [INFO] (bigchaindb.backend.hbase.schema) Create `utxos` table. (MainProcess - pid: 2105)
utxos
[2018-09-06 23:19:27] [INFO] (bigchaindb.backend.hbase.schema) Create `assets` table. (MainProcess - pid: 2105)
assets
[2018-09-06 23:19:27] [INFO] (bigchaindb.backend.hbase.schema) Create `blocks` table. (MainProcess - pid: 2105)
blocks
[2018-09-06 23:19:27] [INFO] (bigchaindb.backend.hbase.schema) Create `metadata` table. (MainProcess - pid: 2105)
metadata
[2018-09-06 23:19:27] [INFO] (bigchaindb.backend.hbase.schema) Create `validators` table. (MainProcess - pid: 2105)
validators
[2018-09-06 23:19:27] [INFO] (bigchaindb.backend.hbase.schema) Create `pre_commit` table. (MainProcess - pid: 2105)
pre_commit
[2018-09-06 23:19:27] [INFO] (bigchaindb.commands.bigchaindb) Starting BigchainDB main process. (MainProcess - pid: 2105)
[2018-09-06 23:19:27] [INFO] (bigchaindb.tendermint.commands)
*************************************************************************
*                                                                       *
*  BIGCHAINDB 2.0.DEV                                                   *
*                                                                       *
*   codename "fluffy cat"                                               *
*   Initialization complete. BigchainDB Server is ready and waiting.    *
*                                                                       *
*   You can send HTTP requests via the HTTP API documented in the       *
*   BigchainDB Server docs at:                                          *
*    https://bigchaindb.com/http-api                                    *
*                                                                       *
*   Listening to client connections on: mongo-master:9985                 *
*                                                                       *
*************************************************************************
```

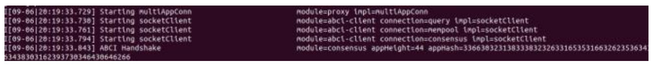
Fig. 3. BigchainDB Start-up.

As we can see above, the components ran successfully and Tendermint opened the required sockets and established the ABCI Handshaking. It compares the application's highest height and the application hash



```
JSON   Raw Data   Headers
Save  Copy
api:
  v1:
    assets:         "/api/v1/assets/"
    docs:           "https://docs.bigchaindb.com/projects/server/en/v2.0.0a6/http-client-server-api.html"
    metadata:       "/api/v1/metadata/"
    outputs:        "/api/v1/outputs/"
    streams:        "ws://mongo-master:9985/api/v1/streams/valid_transactions"
    transactions:   "/api/v1/transactions/"
    validators:     "/api/v1/validators"
docs:               "https://docs.bigchaindb.com/projects/server/en/v2.0.0a6/"
keyring:            []
public_key:         "FoFiNmLFdyMaWXG9oCQK7hxGjTz9rhW7sZJ8D7KRLRp6"
software:           "BigchainDB"
version:            "2.0.0a6"
```

Fig. 4. BigchainDB Web Interface After Startup.



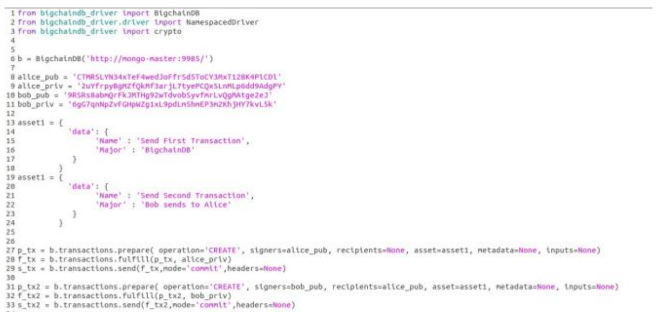
```
I[09-06|20:19:33.729] Starting multiAppConn                        module=proxy impl=multiAppConn
I[09-06|20:19:33.730] Starting socketClient                        module=abci-client connection=query impl=socketClient
I[09-06|20:19:33.761] Starting socketClient                        module=abci-client connection=mempool impl=socketClient
I[09-06|20:19:33.794] Starting socketClient                        module=abci-client connection=consensus impl=socketClient
I[09-06|20:19:33.843] ABCI Handshake                               module=consensus appHeight=44 appHash=336630323138333832326331653531663262353634
6343830316239373034643064626
```

Fig. 5. Tendermint Start-up.

to that stored in HBase in order to confirm that the data is the same.

b) Functionality Experiment

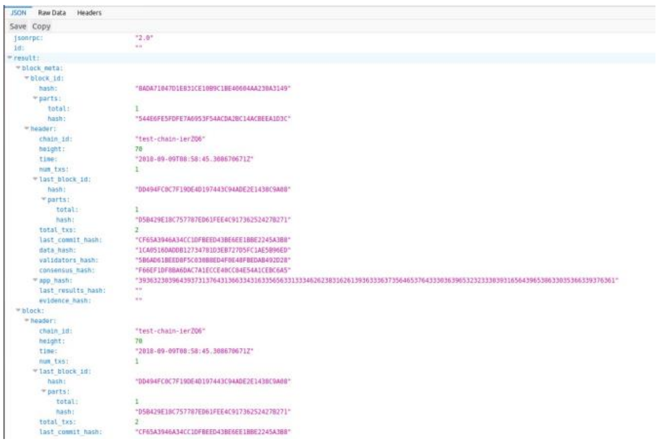
- Purpose: Test if ElasticBloC performs operations normally.
- Required Steps: Testing the functionality of ElasticBloC is done through writing and executing a Python script that gets uses of the bigchaindb_driver library. The steps below show the required steps:
  - Run ElasticBloc components.
  - Write a Python script that creates a transaction, fulfills it with the sender private key, sends the transaction in a commit mode. The following Figure represents the testing Python script.



