ether_sql Documentation

Release 0.0.1

Ankit CHiplunkar

May 28, 2020

Contents:

ether_sql is a Python library to push ethereum blockchain data into an sql database.

It is maintained by Analyse Ether, with the goal of making Ethereum data easily available to everyone. This library can be used as a backbone for creating block explorers or performing data analysis.

CHAPTER 1

Goals

The main focus is to make Ethereum data easily available to everyone, while serving as a backbone for:

- Open block explorers (coming soon. . .)
- Data analysis platforms

1.1 Introduction

1.1.1 ether_sql

ether_sql is a Python library to push ethereum blockchain data into an sql database.

It is maintained by Analyse Ether, with the goal of making Ethereum data easily available to everyone. This library can be used as a backbone for creating block explorers or performing data analysis.

It is written in Python 3.5+, uses web3.py for geting data using JSON-RPC calls and uses SqlAlchemy to connect to a postgressql database.

Goals

The main focus is to make Ethereum data easily available to everyone, while serving as a backbone for:

- Open block explorers (coming soon. . .)
- Data analysis platforms

Buidl Status

This is currently in very alpha stage, and not recommended for production use until it has received sufficient testing. Currently supports Geth, Infura and Parity node, but transaction traces (eg. internal transactions) are currently available only with Parity node.

Follow along the Installation to install the basic setup and checkout the Guides to understand the process.

1.2 Installation Guide

1.2.1 Linux dependencies

Install postgresql as database:

```
$ sudo apt-get install postgresql
```

Install python3 headers:

```
$ sudo apt-get install python3-pip
$ sudo apt-get install python3.6-dev
```

Install redis server:

```
$ sudo apt-get install redis-server
```

Install Rabbit-MQ server:

```
$ sudo apt-get install rabbitmq-server
```

1.2.2 Python dependencies

Clone the **ether_sql** library:

```
$ git clone https://github.com/analyseether/ether_sql.git
$ cd ether_sql
```

Create and activate a virtual environment:

```
$ virtualenv envname
$ source envname\bin\activate
```

Install python libraries:

```
$ pip install -e . -r requirements.txt
```

1.2.3 Database setup

Create a new psql user, this prompts for a user password, use the same password in the variable `SQLALCHEMY_PASSWORD` of the **settings.py** file:

```
$ sudo -u postgres createuser -s -P -e $USER
```

Create the ether_sql database in psql:

```
$ createdb ether_sql
```

We use Alembic to manage tables, you can create the tables by using this command:

```
$ ether_sql sql upgrade_tables
```

1.2.4 Setting up RabbitMQ

To use Celery we need to create a RabbitMQ user, a virtual host and allow that user access to that virtual host:

```
$ sudo rabbitmqctl add_user myuser mypassword

$ sudo rabbitmqctl add_vhost myvhost

$ sudo rabbitmqctl set_user_tags myuser mytag

$ sudo rabbitmqctl set_permissions -p myvhost myuser ".*" ".*" ".*"
```

Substitute in appropriate values for myuser, mypassword and myvhost above and in the settings file.

1.2.5 Node settings

The settings to connect to a node are set in the **settings.py** file using classes.

Infura Settings:

The class **PersonalInfuraSettings** specifies settings to connect to a normal Infura node. You can fill in the value of your API token on **NODE_API_TOKEN**:

```
class PersonalInfuraSettings(DefaultSettings):
    NODE_TYPE = "Infura"
    NODE_API_TOKEN = ""  # your infura api_token
    NODE_URL = 'https://mainnet.infura.io/{}'.format(NODE_API_TOKEN)
```

Local Node settings:

We use the automatic methods in **web3.py** to connect to a node, if a local node is available then only the **NODE_TYPE** is required. The class **PersonalParitySettings** is used to connect to a local Parity node:

```
class PersonalParitySettings(DefaultSettings):
    NODE_TYPE = "Parity"
    # Use this option to parse traces, needs parity with cli --tracing=on
    PARSE_TRACE = True
```

Whereas, the class **PersonalGethSettings** is used to connect to a local Geth node:

```
class PersonalGethSettings(DefaultSettings):
    NODE_TYPE = "Geth"
```

1.2.6 Syncing data

ether_sql has several built in cli commands to facilitate scraping data. To start the sync just type:

```
$ ether_sql scrape_block_range
```

This will by default start pushing the data from an Infura node to the psql database. To switch nodes use the settings flag:

```
$ ether_sql --settings='PersonalParitySettings' scrape_block_range
```

To access other Command Line Interfaces (CLI) checkout the CLI's.

