Bitcoin Blockchain Query API

HingOn Miu
hmiu@andrew.cmu.edu

Block Data Structure:

Magic Number 4 bytes	Block Size 4 bytes	Version Number 4 bytes	
SHA256 Hash of Previous Block 32 bytes			
SHA256 Hash of Merkle Root 32 bytes			
			Mining Time 4 bytes
nBits 4 bytes	Nonce 4 bytes	Transaction Count 1, 3, 5 or 9 bytes	
Transactions ~ bytes			
…			
NULL Padding ~ bytes			
Magic Number 4 bytes	…		

Block Example:

Genesis Block

> hexdump -C -n 297 blk00000.dat

```
f9 be b4 d9 1d 01 00 00  01 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 3b a3 ed fd
7a 7b 12 b2 7a c7 2c 3e  67 76 8f 61 7f c8 1b c3
88 8a 51 32 3a 9f b8 aa  4b 1e 5e 4a 29 ab 5f 49
ff ff 00 1d 1d ac 2b 7c  01 01 00 00 00 01 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 ff ff
ff ff 4d 04 ff ff 00 1d  01 04 45 54 68 65 20 54
69 6d 65 73 20 30 33 2f  4a 61 6e 2f 32 30 30 39
20 43 68 61 6e 63 65 6c  6c 6f 72 20 6f 6e 20 62
72 69 6e 6b 20 6f 66 20  73 65 63 6f 6e 64 20 62
61 69 6c 6f 75 74 20 66  6f 72 20 62 61 6e 6b 73
ff ff ff ff 01 00 f2 05  2a 01 00 00 00 43 41 04
67 8a fd b0 fe 55 48 27  19 67 f1 a6 71 30 b7 10
5c d6 a8 28 e0 39 09 a6  79 62 e0 ea 1f 61 de b6
49 f6 bc 3f 4c ef 38 c4  f3 55 04 e5 1e c1 12 de
5c 38 4d f7 ba 0b 8d 57  8a 4c 70 2b 6b f1 1d 5f
ac 00 00 00 00 f9 be b4  d9 .. .. .. .. .. .. ..
```

Block Fields:

Magic Number
The first 4 bytes of every block in blockchain is 0xD9B4BEF9. The magic number is an identifier to alert parser a new block is right after it. It is in little-endian order.

Block Size
These 4 bytes contain the size of this block, starting from the block header to the end of all transactions. The block size field is in little-endian order.

Version Number
These 4 bytes indicate which set of block validation rules to follow. The version number is in little-endian order.

Previous Block Hash
These 32 bytes contain a SHA256(SHA256()) hash of the previous block's header, such that each block points to its previous block. The previous block hash is in internal byte order.

Merkle Root Hash
These 32 bytes contain a SHA256(SHA256()) hash of the Merkle root, which is derived from the hashes of all transactions in this block, such that none of those transactions can be modified without modifying the header. The Merkle root hash is in internal byte order.

Time
These 4 bytes contain a Unix epoch time when the miner started hashing the header. It must be strictly greater than the median time of the previous 11 blocks. It is in little-endian order.

nBits
These 4 bytes contain an encoded version of the target threshold this block's header hash must be less than or equal to, such that nBits adjusts the difficulty of hashing. It is in little-endian order.

Nonce
These 4 bytes contain an arbitrary number that miners change to modify the header hash in order to produce a hash less than or equal to the target threshold. It is in little-endian order.

Transaction Count
This is a variable length integer that may vary in length to save space. It represents the total number of transactions in this block. It is in little-endian order.

Transactions
These bytes contain raw transactions that each transaction may have multiple inputs and outputs. Each transaction contains version number, input count, list of input transactions, output count, list of output transactions and lock time.

NULL Padding
There are possible zero bytes in between blocks.

Transaction Data Structure:

Version Number 4 bytes	Input TX Count ~ bytes		
Input #1: SHA256 Hash of Previous Transaction 32 bytes			
		Input #1: TX Index 4 bytes	Input #1: Script Size ~ bytes
Input #1: Signature Script ~ bytes			
Input #1: Sequence 4 bytes	Output TX Count ~ bytes	Output #1: Satoshi Amount 8 bytes	
Output #1: Script Size ~ bytes	Output #1: Pub-Key Script ~ bytes		
Lock Time 4 bytes	Version Number 4 bytes	…	

Transaction Example:

Coinbase Transaction of Genesis Block

> hexdump -C -s 89 -n 204 blk00000.dat

```
01 00 00 00 01 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
00 00 00 00 00 ff ff ff  ff 4d 04 ff ff 00 1d 01
04 45 54 68 65 20 54 69  6d 65 73 20 30 33 2f 4a
61 6e 2f 32 30 30 39 20  43 68 61 6e 63 65 6c 6c
6f 72 20 6f 6e 20 62 72  69 6e 6b 20 6f 66 20 73
65 63 6f 6e 64 20 62 61  69 6c 6f 75 74 20 66 6f
72 20 62 61 6e 6b 73 ff  ff ff ff 01 00 f2 05 2a
01 00 00 00 43 41 04 67  8a fd b0 fe 55 48 27 19
67 f1 a6 71 30 b7 10 5c  d6 a8 28 e0 39 09 a6 79
62 e0 ea 1f 61 de b6 49  f6 bc 3f 4c ef 38 c4 f3
55 04 e5 1e c1 12 de 5c  38 4d f7 ba 0b 8d 57 8a
4c 70 2b 6b f1 1d 5f ac  00 00 00 00 .. .. .. ..
```

Signature Script of Coinbase Transaction

> hexdump -C -s 131 -n 77 blk00000.dat

```
.......EThe Time
s 03/Jan/2009 Ch
ancellor on brin
k of second bail
out for banks
```

Transaction Fields:

Version Number

The first 4 bytes of every transaction in a block contain transaction version number. It is in little-endian order.

Input Transaction Count

These bytes indicate the number of inputs in this transaction. It is a variable length integer.

Previous Transaction Hash

These 32 bytes contain the TXID of the transaction holding the output to spend. It is in internal byte order.

Transaction Index

These 4 bytes contain the output index number of the specific output to spend from the transaction. It is in little-endian order.

Script Size

These bytes contain the number of bytes in the script. It is a variable length integer.

Signature Script

These bytes contain a script which satisfies the conditions placed in the outpoint's Pub-Key script.

Sequence Number

These 4 bytes contain 0xffffffff.

Output Transaction Count

These bytes indicate the number of outputs in this transaction. It is a variable length integer.

Satoshi Amount

These 8 bytes contain the amount of Satoshi to spend. It is in little-endian order.

Pub-Key Script

These bytes define the conditions which must be satisfied to spend this output.

Lock Time

These 4 bytes contain a Unix epoch time or block number. It is in little-endian order.

API Documentation:

Testing Base URL: "http://127.0.0.1:9000"
Method: GET

Block Header API

Request block header of the block.

Endpoint: "/blockheader"

Parameters:
$block_hash: 256bit hash of block header

Full URL:
http://[HOST]:[PORT]/blockheader?[BLOCK_HASH]

Success Response:
200 OK, application/json

```
{
    "version":     <block version number>,
    "prev_block":  <hash of previous block header>,
    "mrkl_root":   <hash of all transactions in the block>,
    "time":        <time when miner started hashing header>,
    "bits":        <target threshold for block hash>,
    "nonce":       <arbitrary number to modify block hash>
}
```

Block Transactions API

Request all transactions of the block.

Endpoint: "/blocktransactions"

Parameters:
$block_hash: 256bit hash of block header

Full URL:
http://[HOST]:[PORT]/blocktransactions?[BLOCK_HASH]

Success Response:
200 OK, application/json

```
{
    "tx_count":  <number of transactions>,
    "transactions":
    [ {
        "tx_hash": <256bit hash of transaction>
        "value":   <BTC amount of transaction>
    }, ... ]
}
```

Block Height API

Request block height of the block.

Endpoint: "/blockheight"

Parameters:

$block_hash: 256bit hash of block header

Full URL:

http://[HOST]:[PORT]/blockheight?[BLOCK_HASH]

Success Response:

200 OK, application/json

```
{
    "height":  <Number of blocks since genesis block>
}
```

Main Chain API

Verify the block is in the longest chain.

Endpoint: "/mainchain"

Parameters:

$block_hash: 256bit hash of block header

Full URL:

http://[HOST]:[PORT]/mainchain?[BLOCK_HASH]

Success Response:

200 OK, application/json

```
{
    "main_chain":  <true/false>
}
```

Latest Block API

Request latest block hash in the longest chain.

Endpoint: "/latestblock"

Parameters:

NONE

Full URL:

http://[HOST]:[PORT]/latestblock

Success Response:
200 OK, application/json

{
"hash": <Latest block hash in main chain>
}

Latest Height API

Request current block height in the longest chain.

Endpoint: "/latestheight"

Parameters:
NONE

Full URL:
http://[HOST]:[PORT]/latestheight

Success Response:
200 OK, application/json

{
"height": <Block height of main chain>
}

Transaction Information API

Request information of the transaction.

Endpoint: "/transactioninfo"

Parameters:
$tx_hash: 256bit hash of transaction

Full URL:
http://[HOST]:[PORT]/transactioninfo?[TX_HASH]

Success Response:
200 OK, application/json

{
"block_hash": <256bit hash of block header>
"version": <transaction version number>
"input_tx_count": <number of input transactions>
"output_tx_count": <number of output transactions>
"value": <BTC amount of transaction>
"lock_time": <lock time>
}

Transaction Inputs API

Request input transactions of the transaction.

