Study on Dictionary Metadata in Cloud Data Systems: Usage, Performance, and Storage Formats

Andreas Kongstad (kong@itu.dk) & Carl Bruun (carbr@itu.dk)
Supervisor: Martin Hentschel
Class code: V24KIREPRO1PE872

December 16, 2024

Abstract

Today, cloud-native data systems rely on scalable and cost-efficient storage solutions like Amazon S3 to manage varying workloads and data types. However, while table storage formats like Apache Iceberg and Delta Lake provide advanced table metadata management in data lakes, a challenge remains in the absence of a standardized open format for managing dictionary or non-table metadata. This limitation complicates interoperability and ties organizations to specific data platforms.

This project explores how dictionary metadata can be moved from data systems' internal structures into data lakes with reduced catalog dependencies. Analysis of six data systems revealed significant variation in metadata object and feature support, emphasizing the need for extensibility while balancing implementation complexity. Performance tests showed how metadata structuring impacts efficiency, with DuckDB, Postgres, and Snowflake demonstrating constant performance on metadata point query operations, while SQLite showed linearly scaling characteristics. Based on these findings, we propose baseline requirements and an initial direction for extending metadata management atop Iceberg.

1 Introduction

Cloud computing has revolutionized data storage and processing by providing scalable, flexible, and highly available infrastructure that meets the growing demands of modern businesses. Along with the many advancements in cloud technologies, data lakes have gained prominence as organizations look for solutions to store and manage large volumes of diverse data in open formats.

As data storage requirements have grown to include structured, semi-structured, and unstructured data, workloads have grown to include standard business intelligence analysis and reporting on structured data, as well as data science and machine learning. Diversification of workloads suggests the need for data platforms optimized for specific workloads on data in open formats instead of transferring enormous amounts of data into and out of general-purpose data warehouses or platforms.

Open-source data formats like *Apache Parquet* and *ORC* provide efficient and scalable storage and retrieval from cloud object stores such as AWS S3, Google Cloud Storage, and Azure Blob Storage, allowing for cloud-native platforms with decoupled storage and compute [1]. Similarly, *Apache Iceberg* offers an open table format with SQL-like functionalities for semi-structured and unstructured data [2]. However, while these standards cover data and table formats, they lack support for managing dictionary metadata in cloud-native systems.
That is, while the data itself is often portable, the associated dictionary metadata is typically stored in proprietary formats and service control planes, creating vendor lock-in and making it difficult for organizations to seamlessly migrate data, integrate systems, or switch between platforms without costly and time-consuming conversions[1].

This project contributes to the development of an open-source metadata storage format by addressing the limitations of existing formats in handling non-table metadata. It defines minimum requirements for database object compatibility, features, and metadata performance and explores storage formats to provide initial architectural directions.

The paper is organized as follows: Section 2 addresses inconsistencies in terminology to establish consistency throughout the paper. Section 3 provides an overview of data lake management and storage formats. Section 4 analyses three proprietary and three open-source systems, identifying common metadata objects and features. Section 5 outlines metadata query performance experiments on said systems. Section 6 discusses the results of the experiments and initial directions for an open dictionary metadata format, including limitations and pointers for future work. Finally, Section 7 concludes by summarizing key findings.

2 Terminology

This section presents a set of definitions used throughout the paper to establish a common language and address inconsistencies in terminology in online resources.

Catalog: According to IBM and Google documentation, a *data catalog* is "a detailed inventory of all data assets in an organization "[3][4]. ANSI uses the term "catalog" to refer to databases[5]. In Apache Iceberg a catalog is an entity that contains a meta-store tracking tables and their metadata files using atomic pointers[6]. In PostgreSQL, the *system catalogs* store schema and internal bookkeeping metadata[7]. In Databricks Unity Catalog, "Catalog" objects contain schema objects, which in turn contain objects like table and view.[8]

Database Object: Multiple sources define *database objects* as a structure for storing, managing, and presenting data or metadata in a database. These objects include tables, views, and functions that can be manipulated through data definition language (DDL) statements. Some also distinguish between schema object and non-schema objects.[9][10][7][11].

Dictionary metadata: Not to be confused with a *data dictionary* often defined as some repository storing metadata[12]. *Dictionary metadata* is the metadata stored in and as database objects. That is, everything that can be manipulated using DDL.

Data Definition Language (DDL): commands are used to create, manipulate, and modify database objects: *CREATE/ALTER/DELETE*. Snowflake also includes *DESCRIBE*, *SHOW*, and *USE* under DDL commands[13] while others may categorize these as meta queries[14].

Information Schema Queries: Many systems provide *Information schema queries* that interface for the information provided by *SHOW/DESCRIBE* commands. The information schema views are defined in the SQL standard, and the system-specific objects are queried from somewhere else[15].

Meta-store: in DataBrick's Unity Catalog, a meta-store is a container for metadata inside the Unity Catalog, storing database objects in 3 levels[8]. Hive Metastore is a service that stores metadata in a relational database[16].

Table 1 outlines the specific terminology used throughout this paper.

Term	Description
Catalog	The data lake catalog definition. Engines query through the catalog to access data in a data lake. A catalog tracks and manages tables. Some catalogs also support RBAC.
Database Object	Objects such as Table, View, Function, and User that can be manipulated via DDL commands.
Dictionary Metadata	Metadata represented as database objects, encompassing all data that can be created, altered, or dropped using DDL statements.
DDL	A set of SQL commands that manipulate database objects, including *Create*, *Alter*, *Delete*, *Show*, *Use*, and *Describe*.
Information Schema Queries	Queries that retrieve metadata about database objects from the information schema views. An SQL standard
Meta-store	A relational database, key-value store, or similar that stores dictionary metadata.

Table 1: Terminology overview.

3 Background

As the industry shifts away from a "one size fits all" culture, with an increase in distinct cloud platforms and data lake storage options, solutions are needed to provide seamless service interoperability. In other words, open metadata standards to wrap external data lakes. This section will describe state-of-the-art solutions that have sought to address this new data and metadata management paradigm and their limitations for which proprietary components must still fill a gap.
Open table storage formats like Apache Iceberg, Delta Lake, and Apache Hudi directly enable ACID transactions and DBMS functionality over data lake storage. Cloud object storage offers scalability and cost-efficiency but at the cost of access latency. To mitigate the impact of latency overhead and avoid scanning the entire data lake, open storage formats introduce a metadata layer that tracks file locations, schema, min-max statistics, and other table properties to enable faster query planning.

3.1 Iceberg

Apache Iceberg is a storage format for analytic tables that allows engines to query the tables from data lake file-stores. It supports SQL, schema evolution, time travel, and rollback functionalities[17]. Iceberg stores table metadata in files alongside data files using a hierarchical structure. Manifest files at the bottom level store metadata for data files, while higher-level manifest lists aggregate metadata from manifest files, acting as an index. At the root, metadata files track the table schema, partitioning, custom properties, and snapshots. Iceberg serializes schema and metadata to JSON, storing them in a metadata folder for each table with versioned file names, e.g., $vxx.metadata.json$[18].

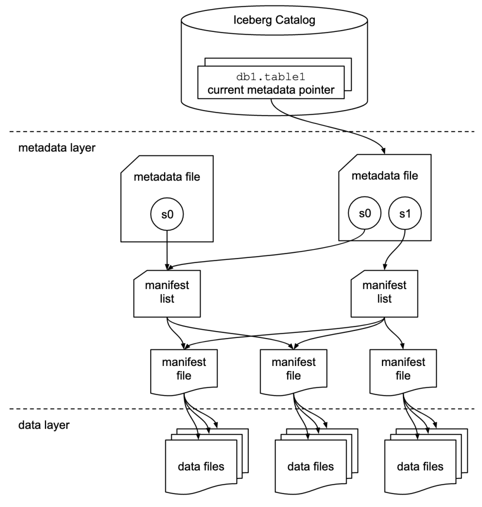

Figure 1: Iceberg Architectural overview [18]

Iceberg Catalogs

Iceberg manages data at the table level, with operations like creating, dropping, and renaming tables handled

through the catalog as depicted in Figure 1. The catalog tracks the current table's metadata address through atomic pointers for consistency, and initializing the catalog is thus the first step when using Iceberg[6]. Compute engines use the catalog to perform operations on Iceberg's metadata layer, such as creating tables, updating metadata, or listing tables.
Multiple Iceberg Catalog implementations are supported and can be plugged into any Iceberg runtime. These are based on different metadata storage backends and include:

- REST: A server-side catalog with a REST API[6]. It provides compatibility with any engine that supports the Iceberg REST Client.

