Reducing Disk Storage with SQLite into BitCoin Architecture

http://dx.doi.org/10.3991/ijes.v3i2.4490

Rodrigue Carlos Nana Mbinkeu[1,2,3], Bernabé Batchakui[1]
[1] University of Yaoundé I, Cameroon
[2]African Centre of Excellence on Information and Communication Technology (CETIC)
[3]DBgroupLab – University of Modena, Italy

***Abstract*—For the past five years, the bitcoin network constantly experience a growth in its size as more communities turn to accept the currency for payment exchanges. Using Flat File and a LevelDB of indices to save blocks on disk, bitcoin users require more memory to save the history of transaction. We focus on issues of memory management and access time in the bitcoin protocol using SQLite DataBase. With all the advantages of SQLite DataBase, it would be efficient if it is fitted in this architecture. The SQLite comes with many flavors one of which is its ability to support sql queries. Thus, instead of parsing indices to search a block from the database, a more powerful query can do the job.**

***Index Terms*—Bitcoin, LevelDB, Flat File, Hash, Query, SQL, SQLite, Index, Memory, Disk.**

I. INTRODUCTION

A brief description of the concepts and architecture schematics of the Bitcoin protocol was published by pseudonymous developer Satoshi Nakamoto. The P2P digital crypto-currency is described in this paper also referred to as the original bitcoin paper, published in 2009 [1]. For the past five years, the bitcoin network constantly experience a growth in its size as more communities turn to accept the currency for payment exchanges.

The Bitcoin is the first electronic payment system that is based on cryptographic proof instead of trust [1]. This crypto-currency platform serves as the foundation of other crypto-currencies [1, 2, 3, 4]. A lot of articles have been written, purporting to explain the details of this crypto-currency [1, 2, 3, 4].

It is rather unfortunate that, at a time when the bitcoin is about exploding, many issues pops in which pose questions that require more research on the protocol in order to answer them. Issues on security and trust in the protocol have been discussed in many forums with more than a hundred articles written. It is obvious that the size of the network keeps increasing as more communities turn to adopt the currency for payment exchange. Using Flat File and a LevelDB of indices to save blocks on disk, bitcoin users require more memory to save the history of transaction [1, 3]. In this article, we focus on issues of memory management and access time in the bitcoin protocol. This is where this article comes in to propose a solution deemed favorable to users and developers of bitcoin.

In this context, our work seeks to better understand the Bitcoin protocol presented in section 1 and the bitcoin architecture in section 2. Through this understanding, explore how we can replace the Flat File and LevelDB with SQLite, section 3. We discuss the perspectives associated with the results obtained in section 4. Finally we make a conclusion.

II. BIT COIN PROTOCOLE

Bitcoin is the first decentralized digital currency, developed in 2009 following the Satoshi Nakamoto's white paper [1]. They are digital coins that are sent from person-to-person via the internet without going through a clearing house [1, 3, 4]. Bitcoins are generated all over the internet by anybody running a free application called a bitcoin miner. Mining requires a certain amount of work for each block of coins. This difficulty (amount of work required to mint a bitcoin) is automatically adjusted by the network such that the coins are always created in an unpredictable and unlimited rate [1, 3].

Technically speaking, a bitcoin, as described in the white paper [1], is a chain of digital signatures. Thus, transactions in the bitcoin network are simply, the exchange and verification of digital signatures. Let's recap a few concepts used to design the bitcoin protocol [1].

A. Bit coin transaction

These are digitally signed section of data that are broadcast to the bitcoin network. Each owner transfers the coin to the next by digitally signing a hash of the previous transaction and the public key (bitcoin address) of the next owner and adding these to the end of the coin. Thus, a transaction typically references previous transactions (inputs) and specifies the amount of bitcoins (outputs) from it to one or more new bitcoin addresses. Transactions are collected in a block. A payee can verify the signatures in order to prove that a particular amount of bitcoin belong to an individual.

