**The following discussion is related to the publicly shared COCO analysis pipeline code:**

The module db.py currently loads vast amounts of ETH transaction, trace, and contract data from .gz files using multiprocessing and file-based caching.

There are six different queries/interactions enabled using the db.py module:

1. get_all_txns_given_hash_list
2. get_all_txn_from_to_address_list
3. get_all_traces_from_to_address_list
4. get_all_traces_given_txn_list
5. get_creation_txn_given_contract
6. get_contracts_code

The above queries follow this workflow with each step optimised using multi-processing:

- Read the JSON / binary files from the query-specific cache folder, and create the required data objects. If all queried objects are covered using the cached data, return the data objects.
- If not, we load the entire JSON .gz (transaction/trace/contract) files and process each record to create the final data objects.
- Store the data objects that have not already been saved in the query-specific cache folder for future queries. Return the queried data objects.

There are two primary bottlenecks in the current approach:

1. Repeated, full-scan parsing of large gzip-compressed JSON files for multiple queries.
2. Lack of indexing capabilities.

Proposed workflow:

A more efficient approach would be to ingest the transaction, trace, and contract data into separate PostgreSQL databases as a one-time effort. We can leverage its powerful indexing and querying capabilities to retrieve required data quickly, avoiding the need to scan entire files. Since we require Python support for the db module, we can use psycopg2 to interact directly with our PostgreSQL databases for CRUD operations. The detailed steps include:

- Create the ‘transactions’ table with indexes on the ‘from_address’ and ‘to_address’ columns.

- Create the ‘traces’ table with indexes on the ‘transaction_hash’, ‘from_address’, ‘to_address’, and (‘trace_type’, ‘to_address’) columns.
- Create the ‘contracts’ table.
- After creating the above tables and indexes, we can support all six of our queries.
- Now, we load the data from gzipped JSON files into the respective tables using the INSERT SQL query. We should use multiprocessing in this step for quicker execution. While reading the data, we can also use a generator helper function that yields one JSON object at a time for memory efficiency. We will store the required fields in the database, including the JSON object string itself, for handling subsequent SQL queries.
- Rewrite the six query functions mentioned before to use SELECT SQL queries for fetching database records and return the required data objects.

With the new approach, we completely move away from file-based caching and redundant parsing of huge data files to adopt indexing over relational PostgreSQL databases for better query computation time.

Code:

Please find the suggested tentative modules:

1) create_db.py - Creates PostgreSQL tables and indexes using data from .gz JSON files.
2) updated_db.py - Replacement for the current db.py module that relies on SQL queries.

Note:

We should tune the postgresql.conf parameters, since the default configuration is very conservative. Parameters such as shared_buffers, max_worker_processes, max_parallel_workers, max_parallel_workers_per_gather, work_mem, and maintenance_work_mem can be set to reflect system hardware and workload.