```python
from bigchaindb_driver import BigchainDB
from bigchaindb_driver.driver import NamespacedDriver
from bigchaindb_driver import crypto


b = BigchainDB('http://mongo-master:9985/')

alice_pub = 'CTMRSLYN34xTeF4wedJoFfrSd5ToCY3MxT128K4PiCDl'
alice_priv = '2uYfrpyBgMZfQkMf3arjL7tyePCQx5LnMLp6dd9AdgPY'
bob_pub = '9RSRsBabmQrFkJMTHg92wTdvobSyvfmrLvQgMAtge2eJ'
bob_priv = '6gG7qnNpZvFGHpWZg1xL9pdLm5hmEP3m2KhjHY7kvL5k'

asset1 = {
            'data': {
                   'Name' : 'Send First Transaction',
                   'Major' : 'BigchainDB'
              }
         }
asset1 = {
            'data': {
                   'Name' : 'Send Second Transaction',
                   'Major' : 'Bob sends to Alice'
              }
         }


p_tx = b.transactions.prepare( operation='CREATE', signers=alice_pub, recipients=None, asset=asset1, metadata=None, inputs=None)
f_tx = b.transactions.fulfill(p_tx, alice_priv)
s_tx = b.transactions.send(f_tx,mode='commit',headers=None)

p_tx2 = b.transactions.prepare( operation='CREATE', signers=bob_pub, recipients=alice_pub, asset=asset1, metadata=None, inputs=None)
f_tx2 = b.transactions.fulfill(p_tx2, bob_priv)
s_tx2 = b.transactions.send(f_tx2,mode='commit',headers=None)
```

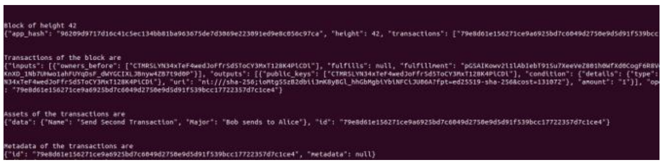
Fig. 6. The Experiment's Python Script

- Results
  The transactions are successfully created and sent. Because the sent transactions are in commit mode, so directly BigchainDB created a block for the transaction. This block was appended to the Tendermint local copy of the blockchain, and the block, including transaction, assets, and metadata are stored in their specific tables at HBase.
  Fig. 7 shows the created block in the blockchain stored at Tendermint.