1.3 Guide's

This section aims to provide basic guides on how to make most use of the library.

1.3.1 Quickstart

This quickstart is a follow up of the Installation instructions.

Syncing the blockchain

The easiest method to start syncing the sql database to the connected node is using the following command.

```
$ ether_sql scrape_block_range
```

This command will check the last block number in your sql database and node and start pushing the remaining blocks into your sql server. To sync blocks in a particular range use the options `--start_block_number` or `--end_block_number` or use the `--help` option to know more about the above command.

```
$ ether_sql scrape_block_range --help
```

Current progress

To get the current status of sync progress you can use the following command to get the highest block number in the sql.

```
$ ether_sql sql blocknumber
```

For more details refer to the API doc on CLI's.

Connecting to Postgresql

Once the database is filled with some blocks you can connect to the psql database using the following command.

```
$ psql ether_sql
```

Once connected to the Postgresql you can start quickly querying the database. Below is a simple code to get the maximum block number in the sql database.

```
ether_sql=# SELECT max(block_number) from blocks;
```

More sample sql examples and their results are available in sql examples. TO know more details about the different tables and their columns refer to sql table api docs.

1.3.2 Syncing the blockchain

The easiest method to start syncing the sql database to the connected node is using the following command.

```
$ ether_sql --settings=YourSettings scrape_block_range
```

The above command picks up node and database settings from `YourSettings`. Then it checks the last block number in sql database and node and start pushing the missing blocks into the database. To sync blocks in a particular range use the options `--start_block_number` or `--end_block_number` or use the `--help` option to know more about the above command. options

```
$ ether_sql scrape_block_range --help
```

Using multiple workers

Syncing the whole blockchain in series would take several months. Hence, to speed up the process we provide options to achieve this task in parallel. We use RabbitMQ or Redash to maintain the queue of blocks to be pushed in the database.

The following command uses the node, database and queue settings provided in `YourSettings` and starts pushing required blocks in the queue.

```
$ ether_sql --settings=YourSettings scrape_block_range --mode=parallel
```

We can then start multiple workers using the following command.

```
$ ether_sql --settings=YourSettings celery start -c4
```

The above command will start 4 workers using the provided settings. Here is a demo of the process: https://www.youtube.com/watch?v=rnkfyAgGJwI&feature=youtu.be where we push first 10k blocks in 30 seconds using 10 workers.

Following the block-chain head

A new block gets added in the ethereum blockchain every 15 seconds. It would be very beneficial if we can keep syncing the database with the blockchain in the backend. This is achieved by running two celery queues, the first queue periodically searches for newly added blocks and pushes them in the second queue, the second queue fetches the block data from the node and pushes it into the database.

The following command starts the periodic queue called `celery_filters`.

```
$ ether_sql --settings=YourSettings celery start -c1 -B -Q celery_filters
```

We scan for new blocks every 30 seconds and put all the blocks which are older than `YourSettings.BLOCK_LAG` into the main queue.

The second queue which pushes data into the database can be started using the following command.

```
$ ether_sql --settings=YourSettings celery start -c4
```

1.3.3 SQL Examples

This is a list of some basic SQL queries written with the synced database. This page is a follow up on the Quickstart page, perform the basic database sync there to start to write these queries.

Block with first transaction

```
ether_sql=# SELECT min(block_number) from blocks where transaction_count>0;
  min
 -------
  46147
 (1 row)
```

Total transactions in 100k blocks

```
ether_sql=# SELECT sum(transaction_count) from blocks where block_number < 100001;
  sum
 ---------
  26970
 (1 row)
```

Maximum transfer of value

```
ether_sql=# SELECT max(value) from transactions where block_number <100001;
  max
 ----------------------------
  11901464239480000000000000
 (1 row)
```

Note: Someone transferred 11.9 million ether!