Endpoint: "/transactioninputs"

Parameters:

$tx_hash: 256bit hash of transaction

Full URL:

http://[HOST]:[PORT]/transactioninputs?[TX_HASH]

Success Response:

200 OK, application/json

```
{
    "input_tx_count":  <number of input transactions>,
    "input_transactions":
    [ {
        "prev_hash":    <256bit hash of previous transaction>
        "sig_script":   <pubkey signature script>
        "seq_num":      <sequence number>
    }, ... ]
}
```

Transaction Outputs API

Request output transactions of the transaction.

Endpoint: "/transactionoutputs"

Parameters:

$tx_hash: 256bit hash of transaction

Full URL:

http://[HOST]:[PORT]/transactionoutputs?[TX_HASH]

Success Response:

200 OK, application/json

```
{
    "output_tx_count":  <number of output transactions>,
    "output_transactions":
    [ {
        "value":       <BTC amount of output transaction>
        "sig_script":  <pubkey signature script>
    }, ... ]
}
```

Implementation:

Before running our server, we have to first run Bitcoin full node to retrieve the latest complete raw Bitcoin blockchain files. They are named blkXXXXX.dat and blk00000.dat is the first file of the raw blockchain. These files are usually located in ~/.bitcoin/blocks/. After downloading those raw blockchain files, we can start our server and run it localhost to listen for HTTP connections on port 9000 for testing purpose. Then, our server should be ready to handle GET requests of various endpoints following the API format of above documentation and respond queries in JSON format.

The absolute path of the directory that holds those raw Bitcoin blockchain files has to be inputted to our server. After starting our server, the setup stage is initiated before our server forks worker threads to handle incoming HTTP connections. The setup stage loads and parses each blockchain file to construct a global block structure hash table, in which each block's previous block hash is the key. Therefore, it is now possible to traverse the blockchain starting from the genesis block, as our server can compute next block's block hash from the block header stored in the hash table. Note that the previous block hash of the genesis block is all zero bytes, and so our server can easily fetch the genesis block from hash table with zero bytes hex string hash.

To compute the block hash of a block, which is the block's block header hash, our server stores version number, previous block hash, Merkle root hash, time, nBits and nonce of each block. These fields are concatenated to run SHA-256 hashing twice to compute the block hash. With the block hash of each block, our server can identify next block using the global hash table. So, same with runtime of the hash table, our server can answer queries like version number, previous block hash, Merkle root hash, time, nBits and nonce of any block in O(1) runtime.

To compute the transaction hash of a transaction, our server stores transaction version number, input transaction count, input transactions, output transaction count, output transactions and lock time of each transaction. These fields are concatenated to run SHA-256 hashing twice to compute the transaction hash. With the transaction hash of each transaction computed and stored, our server can fetch all transaction hashes of a block using the global block hash table. So, same with runtime of the hash table, our server can answer related queries like transaction version number, input transaction count, input transactions, output transaction count, output transactions and lock time of any transaction in O(1) runtime.

For all transactions parsed in each block, our server runs Merkle tree hashing algorithm with all the computed transaction hashes to verify that the Merkle root hash in block header is correct. The bottom-up recursive hashing algorithm runs on every level of the Merkle tree from the bottom leaf hashes. Each pair of child hashes are concatenated to run SHA-256 hashing twice to compute the parent hash. If a level of Merkle tree has an odd number of hashes, the last hash is replicated and appended to the end. The algorithm terminates only when a single hash is left in the tree, which is the Merkle root hash.

Next, our server runs breadth first search from the genesis block to compute the distance of each block from genesis block. So, knowing the distance of each block, our server can identify the longest path from genesis block, which is the main chain of Bitcoin blockchain. This breadth first search pre-processing computation has O(V + E) runtime, so that our server can answer queries like whether a block is in main chain and what the latest block is in main chain in O(1) runtime. Since block height is just the number of blocks counting from the genesis block, our server can answer related queries with computed breadth first search distances of blocks, like what height a block is and what the latest block height is in main chain in O(1) runtime. Note that orphan blocks now have negative distance after breadth first search, since they are disjointed from the genesis block. So, our server can answer query to identify orphan blocks as well. Apparently, after the setup stage, all related queries should be answered by our server in O(1) runtime.

For simplicity of the project, our server uses the built-in Python dictionary for the block structure hash table. As Bitcoin blockchain continues to grow, it is a valid to concern that the built-in Python dictionary type cannot hold all the required blockchain information in memory and get a MemoryError exception in Python runtime. Therefore, a remote in-memory database like Redis could be used instead with the same key-value hash table format if needed.

Source:

https://bitcoin.org/en/developer-reference

https://en.bitcoin.it/wiki/Block_hashing_algorithm