BlockfinBFT Database Schema — //STORE

** Draft Specification. Subject to change **

Introduction

This document describes the database schema for the //STORE blockchain. A relational database is assumed for strict schema enforcement. Since blockchain is immutable, we use append-only design pattern for database tables also. Almost all the tables in the database are immutable — the records once inserted can never be modified or deleted. The immutability ensures simple design, both at the database and application layers.

We assume Postgres database for its NoSQL capability [1], but the schema is (generally) independent of specific database.

History

Version	Description	Date
0.1	Data modeled with LevelDB, a popular key/value database. The progressive nature of BlockfinBFT consensus and multiple relationships among the data made LevelDB less suitable for this job.	Sept. 2018
0.2	Experimented with LMDB for its speed and embeddability. It lacks support for replication. Although there are ways to achieve this, the effort was deemed impractical, given limited engineering resources.	Dec. 2018
0.3	With a desire for strong schema enforcement, we started modeling the data with a relational database. Rich set of reporting tools are also available for relational databases. The implementation efficiency needs to be validated.	Feb. 2019

Schema

The schema is versioned in the schema name itself for clarity. The initial version is 1.0. A "major_minor" ("1_0") schema numbering convention is used. The schema is owned by a specific user, **blockfin**.

```
CREATE SCHEMA blockfin_v1_0 AUTHORIZATION blockfin
```

Tablespace

A separate tablespace is defined for the blockchain database. The tablespace is owned by the same user, who owns the schema. In fact, all database operations are performed with the privileges given to this user account. The tablespace is versioned so as to maintain the data for multiple versions, if needed.

```
CREATE TABLESPACE blockfin_v1_0 OWNER blockfin LOCATION '/data/blockfin_v1_0';
```

Database

The main database for the blockchain is defined as follows.

```
CREATE DATABASE blockfin OWNER blockfin TABLESPACE blockfin_v1_0;
```

Tables

As described in the introduction, almost all the tables in the database are immutable. However, there are a few exceptions arising from the fact that BlockfinBFT is an asynchronous, multi-stage consensus algorithm. The blocks in the blockchain are *built* over many stages. So, the data for each stage is stored in immutable tables whereas some index tables are updated as the stages progress.

An immutable table (a better name is, *append-only* table) is a table where the *row* cannot be updated or deleted after it is inserted. In other words, the table only allows `INSERT` action.

Immutability on tables is achieved by revoking the `UPDATE` and `DELETE` privileges on the respective tables. For example:

```
REVOKE UPDATE DELETE TRUNCATE ON <table name> FROM blockfin;
```

Note that this approach doesn't prevent superusers from performing revoked tasks. The idea is to prevent such actions in the application flow, where the queries are executed with the privileges granted to the user, `blockfin`. So, even if there is a bug in the application code where an update or delete action is performed on immutable tables, the action is rejected at the database level.

For the rest of this document, we only flag whether the table is immutable or not. We don't explicitly repeat the `REVOKE` statement discussed above.

Block table

The `block` table models individual blocks of //STORE blockchain. This table is not immutable as the block is built progressively by the BlockfinBFT consensus algorithm. This table contains mostly *pointers* to the data in other (immutable) tables. Each row in this table represents a single block. At any time, the row can be examined to learn about the progress of block assembly and validation process.

Each block is identified by a sequence number. A `SEQUENCE` is created to produce the next block id.

```
CREATE SEQUENCE blockfin_v1_0.block_id_seq;
```

Each block has a block type, which describes the type of the block. The types can be as follows:

```
CREATE TYPE block_type AS ENUM ('genesis', 'identity', 'transaction');
```

where:

- **genesis** — Genesis block. Structurally similar to identity block described below. This type specifically identifies block 0.
- **identity** — The block contains the identity information about the miners. This information is used to verify the signatures of the miners in the transaction blocks. All transaction blocks following the identity block use the miner identities described in this block. An identity block doesn't contain any transactions.
- **transaction** — The block containing transactions. This is the most common block type. This block also contains the miner signatures as they build and validate the block.