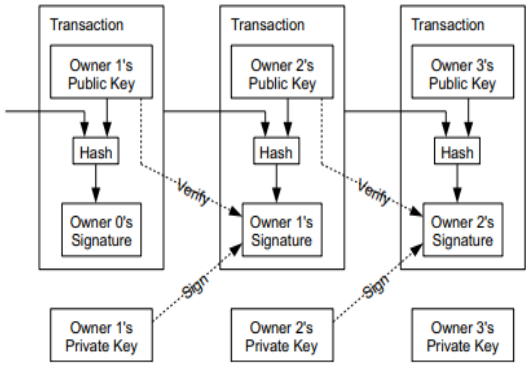

Figure 1. Chain of digital signatures

TABLE I.
GENERAL FORMAT OF A TRANSACTION IN BLOCK

Field	***Size****/bytes*
Version No	4
In-Counter	1-9
List of Inputs	Data size
Out-Counter	1-9
List of Outputs	Data size
Lock time	4

B. Bitcoin Address

Payments are made to bitcoin addresses – a string of 27 – 34 alphanumeric characters, beginning with a 1 or 3. Properly generated bitcoin addresses have at least one secrete information, private key, which is used to verify ownership and validity of bitcoins and transactions respectively [1].

C. Network protocol

At the core of the bitcoin network, a bitcoin is seen as a digital file that list accounts and money like a "ledger" (in the traditional sense). A copy of this file is maintained on every computer in the bitcoin network. To send money, one simply broadcast the information (transaction) to the network. Nodes of computers in the network apply this information to their copies of the ledger and then pass on the transactions to other nodes. With this update of the ledger by every node in the network, security issues are handled [1]. Thus, opposed to banking transactions in which one only knows about his/her own transactions, in bitcoin, everyone knows about everyone else's transactions. Hence, bitcoin transactions take place between anonymous strangers; the system is designed so that no trust is needed. The network uses special mathematical algorithms to protect various aspects of the system [1].

D. Timestamp server

A timestamp is the time at which a transaction is recorded by a computer. In the bitcoin network, a timestamp server works by taking a hash of a block of transactions to be timestamp, and broadcasting the hash to every node in the network. This proves that the transaction must have existed at the time, in order to get into the hash. Each timestamp includes the previous timestamp in its hash, forming a chain, with each additional timestamp reinforcing the ones before it [1].

E. Proof-of-work

The bitcoin protocol uses a proof-of-work system to deter denial of service attacks [1, 2, 3]. The order of transactions is a very crucial issue in the network. In the bitcoin network, transactions are ordered by placing them in groups called blocks which are then linked together in what bitcoin refers to as a block chain [1, 5].

Each block has a reference to previous blocks. Transactions in the same block are considered to have happened at the same time. Transactions not yet in any block are called "unconfirmed or unordered" transactions. Nodes in the network collect a set of unconfirmed transactions into a block and broadcast it to the network as a suggestion to which block is to be the next in the chain. Observe that multiple nodes could create blocks at the same time, thus, several options to choose from as to which block is to be the next in the chain. In such a case, the network uses the proof-of-work system to decide which block to choose. With this system, each valid block is to contain a very special mathematical problem. Computers in the network run an entire text to solve this problem by doing some random guess. The first person to find the solution to this problem then broadcast its block to the network as the only valid candidate to be the next in the chain. Hence, the randomness in the solution makes it unlikely that two people can solve a block at the same time.

F. Bitcoin Infrastructure

Bitcoind deamon is the first bitcoin client. It is the first computer program that implements the bitcoin protocol for command line and remote procedure call (RPC) use. It is now bundled as a pack with the Bitcoin-Qt. We will use this product to illustrate our findings. When a bitcoin client is first intalled, it downloads every transactions ever made and checks their validity [1, 5]. The "bitcoind daemon" is specially implemented to support this feature of the protocol.