- JDBC: A simple relational database like Postgres for metadata storage and management.[6]

A benefit of the REST catalog is that it decouples the REST server from the backend storage, allowing one client to interact with any catalog backend with a single client implementation. As such, the REST catalog is not just a deployable catalog but an API specification that provides new catalog implementations - like Apache Polaris - with an interface for table-level operations. This reduces the development burden by eliminating the need to create separate clients in multiple engine languages.

3.2 Apache Polaris

Apache Polaris is an open-source catalog implementation for Apache Iceberg, specifically an implementation of the REST API catalog[19]. Polaris extends standard Iceberg catalog functionality with Role-based access control using ”credential vending”[20]. It implements service connections to REST-compatible engines like Flink, Spark, and Snowflake such that, as an example, user A, given the proper privileges, can create and populate an Iceberg table using Spark, after which user B can read the newly created table using Snowflake. Polaris extends Iceberg with additional objects, including Principal Identities that represent users or services, Principal Roles that can be granted to principals, and Catalog Roles that can be granted to catalogs. Principals and Roles form the mechanism for granting access to entities within the catalog[20].
The Polaris code base defines *PolarisMetaStoreSession* as an interface to the Polaris metadata store. The metadata store persists Polaris entity metadata and grants between these, which is the foundation for the Polaris Role Based Access Control model (RBAC). PolarisMetaStoreSession abstracts the meta-store and supports databases like Postgres or simpler key-value stores[21].

3.3 Delta Lake

Delta Lake is a transactional storage layer that optimizes data lakes by providing ACID transactions, data versioning, and schema enforcement through Delta Tables inside cloud object stores. Delta Lake's client libraries - equivalent to Iceberg Catalog implementations - automatically reject commits that violate these expectations.[22].

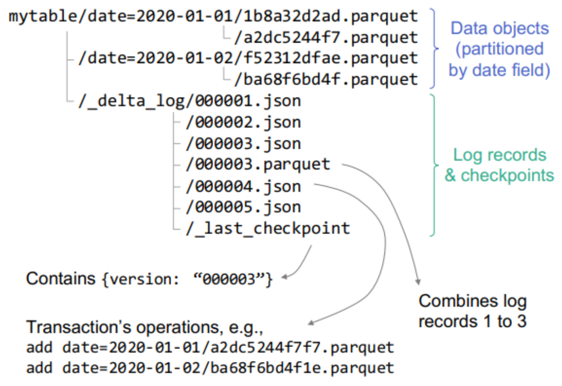
mytable/date=2020-01-01/1b8a32d2ad.parquet
/a2dc5244f7.parquet
/date=2020-01-02/f52312dfae.parquet
/ba68f6bd4f.parquet
/_delta_log/000001.json
/000002.json
/000003.json
/000003.parquet
/000004.json
/000005.json
/_last_checkpoint
Data objects (partitioned by date field)
Log records & checkpoints
Contains {version: "000003"}
Combines log records 1 to 3
Transaction's operations, e.g.,
add date=2020-01-01/a2dc5244f7f7.parquet
add date=2020-01-02/ba68f6bd4f1e.parquet

Figure 2: Sample Delta Table([1] page 3414, Figure 2)

As shown in Figure 2, Delta tables are directories of table data stored in Parquet and with a transaction log that tracks metadata and data organization. The transaction log ensures serializability and snapshot isolation by managing object store operations through the client library. Each Delta table uses a write-ahead log to track data files and provide metadata such as:

- **Schema Definitions:** Describes the data organization to support schema enforcement.

- **File Locations and Statistics:** Contains the paths to data files stored in the data lake and min-max statistics.

- **Version History:** Tracks changes to data and metadata, supporting features such as time travel and rollback.

Delta tables act as wrappers around data objects to optimize query processing closer to the source. In other words, they provide operational metadata at the table level, enabling clients to run queries over object storage by launching servers with client library support, such as Unity Catalog for Databricks[8], as a separate compute unit.
One limitation lies in the table-level transaction logs, in which consistency cannot be ensured across multiple tables as it would result in contention over sharing logs [1]. Also, though Delta Lake decouples operational metadata from the compute engine, data governance and addressing dependencies remain. The Databricks paper, 'Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores' [1], does not explicitly discuss centralized governance, highlighting a limitation in reliance on third-party solutions like Unity Catalog. As an example, managed tables - Databricks' default table - use Unity Catalog and Delta Tables together, providing data access control, auditing, and lineage [8].

3.4 Summary of Key Takeaways

As described in this section, formats such as Iceberg and Delta Lake provide features like ACID transactions, time travel, schema enforcement, and query optimization using statistics. These features help reduce latency in accessing metadata stored separately from the data by keeping metadata in the storage layer. By decoupling table metadata management from specific storage and computing services, these formats allow multiple platforms to interact with the data lake data.

However, catalogs outside the data lakes are left to handle non-table metadata. That is, functional dependencies remain for all formats in centralized meta-stores like the Iceberg Catalog and Unity Catalog, which manage metadata such as access control, governance, and addressing. These dependencies complicate interoperability between different systems and data lakes, where catalog metadata layers may not be universally compatible, maintaining vendor lock-in.

Though some of these dependencies are inherently necessary for current formats, a question is raised about to what extent the catalog metadata management layer could be standardized and what trade-offs such an implementation might introduce. To make a case for a different approach, we explore what object types are essential for metadata management across data systems, their supported features, and time complexity.

4 Data System Metadata Features and Objects

This section examines six systems' database objects and features, including three open-source and three proprietary options. It includes OLTP and OLAP-optimized systems, along with traditional and cloud-native architectures. This diverse selection aims to provide a comprehensive perspective on metadata management design and support across various platforms, data systems, environments, and workloads.

4.1 SQLite

SQLite is an open-source light-weight, self-contained transactional (ACID) database engine in which the complete state of the database is usually contained in a single ”main database file” on disk, simplifying management and portability, with an option of being used as an in-memory database engine[23].

SQLite supports only *CREATE*, *ALTER*, and *DROP* DDL commands on Table, Index, View, and Trigger database objects[24]. The metadata is stored within the ”sqlite_master” table, which is automatically created and maintained for each SQLite database and can be queried like any other database table either directly or through information schema views. The master table contains metadata for all database objects excluding itself.[25]. The master table schema is depicted in Figure 3.

```
CREATE TABLE sqlite_master (
  type      TEXT,    -- either "table" or "index"
  name      TEXT,    -- name of this table or index
  tbl_name  TEXT,    -- for indices: name of associated table
  sql       TEXT     -- SQL text of the original CREATE statement
)
```

Figure 3: ”sqlite_master” table[25]

One thing to note is the lack of access control statements, such as *GRANT* and *REVOKE*, as SQLite is a server-less, self-contained database engine. Since it operates on a single-user basis with direct access to the database file, there is no concept of role-based user permissions. SQLite does not even support user objects or retain historical or versioning data, emphasizing its simple design.

4.2 PostgreSQL

PostgreSQL is an open-source object-relational database system optimized for OLTP workloads. According to the stack overflow developer survey 2024 of 41.000 respondents, 51% of professional developers use PostgreSQL, making it the most popular database for the second year in a row[26].

In terms of architecture, PostgreSQL uses a client-server model. A server process manages the database files, accepts connections, and performs database actions that client applications want to perform[27].

Database object metadata is stored in the *system catalogs*. In PostgreSQL, system catalogs are regular tables that can be accessed and modified using standard SQL commands. A subset of the system catalogs is listed in Table 2. Catalogs like *pg_class*, which store metadata about tables, sequences, views, and similar objects, are database-specific. Other catalogs, such as *pg_database*, which stores information about available databases, and *pg_authid*, which contains Role objects, are shared across all databases in a cluster. Interestingly, a Role subsumes the concepts of ”users” and ”groups”. A typical ”user” is instead a Role with the *rolcanlogin* flag set. If one executes ‘*CREATE USER*‘ or ‘*CREATE GROUP*‘, Role objects are instead created[28][7]. Users may define access control rules using the *GRANT* and *REVOKE* commands on Role object[29].

PostgreSQL supports a list of database objects, which include Database, Table, View, Function, Procedure, Operator, Type, Domain, Trigger, Rule, User, and Role. The DDL commands on database objects in PostgreSQL are: *ALTER*, *CREATE*, *DROP*, and *SHOW*[29].

Catalog Name	Purpose
`pg_attribute`	Table columns (“attributes”)
`pg_authid`	Authorization identifiers (roles)
`pg_class`	Tables, indexes, sequences, views
`pg_database`	Databases within this database cluster
`pg_description`	Descriptions or comments on objects
...	...

Table 2: Subset of PostgreSQL system catalog tables.

