Reflecting on

Metadata Consistency in Open-Source Catalogs

Author: Benjamin Storm Larsen

Supervisor: Martin Hentschel

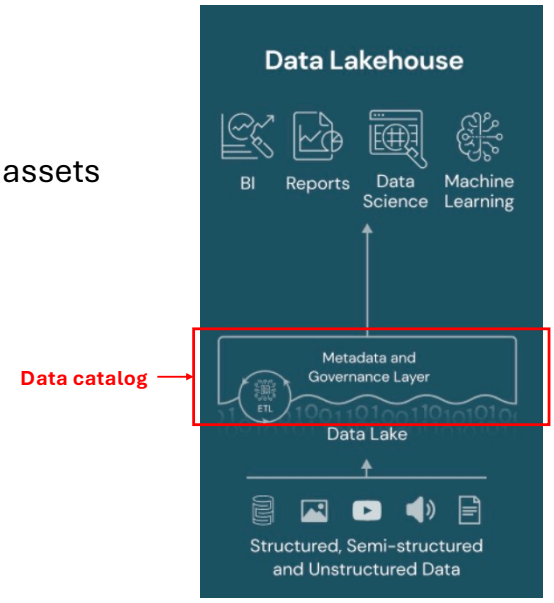
POLARIS CATALOG

Unity Catalog

I'm investigating metadata consistency in open-source data catalogs. The main goal was to benchmark how well Apache Polaris and Databricks' Unity catalog enforce consistency across their metadata APIs. This is done with a custom-made benchmark driver, which was built using Golang.

Data Catalog

- Centralized repository for data assets
- Data about data
- Features
  - Discovery
  - Governance
  - Organize data
- Crucial in the Data Lakehouse



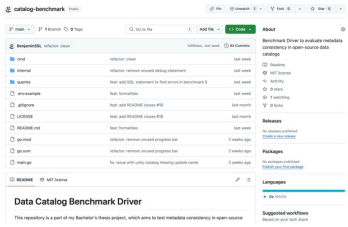
A data catalog is a centralized repository that store and organize metadata about data assets within an organization.
Their primary purpose is to facilitate data discovery, governance, and access control, ensuring that users
can efficiently search, understand, and access data across multiple systems. By serving as a unified interface for
accessing metadata, data catalogs enable data consumers, such as analysts, data engineers, or data scientists, to
discover datasets without having to repeatedly search across separate systems.

They are crucial for the Data Lakehouse architecture as they sit between the clients and the actual data, orchestrating access and retrieval between the parties.

Benchmark driver

Driver

- Developed in Golang
- Implements the Catalogs API
- Scales to thousand of goroutines
- Thousands of request per seconds
- Tests multiple entities of the data catalogs
- Open-source on GitHub



https://github.com/BenjaminSSL/catalog-benchmark

The driver was an important tool, which enabled me to target the catalogs with specific operations and metadata.
It works by running various benchmark scenarios against the data catalog, to test various aspects of it. At different durations, and with n number of clients.

When started it creates the specified number of threads, which each send operations to the data catalogs APIs, on their entities. Golang is well know for its excellence support for async programming, which is part of the reason for developing the driver with it.

The driver is uploaded to GitHub to allow maintainers to inspect the code and run against their data catalogs and potentially implement the interface for other catalogs.

What is tested?

- Consistency in the data catalog
  - When an operation is executed does any errors occur?
  - Is the error transparent in its message?

- Benchmark scenarios
  - Targets specific operations on the data catalogs
  - Entity-agnostic which allows the benchmark to run on all the catalog entities (important)
  - 795 unique benchmarks

Benchmark ID	Benchmark Name	Description
1	Create `entity`	Repeatedly creates `entity`
2	Create & Delete `entity`	Repeatedly creates and deletes `entity`
3	Update `entity`	Repeatedly updates the same `entity`
4	Create & Delete & List `entity`	Repeatedly creates, deletes, and lists `entity`
5	Create & Update & Get `entity`	Repeatedly updates, and gets `entity`

Consistency in the data catalogs, is in my thesis defined by the errors. When sending an operation to the data catalog it responds with a success or an error. If it's an error, then it is analyzed to uncover why it happened and what went wrong. It is also examined if the error message is transparent for the user, as this also plays an important role for debugging the issues.