G. Architecture

The entire protocol is built on the bitcoind daemond suit. This suit is like the core of the platform and support all activities of the protocol; database management, transaction management, message signing, mining, etc. There are various modules that are implemented in the "bitcoind daemon" to support the bitcoin protocol [1]. The following modules of the architecture are of interest:

- Database module
- Network module
- Serialization module

Database Module

The bitcoin daemon is equiped with various embedded databases that support the management of wallets, transactions, blocks, and bitcoin addresses in the bitcoin network.

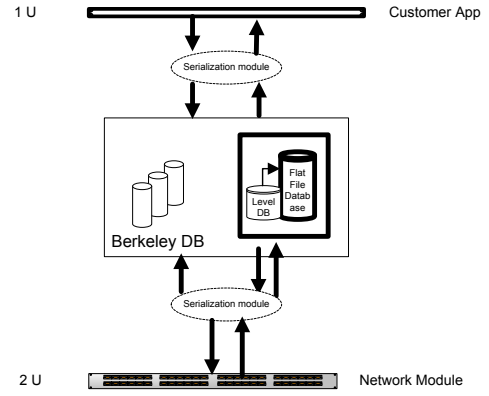

Figure 2. Storage mechanism of the bitcoin daemon.

- **Berkeley Database Engine:** At the lowest level of this program is a Berkeley DB that keeps history of all wallets that are used in the protocol. Berkeley DB (BDB) is a high-performance embedded database engine of key/value type produced by Oracle. In the bitcoin protocol, this database engine manages wallets which contain vital informations pertaining to the user such as private/public keys, name, address, account balances, and more.
- **Flat File Database:** Next in this protocol is a flat database file. Blocks, containing transactions, are written into this file in raw database format. A wrapper class for the C file descriptor, which im-

plements serialization writing and reading byte streams to and from the file respectively is used to read and write blocks to these files respectively. These block files are later encrypted to prevent access by an external application process.

- **LevelDB:** To improve on the access time to the information stored on the flat database, the protocol implements this open source persistent key/value database – leveldb. With an increase in size of the bitcoin network, accessing information on the flat file becomes costly, in terms of speed. Reading blocks is made faster through the use of indices managed by LevelDB. The bitcoin daemon uses a leveldb database engine to store index values which are used for accessing information on the file.

When launched, the daemon calls the **"OpenBLockFIle()"** method from **"main.h"** which returns the **"OpenDiskFile()"** method. With the help of the "boost filesystem" library, the flat file is created in this method.

```
FILE* OpenDiskFile(const CDiskBlockPos &pos,
const char *prefix, bool fReadOnly)
{
...
boost::filesystem::path path = GetDataDir() / "blocks" / strprintf("%s%05u.dat",
prefix, pos.nFile);
...
}
```

Serialization module

Observe that, the flat file stores data in a raw network format. Hence, it is necessary to translate all data structures or objects in the bitcoin network into a format that can be stored [1, 5]. The data can then be reconstructed later in the same or another computer environment. The serialization module is equiped with methods for the serialization/unserialization of data structures in various formats. Data is written and read to and from the database module respectively. The WriteBlockToDisk() and ReadBlockFromDisk() menthods of main.h enable the writing and reading of blocks to and from the flat file database respectively. The data structure has to be serialized/unserialized as the need arises. With the above configuration, reading block form the flat file requires parsing the block's position as a parameter to the ReadBlockFromDisk() method. This index value is stored in the leveldb. Thus, a block in the database is both its index value and its data.

III. USING SQLITE INTO BITCOIN ARCHITECTURE

SQLite, by design, is engineered to be a portable, efficient SQL storage engine that offers maximum convenience, simplicity, in a small footprint [6, 7]. SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine [6, 7]. SQLite is an embedded SQL database engine. Unlike most other SQL databases, SQLite does not have a separate server process. SQLite reads and writes directly to ordinary disk files [6, 7]. A complete SQL database with multiple tables, indices, triggers, and views, is contained in a single disk file [6, 7]. The database file format is cross-platform; you can freely copy a database between 32-bit and 64-bit systems architectures. These features make SQLite a popular choice as an Application File Format [6, 7].

