Benchmarking and Tuning Log-Structured Table Formats

Jesús Camacho-Rodríguez, Ashvin Agrawal, Anja Gruenheid, Ashit Gosalia, Cristian Petculescu, Josep Aguilar-Saborit, Avrilia Floratou, Carlo Curino, Raghu Ramakrishnan*

Technical Fellow, CTO for Data
Comparing Log-Structured Table Formats
Delta Lake, Apache Hudi, and Apache Iceberg

• Popular OSS projects for updatable tables: **Delta, Hudi, and Iceberg**

  • **Goal:** Provide additional functionality (transactions, indexes, time travel, cloning) on top of low-cost data lake storage

• Although they share the same goal, architecture and implementation is different

  • Each one has its own strengths/weaknesses
  • Metadata caching, distributed planning, CoW and MoR support, storage optimizations
  • Maturity of integration with supported engines (e.g., Spark, Presto) differs
• **Overview of Log-Structured Table Storage**
  • Storage format
  • Concurrency control

• **LSTBench**
  • Key ideas
  • Results
Types of Data in LSTs

- Many types of data are required to represent a log-structured table
  - Logical metadata (e.g., table definitions)
  - Physical metadata (how to find data for a table)
  - Data files (parquet type, column oriented, store user data)
  - Delete bitmaps (describes which rows are deleted)
  - Clone info (reference counting for files)
- Where this data is stored
  - Files in cloud storage
- Approach to updates—NOT in-place in page-files!
  - Copy-on-Write (CoW)—affected data files copied over with changes reflected
  - Merge-on-Read (MoR)—for affected data files, ONLY changes recorded; reads must merge changes
Table Format
Row-Oriented Page Files on Disks
Table Format
Row-Oriented Page Files on Disks

(1) X1. Loaded 3 rows and committed
(2) X2 bulk inserts 2 rows
(3) X2 deletes one row, then inserts 2 more rows
Indexes
Auxiliary structures that support direct access to rows with given values in indexed columns.
Table Format
Column-Oriented
Log-Structured Files
Table Format

Column-Oriented

Log-Structured Files
Table Format
Column-Oriented Log-Structured Files
Storage Reorganizations Required (Examples)

- **Poor storage quality**
  - Over time, trickle updates/deletes create sub-par storage layouts
  - Deletes also lead to inefficient scans
- **Data compaction**
  - Read only those files known to contribute to poor scan performance
  - Write those files into a new set of files with optimal layout
  - Does not change data, only changes physical structure
- **Metadata handling**
  - Can have a significant impact!
Concurrency Control—SI and MVCC

- Given mostly-read workloads and immutable data files, optimistic or versioned approaches
  - NOT lock-based CC!
- With Snapshot Isolation and MVCC, we only need to track write/write conflicts
  - Snapshot isolation protects us from read after write and write after read conflicts
  - Typically, tracked at modified data/parquet file granularity
  - Finer-granularity tracking/detection introduces more overhead, improves concurrency
Table/Database Clone

- Copy logical/physical metadata of a table (or database of tables)
  - Transactionally consistent
  - Metadata only – data is shared
- Once clone occurs, tables have separated futures
  - DML/DDL against one does not affect the other
  - Requires ref counting of data files for GC and retention
- Backup and restore of database implemented as clone
  - Could have one database (or subset of tables) current, one with data from a week ago
Time Travel Queries

- Normal transaction
  - Takes read snapshot of physical metadata
  - Reads the whole sequence to get latest visible snapshot of data
- Time travel query
  - Reads only a prefix of the physical metadata
  - Still transactionally consistent, but no longer correct for the current txn snapshot
  - Requires versioned logical metadata (if ALTER TABLE occurs)
- Clone and backup/restore a kind of time travel
  - Table clone allows you to keep a backup copy of the state of a table
  - Not copying data, just metadata
  - Only possible due to multi-version, immutable data files
Existing Approaches to Evaluate LSTs

• Numerous blog posts and papers comparing LSTs
  • Theoretical evaluation: Features, OSS community
  • Experimental evaluation: LH-Bench, Brooklyn Data, DataBeans, OneHouse
    • Typically rely on TPC-DS, standard OLAP benchmark
    • May involve limited number of queries or handcrafted queries to test certain operations

• Limited work on a comprehensive framework to evaluate LSTs
  • Complexities of continuously changing performance in long-running deployments
  • Effect of running maintenance operations concurrently with other queries
TPC-DS Overview

**Workload**

TPC-DS tasks
- Load
- Single User
- Data Maintenance

**Performance Metric**

- Normalized query throughput per hour

\[ Q_{phDS@SF} = \frac{SF \times Q}{\sqrt{TP_{PT} \times TP_{TT} \times TP_{DM} \times TP_{LD}}}. \]

Product of total # of queries executed and scale factor

Geometric mean of elapsed time for load, power, throughput, and data maintenance phases
**Limitation 1:** Failure to expose important characteristics of LSTs that are crucial in actual customer scenarios.
**TPC-DS Metrics**

**Limitation 2**: Lack of metrics to expose important aspects when comparing different LST implementations

\[ Q_{phDS@SF} = \left[ \frac{SF \times Q}{\sqrt[4]{T_{PT} \times T_{TT} \times T_{DM} \times T_{LD}}} \right] \]