4.3 DuckDB

DuckDB is an another open-source, embedded, in-process relational DBMS, however optimized for OLAP workloads in contrast to SQLite. DuckDB supports SQL queries with ACID guarantees and is compatible with Parquet files, allowing for direct querying of Parquet data[30]. Furthermore, it offers an extension mechanism that allows for the definition of new data types, functions, file formats and new SQL syntax[31].

DuckDB supports the DDL commands *CREATE*, *ALTER*, and *DROP* on *"catalog objects"*. These objects are:
Database object metadata is store is stored in an in-memory hash key-value structure. Index, Macro, Schema, Secret, Sequence, Table, View, and Type[32]. Furthermore, it supports Describe and Show commands on table objects with *Metadata Functions*, which are table functions that return metadata from the current database[33]. Moreover, like SQLite, DuckDB is a single-file, single-user database. Thus, it does not define User objects and does not need access control commands.

4.4 Snowflake

Snowflake is a cloud-native OLAP-optimized data warehouse. It uses a 3 tier architecture with separate storage and compute, below a managed control plane comprising different cloud services. Metadata objects are stored in a transactional key-value store in the control plane[34].

Snowflake supports three overall object types. *Account objects* include User, Role, Database, and Warehouse. *Database objects* include Table, View, Function, and File. Finally, *Classes* are objects with a public API of stored procedures and functions, and they include Budget and ML-specific objects[13][35].

Snowflake supports standard and extended SQL on structured and semi-structured data, including support for *DDL* commands *CREATE*, *ALTER*, *DROP*, *UNDROP*, *SHOW*, *USE*, and *DESCRIBE* on account and database objects. SQL extension includes support for user-defined-functions in popular programming languages and Iceberg tables that use the *Apache Iceberg* format to allow Snowflake to operate on data lakes[36]. Furthermore, Snowflake supports features such as object-level access control, time travel queries, and zero-copy data sharing between accounts[36].
At object creation, metadata fields are captured and added to the meta-store. The most common metadata fields are (1) object definitions, such as a policy, an external function, or a view definition, and (2) object properties, such as an object name or an object comment or, in other words, any fields in the DDL command that creates or modifies the object[37].

4.5 Databricks

Databricks is an analytics platform that employs the Lakehouse architecture to integrate the strengths of both data lakes and data warehouses. Databricks operates out of a control plane (managed backend services) and compute plane[38].
As a complement or alternative to Delta Lake, described in Section 3.3, metadata such as catalog management and access control policies is handled by Databricks' Unity Catalog, which operates through a centralized meta-store at the control plane. The meta-store contains metadata about all databases, tables, views, and permissions across multiple Databricks workspaces and cloud regions in a three-level database object hierarchy[39]. Level one consists of *Catalogs* and *Non-data securable objects*. The catalog objects consist of schemas (commonly known as databases) that include Volume, Table, View, Function, and Model objects. Non-data securable objects include data governance objects including Storage Credentials and Connections.[40]. Each catalog in Unity Catalog provides a standard SQL schema, *INFORMATION_SCHEMA*, allowing users to query meta-store data through its various views to access metadata about objects, their privileges, and the current meta-store based on what the user is privileged to see[41]. In addition to the standardized information schema access, metadata can be queried directly from the meta-store via SQL *DESCRIBE* (or "SHOW") commands, similar to Snowflake's *SHOW* DDL commands[42].
Databricks additionally supports DDL queries *CREATE*, *ALTER*, *DROP*, *COMMENT*, *DESCRIBE*, *GRANT*, *REVOKE*, *SHOW*, *TRUNCATE*, *USE*, *UNDROP*, and *REFRESH* (Foreign)[42].

4.6 Oracle

Oracle Database is a RDBMS that supports standard SQL with proprietary extensions. It is the core engine for various Oracle services, available for on-premises deployments and through Oracle Cloud Infrastructure (OCI)[43].

The Oracle Database engine supports two types of database objects: Schema Objects and Non-Schema Objects. A Schema is a collection of schema objects owned by (and named after) a database user, where each user owns a single Schema inside a logical storage container called a user-tablespace. The Schema Objects include Tables, Views, and Indexes. The Non-Schema Objects exist within a centralized system tablespace owned by a special "sys" user requiring a higher level of privilege and includes the data dictionary for administrative metadata, which is further described below[44][45].

The *Data Dictionary* is a read-only set of tables containing administrative metadata of the database, such as the definitions of all Schema Objects, space allocated and used by these objects, and users, roles, and privileges. There are no built-in system commands to access the metadata, such as the previously seen "Show" command. However, as the dictionary data is stored in tables, it can be accessed and read through SQL statements. The base tables are, however, rarely accessed due to their complexity and low readability. Instead, SQL queries are written to access specific views, much like the information schema seen for

Object	SQLite	Postgres	DuckDb	Snowflake	Databriks	Oracle
Table	✓	✓	✓	✓	✓	✓
View	✓	✓	✓	✓	✓	✓
Function	✗	✓	✓*	✓	✓	✓
Index	✓	✓	✓	✗	✗	✓
Mat. View	✗	✓	✗	✓	✓	✓

Table 3: Subset of Database Object Matrix (*, called Macro instead)

Feature	SQLite	Postgres	DuckDb	Snowflake	Databriks	Oracle
Alter	✓	✓	✓	✓	✓	✓
Create	✓	✓	✓	✓	✓	✓
Drop	✓	✓	✓	✓	✓	✓
Comment	✗	✓	✓	✓	✓	✓
Describe	✗	✓	✓	✓	✓	✗

Table 4: Subset of DDL Command Matrix

the other databases[46]. Supported DDL features include *CREATE*, *ALTER*, *DROP*, *FLASHBACK* (equivalent to *UNDROP*), *TRUNCATE*, *GRANT*, *REVOKE*, and *COMMENT*[47].

4.7 Summary of Key Takeaways

This section concludes with two matrices that compare the database objects and metadata features each data system supports. The matrices are the outcome of this section's analysis and aim to provide key insights for defining minimum requirements for an alternative open table format.
Figure 3 showcases five out of 121 rows, each representing a distinct database object type. Though only Table and View database types are supported by all systems, other object types, such as Functions and Materialized Views, show more significant variance in support. For instance, Functions are universally supported except by SQLite, which highlights its simple and lightweight design. Similarly, Materialized Views are only supported by PostgreSQL, Snowflake, Databricks, and Oracle, reflecting their more advanced capabilities.
Figure 4 highlights the metadata-related features and shows the first 5 rows of 12 distinct commands examined. The DDL commands *CREATE*, *ALTER*, and *DROP* are supported across all systems, showing their core role in managing database objects. In contrast, features such as COMMENT and DESCRIBE have more limited support. The rows of Table 3 are organized by the level of support, with the five database objects highlighted representing the most widely supported types among 121 identified database objects. This points to a key insight from the analysis: the considerable variation across these systems makes alignment and seamless interoperability more challenging. However, it does provide valuable insights into establishing baseline support for an open table format, where adaptability and extensibility must be prioritized.
The full matrices are in Appendix A.2.

5 Experiments

Section 4 provided a comparative analysis of six data systems, focusing on object definitions and metadata management functionality. Depending on its intended use case, each system is optimized for either OLTP or OLAP workloads. However, such optimization does not necessarily extend to metadata operations.

This section outlines the design of an experiment that assesses the performance of the most common DDL features, as identified in Section 4, under synthetic workloads. This experiment aims to scope the performance requirements of metadata operations for an open metadata format.

5.1 Experimental Design

The experimental design roughly follows the general approach outlined by Raj Jain in his book on systems performance analysis [48]. This systematic approach streamlines the experimental methodology to prevent unintentional mistakes due to common pitfalls and gaps in knowledge about systems performance analysis.

The experiment will involve four of the 6 data systems discussed in Section 4, based on the simple criterion that they incur no additional costs for the authors. The four systems are SQLite, DuckDB, PostgreSQL, and Snowflake.

System Specifications

The experiment will include both hosted and local data systems. All systems, system versions, and Python connector versions are listed in Table 5:

Data System	DB Version	Connector	Cloud
SQLite	3.45.3	sqlite3-v3.45.3	False
DuckDB	1.1.3	duckdb-v1.1.3	False
Postgres	17.0	psycog2-v2.9.10	False
Snowflake	8.40.1	snowflake-connector-python-v3.12.3	True

Table 5: Data Systems and Versions

Each of the systems listed in Table 5 will run on a Mac machine. Table 6 provides an overviews of the system specifications.

Due to cloud provisioning, some hardware specifications are inaccessible for Snowflake. However, since DDL queries modify metadata in a transactional key-value store inside the Snowflake cloud service layer[34], the number of warehouses does not impact performance. Thus, Snowflake is configured with a minimum of two vCPUs/warehouses.