Transaction hash of maximum value transfer

```
ether_sql=# SELECT transaction_hash from transactions where value =␣
↪11901464239480000000000000;
  transaction_hash
 --------------------------------------------------------------------
  0x9c81f44c29ff0226f835cd0a8a2f2a7eca6db52a711f8211b566fd15d3e0e8d4
 (1 row)
```

Total smart contracts in 100k blocks

```
ether_sql=# SELECT count(1) from traces where contract_address is not null and block_
↪number < 100001;
  count
 -------
  49393
 (1 row)
```

Top miners in first 100k blocks

```
ether_sql=# SELECT miner, count(*) AS num, count(1)/100000.0 AS PERCENT
ether_sql-# FROM blocks
ether_sql-# WHERE block_number<=100000
ether_sql-# GROUP BY miner
ether_sql-# ORDER BY num DESC
ether_sql-# LIMIT 15;
                     miner                    | num  |        percent
  --------------------------------------------+------+------------------------
  0xe6a7a1d47ff21b6321162aea7c6cb457d5476bca | 9735 | 0.09735000000000000000
  0xf927a40c8b7f6e07c5af7fa2155b4864a4112b13 | 8951 | 0.08951000000000000000
  0xbb7b8287f3f0a933474a79eae42cbca977791171 | 8712 | 0.08712000000000000000
  0x88d74a59454f6cf3b51ef6b9136afb6b9d405a88 | 4234 | 0.04234000000000000000
  0x9746c7e1ef2bd21ff3997fa467593a89cb852bd0 | 3475 | 0.03475000000000000000
  0xf8e0ca3ed80bd541b94bedcf259e8cf2141a9523 | 2409 | 0.02409000000000000000
  0xa50ec0d39fa913e62f1bae7074e6f36caa71855b | 1627 | 0.01627000000000000000
  0xbcb2e3693d246e1fc00348754334badeb88b2a11 | 1537 | 0.01537000000000000000
  0xeb1325c8d9d3ea8d74ac11f4b00f1b2367686319 | 1390 | 0.01390000000000000000
  0x1b7047b4338acf65be94c1a3e8c5c9338ad7d67c | 1335 | 0.01335000000000000000
  0xf2d2aff1320476cb8c6b607199d23175cc595693 | 1141 | 0.01141000000000000000
  0x47ff6576639c2e94762ea5443978d7681c0e78dc | 1131 | 0.01131000000000000000
  0xbb12b5a9b85d4ab8cde6056e9c1b2a4a337d2261 | 1102 | 0.01102000000000000000
  0x0037ce3d4b7f8729c8607d8d0248252be68202c0 |  917 | 0.00917000000000000000
  0x580992b51e3925e23280efb93d3047c82f17e038 |  874 | 0.00874000000000000000
  (15 rows)
```

1.3.4 Contributing

Thank you for your interest in contributing! Please read along to learn how to get started.

Running the tests

We create a seperate database to run our tests, so it does not interfere with the current synced database.

Use the following command to create a new database:

```
$ createdb ether_sql_tests
```

If you are testing using Infura node use the command:

```
$ python -m pytest tests/infura
```

If you are using a local parity node use the command:

```
$ python -m pytest tests/parity
```

Updating the database tables

We use Alembic to handle database migrations.

You can create new tables by adding a new class in the `ether_sql/models` module. More details on available columns are available at SQLAlchemy guides

To create SQL commands that can reflect the changes in the database, run the following command.

```
$ ether_sql sql migrate -m "message for changes"
```

Next upgrade the database using the following command:

```
$ ether_sql sql upgrade
```

Updating the docs

We suggest to create different virtual enviornment for updating the docs.

```
$ virtualenv venvdocs
$ source venvdocs/bin/activate
$ pip install -e . requirements.txt
```

We use Sphinx to automate the documentation of python modules and sphinx-click to automate building docs of click commands.

Pull Requests

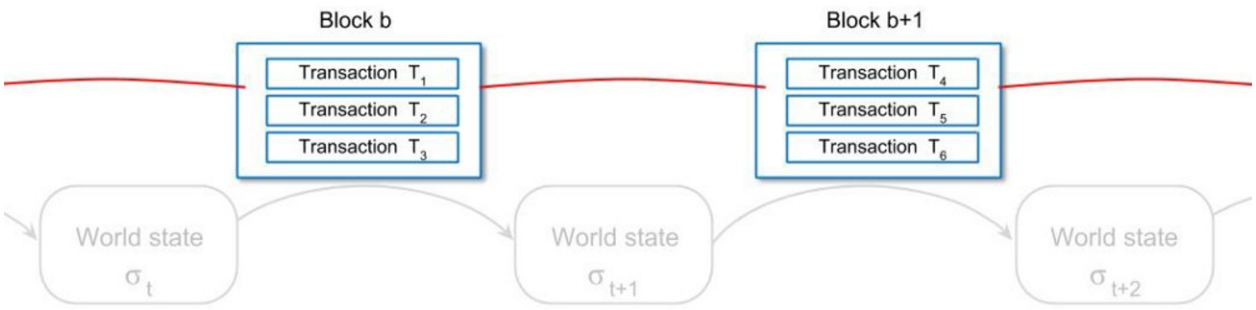
Once all the tests are passing generate a pull request and we will merge the contribution after a discussion.

