cole evaluation notes

Yasmine Vaziri

July 2024

1 Experiment Setup

1. Baselines:
   - MPT: Utilized by Ethereum for indexing blockchain storage.
   - LIPP: State-of-the-art learned index supporting in-place data writes, applied to blockchain storage without the column-based design.
   - CMI (Column-based Merkle Index): Uses column-based design with traditional Merkle indexes. It has a two-level structure with an upper index as a non-persistent MPT and a lower index using an MB-tree.
2. Implementation and Parameter Setting:
   - Implemented in Rust, utilizing the Rust Ethereum Virtual Machine (EVM) for transaction execution.
   - Source code is available on GitHub.
   - Baselines use RocksDB for storage, while COLE uses simple files.
   - System parameters, such as the number of generated blocks and size ratio, are specified.
3. Workloads and Evaluation Metrics:
   - Workloads: SmallBank and KVStore from Blockbench, with scenarios like Read-Write, Read-Only, and Write-Only.
   - Evaluation Metrics: Average transaction throughput, tail latency, storage size, and for provenance queries, CPU time and proof size.

2 Experimental Results

1. Overall Performance: COLE significantly reduces storage size compared to MPT and improves throughput by 1.4×-5.4×. COLE* (with asynchronous merge) performs slightly worse than COLE due to the overhead.

2. Impact of Workloads: The system's performance remains stable across different workloads, with optimal settings identified for size ratio and MHT fanout.

3. Tail Latency: Both COLE and COLE* show a U-shaped trend in latency with varying size ratios, indicating optimal settings for minimal latency.

4. Provenance Query Performance: COLE and COLE* demonstrate sublinear growth in CPU time and proof size for provenance queries, outperforming MPT, which grows linearly.

3 Benchmarks Used

The benchmarks used in the evaluation include:

- SmallBank: Simulates account transfers.
- KVStore: Utilizes YCSB for read/write tests and is used for provenance queries to simulate frequent data updates.

These benchmarks provide a comprehensive assessment of COLE's performance under various transaction loads and query scenarios.

4 Our Experiment

To calculate the storage consumption versus the number of blocks (block height) for various baselines as shown in Figure 9, we need to understand the configuration parameters in the JSON files and then run experiments with these configurations to measure the storage consumption.

4.1 Understanding JSON Parameters

index name: Specifies the type of indexing used (e.g., "mpt" for Merkle Patricia Tree).

contract name: The specific contract used in the experiment (e.g., "smallbank").

scale: The scale of the dataset (number of operations or data points).

ycsb path: The path to the YCSB (Yahoo! Cloud Serving Benchmark) data file.

ycsb base row number: The base number of rows in the YCSB benchmark.

num of contract: The number of contracts involved in the experiment.

tx in block: Number of transactions per block.

db path: Path where the database files are stored.

mem size: Memory size allocated for the operations.

size ratio: Ratio used in the storage or indexing strategy.

epsilon: A parameter related to indexing.
mht fanout: The fanout for Merkle hash trees.
result path: Path where the results or logs are stored.
fix window size: Indicates whether a fixed window size is used.

4.2 Experiment Steps

1. Setup Environment: Ensure that the necessary environment and dependencies (e.g., YCSB benchmark, database setup) are configured.

2. Load Data: Load the dataset as per the JSON configurations. Adjust the scale, ycsb base row number, and num of contract to match the experimental conditions.

3. Run Transactions: Execute the transactions in blocks as specified by tx in block.

4. Measure Storage: After executing the workload, measure the size of the database files located at db path.

5. Record Results: Note the storage size for each experiment and compare it with the corresponding block height.

5 Understanding write storage json function:

This function creates a JSON file containing storage size information for different parts of the database (e.g., tree meta, level meta, state size, mht size, model size, filter size, total size).

The JSON files are created in the directory corresponding to each workload (e.g., ./smallbank/ for the smallbank workload). The filename format helps identify the index and scale, e.g., smallbank-mpt-1k-fan4-ratio4-mem64-storage.json. Extracting Storage Consumption Data:

Each JSON file contains key-value pairs representing different components of storage and their sizes in bytes. The total size field in the JSON file represents the overall storage consumption for that configuration.

Step 1: Locate the JSON files in the appropriate directories (e.g., ./smallbank/).

Step 2: Read each JSON file to extract the total size value.

Step 3: Aggregate the total size values for different indexes and scales to plot the storage consumption as shown in Figure 9.