Specification	**M1 Mac**
Processor	M1
Clock Speed	3.2 GHz
Cores	8 (4 perf, 4 eff)
RAM	16 GB
Operating System	MacOS 15.0.1
Instruction set	ARMv8.4-A

Table 6: System specifications M1 mac. "Perf": High performance cores, "eff": High-efficiency cores.

Metrics, Parameters & Levels:
Response time provides insight into the performance of DDL operations across multiple systems. Therefore, to compare the runtime performance of the most common DDL features, the total elapsed **response time** of queries will be collected and evaluated. For Snowflake, a cloud-hosted system, response time is influenced by network latency between the client and Snowflake servers, which should be considered when interpreting results.

Table 7 outlines the parameters and corresponding levels used in the metadata operation performance experiment.

Parameters	**Levels**
Object Granularity	1, 1.000, 10.000, 100.000
DDL Type	CREATE, ALTER, SHOW, COMMENT
Object Type	Table

Table 7: Levels of Parameters

Object Granularity indicates the number of database objects created before the Alter, Show, and Comment DDL commands are executed. *DDL Type* specifies the specific DDL command executed. *Object Type* describes the database objects targeted, only TABLE in this particular experiment.

Instead of a standard brute-force approach, the experiment follows a targeted sequence of runs structured as follows: It begins with an exclusive assessment of *CREATE TABLE* operations across all specified levels of object granularity. This is followed by four DDL queries at each level to capture potential performance differences: two *ALTER* statements (add a column and add a comment) and two metadata read commands (list all tables via *SHOW* and list all tables via Information Schema queries). Note that for systems without a *SHOW* command, namely SQLite and PostgreSQL, the sqlite_master table and PostgreSQL system catalogs were queried instead. These queries directly target the underlying metadata, which is assumed to align with how a *SHOW* command would be implemented in these systems.

The experiment is structured as follows: Granularity * *CREATE* Table statements are executed, followed by each DDL command. The DDL execution sequence is repeated three times to ensure consistent results and minimize deviations in performance measurements. After a command is executed, the total elapsed runtime is recorded.

Workloads

These experiments focus on table objects as these are supported by all target data systems. Simple synthetic workloads that perform common DDL queries against varying object granularities are generated with a Python script that initializes language-to-DB connectors for all four systems and executes DDL commands through these connectors. The connectors simplify benchmark implementation by enabling native Python timing functionality across all systems and executing the same commands on different systems by passing connectors as parameters. We depend on python-to-db interaction overhead, which may be relative to connector implementations. However, we regard this as negligible to our experiments as it should not impact the runtime complexity of the results.

All workloads except '*CREATE*' are repeated three times to mitigate and investigate outliers resulting from conditions such as cold starts, cache behavior, and influence from external processes on the benchmarking machine. Furthermore, we drop the database schema when moving from one level to another, so each experiment starts with the same database state.

Contrary to SQL, Python is procedural and supports iterative logic through loops. Algorithm ***create_tables()*** shows how the experiment script loops over the object granularity and creates tables through a python-db-connector (conn).

create_tables$(conn, system, num\_objects)$

```
for $i = 0$ to $num\_objects - 1$ do
    query ← "CREATE TABLE $t\_\{i\}$
     (id INTEGER PRIMARY KEY, value TEXT);"

    start_time, end_time, query_time ←
      execute(conn, query, system)

    recorder.record(system, "CREATE", query,
     "TABLE", granularity, num_exp, elapsed,
    start_time, end_time)
end for
```

The alter commands are point queries targeting a specific table and adding a column. Comparing the *alter_tables()* algorithm to create_tables, notice that the core logic remains mostly the same with the object granularity loop removed and query change, which minimizes work when adding commands.

```
alter_tables(conn, system, granularity, num_exp)
  table_num ← random(0, granularity.value - 1)

  query ← "ALTER TABLE t_{table_num} ADD COLUMN altered_{num_exp} TEXT;"

  start_time, end_time, query_time ← execute(conn, query, system)

  recorder.record(system, "CREATE", query, "TABLE", granularity, num_exp, elapsed, start_time, end_time)
```

The data_recorder object records experiment runtime results with metadata like query text, command, and granularity. It defines enums for the levels in table 7, initializes result tables for each system, and records experimental result records in an SQLite database with the record() method. The SQL snippet below shows the table definition for experimental result records.

```
CREATE TABLE IF NOT EXISTS {system_name}(
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    system_name TEXT, ddl_command TEXT,
    query_text TEXT, target_object TEXT,
    granularity INTEGER,repetition_nr INTEGER,
    query_runtime REAL,start_time DATETIME,
    end_time DATETIME);
```

5.2 Results

This section presents the experimental results across different database systems and levels of object granularity. As outlined in Section 5, the focus is on evaluating the performance and scalability of *CREATE*, *ALTER*, *SHOW* operations under varying workloads. We disregard "DROP" despite being supported in all six systems and categorize it as a metadata point query similar to "ALTER." The results offer insights into the performance of metadata management strategies employed by each system: SQLite, DuckDB, PostgreSQL, and Snowflake.

The findings presented in this section provide the foundation for the discussion in Section 6, which will outline considerations for a standardized open metadata format.

DDL: Create

When creating only one table or a small batch, the initial runs take slightly longer due to one-time setup tasks, such as allocating resources and initializing internal structures. This initial overhead significantly impacts the average when only a few tables are created, such as at granularities of 1 or 10, making the first *CREATE* operations appear slower.
As seen in Figure 4, Snowflake, the only cloud-hosted system, maintained consistent response times regardless of object granularity, with a network delay of approximately 255 ms - calculated as the average response time from ten constant-time SELECT queries. Locally hosted systems DuckDB and Postgres also showed consistent performance. SQLite, however, had a linear increase in run time as the number of objects increased. Furthermore, DuckDB performed notably better in the run time of *CREATE* statements than the other systems. DuckDB had an average query runtime of 0.00017s, which is 33x faster than SQLite, 2.715x faster than Snowflake, and 6x faster than Postgres.

Figure 4 also shows significant outliers in elapsed runtime among the 100k *CREATE* queries executed across all systems. These may result from many factors, including external processes running on the experiment system, amortized constant operations on internal data structures, and unstable network connection.

DDL: Alter

A similar pattern can be observed for *ALTER* table queries in Figure 5. Snowflake is constant in running time and bound by network latency; SQLite increases in runtime as object granularity increases while DuckDB and Postgres remain constant, with DuckDB performing notably better than other systems. This indicates that each system's table creation and modification employ different metadata storage structures and designs, for which adding new entrants and manipulating existing entrants are affected.

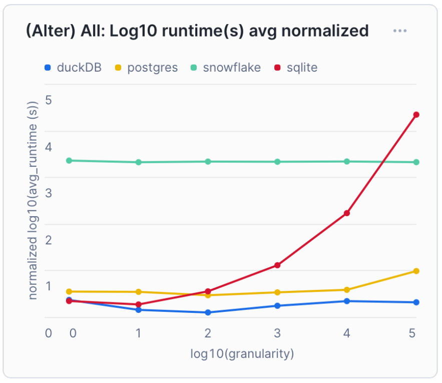

Figure 5: ALTER results with log10 scaled and normalized x and y axis.

DDL: Show

Figure 6 log scales and normalizes the runtime of *SHOW* commands to fit a single diagram. All systems, except Snowflake, show an expected linear increase in running time as the number of objects increases.

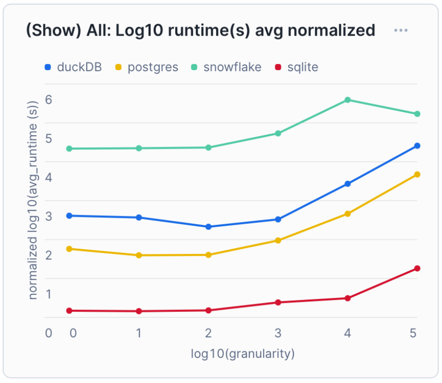

Figure 6: SHOW results with log10 scaled and normalized x and y axis.