1.4 API's

This section aims to provide a detailed description of all the API's in the library.

1.4.1 SQL Tables

This section aims at giving a detailed description of the psql tables in the database and their corresponding helper functions.



Blocks

As visible from the image above, a blockchain is literally a chain of blocks. A block contains a list of transactions, few features to prove the work done by a miner and a list of uncles.

class ether_sql.models.blocks.**Blocks**(***kwargs*)
    Class mapping a block table in the psql database to a block in ethereum node.

Parameters

- **block_number** (*int*) – Quantity equal to number of blocks behind the current block
- **block_hash** (*str*) – The Keccak 256-bit hash of this block
- **parent_hash** (*str*) – The Keccak 256-bit hash of the parent of this block
- **difficulty** (*int*) – Difficulty level of this block
- **gas_used** (*int*) – Total gas used by the transactions in this block
- **miner** (*str*) – Address to which all block rewards are transferred
- **timestamp** (*datetime*) – Unix time at the at this blocks inception
- **sha3uncles** (*str*) – Keccak 256-bit hash of the uncles portion of this block
- **extra_data** (*str*) – Byte array of 32 bytes or less containing extra data of this block
- **gas_limit** (*int*) – Current maximum gas expenditure per block
- **uncle_count** (*int*) – Number of uncles in this block
- **transaction_count** (*int*) – Number of transactions in this block

classmethod add_block(*block_data*, *iso_timestamp*)

Creates a new block object from data received from JSON-RPC call eth_getBlockByNumber.

Parameters

- **block_data** (*dict*) – data received from the JSON RPC call
- **iso_timestamp** (*datetime*) – timestamp when the block was mined

classmethod missing_blocks(*max_block_number=None*)

Return missing blocks in the blocks table between 0 to block_number

Parameters **max_block_number** (*int*) – Maximum block number that we want to find missing blocks from

Transactions

A transaction is the basic method for Ethereum accounts to interact with each other. The transaction is a single cryptographically signed instruction sent to the Ethereum network and has the capacity to change the world state.

class ether_sql.models.transactions.**Transactions**(***kwargs*)

Class mapping a transaction table in the psql database to a transaction in ethereum node.

Parameters

- **transaction_hash** (*str*) – The Keccak 256-bit hash of this transaction
- **block_number** (*int*) – Number of the block containing this transaction
- **nonce** (*int*) – Number of transactions sent by this sender
- **sender** (*str*) – Address of account which initiated this transaction
- **start_gas** (*int*) – Maximum amount of gas to be used while executing this transaction
- **value_wei** (*int*) – Number of wei to be transferred to the receiver of this transaction
- **receiver** (*str*) – Address of the recepient of this transaction, null if transaction creates a smart-contract

- **data** (*str*) – Unlimited size text specifying input data of message call or code of a contract create
- **gas_price** (*int*) – Number of wei to pay the miner per unit of gas
- **timestamp** (*int*) – Unix time at the at this transactions blocks
- **transaction_index** (*datetime*) – Position of this transaction in the transaction list of this block

classmethod add_transaction(*transaction_data*, *block_number*, *iso_timestamp*)
Creates a new transaction object from data received from JSON-RPC call eth_getBlockByNumber.

Parameters

- **transaction_data** (*dict*) – data received from JSON RPC call
- **iso_timestamp** (*datetime*) – timestamp when the block containing the transaction was mined
- **block_number** (*int*) – block number of the block where this transaction was included

Uncles

Due to ethereum block-chains fast block propagation time (~15 seconds), the probability of a block with sufficient proof-of-work becoming stale becomes quite high. This reduces the security and miner decentralization of block-chain. To rectify this issue ethereum proposes a modified-GHOST protocol by including and rewarding uncles (ommers) or stale blocks not included in the blockchain.

class ether_sql.models.uncles.**Uncles**(***kwargs*)
Class mapping an uncle table in the psql database to an uncle (ommer) in ethereum node.