```
JSON   Raw Data   Headers
Save  Copy
  jsonrpc:                   "2.0"
  id:                        ""
▾ result:
  ▾ block_meta:
    ▾ block_id:
        hash:                "8ADA71047D1EB31CE10B9C1BE40604AA230A3149"
      ▾ parts:
          total:             1
          hash:              "544E6FE5FDFE7A6953F54ACDA2BC14ACBEEA1D3C"
    ▾ header:
        chain_id:            "test-chain-ierZQ6"
        height:              70
        time:                "2018-09-09T08:58:45.308670671Z"
        num_txs:             1
      ▾ last_block_id:
          hash:              "DD494FC0C7F19DE4D197443C94ADE2E1438C9A08"
        ▾ parts:
            total:           1
            hash:            "D5B429E18C757787ED61FEE4C917362524278271"
        total_txs:           2
        last_commit_hash:    "CF65A3946A34CC1DFBEED43BE6EE1BBE2245A3B8"
        data_hash:           "1CA0516DADDB127347B1D3EB72705FC1AE5B96ED"
        validators_hash:     "5B6AD61BEED8F5C038B8ED4F0E48FBEDAB492D28"
        consensus_hash:      "F66EF1DF8BA6DAC7A1ECCE48CC84E54A1CEBC6A5"
      ▾ app_hash:            "3936323039643937313764313663343163356563313334626238316261393633363735646537643330363965323233303931656439653863303536633937636361"
        last_results_hash:   ""
        evidence_hash:       ""
  ▾ block:
    ▾ header:
        chain_id:            "test-chain-ierZQ6"
        height:              70
        time:                "2018-09-09T08:58:45.308670671Z"
        num_txs:             1
      ▾ last_block_id:
          hash:              "DD494FC0C7F19DE4D197443C94ADE2E1438C9A08"
        ▾ parts:
            total:           1
            hash:            "D5B429E18C757787ED61FEE4C917362524278271"
        total_txs:           2
        last_commit_hash:    "CF65A3946A34CC1DFBEED43BE6EE1BBE2245A3B8"
```

Fig. 7. The Appended Block in Tendermint Blockchain.



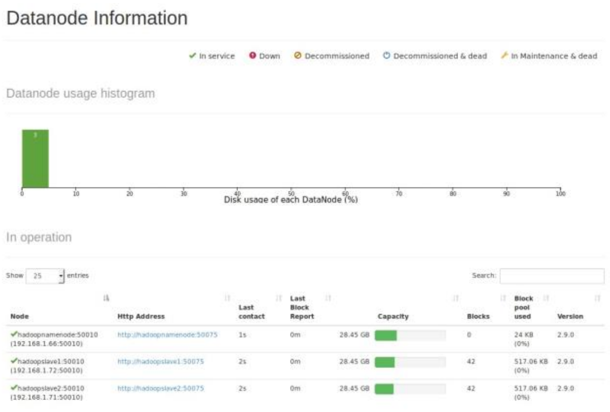
```
Block of height 42
{"app_hash": "96209d9717d16c41c5ec134bb81ba963675de7d3069e223091ed9e8c056c97ca", "height": 42, "transactions": ["79e8d61e156271ce9a6925bd7c6049d2750e9d5d91f539bcc

Transactions of the block are
{"inputs": [{"owners_before": ["CTMRSLYN34xTeF4wedJoFfrSd5ToCY3MxT128K4PiCDi"], "fulfills": null, "fulfillment": "pGSAIKowv2i1lAb1ebT91Su7XeeVeZ801h0WfXd0CogF6R8V
KnXD_1Nb7UHwo1ahFUYqDsF_dWYGCIXLJBnyw4Z87t9d0P"}], "outputs": [{"public_keys": ["CTMRSLYN34xTeF4wedJoFfrSd5ToCY3MxT128K4PiCDi"], "condition": {"details": {"type":
N34xTeF4wedJoFfrSd5ToCY3MxT128K4PiCDi"}, "uri": "ni:///sha-256;ioMtg55zB2dbii3mK8yBGl_hhGbMgbiYbiNFCiJU86A?fpt=ed25519-sha-256&cost=131072"}, "amount": "1"}], "op
: "79e8d61e156271ce9a6925bd7c6049d2750e9d5d91f539bcc17722357d7c1ce4"}