The `block` table has `block_id` as its primary key.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.BLOCK(
 /* A sequenced block id */
 block_id INTEGER PRIMARY KEY DEFAULT nextval('blockfin_v1_0.block_id_seq'),
```

```
 /* Block type, identifying the type of the block */
 block_type block_type,

 /* The pointer to the block header created at the time of block creation.
  * This references the block_hash column in the block_creation_header table. So,
  * the value for this column cannot exist without the block header created first.
  */
 create_header VARCHAR(32) REFERENCES blockfin_v1_0.BLOCK_CREATION_HEADER(block_hash),

 /* The pointer to the block header created at the time of block assembly.
  * This references the block_hash column in the block_assembly_header table.
  */
 assembly_header VARCHAR(32) REFERENCES blockfin_v1_0.
                           BLOCK_ASSEMBLY_HEADER(block_hash),

 /* The pointer to the block header created at the time of block validation.
  * This references the block_hash column in the block_validation_header table.
  */
 validate_header VARCHAR(32) REFERENCES blockfin_v1_0.
                           BLOCK_VALIDATION_HEADER(block_hash),

 /* The pointer to the block header created at the time of block sealing phase.
  * This references the block_hash column in the block_validation_header table.
  */
 seal_header VARCHAR(32) REFERENCES blockfin_v1_0.
                           BLOCK_SEALING_HEADER(block_hash),

 /* The pointer to the transaction batches included in this block. */
 transaction_batches VARCHAR(32) REFERENCES blockfin_v1_0.
                           TRANSACTION_BATCHES(batch_hash),

 /* The pointer to the Storage miner signatures added during block creation. */
 create_signatures VARCHAR(32) REFERENCES blockfin_v1_0.
                           CREATE_SIGNATURES(signature_hash),

 /* The pointer to the Validation miner signatures added during block validation. */
 validate_signatures VARCHAR(32) REFERENCES blockfin_v1_0.
                           VALIDATE_SIGNATURES(signature_hash),

 /* The pointer to the sealing signatures added during block sealing phase. */
 sealing_signatures VARCHAR(32) REFERENCES blockfin_v1_0.
                           SEALING_SIGNATURES(signature_hash)

) TABLESPACE blockfin_v1_0;
```

This table essentially consists of references to other tables. The columns are populated as the consensus progresses through its stages.

Block_Creation_Header table

This table models the block header created at the time of block creation. Unlike the blocks in the traditional blockchains, which contain a single block header section, the blocks in //STORE blockchain will have multiple headers. Each header represents the data created at different stages of BlockfinBFT consensus algorithm.

The `BLOCK_CREATION_HEADER` table contains the data produced when an empty block is created and the block is added to the blockchain.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.BLOCK_CREATION_HEADER(
 block_id INTEGER PRIMARY KEY REFERENCES blockfin_v1_0.BLOCK(block_id),
 create_time TIMESTAMP NOT NULL,

 /* If the block is genesis or identity type, the identities of the miners is
  * included in the header.
  */

 identities VARCHAR(32) REFERENCES blockfin_v1_0.IDENTITIES(identity_hash),

 /* The hash created with this block header data + block_hash of BLOCK_CREATION_HEADER
  * of the previous block.
  */
 block_hash VARCHAR(32) NOT NULL
) TABLESPACE blockfin_v1_0;
```

Identities table

The `IDENTITIES` table contains miner identities. The table is described as follows.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.IDENTITIES(
 /*
  * The identities of Validation miners. This is a JSON data structure with the
following
  * format:
  * {“public key”: “Know Your Voter link on Github that proves the identity”}
  */
 validators json NOT NULL,

 /* Storage miner, backup validators and messagenode identities are similar. */
 messagenodes json NOT NULL,
 backup_validators json NOT NULL,
 backup_messagenodes json NOT NULL,

 /* The block id at which this new set of miners will take charge. */
 starting_block_id INTEGER NOT NULL,
```

```
 /* Merkle root of the identities of all miner types. The construction of this
  * field is described separately, but for the sake of this specification, it is
  * safe to assume that this acts like a hash, so every block can verify that the
  * right miners signed the blocks.
  */
 identity_hash VARCHAR(32) PRIMARY KEY NOT NULL