<table>
<thead>
<tr>
<th>LST</th>
<th>Throughput-QphDS</th>
<th>Inter-test Degradation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delta</td>
<td>511K</td>
<td></td>
</tr>
<tr>
<td>Hudi-CoW†</td>
<td>262K</td>
<td></td>
</tr>
<tr>
<td>Hudi-MoR‡</td>
<td>112K</td>
<td></td>
</tr>
<tr>
<td>Iceberg-CoW†</td>
<td>549K</td>
<td></td>
</tr>
<tr>
<td>Iceberg-MoR‡</td>
<td>493K</td>
<td></td>
</tr>
</tbody>
</table>

† Copy-on-Write mode  ‡ Merge-on-Read mode

SF=1000
Spark (3.3.1), 16 workers, 8 cores, 64 GB
Delta (2.2.0), Iceberg (1.1.0), Hudi (0.12.2)
LST-Bench

A benchmark specifically tailored to evaluate LSTs

- Builds on TPC-DS
- Proposes new *workload patterns* and *metrics* extensions relevant for LSTs
- **Open-source implementation:** [https://github.com/microsoft/lst-bench](https://github.com/microsoft/lst-bench)
- Evaluation of OSS Spark + Delta Lake, Apache Hudi, Apache Iceberg
Extending Workloads Beyond TPC-DS

Workload components:

Extensions to enhance TPC-DS workload:

- Configurable sequence of phases
- Ability to run multiple different tasks concurrently within a phase
- New tasks:
  - Optimize
  - Single User (Time Travel)
LST-Bench Workload Patterns

Gain insights into LST aspects overlooked by the base workload

- **WP1**: Data modifications over a long period of time

  ![WP1 Diagram](image)

- **WP2**: Multiple data modifications of varying sizes in a regularly optimized table

  ![WP2 Diagram](image)

- **WP3**: Multiple sessions reading and writing data simultaneously

  ![WP3 Diagram](image)

- **WP4**: Querying data at different points in time

  ![WP4 Diagram](image)
LST-Bench Metrics

Traditional Metrics
- **Performance:** *Latency, Throughput*
- **Cloud Storage Efficiency:** *Capacity Utilization, API Call Count, Total IO*
- **Compute Engine Efficiency:** *CPU Utilization, Memory Utilization, Disk Utilization*

Other Metrics
- **Stability** - System’s ability to sustain consistent performance and efficiency with minimal degradation: *Degradation Rate*

\[
S_{DR} = \frac{1}{n} \sum_{i=1}^{n} \frac{M_i - M_{i-1}}{M_{i-1}}
\]

where
- \(M_i\) is metric value of the \(i^{th}\) iteration of a workload phase,
- \(n\) is the number of iteration of the phase, and
- \(S_{DR}\) is the degradation rate.
LST-Bench Implementation

- Java Client Application
  - Customizable / extensible via config files
  - Connects to engine via JDBC

- Python Metrics Processor

- Open-source available: https://github.com/microsoft/lst-bench
Evaluation

- **OSS Spark 3.3.1** cluster (Azure VMSS) with 1 head and 16 worker nodes
  - Each node with 8 virtual cores and 64GB RAM
- Data stored in Azure Data Lake Storage Gen2 (ADLS)
  - TPC-DS SF1000
- Azure Monitor to collect telemetry and Logs Analytics to execute queries against it

- No special tuning for any of the LSTs we evaluated:
  - Delta Lake v2.2.0, Apache Hudi v0.12.2, Apache Iceberg v1.1.0

- Important remarks
  - Results subject to change and improvements due to further tuning and future developments
  - Insights drawn for Spark may not apply to the LST on different engines (Trino, Presto)
Evaluation WP1

Significant slowdown across iterations (up to $6.8x$)

Significant differences between MoR vs CoW regarding interaction with the storage layer
Table maintenance has a big impact on Delta and Iceberg performance stability (zig-zag pattern).

Hudi maintains stable performance without user-triggered maintenance by doing work upfront.

Iceberg's default file grouping for compaction significantly increases compaction time (potentially minimizes read query disruptions). Tuning LSTs involves trade-offs based on user goals.
Running maintenance operations on the cluster while executing read queries has negligible impact on query latency.
No significant differences found between queries on the latest data version and time travel queries on same version after data modifications.
## Stability Evaluation

<table>
<thead>
<tr>
<th></th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
<th>Combined</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iceberg-MoR</td>
<td>0.41</td>
<td>0.88</td>
<td>0.89</td>
<td>0.73</td>
</tr>
<tr>
<td>Iceberg-CoW</td>
<td>0.19</td>
<td>0.64</td>
<td>0.69</td>
<td>0.51</td>
</tr>
<tr>
<td>Delta</td>
<td>0.29</td>
<td>0.41</td>
<td>0.30</td>
<td>0.34</td>
</tr>
<tr>
<td>Hudi-MoR</td>
<td>0.05</td>
<td>0.39</td>
<td>0.23</td>
<td>0.23</td>
</tr>
<tr>
<td>Hudi-CoW</td>
<td>0.07</td>
<td>0.38</td>
<td>0.20</td>
<td>0.22</td>
</tr>
</tbody>
</table>

- **Hudi** shows highest stability
- **Iceberg** shows lowest stability
Next Up ...

• Evaluating existing LSTs and their engine integrations to identify strengths and areas for improvement

• Extensions to LST-Bench to explore aspects overlooked in proposed workload patterns

• Developing best practices and tuning configurations to optimize LSTs performance
Thank you!