Assets of the transactions are
{"data": {"Name": "Send Second Transaction", "Major": "Bob sends to Alice"}, "id": "79e8d61e156271ce9a6925bd7c6049d2750e9d5d91f539bcc17722357d7c1ce4"}

Metadata of the transactions are
{"id": "79e8d61e156271ce9a6925bd7c6049d2750e9d5d91f539bcc17722357d7c1ce4", "metadata": null}
```

Fig. 8. The Appended Block and Its Details in HBase.

Fig. 8 shows the result of the submitted transaction with its details retrieved from HBase.

- Discussion
  The experiment is conducted successfully and the architecture is working finely.
  One of the positive aspects of the architecture is that creation, fulfillment, sending, and validating of the transaction, with the block creation and appending to the blockchain, and its storage in HBase took around only one second.
  One major limitation of the experiment is that these experiments were conducted on a limited blockchain dataset. The reason behind this is that there was neither possibility to access a large blockchain dataset nor time to build our own large dataset. Instead, we take into consideration previous benchmarks and experiments were done using huge bulks of data in which it helps us to evaluate our architecture.

c) Scalability Experiment
- Purpose
  Test if ElasticBloC can scale massively.
- Required Steps
  Massive scalability means that ElasticBloC has the ability to scale as much as it needs on its physical layer. This could be ensured if we succeeded in adding new data nodes to the ElasticBloC node. To do that we tried to add new Hadoop nodes to the Hadoop cluster either while ElasticBloC is running or when it is offline.
- Results
  The previous experiment is conducted successfully and ElasticBloC runs normally.



| Node | Http Address | Last contact | Last Block Report | Capacity | Blocks | Block pool used | Version |
|---|---|---|---|---|---|---|---|
| ✓hadoopnamenode:50010 (192.168.1.66:50010) | http://hadoopnamenode:50075 | 1s | 0m | 28.45 GB | 0 | 24 KB (0%) | 2.9.0 |
| ✓hadoopslave1:50010 (192.168.1.72:50010) | http://hadoopslave1:50075 | 2s | 0m | 28.45 GB | 42 | 517.06 KB (0%) | 2.9.0 |
| ✓hadoopslave2:50010 (192.168.1.71:50010) | http://hadoopslave2:50075 | 2s | 0m | 28.45 GB | 42 | 517.06 KB (0%) | 2.9.0 |

Fig. 9. The New Data nodes Cluster.

- Discussion
  As ElasticBloC scales by adding a new server to the Hadoop cluster, it is feasible to add data nodes either on fly or offline. This means that ElasticBloC has the ability to scales massively.

d) Performance Evaluation
Concerning the large blockchain dataset, we were not able to access a large blockchain dataset rather than building it. For that, we could not test ElasticBloC on a massive scale dataset. Accordingly, we rely on this section on some previously conducted experiments and workbenches that give us, theoretically, a clear idea on the performance of the overall architecture.
The overall performance of ElasticBloC is evaluated by its components, in particular, the performance of Tendermint as a consensus engine and HBase as a backend database.
Tendermint acts as a high-performant in a large distributed environment. According to Cosmos white paper [18]:
"Despite its strong guarantees, Tendermint provides exceptional performance. In benchmarks of 64 nodes distributed across 7 data centers on 5 continents, on commodity cloud instances, Tendermint consensus can process thousands of transactions per second, with commit latencies on the order of one to two seconds. Notably, the performance of well over thousands of transactions per second is maintained even in harsh adversarial conditions, with validators crashing on broadcasting maliciously crafted votes."
On the other side, building MongoDB on the top of HDFS is less efficient than building HBase on the top of the mention file system. The reason behind this argument is that HBase is natively developed to run on the top of HDFS, while MongoDB needs a connector as the third party to be built on the top of HDFS.
Moreover, based on several benchmarks, such as [11] and [12], HBase acts more efficiently than MongoDB in large