) TABLESPACE blockfin_v1_0;
```

Block_Assembly_Header table

This table models the block header created at the time of block assembly. The `BLOCK_ASSEMBLY_HEADER` table contains the data produced when an empty block is assembled by the Storage miners and the block is ready for validation. At this time, the transaction batches are already available.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.BLOCK_ASSEMBLY_HEADER(
 block_id INTEGER PRIMARY KEY REFERENCES blockfin_v1_0.BLOCK(block_id),
 assembly_time TIMESTAMP NOT NULL,

 /* The hash created with this block header data + block_hash of BLOCK_ASSEMBLY_HEADER
  * of the previous block.
  */
 block_hash VARCHAR(32) NOT NULL

) TABLESPACE blockfin_v1_0;
```

Block_Validation_Header table

This table models the block header created at the time of block validation. The `BLOCK_VALIDATION_HEADER` table contains the data produced when an assembled block is validated by the validators and the block is finalized.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.BLOCK_VALIDATION_HEADER(
 block_id INTEGER PRIMARY KEY REFERENCES blockfin_v1_0.BLOCK(block_id),
 validate_time TIMESTAMP NOT NULL,

 /*
  * Transaction exceptions. In BlockfinBFT, a block is finalized even if it contains
  * invalid transactions. But such transactions are called out by the validators.
  * This list can be empty if all transactions are valid, but if this list is
  * non-empty, the specified transactions are deemed failed. Each failed transaction
  * is identified by the hash of the transaction, so the array, if not empty, will
  * contain a list of hashes.
  */
```

```
 failed_transaction VARCHAR ARRAY,

 /* The hash created with this block header data + block_hash of
  * BLOCK_VALIDATION_HEADER of previous block.
  */
 block_hash VARCHAR(32) NOT NULL

) TABLESPACE blockfin_v1_0;
```

Block_Sealing_Header table

This table models the block header created at the time of block sealing. The `BLOCK_SEALING_HEADER` table contains the data produced when a finalized block is sealed by dGuards. The early releases of //STORE blockchain won't have dGuards and hence there is no sealing phase, but this table is defined for completeness.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.BLOCK_SEALING_HEADER(
 block_id INTEGER PRIMARY KEY REFERENCES blockfin_v1_0.BLOCK(block_id),
 sealing_time TIMESTAMP NOT NULL,

 /* The hash created with this block header data + block_hash of
  * BLOCK_SEALING_HEADER of previous block.
  */
 block_hash VARCHAR(32) NOT NULL

) TABLESPACE blockfin_v1_0;
```

Transactions table

The `TRANSACTIONS` table models the transactions submitted to Validation miners. The transactions are signed by the senders. The table is keyed with transaction hash, which is referenced everywhere else.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.TRANSACTIONS(
 /* From address. Addresses are similar to Bitcoin addresses and are 35 characters
  * in length.
  */
 from_address CHAR(36) NOT NULL,

 /* To address. */
 to_address CHAR(36) NOT NULL,

 /* Transaction nonce */
 nonce INTEGER NOT NULL,

 /* Transaction creation timestamp. This is client-provided and not interpreted. */
 timestamp TIMESTAMP NOT NULL,
```

```
 /* Transaction value in Edisons. An INTEGER is used to avoid floating point
  * related problems.
  */
 value INTEGER NOT NULL,

 /* Sender's public key. */
 public_key CHAR(44) NOT NULL,

 /* Transaction signature. */
 signature CHAR(88) NOT NULL,

 /* Transaction hash. SHA256(signature). */
 transaction_hash CHAR(32) PRIMARY KEY NOT NULL

) TABLESPACE blockfin_v1_0;
```

The transaction table can be queried for specific `from_address` and `to_address` or a combination of both. So, the following indexes are defined.