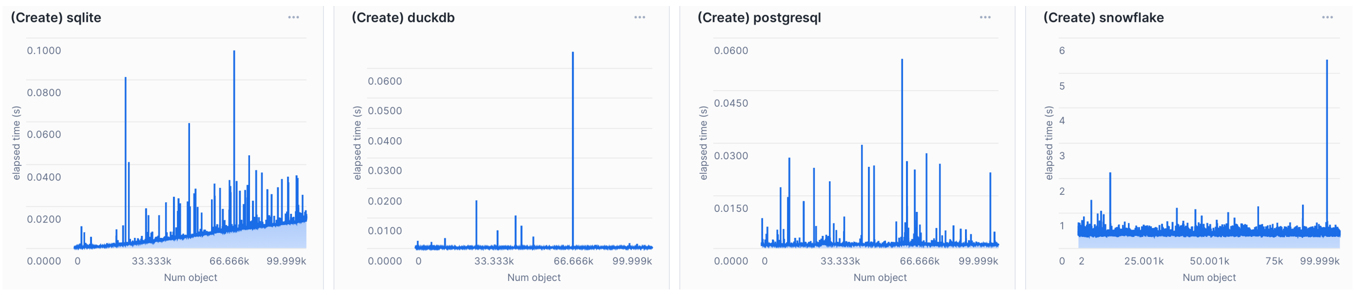

Figure 4: Elapsed runtime of Create table queries by number of objects for Sqlite, DuckDB, PostreSQL, and Snowflake.

Notably, Snowflake performs better at 100k than 10k, partly due to putting a 10k row restriction on *SHOW* commands. If this restriction was removed, we expect the same linear increase pattern observed with the information schema select command in appendix figure 13. Surprisingly, we observe an approximate 60% improvement in running time, going from 100k to 10k. This is a significant difference, assuming the returned rows are the same, and the only difference is whether the 10k limit is reached.Finally, SQLite performed notably better on *SHOW* queries than Snowflake and the local systems with an average *SHOW* query runtime of 0.000069s, 1066x faster than DuckDB, 193x faster than Postgres, and 16048x faster than Snowflake.
With *ALTER* and *CREATE* performance results in mind, the SQLite metadata management system seems to sacrifice point query performance in favour of sequential scan performance. Furthermore, scan performance becomes increasingly important when one seemingly performs one with every point query.

Variance Between Repetitions in PostgreSQL

Figure 7 shows all alter command results using Postgres. Total elapsed runtime is constant over all object granularities except 100k, which varies significantly between repetitions. This may result from the Postgres instance approaching the maximum allotted memory. We experienced some difficulty running the experiment with 100k granularity, which was solved by allotting the PostgreSQL instance additional memory and increasing max_locks_per_transaction. Both of these may cause issues if the instance contains a lot of tables[49].

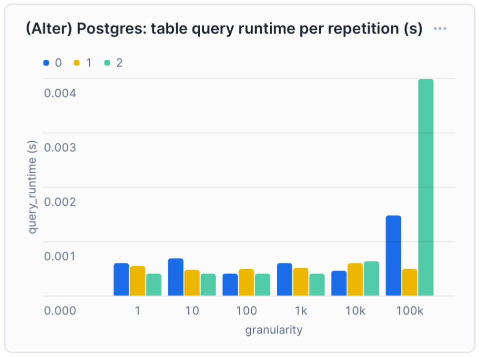

Figure 7: Postgres ALTER result for with individual repetitions.

6 Discussion

This section reflects on the key findings from the report, bringing together insights from the analysis of state-of-the-art formats like Iceberg and Delta Lake, the investigation of six data systems along with their metadata features and objects, and the results of experiments conducted on these systems. Because of disparities in performance results, it investigates the code bases of DuckDB and SQLite to find out how differences in metadata management approaches affect runtime complexity.
Additionally, it outlines key baseline requirements and considerations for a format intended to minimize dependence on external metadata layers, promote broader interoperability across systems, and act as an alternative or an extension to formats like Iceberg and Delta Lake, addressing limitations regarding dictionary metadata storage.
Finally, it acknowledges some of the limitations of the project and presents possible directions for future work.

6.1 How Metadata Structures Affect DDL Performance

As highlighted in the experimental results from Section 5.2, there were significant differences in execution response times across the systems. This section will explore the metadata structures of DuckDB and SQLite, discussing how these structures impact performance.

DuckDB

Reviewing DuckDB's source code reveals its approach to adding to and manipulating its metadata structure and how it supports consistent performance across all levels of object granularity.
The *CreateTable* method in the *Catalog* class ([50], line 142) of DuckDB starts the *CREATE TABLE* process by first retrieving the appropriate schema using the *GetSchema* function (line 143). As described in Section 4.3, DuckDB organizes its metadata at the schema level, meaning that when the correct schema is identified, it calls the schema's *CreateTable* function (line 139). The *CreateTable* is implemented in *duck_schema_entry.cpp*, which calls *DuckSchemaEntry::AddEntryInternal* ([51], line 100). Inside *AddEntryInternal*, DuckDB performs conflict checking to ensure no other entry, table in this case, with the same name exists. This is done using the *CreateEntry* (line 145) function on a *CatalogSet*,

DuckDB's internal structure for managing database object entries. The *CatalogSet*[52] is backed by a *CatalogEntryMap*, a key-value data structure that allows for constant-time lookups. This setup avoids needing a full scan of existing entries, giving consistent performance even as the number of tables increases.
Both adding new entries and updating existing ones are managed with constant time complexity. This explains why DuckDB maintains consistent performance for both *CREATE* and *ALTER* commands, regardless of object granularity.

SQLite

The experiment in Section 5 revealed two notable features of the SQLite metadata management approach. First, the runtime of point queries like *CREATE*, *ALTER*, and *COMMENT* scale linearly with the number of database objects, implying that a scan is executed. Second, listing all tables through *SHOW* functionality was highly performant, indicating some scan optimization or optimized metadata format.

To investigate findings from the experiment, we review the SQLite codebase. The function src/build.c/sqlite3StartTable() is called in response to a *CREATE* table statement. It constructs the table representation in-memory and generates the code that adds the table record into the main metadata storage structure - the schema_table, also known as sqlite_master[53][54]. As one of the first steps in table creation, sqlite3StartTable checks that the new table name does not collide with index or table names in the same database using sqlite3FindTable() and sqlite3FindIndex (lines 1260 and 1272), which locates the in-memory structure that describes a particular table. Given the name of the table, these lookups loop over all the SQLite databases and perform the sqlite3HashFind() operation to find the table name in a hash structure used to optimize lookups: *"Hash tblHash /* All tables indexed by name */"*[55][56]. This raises the question: Considering the use of in-memory hash structures, why do we still experience a linear increase in runtime for point queries, as seen in Figure 6 and 5?

In the interest of time, we leave the definitive diagnosis to future studies and present our current assumptions: sqlite3HashFind() calls findElementWithHash(*pH, *pKey) (hash.c lines 218 and 147), which searches the hash structure *pH* containing tables indexed by name, for the table name *pKey*. To this point, we assume a complexity of O(n) from SQLite3FindTable with n being databases, and amortized O(1) for tables because of the internal hash structure. However, there are two ways findElementWithHash may perform a near O(n) scan, with n being the number of tables. After searching the entire structure, the function returns the requested element if found; otherwise, it returns null. Since no table names are repeated in our experiment, the function should always return the null element.
findElementWithHash has two modes depending on the value of $pH \rightarrow ht$, where ht is a struct that stores a *count* - the number of entries in this hash bucket - and a pointer called *chain* that points to the first element with this hash[57]. First, *ht* may not be allocated, in which case it performs a linear scan on a linked list. Second, *ht* is set, where *htsize* defines the number of buckets in the hash table. When ht is defined, findElementWithHash() can jump to a bucket with a specific hash and then search that for the element, eliminating the linear search. However, many tables may end in the same bucket depending on the hash function, bucket size, and table names - all tables in our experiment are named T_*, with (*) being 1 through the value of object granularity. Finally, this could result in a more extensive scan when the number of tables becomes sufficiently large.

We leave the discussion at that but note that our performance results may be entirely related to another part of the system or experiments like locking behaviour or opening and closing database connections.

The second notable finding was the excellent performance of "*SHOW*" statements, which in SQLite is equivalent to performing the following query: *SELECT name FROM sqlite_master WHERE type = 'table';*. Instead of scanning an in-memory metadata data structure for all tables, it can leverage the speed of its select statements to output the list of tables. Furthermore, the sqlite_master table in our experiment should not have contained anything but the created tables. Thus, being a row-store, SQLite may gain performance from sequential prefetching.

6.2 Minimum Requirements for an Open Metadata Format

This section highlights the core requirements for an open metadata format, emphasizing the components necessary to support varying data systems and enable efficient metadata management. The aim is to define a foundation that addresses the gaps state-of-the-art solutions, Iceberg and Delta Lake, were not designed for while being consistent in performance, through tabular representations of the requirements within the three categories, database object support, database feature support, and performance.

Database Object Support

Table 8 shows the key database object types that such a format should support off-the-shelf. These object types are based on the most widely supported database objects identified in Section 3. Special focus is given to governance-related objects, as this was a notable limitation in existing solutions and thus their reliance on catalog meta-stores outside the data lakes.