clusters. For instance, End Point [23] performed a series of tests for the performance of several NoSQL databases including HBase and MongoDB. The following are some comparison results for ‘the performance of HBase and MongoDB in different tests based on [11].

TABLE III
Throughput Comparison while Loading Data

| Number of Nodes | HBase (operation/sec) | MongoDB (operation/sec) |
|---|---|---|
| 1 | 15617.98 | 8368.44 |
| 2 | 23373.93 | 13462.51 |
| 4 | 38991.82 | 18038.49 |
| 8 | 74405.64 | 34305.30 |
| 16 | 143553.41 | 73335.62 |
| 32 | 296857.36 | 134968.87 |

TABLE IV
Throughput Comparison while Retrieving Data

| Number of Nodes | HBase (operation/sec) | MongoDB (operation/sec) |
|---|---|---|
| 1 | 428.12 | 2149.08 |
| 2 | 1381.06 | 2588.04 |
| 4 | 3955.03 | 2752.40 |
| 8 | 6817.14 | 2165.17 |
| 16 | 16542.11 | 7782.36 |
| 32 | 20020.73 | 6983.82 |

TABLE V
Throughput in Balanced Read/Write

| Number of Nodes | HBase (operation/sec) | MongoDB (operation/sec) |
|---|---|---|
| 1 | 527.47 | 1278.81 |
| 2 | 1503.09 | 1441.32 |
| 4 | 4175.8 | 1501.06 |
| 8 | 7725.94 | 2195.92 |
| 16 | 16381.78 | 1230.96 |
| 32 | 20177.71 | 2335.14 |

TABLE VI
Throughput in Read/Update/Write Operations.

| Number of Nodes | HBase (operation/sec) | MongoDB (operation/sec) |
|---|---|---|
| 1 | 324.8 | 1261.94 |
| 2 | 961.01 | 1480.72 |
| 4 | 2749.35 | 1754.30 |
| 8 | 4582.67 | 2028.06 |
| 16 | 10259.63 | 1114.13 |
| 32 | 16739.51 | 2363.69 |

TABLE VII
Throughput Comparison in Mixed Operational and Analytical Workload

| Number of Nodes | HBase (operation/sec) | MongoDB (operation/sec) |
|---|---|---|
| 1 | 269.30 | 939.01 |
| 2 | 333.12 | 30.96 |
| 4 | 1228.61 | 10.55 |
| 8 | 2151.74 | 39.28 |
| 16 | 5986.65 | 337.4 |
| 32 | 8936.18 | 227.80 |

The above experiment is tangible evidence of how HBase is more efficient than MongoDB.
Hence, according to theory and pre-existing experiments, HBase would also enhance the overall performance of ElasticBloC. However, in its worth case, replacing MongoDB with HBase will not downgrade the performance of ElasticBloC.

## G. Conclusion & Future Work

Blockchain has not been adopted widely until now except for cryptocurrency applications, however, it has been identified as potential technologies for several areas that need trust and security.

It is relatively a new technology that needs various improvements to reach to a maturity level. It has several limitations that are the main barriers to the wider adoption of this technology. Of all, scalability and performance are the major limitations that must be addressed.

This research primarily aims at addressing the scalability problem. There are some solutions that offer techniques methods, and guidelines for scalable blockchain. However, we found that state-of-the-art technologies focus on scalability at the logical level which is an inadequate approach if scalability at the physical level is to be guaranteed. In this paper, we designed and implement a scalable architecture called ElasticBloC which enables users to build a highly scalable blockchain-based ecosystem consisting of tens or more of physical nodes.

In this paper, we proposed a solution for the scalability limitation of blockchain technology. We discussed our solution ElasticBloC which is a scalable architecture for building blockchain-based applications such as payment system, notarization, the smart contract can be implemented by ensuring scalability. ElasticBloC is a built on cluster computing paradigm that building infrastructure with a massive number of nodes. We presented the components with a detailed description. We discussed the workflow of ElasticBloc. We discussed technologies that we used in implementing the proposed solution.

We tested ElasticBloc to evaluate the scalability. Our experiment shows that ElasticBloc has the ability to scale up; it is flexible for adding as many servers. We discussed the results of our experiments in this paper. Additionally, we provided some previous workbenches to provide a comparative view of the abilities of ElasticBloc.