```
CREATE INDEX FROM_ADDRESS ON TRANSACTIONS (FROM_ADDRESS);
CREATE INDEX TO_ADDRESS ON TRANSACTIONS (TO_ADDRESS);
CREATE INDEX TRANSACTION ON TRANSACTIONS (FROM_ADDRESS, TO_ADDRESS);
```

With these indexes, the following types of queries can be served efficiently.

- Find all transactions originating from <sender>.
- Find all transactions sent to <receiver>.
- Find all transactions sent by <sender> to <receiver>.

Transactions_Batches table

The `TRANSACTION_BATCHES` table models the transaction batches created by the Validation miners. The validators wait for a finite window and collect the transactions received in that window before sending them to the Storage miners. The Storage miners treat a transaction batch as a single entity while assembling the block. In other words, the block contains transaction batches, instead of individual transactions.

Each transaction batch contains the signature of the Validation miner producing the batch, the hash of which serves as its primary key.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.TRANSACTION_BATCHES(
 /* Validation miner signature. */
 validator_signature CHAR(88) NOT NULL,

 /* SHA256(validator_signature). */
```

```
 batch_hash CHAR(32) PRIMARY KEY NOT NULL,

 /* The transaction hashes are referenced here. */
 transactions VARCHAR ARRAY,

 /* Transaction batch nonce. The validator increments nonce for each batch. */
 nonce INTEGER NOT NULL,

 /* Transaction batch creation timestamp. This is validator-provided
  * and not interpreted. */
 timestamp TIMESTAMP NOT NULL,

 /* Validation miner's public key. */
 public_key CHAR(44) NOT NULL

) TABLESPACE blockfin_v1_0;
```

Create_Signatures table

The `CREATE_SIGNATURES` table models the Storage miner signatures created by each Storage miner as it attests the newly created block. Since each Storage miner attests the block asynchronously, each row represents the signature of one Storage miner for a given block.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.CREATE_SIGNATURES(

 /* The block for which signatures are produced.
 block_id INTEGER PRIMARY KEY REFERENCES blockfin_v1_0.BLOCK(block_id),

 /* Storage miner signature. */
 messagenode_signature CHAR(88) NOT NULL,

 /* Storage miner's public key. */
 public_key CHAR(44) NOT NULL

) TABLESPACE blockfin_v1_0;
```

Assemble_Signatures table

The `ASSEMBLE_SIGNATURES` table models the Storage miner signatures created by each Storage miner as it assembles transaction batches into an empty block. Since each Storage miner attests the block asynchronously, each row represents the signature of one Storage miner for a given block.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.ASSEMBLE_SIGNATURES(

 /* The block for which signatures are produced.
 block_id INTEGER PRIMARY KEY REFERENCES blockfin_v1_0.BLOCK(block_id),
```

```
 /* Storage miner signature. */
 messagenode_signature CHAR(88) NOT NULL,

 /* Storage miner's public key. */
 public_key CHAR(44) NOT NULL

) TABLESPACE blockfin_v1_0;
```

Validate_Signatures table

The `VALIDATE_SIGNATURES` table models the Validation miner signatures created by each Validation miner as it validates the blocks and finalizes it. Since each Validation miner validates the block asynchronously, each row represents the signature of one Validation miner for a given block.

```
CREATE IF NOT EXISTS TABLE blockfin_v1_0.VALIDATE_SIGNATURES(

 /* The block for which signatures are produced.
 block_id INTEGER PRIMARY KEY REFERENCES blockfin_v1_0.BLOCK(block_id),

 /* Validation miner signature. */
 validator_signature CHAR(88) NOT NULL,

 /* Validation miner's public key. */
 public_key CHAR(44) NOT NULL

) TABLESPACE blockfin_v1_0;
```

Next steps

1. Create a .sql file with the schema and SQL commands to create the tables on Postgres database.
2. Populate test data to understand indexing needs and query performance. List most common queries performed during consensus.
3. Audit the schema design with third party database experts.
4. Setup and study Postgres' synchronous replication for its efficiency. Considering that most tables are append-only, this is expected to have acceptable performance.

References

1. http://info.enterprisedb.com/rs/enterprisedb/images/EDB_White_Paper_Using_the_NoSQL_Features_in_Postgres.pdf.