According to the site "http://www.sqlite.org/about.html", with all features enabled, the library size can be less than 500KiB, depending on the target platform and compiler optimization settings. If optional features are omitted, the size of the SQLite library can be reduced below 300KiB. SQLite can also be made to run in minimal stack space (4KiB) and very little heap (100KiB), making SQLite a popular database engine choice on memory constrained gadgets such as Smartphones, cellphones and MP3 players. There is a tradeoff between memory usage and speed. SQLite generally runs faster the more memory you give it. Nevertheless, performance is usually quite good even in low-memory environments [6, 7].

Unlike client–server database management systems, the SQLite engine has no standalone processes with which the application program communicates. Instead, the SQLite library is linked in and thus becomes an integral part of the application program. Several computer processes or threads may access the same database concurrently [6, 7]. Several read accesses can be satisfied in parallel. Applications interact with the SQLite library through function calls managed by the application process which is more efficient than inter process communications. Some of these functions include:

- sqlite3_open(): Opens an SQLite database file.
- sqlite3_close(): Closes a database file.
- sqlite3_exec(): Executes SQL statements.
- sqlite3_prepare(): Used for carrying out multiple inserts.
- sqlite3_db_mutex(): Sets the database with mutex enabled.
- sqlite3_db_filename(): Returns a pointer to a file name associated with a given database.

A. SQLite in Bitcoin

The decentralized paradigm of Bitcoin requires each node of the network to retain the blockchain (i.e., entire transaction history) [1, 8]. The block database rapidly grows in size as new blocks with an average size of about 50 KB are added to the database after an average of 10 minutes. With this increase in size, it becomes costly to read a particular block from the database. As a solution, the bitcoin daemon implements a leveldb database whose contents are indices for the blocks in the flat file. Keeping indices and data value is yet another problem as it makes use of resources – disk space. With all the advantages of relational databases and sql–like database engines, sqlite, is proposed in this architecture. Instead of storing and retrieving serialized and unserialized data, a relational database of tables is used to store information.

The use of SQLite make that the old system of storage of indices (in leveldb) will be eliminated. So, this has an effect to reduce the use of disk storage. The access time of data may be reduced by suppressing the module of serialization/unserialization. Therefore, we are expecting to obtain some reduction memory by using SQLite.

Database Design

The goal is to store/retrieve blocks. From the bitcoin protocol specification, a block's structure is comprised of

a header and some transactions. The header stores the current block header version (nVersion), a reference to the previous block (HashPrevBlock), the root of the Merkle tree (HashMerkleRoot), a timestamp (nTime), a target value (nBits) and a nonce (nNonce).

In the relational representation of a block, the above structure is stored using 5 relations, namely:

- Block(block_id, hashMerkleRoot, txn_counter)
- BlockHeader(id, nVersion, hashPrevBlock, hashMerkleRoot, nTime, nBits, nonce)
- Txn(txn_id, nVersion, inCounter, outCounter, lock_time, block_id, id_file)
- TxnIn(id, hashPrevTxn, txnOut_id, scriptLen, scriptSig, seq_no, txn_id)
- TxnOut(id, value, scriptLen, scriptSig, txn_id)

From the above relational schema, we observe the following:

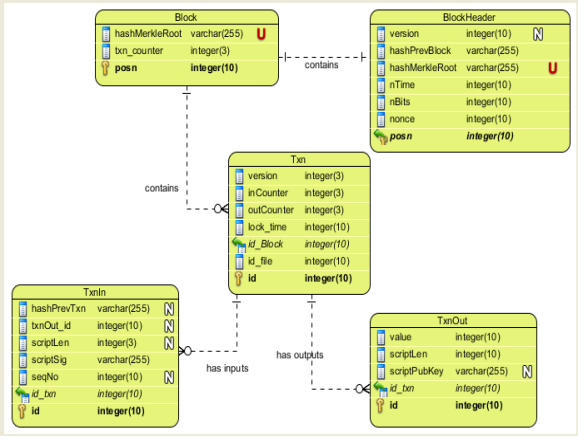

Figure 3. Entity Relationship Diagram of the block in bitcoin

The entity relationship diagram of the above relational schema is given below. A block is made up of one and only one header which is in turn included in one and only one block. The hashMerkleRoot attribute serves as a foreign key relating a block to its header meanwhile txn_counter holds the number of transactions being stored in the block. A BlockHeader keeps its attributes according to the diagram for the block.

Transactions (Txn) found in a block also has a list of inputs and outputs, the inCounter and outCounter attributes keep the number of inputs and outputs respectively contained in a transaction. The attribute id_Block is a foreign key that references the id of the block in which the transaction is contained while id_file keeps the file number in which the transaction is written on disk. A Block can contain many transactions but a transaction is found in one and only one block.

A transaction input (TxnIn) belongs to a single transaction, id_txn is a foreign key referencing the id transaction in to which the input belongs. Just like the input, a transaction output belongs to a single transaction and this relationship is expressed through the possession of foreign key referencing the transaction to which it belongs. The id attributes of all tables act as primary keys and are set to auto-increment; this is to enable indexing the tables and make queries less complex to obtain desired data.

Bitcoin System Design with SQLite

The logical architecture will seem simple. No serialization is needed in this architecture. Blocks are written and read to and from the database by parsing sql-like statements. The following figure gives the logical architecture of the system.

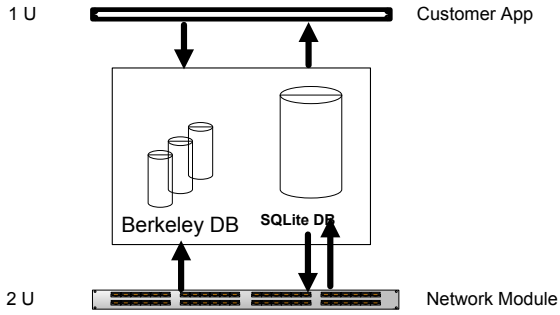

Figure 4. Logical architecture of bitcoind daemon with SQLite

With this configuration, to read or write data, no serialization is needed since the database contains well-structured data. Data is retrieved by writing SQL queries. The figures below give a series of messages that are passed when writing/reading blocks from the database.

IV. RESULTS

SQL creation scripts are written to create the database file and its handler. This handler is then passed to functions. A wrapper class with methods to perform SELECT, INSERT, UPDATE and DELETE is coded to support the queries passed to the program. Look below an example of method developed into our bitcoin deamon with SQLike.

```
bool FindBlockPos(CValidationState &state,
        CDiskBlockPos &pos, unsigned int nAddSize,
unsigned int nHeight, uint64_t nTime, bool fKnown = false)
    {...
        CSQLiteWrapper db = CSQLiteWrapper();

        db.OpenConnection(strprintf("blk%05u.dat",pos.nFile),Get
                DataDir() / "blocks" );
int n = db.Execute("SQL QUERY");
  ... }
```

A. Results and Perspectives

DataBase Access

Bitcoin with flat file and LevelDB: requires iterating through a bulk of data item using an index value. For example, reading a block from the file from the file requires looking for the block's position in the database. This requires iterating through a list of blocks which is time consuming. We remember that for saving or reading data, the system use the serialization/deserialization functions which are also factors that increment time consuming.