Parameters

- **uncle_hash** (*str*) – The Keccak 256-bit hash of this uncle
- **uncle_blocknumber** (*int*) – Number of blocks behind this uncle
- **parent_hash** (*str*) – The Keccak 256-bit hash of the parent of this uncle
- **difficulty** (*int*) – Difficulty level of this block
- **current_blocknumber** (*int*) – Block number where this uncle was included
- **gas_used** (*int*) – Total gas used by the transactions in this uncle
- **miner** (*str*) – Address of account where all corresponding uncle rewards are transferred
- **timestamp** (*datetime*) – Unix time at the at this uncles inception
- **sha3uncles** (*str*) – Keccak 256-bit hash of the uncles portion of this uncle
- **extra_data** (*str*) – Byte array of 32 bytes or less containing extra data of this block
- **gas_limit** (*int*) – Current maximum gas expenditure per block

classmethod add_uncle(*uncle_data*, *block_number*, *iso_timestamp*)
Creates a new block object from data received from JSON-RPC call eth_getUncleByBlockNumberAndIndex.

Parameters

- **uncle_data** (*dict*) – uncle data received from JSON RPC call
- **block_number** (*int*) – block number where this uncle was included

- **iso_timestamp** (*datetime*) – timestamp when the block was mined

Receipts

Receipts information concerning the execution of a transaction in the block-chain. They can be useful to form a zero-knowledge proof, index and search, and debug transactions. The status column was included after the Byzantinium hardfork.

class ether_sql.models.receipts.**Receipts**(***kwargs*)

Class mapping a log table in the psql database to a log in ethereum node.

Parameters

- **transaction_hash** (*str*) – The Keccak 256-bit hash of this transaction
- **status** (*bool*) – Success or failure of this transaction, included after the Byzantinium fork
- **gas_used** (*int*) – Amount of gas used by this specific transaction alone
- **cumulative_gas_used** (*int*) – Total amount of gas used after this transaction was included in the block
- **contract_address** (*str*) – Contract address create if transaction was a contract create transaction, else null
- **block_number** (*int*) – Number of the block containing this transaction
- **timestamp** (*datetime*) – Unix time at the at this transactions blocks
- **transaction_index** (*int*) – Position of this transaction in the transaction list of this block

classmethod add_receipt(*receipt_data*, *block_number*, *timestamp*)

Creates a new receipt object from data received from JSON-RPC call eth_getTransactionReceipt.

Parameters

- **receipt_data** (*dict*) – receipt data received from the JSON RPC callable
- **timestamp** (*int*) – timestamp of the block where this transaction was included
- **block_number** (*int*) – block number of the block where this transaction was included

Logs

The logs table contains the logs which were accrued during the execution of the the transaction, they are helpful in deciphering smart-contract executions or message calls.

class ether_sql.models.logs.**Logs**(***kwargs*)

Class mapping a log table in the psql database to a log in ethereum node.

Parameters

- **transaction_hash** (*str*) – Hash of the transaction which created this log
- **address** (*str*) – Address from which this log originated
- **data** (*str*) – Contains one or more 32 Bytes non-indexed arguelents of the log
- **block_number** (*int*) – The block number where this transaction was included
- **timestamp** (*datetime*) – Timestamp when the block was mined

- **transaction_index** (*int*) – Position of the transaction in the block
- **log_index** (*int*) – Position of the log in the block
- **topics_count** (*int*) – Total number of topics in this log
- **topic_1** (*str*) – First topic in the log
- **topic_2** (*str*) – Second topic in the log
- **topic_3** (*str*) – Third topic in the log
- **topic_4** (*str*) – Fourth topic in the log

classmethod add_log(*log_data*, *block_number*, *iso_timestamp*)
Creates a new log object from data received from JSON-RPC call eth_getTransactionReceipt.

Parameters

- **log_data** (*dict*) – data received from receipt JSON RPC call
- **block_number** (*int*) – block number of the block containing the log
- **iso_timestamp** (*datetime*) – timestamp when the block containing the transaction was mined

classmethod add_log_list(*current_session*, *log_list*, *block_number*, *timestamp*)
Adds a list of logs in the session

Traces

The trace module is for getting a deeper insight into transaction processing, can be used to debugging transactions and also access the internal transactions which are not included in a block.

class ether_sql.models.traces.**Traces**(***kwargs*)
Class mapping a traces table in the psql database to a trace in ethereum node.