Several works are lined up for a future extension of ElasticBloC. However, to the best of our knowledge, the imminent critical task that must be accomplished in the near future is extending the functional capabilities of our solution. We planned to enhance add new modules to ElasticBloC to enable users to develop permission-less blockchain-based applications or for permission blockchain-based applications.

## REFERENCES

[1] Anonymous. Available Retrieved from https://tendermint.readthedocs.io/en/latest/introduction.html.

[2] Branden, J. V. Building a Performance Model of the Tendermint Concensus Algorithm.

[3] Buchman, E. (2016). Tendermint: Byzantine fault tolerance in the age of blockchains (Doctoral dissertation).

[4] Buterin, V. (2014). A next-generation smart contract and decentralized application platform.

[5] Castro, M., & Liskov, B. (2003). U.S. Patent No. 6,671,821. Washington, DC: U.S. Patent and Trademark Office.

[6] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., ... & Gruber, R. E. (2008). Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2), 4.

[7] Condos, J., Sorrell, W. H., & Donegan, S. L. (2016). Blockchain technology: Opportunities and risks.

[8] DBS Group Research. (2016). Understanding blockchain technology and what it means for your business.

[9] Dorri, A., Kanhere, S. S., Jurdak, R., & Gauravaram, P. (2017). LSB: A Lightweight Scalable BlockChain for IoT Security and Privacy. arXiv preprint arXiv:1712.02969.

[10] Ehmke, C., Wessling, F., & Friedrich, M. C. (2018). Proof-of-property: a lightweight and scalable blockchain protocol. In Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain. (pp. 48-51). ACM.

[11] End Point. (2015). Benchmarking Top NoSQL Databases: Apache Cassandra, Couchbase, Hbase, and MongoDB.

[12] Gandini, A., Knottenbelt, W. J., Osman, R., & Piazolla, P. (n.d.). Performance evaluation of NoSQL databases.

[13] Gao, Z., Xu, L., Chen, L., Shah, N., Lu, Y., & Shi, W. (2017, December). Scalable blockchain based smart contract execution. In Parallel and Distributed Systems (ICPADS), 2017 IEEE 23rd International Conference on (pp. 352-359). IEEE.

[14] Gencer, E. A., Sirer, G. E., Van Renesse, R., & Eyal, I. (2016). Bitcoin-NG: A Scalable Blockchain Protocol. In NSDI.

[15] George, L. (2011). HBase: the definitive guide: random access to your planet-size data. " O'Reilly Media, Inc.".

[16] Gunicorn - Python WSGI HTTP Server for UNIX. (n.d.). Retrieved from https://gunicorn.org.

[17] The HTTP Client-Server API — BigchainDB Server 0.8.2 documentation. (n.d.). Retrieved from http://docs.bigchaindb.com/projects/server/en/v0.8.2/drivers-clients/http-client-server-api.html.

[18] Internet of Blockchains - Cosmos Network. (n.d.). Retrieved from https://cosmos.network/resources/whitepaper.

[19] James-Lubin, K. (2015, January 22). Blockchain scalability. Retrieved from https://www.oreilly.com/ideas/blockchain-scalability.

[20] Kwon, J. (2014). Tendermint: Consensus without Mining.

[21] McConaghy, T., Marques, R., Müller, A., De Jonghe, D., McConaghy, T., McMullen, G., ... & Granzotto, A. (2016). BigchainDB: a scalable blockchain database. white paper, BigChainDB.

[22] Out of Asia. (2017, December 27). Five Issues Preventing Blockchain From Going Mainstream: The Insanely Popular Crypto Game Etheremon Is One Of Them. Retrieved from https://www.forbes.com/sites/outofasia/2017/12/22/five-issues-preventing-blockchain-from-going-mainstream-the-insanely-popular-crypto-game-etheremon-is-one-of-them/#6d364bb66fad.

[23] Secure Business Solutions — End Point. (n.d.). Retrieved from http://www.endpoint.com/.

[24] Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010, May). The hadoop distributed file system. In Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on (pp. 1-10). Ieee.

[25] Vukolić, M. (2015, October). The quest for scalable blockchain fabric: Proof-of-work vs. BFT replication. In International Workshop on Open Problems in Network Security (pp. 112-125). Springer, Cham.

[26] Zamani, M., Movahedi, M., & Raykova, M. (n.d.). RapidChain: A Fast Blockchain Protocol via Full Sharding.

[27] ZooKeeper, A. (2017). The Apache Software Foundation. Accessed December, 29, 2017.