The driver has 5 built-in benchmark scenarios. These test various aspects of the data catalog, including running multiple operations in parallel to analyze the interplay between them and the effect on metadata consistency. Important to note, is that they are entity-agnostic, which means that they can run on any entity. The reason why it is important is because some entities are implemented differently from others, and they might not behave the same. With a benchmark suite, altering multiple parameters, I was able to run 795 unique benchmarks against the catalogs.

Fermi method

- Used to estimate load in a production environment
- Helpful for justifying the benchmark parameters
- Estimation parameters
  - Users/services request per second
  - Request Types
  - Error tolerances
  - Size of host machine
  - Etc.

Given that I did not have any data on accurate loads, I had to figure out my own parameters for testing the data catalogs. The Fermi method assisted me in finding the appropriate parameters for testing the data catalogs.
I used the following parameters in my estimation: Users/services request per second, Request Types, Error tolerances, Size of host machine and more.

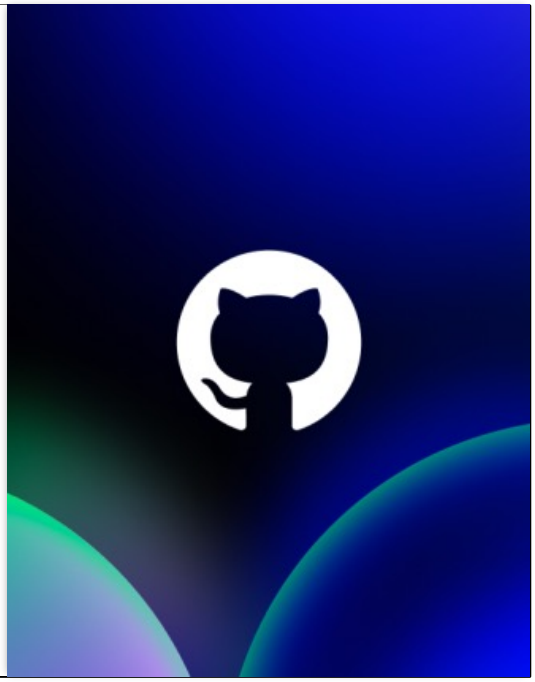
This made the load slightly more realistic, given the limitation of not having an accurate standpoint, in terms of real production load on a data catalog.

Results

Found three critical consistency issues in the Polaris Catalog

- Inconsistent listing of entities
- Concurrent updates cause consistency issues
- Creating and deleting entities causes entity-tree errors

All bugs are reported on Polaris GitHub repository:
https://github.com/apache/polaris



The results includes three inconsistencies found in the Polaris data catalog. These inconsistencies were found to have a significant impact on the data catalog, with very vague error messages. They were reported to the maintainers on GitHub, to allow them to resolve the issue in future releases.

All the bug reports can be found in the Apache Polaris GitHub repository.

Biases

- Selection Bias
  - Tested only the two most used and established data catalogs in the open-source space.
- Cognitive Bias
  - Unity Catalog is more coherent in its code and documentation
  - Polaris is messy and its documentation is cluttered
    - My supervisor previously worked for the company that created Polaris.

This thesis is subjected to selection bias, as it focuses exclusively on two of the most widely used and well-established open-source data catalogs. While the thesis is not directly comparing them, it might not provide a generalized result for smaller or newly established data catalogs.

It is also subjected to cognitive bias, which may have influenced perceptions, as Unity catalog displays a clean design with good documentation, while Polaris felt messy. Additionally, the fact that my supervisor previously worked with Snowflake, which created Polaris, might have introduced unconscious bias in my research.

Limitations

- Limited to only two data catalogs
- Focused benchmarks does not model a real production environment
  - Database and external cloud storage
  - Authentication was not tested
  - Work loads were estimated
- Does not examine the underlying metastore

In extension to the selection bias, I have included the fact that only testing two catalogs is a limitation of the thesis.

The focused benchmark does not model a real production environment, in the sense that a real data catalogs typically handle a heterogeneous mix of request from various clients simultaneously. This is difficult to implement and analyze, but doing so could uncover additional inconsistencies.

Additionally, the testing environment was fully local and disconnected from the network. This does not reflect real-world conditions, where data catalogs are usually hosted in the cloud and uses external storage such as S3. Authentication is an important factor, but since it depends on an external Identity and Access Management (IAM) solution, it could have introduced additional variables into the experiments.

The inconsistencies only covers what the API returns to the driver, the metastore (database) was not examined for ensuring that the commits were correct. This is also included in the future work section of my thesis.