Parameters

- **block_number** (*int*) – Number of the block containing this trace
- **transaction_hash** (*str*) – The of the transaction containing this trace
- **trace_type** (*str*) – Type of trace available types; ‘call’, ‘create’, ‘suicide’ and ‘reward’
- **trace_address** (*str*) – Array of integers specifying the address of the trace in this transaction
- **subtraces** (*int*) – Number of subsequent traces
- **transaction_index** (*int*) – Position of the transaction in this block
- **sender** (*str*) – Address of account which initiated this trace
- **receiver** (*str*) – Address of recepient of this trace, null for trace_type = ‘create’ or ‘suicide’
- **value** (*int*) – Number of wei to be transferred to the receiver of this trace
- **start_gas** (*int*) – Maximum amount of gas to be used while executing this trace
- **input_data** (*str*) – Unlimited size text specifying input data of message call or code of a contract create
- **gas_used** (*int*) – The amount of gas utilized by this step

- **contract_address** (*str*) – Address of created contract if trace_type = 'create' else null
- **output** (*str*) – Output of this trace
- **error** (*str*) – Error message if this step resulted in an error

Note: This needs proper documentation from team parity

classmethod add_trace(*dict_trace*, *transaction_hash*, *transaction_index*, *block_number*, *timestamp*)

Creates a new trace object from data received from JSON-RPC call trace_transaction.

Parameters

- **dict_trace** (*dict*) – trace data received from the JSON RPC callable
- **timestamp** (*int*) – timestamp of the block where this trace was included
- **block_number** (*int*) – block number of the block where this trance was included

classmethod add_trace_list(*current_session*, *trace_list*, *transaction_hash*, *transaction_index*, *block_number*, *timestamp*)

Adds a list of traces in the sql session

StateDiff

The state diff table contains information about the change in state after each transaction or block

class ether_sql.models.state_diff.**StateDiff**(***kwargs*)

Class mapping a state_diff table in psql to a difference in state after transactions

Parameters

- **block_number** (*int*) – Number of the block containing this StateDiff
- **timestamp** (*timestamp*) – Unix time at the mining of this block
- **transaction_hash** (*str*) – The transaction hash if this was created by a transaction
- **transaction_index** (*int*) – Position of this transaction in the transaction list of the block
- **address** (*str*) – Account address where the change occoured
- **balance_diff** (*int*) – Difference in balance due to this row
- **nonce_diff** (*int*) – Difference in nonce due to this row
- **code_from** (*str*) – Initial code of this account
- **code_to** (*str*) – Final code of this account

classmethod add_mining_rewards(*current_session*, *block*, *uncle_list*)

Adds the mining and uncle rewards to the state_diff table

classmethod add_state_diff(*balance_diff*, *nonce_diff*, *code_from*, *code_to*, *address*, *transaction_hash*, *transaction_index*, *block_number*, *timestamp*, *miner=None*, *fees=None*, *state_diff_type=None*)

Creates a new state_diff object

classmethod add_state_diff_dict(*current_session*, *state_diff_dict*, *transaction_hash*, *transaction_index*, *block_number*, *timestamp*, *miner*, *fees*)
Creates a bunch of state_diff objects upon receiving them as a dictionary and adds them to the current db_session

StorageDiff

The storage diff table contains information about the change in storage after each contract execution

class `ether_sql.models.storage_diff.`**StorageDiff**(***kwargs*)
Class mapping the storage_diff table in psql to difference in storage due to transactions

Parameters

- **block_number** (*int*) – Number of the block containing this StorageDiff
- **timestamp** (*timestamp*) – Unix time at the mining of this block
- **transaction_hash** (*str*) – The transaction hash if this was created by a transaction
- **transaction_index** (*int*) – Position of this transaction in the transaction list of the block
- **state_diff_id** (*int*) – Id in state_diff table which caused this change in storage
- **address** (*str*) – Contract address where the change occoured
- **position** (*str*) – Position in the contract address where this change occoured
- **storage_from** (*str*) – Initial value of the storage
- **storage_to** (*str*) – Final value of the storage

classmethod add_storage_diff(*storage_diff_row*, *position*, *address*, *transaction_hash*, *transaction_index*, *block_number*, *timestamp*, *state_diff_id*)
Creates a new storage_diff object

classmethod add_storage_diff_dict(*current_session*, *storage_diff_dict*, *address*, *transaction_hash*, *transaction_index*, *block_number*, *timestamp*, *state_diff_id*)
Creates a bunch of storage_diff objects upon receiving them as a dictionary and adds them to the current db_session

1.4.2 Command Line Interface (CLI)

ether_sql has several built in CLI commands to interact with the node and sql table. This section aims at detailing the various cli options available in the library in detail.