6 the evaluated script

```
import os
import json
```

```
#  workloads, indexes, and scales
workloads = ["smallbank"]
indexes = ["mpt", "cole", "cole_star", "non_learn_cmi"]
scales = [1000, 10000, 100000, 1000000, 10000000]

def get_storage_data(workload, index, scale, fanout, ratio, mem_size):
    # construct the JSON file name
    file_name =
        f"./{workload}/{workload}-{index}-{scale//1000}k-fan{fanout}-

    ratio{ratio}-mem{mem_size}-storage.json"
    if os.path.exists(file_name):
        with open(file_name, 'r') as f:
            storage_data = json.load(f)
            return storage_data.get("total_size", 0)
    return None


storage_results = {}

# loop through each workload, index, and scale to extract storage data
for workload in workloads:
    storage_results[workload] = {}
    for index in indexes:
        storage_results[workload][index] = {}
        for scale in scales:
            if index == "non_learn_cmi" and scale == 10000000:
                continue
            # assume default values for fanout and ratio as given in the
                original script
            fanout_default = 4
            ratio_default = 4
            mem_size = 450000 if index in ["cole", "cole_star"] else 64
            storage_size = get_storage_data(workload, index, scale,
                fanout_default, ratio_default, mem_size)
            if storage_size is not None:
                storage_results[workload][index][scale] = storage_size
            else:
                storage_results[workload][index][scale] = "Database path
                    not found or storage file not generated"

# print the results
for workload, indexes_data in storage_results.items():
    print(f"Workload: {workload}")
    for index, scales_data in indexes_data.items():
        print(f" Index: {index}")
        for scale, storage_size in scales_data.items():
            if storage_size == "Database path not found or storage file
                not generated":
```

```
            print(f"   Scale: {scale}, {storage_size}")
        else:
            print(f"   Scale: {scale}, Storage Size: {storage_size /
                (1024 * 1024):.2f} MB")
```

```
  Workload: smallbank
Index: mpt
  Scale: 1000, Storage Size: 0.39 MB
  Scale: 10000, Storage Size: 7.07 MB
  Scale: 100000, Storage Size: 108.67 MB
  Scale: 1000000, Storage Size: 1465.25 MB
  Scale: 10000000, Storage Size: 19046.84 MB
Index: cole
  Scale: 1000, Storage Size: 0.15 MB
  Scale: 10000, Storage Size: 1.51 MB
  Scale: 100000, Storage Size: 15.06 MB
  Scale: 1000000, Storage Size: 120.68 MB
  Scale: 10000000, Storage Size: 1125.18 MB
Index: cole_star
  Scale: 1000, Storage Size: 0.15 MB
  Scale: 10000, Storage Size: 1.51 MB
  Scale: 100000, Storage Size: 15.06 MB
  Scale: 1000000, Storage Size: 228.15 MB
  Scale: 10000000, Storage Size: 1575.77 MB
Index: non_learn_cmi
  Scale: 1000, Storage Size: 0.61 MB
  Scale: 10000, Storage Size: 9.41 MB
  Scale: 100000, Storage Size: 125.43 MB
  Scale: 1000000, Storage Size: 1734.58 MB
```

7 Tables

Scale	Storage Size (MB)
1000	0.39
10000	7.07
100000	108.67
1000000	1465.25
10000000	19046.84

Storage Size for MPT at different scales

Scale	Storage Size (MB)
1000	0.15
10000	1.51
100000	15.06
1000000	120.68
10000000	1125.18

Storage Size for COLE at different scales

Scale	Storage Size (MB)
1000	0.15
10000	1.51
100000	15.06
1000000	228.15
10000000	1575.77

Storage Size for COLE* at different scales

Scale	Storage Size (MB)
1000	0.61
10000	9.41
100000	125.43
1000000	1734.58

Storage Size for Non-learn CMI at different scales

8 Provided Results vs. Chart Data

MPT:

Scale 1000: Provided 0.39 MB vs. Chart 0.1 MB
Scale 10000: Provided 7.07 MB vs. Chart 2 MB
Scale 100000: Provided 108.67 MB vs. Chart 30 MB
Scale 1000000: Provided 1465.25 MB vs. Chart 200 MB
Scale 10000000: Provided 19046.84 MB vs. Chart 1000 MB
COLE:
Scale 1000: Provided 0.15 MB vs. Chart 0.1 MB
Scale 10000: Provided 1.51 MB vs. Chart 1 MB
Scale 100000: Provided 15.06 MB vs. Chart 10 MB
Scale 1000000: Provided 120.68 MB vs. Chart 100 MB
Scale 10000000: Provided 1125.18 MB vs. Chart 1000 MB
COLE*:
Scale 1000: Provided 0.15 MB vs. Chart 0.1 MB
Scale 10000: Provided 1.51 MB vs. Chart 1 MB
Scale 100000: Provided 15.06 MB vs. Chart 10 MB
Scale 1000000: Provided 228.15 MB vs. Chart 100 MB
Scale 10000000: Provided 1575.77 MB vs. Chart 1000 MB
CMI:
Scale 1000: Provided 0.61 MB vs. Chart 0.5 MB
Scale 10000: Provided 9.41 MB vs. Chart 5 MB
Scale 100000: Provided 125.43 MB vs. Chart 70 MB
Scale 1000000: Provided 1734.58 MB vs. Chart 800 MB
Observations:
MPT and COLE*:

The provided storage sizes for "mpt" and "cole star" are generally higher than the chart values.

COLE:

The provided storage sizes for "cole" match reasonably well with the chart values.

CMI:

The provided storage sizes for "cmi" (non learn cmi) are somewhat consistent but still generally higher than the chart values.