Database Object	Supported Systems (out of 6)
Table	6
View	6
Function	5
Sequence	5
Materialized View	4
Database	3
Role	3
User	3
Connection	2
Schema	2
Share	2

Table 8: Minimum Required Database Objects for a Cloud-Based Open Metadata Format

As support for all 121 identified objects would be out of scope and unnecessary for most use cases, it is important to balance and prioritize what would suffice for a baseline implementation and then emphasize extendability during implementation.

All database objects supported simultaneously by both Snowflake and Databricks have been included, as they form essential components for interoperability in a cloud-native environment, or at least within the scope investigated in this work. This includes Table, View, Function, Materialized View, Schema, Connection, and Share. The most commonly supported objects across all six systems have also been included to ensure more extensive compatibility. Moreover, objects like User and Role have been included to provide access-control capabilities, which is lacking from Iceberg and Delta Lake.

Also, while traditional Index objects are supported by four out of six systems (all except Snowflake and Databricks), they do not inherently fit within data lakes. Data lakes store files, not rows, and rely on their statistic-based pruning techniques for optimization. However, if the goal were to enable seamless data and metadata migration out of the data lake, it could be reasonable to support index metadata, as it would allow external systems to reconstruct indexes. For now, however, the focus remains on ensuring high compatibility and extendability across systems within the data lake. Focusing on migrating the data would somewhat undermine this goal in a proof-of-concept implementation. However, we note that it may become a requirement for future work.

Database Feature Support

Certain database features must be included to support the operations required for the objects listed in Table 8. Table 9 lists the essential, or baseline, commands and operations that users and systems must be able to perform on the objects for consistent control.

Feature	Supported Systems (out of 6)
Create	6
Alter	6
Drop	6
Show	5
Describe	4
Grant	4
Revoke	4
Undrop	3

Table 9: Minimum Required Database Features for a Cloud-Based Open Metadata Format

The most common DDL operations, such as *CREATE*, *ALTER*, and *DROP*, are included to support basic object definition and modification. Additionally, to accommodate governance-related objects like User and Role included in the minimum supported objects, also features *GRANT* and *REVOKE* have been included for access control. *SHOW* and *DESCRIBE* provide essential visibility over the metadata, ensuring users can easily query and inspect the stored metadata instead of manually querying the format. Finally, support for *UNDROP* is included to remain consistent with the time travel features of Databricks and Snowflake as well as the Iceberg and Delta Lake formats. Once again, this is a take on the baseline essential functionality, where extendability remains a required priority in an actual implementation for further compatibility.

While this list defines baseline functionality, arguably, other features like *COMMENT* could be included to improve compatibility and usability. However, such features are currently seen as less critical in a minimal implementation.

Performance Requirements

Table 10 lists the expected performance standards for DDL operations. Since seamless interoperability is a key goal for a new metadata format, performance must stay consistent to prevent systems from disregarding its use, in which it would never become a ”standard.” Also, strong internal performance is necessary to offset network latency if the use case lies in cloud environments. The objective is not to achieve a specific target performance for metadata operations but to establish a reference baseline that aligns with the performance of other systems to serve as a guideline or evaluation metric for future implementation. Instead, the primary focus for implementation should be the runtime complexity of commonly used operations like *CREATE* and *ALTER* to guarantee consistent performance regardless of object granularity. Thus avoiding the linear scaling behaviour for point query operations observed in SQLite. However, we note that if a widely used system such as SQLite only guarantees O(n) runtime for metadata point queries on the experiment workload, the requirements defined in Table 10 could be relaxed depending on the workload priority - for example, whether the system is optimized for analytical, transactional, or a mixed-use case. Defining a workload priority does, however, propose trade-offs that are not considered here.

Operation	Complexity	Average Performance (Cloud)	Average Performance (Local)
Create Table	O(1)	0.46s	0.0010s
Alter Table	O(1)	0.32s	0.000755s
Show Tables	O(n)	-	-

Table 10: Baseline Performance Requirements for DDL Operations based on Snowflake (cloud) and the worst performing data system that adheres to the complexity requirement (local).

We can only specify minimum performance requirements for the operations evaluated in this project, such as *CREATE Table*, *ALTER Table*, and *SHOW*. However, minimum performance requirements for all objects and features supported should at some point be defined as well, but as these have not been tested, no proper baseline can be strictly defined. Furthermore, from these experiments alone, we cannot say whether the data systems include optimizations for tables and if they also apply to other database objects or, contrarily, other optimizations exist for those objects.
For now, *CREATE Table* and *ALTER Table* are bound to a constant complexity, with reference average response times in a cloud environment defined as 0.46 and 0.32 seconds, respectively. These values are based on the average response times measured across all object granularities in Snowflake, which includes an approximate 255 ms client-to-snowflake network latency. Table 10 also includes reference average response times for local systems, 0.0010s and 0.000755s, respectively. These are computed as the average runtime of all experimental runs with the respective command for the worst-performing DB that adheres to the complexity requirement.
Baseline performance expectations for *SHOW* do not include reference response time. Snowflake's average observed performance is 1107 ms (rounded), including network latency. However, as this is not a constant time operation, using an average from a varying object granularity gives considerably more weight to executions involving more objects, which would be misleading to reference - this is also the case for local *SHOW* performance. Instead, the requirements include an O(n) complexity bound across the investigated systems.

6.3 Initial Baseline and Design Considerations

To refresh the motivation: while formats like Iceberg and Delta Lake take advantage of the scalability and cost-efficiency of data lakes by decoupling table metadata from storage and compute and offering database-like features over data lake storage, they still rely on centralized meta-stores. Catalogs like Unity Catalog and Iceberg Catalog remain necessary for managing non-table metadata such as access control and governance, which introduces functional dependencies that limit full interoperability, as the catalog metadata is not necessarily compatible across different systems.
This section does not present systematically constructed architectural designs for a metadata management format but rather initial structural ideas that allow users to push their non-table database objects to their data lake storage, reducing vendor lock-in. Furthermore, supporting and integrating with commonly used data systems and engines poses additional challenges that may not be considered in this discussion.

Of the state-of-the-art systems investigated throughout this project, Apache Polaris proposes a solution closest to our vision. It allows users to use any Iceberg REST catalog-compatible engine to create, write to, and read from the same iceberg tables in Polaris. Furthermore, it manages RBAC through identity-related database objects like Role and Principle such that the security configuration is shared across engines. However, it stores its database objects inside an internal meta-store - in most cases, PostgreSQL or a transactional key-value store - not with the iceberg table data itself. Still, it serves as an inspiration for our format ideas.

Initial considerations include:

Can we store non-table metadata in current data lake formats? No. Some centralized global storage is in the nature of access control and governance, which is not directly practical through the current table-level architectures employed by Iceberg and Delta Lake. For example, extending Delta Lake with a global file instead of an external catalog would handle all non-table metadata as a single table, which would be inefficient in terms of the ACID properties implemented over delta tables. Reflecting over extending Iceberg with a root quickly shows a bottleneck in the reliance on atomic pointers to data lake files.
Insight: Since non-table metadata cannot be stored alongside table-level metadata, and extending current architectures with a root structure is not practical, introducing a modular meta-store that operates alongside existing architectures in the data lake seems the best approach. Alternatively, a re-implementation or core modification of formats like Iceberg or Delta Lake would be required, significantly increasing implementation complexity.

Is the dependency to a catalog external to the data lake then necessary? Yes. Addressing and endpoint definitions must be accessible for the systems using the format, meaning a centralized coordination layer is necessary. Alternatively, engines would need to be extended to store these pointers internally and implement a client library to support the relevant procedures in regards to maintaining the properties of current formats. Moreover, while Delta Lake can operate without an "external" catalog like Unity Catalog, if addressing is handled another way, extending a format like Iceberg would introduce challenges in replacing the reliance on atomic pointers for consistency if the catalog was to be removed.
Insight: A catalog external from the data lake is necessary. However, by reducing the responsibility of catalogs to just maintaining pointers and endpoints, interoperability would still be greatly improved. Nonetheless, the

intended outcome is a format for storing dictionary metadata in the data lake, not necessarily including anything but a future reference implementation conforming to these considerations.

We have explored multiple approaches to dictionary metadata storage throughout this report. SQLite stores all its database objects in a table called sqlite_master or sqlite_schema. Postgres stores its metadata in an array of tables dubbed system catalogs. The approach to a metadata storage format for the cloud that we envision in figure 8 stems from this approach of using database tables instead of an internal data structure for metadata management. Furthermore, we also note that we should design for extensibility. So, instead of mimicking the table structures from SQLite or Postgres, we store every metadata object in object-specific tables, aiming to make extending the metadata store with a new object as simple as defining a new table.
It is important to note that the minimum requirement for standardizing the format in figure 8 and 9 would be that all engines follow the exact specification for the data stored inside database objects and write dictionary metadata to the system tables in the same way. Therefore, future work should define standard and extendable specifications for data stored inside common dictionary metadata objects like User, Role, and Function.