We use the Click library to generate CLI groups and their nested commands in a tree structure.

Group: ether_sql

ether_sql

`ether_sql` is the most basic CLI group with 4 subsequent commands.

```
ether_sql [OPTIONS] COMMAND [ARGS]...
```

Options

--settings <settings>
Settings to run ether_sql, choose from DefaultSettings, ParityTestSettings, PersonalGethSettings, PersonalInfuraSettings, PersonalParitySettings, TestSettings [default: DefaultSettings]

Commands

celery
Manages the celery workers (start and stop…

ether
Manages the ether node (query the node).

scrape_block
Pushes the data at block=block_number in the…

scrape_block_range
Pushes the data between start_block_number…

sql
Manages the sql (create/drop/query tables).

SubGroup: ether_sql ether

ether_sql ether

Manages the ether node (query the node).

```
ether_sql ether [OPTIONS] COMMAND [ARGS]...
```

blocknumber

Gives the most recent block number in the ether node

```
ether_sql ether blocknumber [OPTIONS]
```

SubGroup: ether_sql sql

ether_sql sql

Manages the sql (create/drop/query tables).

```
ether_sql sql [OPTIONS] COMMAND [ARGS]...
```

blocknumber

Gives the current highest block in database

```
ether_sql sql blocknumber [OPTIONS]
```

create_tables

This is a depreceated function. Alias for *ether_sql sql upgrade_tables*

```
ether_sql sql create_tables [OPTIONS]
```

drop_tables

Alias for 'alembic downgrade base'. Downgrade to no database tables

```
ether_sql sql drop_tables [OPTIONS]
```

export_to_csv

Export the data pushed into sql as csv

```
ether_sql sql export_to_csv [OPTIONS]
```

Options

--directory <directory>
    Directory where the csv should be exported

--mode <mode>
    Choose single is using same thread or parallel if using multiple threads

migrate

Alias for 'alembic revision –autogenerate' Run this command after changing sql tables

```
ether_sql sql migrate [OPTIONS]
```

Options

-m, --message <message>
    Write a message specifying what changed

upgrade_tables

Alias for 'alembic upgrade head'. Upgrade to latest model version

```
ether_sql sql upgrade_tables [OPTIONS]
```

SubGroup: ether_sql celery

ether_sql celery

Manages the celery workers (start and stop celery).

```
ether_sql celery [OPTIONS] COMMAND [ARGS]...
```

shutdown

Stops the celery workers

```
ether_sql celery shutdown [OPTIONS]
```

start

Starts the celery workers, also allows for passing celery specific arguements.

```
ether_sql celery start [OPTIONS]
```

Options

-l, --loglevel <loglevel>
Specifies the log level for the celery workers

-c, --concurrency <concurrency>
Number of parallel workers

Command: ether_sql scrape_block_range

ether_sql scrape_block_range

Pushes the data between start_block_number and end_block_number in the database. If no values are provided, the start_block_number is the last block_number+1 in sql and end_block_number is the current block_number in node. Also checks for missing blocks and adds them to the list of required block numbers

param int start_block_number starting block number of scraping

param int end_block_number end block number of scraping

param str mode Mode of data sync 'parallel' or single

param bool fill_gaps If switched on instructs to also fill missing blocks

```
ether_sql scrape_block_range [OPTIONS]
```

Options

--start_block_number `<start_block_number>`
start block number

--end_block_number `<end_block_number>`
end block number

--mode `<mode>`
Choose single is using same thread or parallel if using multiple threads

--fill_gaps, --no-fill_gaps

Command: ether_sql scrape_block

ether_sql scrape_block

Pushes the data at block=block_number in the database

```
ether_sql scrape_block [OPTIONS]
```

Options

--block_number `<block_number>`
block number to add

CHAPTER 2

Indices and tables

- genindex
- search

Index