Bitcoin with SQLite: SQLite stores data in structured format, so it will be easier to find a record from multiple set of records which is very tedious process in case of flat file. With relational model, it is easy to construct ad-hoc queries against the data in the database. SQLite is good for performing complex operations on large amount of data. For example, to read a block from the database, the following four queries are used to return a vector of the content of a block:

- *"SELECT * FROM BlockHeader WHERE id=%d",pos.nPos*
- *"SELECT * FROM Block WHERE id=%d",pos.nPos*
- *"SELECT * FROM TxnIn WHERE id_txn=%d",vIdTxns[i]*
- *"SELECT * FROM TxnOut WHERE id_txn=%d",vIdTxns[i]*

Disk Management

Using the flat file, the size of a block on disk, as of 2009, is approximately 10KB according to the site *"http://blockchain.info"* Using SQLite, for 500 blocks downloaded, we have an approximated 368 KB size on disk. Analyzing and comparing, we observed that with SQLite, a block has a size of approximately 0.8KB i.e. 800B. With this size we conclude that SQLite brings with it an approximated 92% reduction of block size on disk. The possible reasons for this reduction are:

- removal of redundant data opposed to inclusion of redundant data during serialization in case of flat file;
- removal of indices used to search block into the old system storage (levelDB);
- Saving only relevant information leaving out information generated by the protocol.

B. Perspectives

Our tests and preliminary results give us the opportunity to improve further this new architecture and analyze deeply how we obtain this high reduction of block size on disk. We are sure that these preliminary results are very important because it will be easy now to implement a bitcoind daemon client with SQLite for small devices like as smartphones will can be downloaded more blocks from blockchain data than before and their limit is only the CPU computing power.

V. CONCLUSIONS

At this point, we conclude by encouraging, with all the advantages that it brings, SQLite in the bitcoin architecture. One of its features – relational model is appreciable. The relational model structures data in a manner that avoids complexity i.e accessing data in a database does not require navigating a rigid pathway through a tree or hierarchy like as done in the levelDB. This research work is the preliminary investigation in bitcoin architecture by replacing Flat file and LevelDB with SQLite. We obtain some important results as memory reduction on disk and time access of data. In the future works, we are planning to analyze deeply the implication for using SQLite Database in the bitcoin architecture.

REFERENCES

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

[2] D. Drainville., An Analysis of the Bitcoin Electronic Cash System, 2013 https://math.uwaterloo.ca/combinatorics-and-optimization/sites/ca.combinatorics-and-optimization/files/uploads/files/Drainville,n%20Danielle.pdf

[3] Bitcoin Gateway, A Peer-to-peer Bitcoin Vault and Payment Network, 2011. Available from http://arimaa.com/bitcoin/.

[4] Beverly Yang and Hector Garcia-Molina. Ppay: micropayments for peer-to-peer systems. In Proc. of Computer and communications security, 2003. http://dx.doi.org/10.1145/948109.948150

[5] Erik R. Barnett; Virtual Currencies: Safe for Business and Consumers or just for Criminals? 13th European Security Conference & Exhibition. The Hague April 2, 2014.

[6] Chunyue Bi. Research and Application of SQLite Embedded Database Technology. WSEAS TRANSACTIONS on COMPUTERS, Issue 1, Volume 8, January 2009.

[7] Sunguk Lee. Creating and Using Databases for Android Applications. International Journal of Database Theory and Application Vol. 5, No. 2, June, 2012.

[8] Spagnuolo, M.: Bitiodine: Extracting intelligence from the bitcoin network. Master's thesis, Politecnico di Milano (December 2013).

AUTHORS

Nana MBinkeu R. C. is a Senior Lecturer in the National Advanced School of Engineering, P.O Box 8390 ENSP, University of Yaoundé I, Cameroun (e-mail: nanambinkeu@gmail.com).

Batchakui B. is a Senior Lecturer in the National Advanced School of Engineering, P.O. Box 8390 ENSP, University of Yaoundé I, Cameroun (e-mail: bbatchakui@gmail.com).

Submitted 24 February 2015. Published as resubmitted by the authors 12 May 2015.