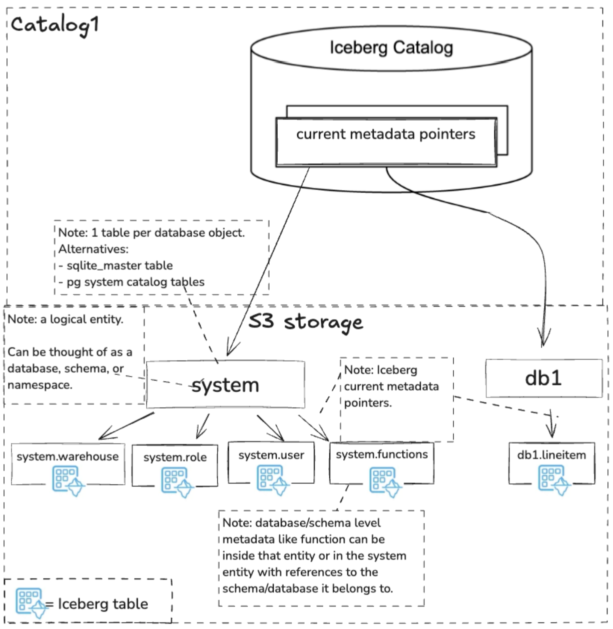
Catalog1
Iceberg Catalog
current metadata pointers
Note: 1 table per database object.
Alternatives:
- sqlite_master table
- pg system catalog tables
Note: a logical entity.
Can be thought of as a database, schema, or namespace.
S3 storage
system
Note: Iceberg current metadata pointers.
db1
system.warehouse
system.role
system.user
system.functions
db1.lineitem
Note: database/schema level metadata like function can be inside that entity or in the system entity with references to the schema/database it belongs to.
= Iceberg table

Figure 8: Initial design for a dictionary metadata format that uses Iceberg Tables

Figure 8 shows the metadata storage format. It stores metadata in Iceberg tables, thereby leveraging Iceberg's ability to operate on cloud object store data as well as its consistency, time travel, and rollback features. Database objects belong to the "system" namespace/schema/database, and regular table data belong to user-defined namespaces like *"db1"*. Note that objects like *functions* that may logically belong to a specific database or schema may also be stored next to the data tables. However, centralizing all dictionary metadata inside the system namespace and storing references in tables like "function" that specifies where the object belongs is preferred for simplicity and avoids mixing metadata with data.
Writing dictionary metadata to Iceberg tables in this way has several limitations. For instance, consider catalog1 a Spark JDBC catalog with Snowflake and Spark as engines. Spark can write dictionary metadata to the system tables in this setup, and Snowflake can integrate the catalog as an external catalog. However, Snowflake does not support writing to the tables of the external Iceberg catalog[58]. Hence, the format in figure 8 may have the limitation that only the engine associated with the internal catalog can directly write to the system table. However, depending on the use case and requirements, the significance of this limitation varies.
Figure 8 presents Roles and Users as example database objects in the system tables. However, implementing RBAC across engines may not be feasible without introducing an additional governance layer since engines would have to query the information in the data lake to check whether they can read/write from the data lake. Furthermore, moving security objects like Users and Roles to the data lake requires users to manage IAM policies on the object store to prevent tampering with permissions from within the data lake.

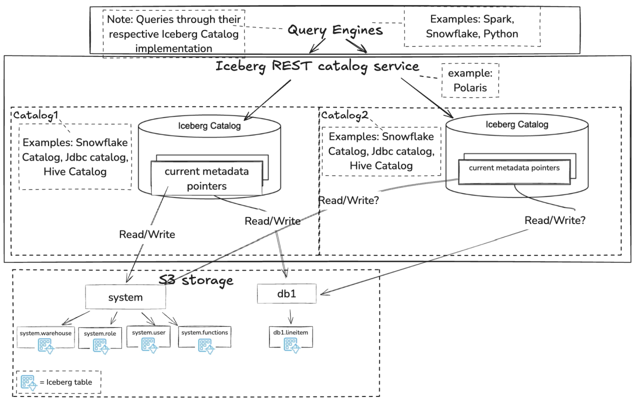
Note: Queries through their respective Iceberg Catalog implementation
Query Engines
Examples: Spark, Snowflake, Python
Iceberg REST catalog service
example: Polaris
Catalog1
Examples: Snowflake Catalog, Jdbc catalog, Hive Catalog
Iceberg Catalog
current metadata pointers
Read/Write
Read/Write
Catalog2
Examples: Snowflake Catalog, Jdbc catalog, Hive Catalog
Iceberg Catalog
current metadata pointers
Read/Write?
Read/Write?
S3 storage
system
db1
system.warehouse
system.role
system.user
system.functions
db1.lineitem
= Iceberg table

Figure 9: The format from figure 8 in a Iceberg REST Catalog Service

Figure 9 shows the format from figure 8 queried through an Iceberg REST catalog implementation that supports multiple Iceberg catalogs. It is inspired by Apache Polaris, which integrates different catalogs and performs RBAC on them and the tables inside, allowing cross-engine reading and writing. However, Polaris stores its database objects inside a Postgres or key-value meta-store and does not support extensions with additional database objects.
On a final note, while storing metadata in user-managed cloud storage gives users greater freedom and control over their data, it also removes certain fail-safe and security guarantees provided by managed platforms like Snowflake. This approach represents one attempt to integrate with the current state-of-the-art formats. However, alternative format specifications may forego the features of iceberg tables and define features like object serialization and time travel specifically for metadata.

6.4 Limitations & Future Work

When researching state-of-the-art formats and data systems as well as designing the metadata performance experiment, we aimed to produce a set of minimal guidelines and requirements for a dictionary metadata format. However, after presenting these along with our initial outlook on such a format, we realize that our focus may have become too wide, making it harder to find space and time to extend experiments and investigate findings extensively. Perhaps this report should have been logically split into 2:
First, a project that focuses on state-of-the-art systems and formats and further investigates these along with others like Apache Gravitino. Then, it could experiment with the performance and usability of Iceberg with different catalogs, followed by a discussion that proposes plausible architectures and interprets findings.
Second, a project that includes investigating different data systems with DDL/Object performance experimentation, including more objects and features, discussing the differences in performance/implementation approach, and presenting more extensive requirements. Thus allowing for further investigation and experimentation with systems.

On a different note, despite producing findings allowing for a useful discussion on metadata management approaches, the experiment lacks an additional cloud-based data platform to create equal ground for comparison with Snowflake. The exclusion of such a system was mostly to avoid the monetary cost and, thus, intentional. Although it could improve the quality of the experiment.

Even among open-source projects, documentation, blog posts, and other information regarding metadata management varies. In the case where no information is readily available, like for our investigation of SQLite and DuckDB, studying a large unknown codebase to find specific details is time-consuming.

Finally, the metadata operation performance experiment could be extended to include additional metrics like consistency or correctness under time travel queries. Furthermore, a study could investigate, build upon, and test the feasibility of the initial ideas for a dictionary metadata format shown in figures 8 and 9 or seek another direction that does not attempt to integrate with current state-of-the-art data lake formats.

7 Conclusion

This project explored state-of-the-art data lake table formats and examined metadata management approaches in widely used local and cloud data systems to provide minimum performance requirements and initial guidelines for a standardized dictionary metadata storage format.

The investigation of 6 data systems revealed notable differences in how and what metadata objects and features are supported. Furthermore, it identified 121 distinct database objects, suggesting that any universal standard must balance essential objects without introducing unnecessary complexity. Additionally, this variety in database object support signifies the need for extendability in creating an interoperable standard format.

We conducted a metadata operation performance experiment that evaluated the runtime performance of the most commonly supported metadata operations as the number of database objects increased in four popular data systems. The experiment demonstrated how metadata management approaches impact efficiency. For instance, on the *ALTER* and *CREATE* operations, DuckDB, Postgres, and Snowflake showed constant performance as the number of database objects increased; on the other hand, SQLite demonstrated a linear increase in execution time. However, SQLite performed significantly better than other systems on Show operations.
Based on our experiment results, we presented minimum metadata operation complexity requirements along with average performance guidelines for *CREATE*, *ALTER*, and *SHOW* commands.

Following the differences observed in runtime complexity on point query operations, an investigation of the DuckDB and SQLite code bases revealed that DuckDB uses a key-value structure to manage database object entries. SQLite stores all its metadata objects in a single table and maintains an internal hash structure for faster collision checks. These findings highlighted the impact of metadata structure designs on metadata management performance.

Drawing inspiration from SQLite's and Postgres' table-based approach to metadata object storage and Apache Polaris' cross-engine query and RBAC support, using the metadata object and operation requirements, we presented initial design ideas for an extendable in-data-lake dictionary metadata format using Apache Iceberg tables.

Finally, we acknowledge the opportunity for extension of our proposed requirements and the immaturity of our initial design ideas. Therefore, we encourage future work to extend our experiment with additional objects and features, to extend our database object requirements by defining standard and extendable specifications for what metadata is stored by common dictionary metadata objects, and to challenge the feasibility of our ideas for a data lake dictionary metadata storage format.

References

[1] M. Armbrust, T. Das, L. Sun, *et al.*, “Delta lake: High-performance acid table storage over cloud object stores,” *Proceedings of the VLDB Endowment*, vol. 13, Aug. 2020. DOI: 10.14778/3415478.3415560.

[2] Amazon Web Services, *What is apache iceberg?* Accessed: 2024-09-23. [Online]. Available: https://aws.amazon.com/what-is/apache-iceberg/.

[3] IBM Corporation, *What is a data catalog?* Accessed: 2024-12-06, 2024. [Online]. Available: https://www.ibm.com/topics/data-catalog.

[4] Google Cloud, *Data catalog: Overview*, Accessed: 2024-12-06, 2024. [Online]. Available: https://cloud.google.com/data-catalog/docs/concepts/overview.

[5] Snowflake Inc., *Information schema in snowflake*, Accessed: 2024-12-06, 2024. [Online]. Available: https://docs.snowflake.com/en/sql-reference/info-schema.

[6] Apache Iceberg, *Apache iceberg - catalog concepts overview*, Accessed: 2024-11-17, 2024. [Online]. Available: https://iceberg.apache.org/concepts/catalog/%5C#overview.

[7] PostgreSQL, *System catalogs - postgresql documentation*, https://www.postgresql.org/docs/current/catalogs.html, Accessed: 2024-10-02, 2024.

[8] Databricks. “Unity catalog: Fine-grained governance for data and ai.” Accessed: 2024-11-17. (2024), [Online]. Available: https://docs.databricks.com/en/data-governance/unity-catalog/index.html (visited on 11/17/2024).

[9] Wikipedia contributors, *Database object*, Accessed: 2024-12-06, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Database_object.

[10] Snowflake Inc., *Databases in snowflake*, Accessed: 2024-12-06, 2024. [Online]. Available: https://docs.snowflake.com/en/user-guide/databases.

[11] Oracle Corporation, *Database objects*, Accessed: 2024-12-06, 2024. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/Database-Objects.html%5C#GUID-31BE00A7-7FF9-41CB-852A-F1416912CA9E.

[12] Wikipedia contributors, *Data dictionary*, Accessed: 2024-12-06, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Data_dictionary.

[13] Snowflake, *Ddl summary*, https://docs.snowflake.com/en/sql-reference/sql-ddl-summary, Accessed: 2024-10-02, 2024.

[14] DuckDB Development Team, *Describe statement in duckdb*, Accessed: 2024-12-06, 2024. [Online]. Available: https://duckdb.org/docs/guides/meta/describe.

[15] PostgreSQL Global Development Group, *Information schema in postgresql*, Accessed: 2024-12-06, 2024. [Online]. Available: https://www.postgresql.org/docs/current/information-schema.html.

[16] IBM, *Hive metastore overview*, Accessed: 18 November 2024, 2024. [Online]. Available: https://www.ibm.com/docs/en/watsonx/watsonxdata/1.0.x?topic=components-hive-metastore-overview.

[17] Apache Software Foundation, *Apache iceberg documentation*, Version 1.5.2. [Online]. Available: https://iceberg.apache.org/docs/1.5.2/.

[18] A. S. Foundation, *Apache iceberg specification*, Accessed: 17 November 2024, 2024. [Online]. Available: https://iceberg.apache.org/spec/%5C#overview.

[19] The Apache Software Foundation, *Apache polaris*, https://polaris.apache.org, Accessed: 2024-12-06, 2024.

[20] Apache Software Foundation, *Entities - apache polaris documentation (unreleased)*, https://polaris.apache.org/in-dev/unreleased/entities/, Accessed: 2024-11-18, 2024.

[21] Apache Software Foundation, *Polarismetastoresession.java: Metadata store session implementation*, Accessed: 2024-12-06, 2024. [Online]. Available: https://github.com/apache/polaris/blob/main/polaris-core/src/main/java/org/apache/polaris/core/persistence/PolarisMetaStoreSession.java%5C#L49.

[22] M. Zaharia, A. G. 0002, R. Xin, and M. Armbrust, “Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics,” in *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*, www.cidrdb.org, 2021. [Online]. Available: http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf.

[23] SQLite Documentation, *SQLite Home Page*, https://www.sqlite.org/index.html, Accessed: 2024-10-09, 2024.

[24] SQLite Documentation, *SQL As Understood By SQLite*, `https://sqlite.org/lang.html`, Accessed: 2024-10-09, 2024.

[25] SQLite Documentation, *The Virtual Database Engine of SQLite*, `https://sqlite.org/vdbe.html`, Accessed: 2024-10-09, 2024.

[26] S. Overflow, *Stack overflow developer survey 2024 - most popular technologies - databases*, `https://survey.stackoverflow.co/2024/technology#most-popular-technologies-database-prof`, Accessed: 2024-10-02, 2024.

[27] PostgreSQL, *Architectural fundamentals*, `https://www.postgresql.org/docs/current/tutorial-arch.html`, Accessed: 2024-10-02, 2024.

[28] PostgreSQL, *Managing users and roles - postgresql documentation*, `https://www.postgresql.org/docs/current/user-manag.html`, Accessed: 2024-10-02, 2024.

[29] PostgreSQL, *Sql commands - postgresql documentation*, `https://www.postgresql.org/docs/current/sql-commands.html`, Accessed: 2024-10-02, 2024.

[30] DuckDB, *File formats*, `https://duckdb.org/docs/guides/performance/file_formats.html`, Accessed: 2024-10-02, 2024.

[31] DuckDB, *Why duckdb?* `https://duckdb.org/why_duckdb`, Accessed: 2024-10-02, 2024.

[32] DuckDB, *Drop statement*, `https://duckdb.org/docs/sql/statements/drop`, Accessed: 2024-10-02, 2024.

[33] DuckDB, *Duckdb%metadatafunctions*, `https://duckdb.org/docs/sql/meta/duckdb_table_functions`, Accessed: 2024-10-02, 2024.

[34] B. Dageville, J. Huang, A. Lee, *et al.*, "The snowflake elastic data warehouse," Jun. 2016, pp. 215–226. DOI: `10.1145/2882903.2903741`.

[35] Snowflake, *Create ¡object¿*, `https://docs.snowflake.com/en/sql-reference/sql/create`, Accessed: 2024-10-02, 2024.

[36] Snowflake, *Overview of key features*, `https://docs.snowflake.com/en/user-guide/intro-supported-features`, Accessed: 2024-10-02, 2024.

[37] Snowflake, *Metadata fields in snowflake*, `https://docs.snowflake.com/en/sql-reference/metadata`, Accessed: 2024-10-02, 2024.

[38] Databricks, *Getting started - databricks documentation*, `https://docs.databricks.com/en/getting-started/overview.html`, Accessed: 2024-10-06, 2024.

[39] Databricks, *Unity catalog metastore - databricks documentation*, `https://docs.databricks.com/en/data-governance/unity-catalog/index.html#metastore`, Accessed: 2024-10-06, 2024.

[40] Databricks, *Unity catalog - databricks documentation*, `https://docs.databricks.com/en/data-governance/unity-catalog/index.html#other-securables`, Accessed: 2024-10-06, 2024.

[41] Databricks, *Information schema*, Accessed: 2024-10-06, 2024. [Online]. Available: `https://docs.databricks.com/en/sql/language-manual/sql-ref-information-schema.html`.

[42] Databricks, *Describe statements*, Accessed: 2024-10-06, 2024. [Online]. Available: `https://docs.databricks.com/en/sql/language-manual/index.html%5C#describe-statements`.

[43] Q. Software, *What is oracle database?* `https://www.quest.com/learn/what-is-oracle-database.aspx`, Accessed: 2024-10-09, 2023.

[44] O. Corporation, *Schema Objects*. 2023, Accessed: 20-Oct-2024. [Online]. Available: `https://docs.oracle.com/cd/B13789_01/server.101/b10743/schema.htm`.

[45] O. Corporation, *Oracle database concepts, 21c: Logical storage structures*, Accessed: October 17, 2024, 2021. [Online]. Available: `https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/logical-storage-structures.html`.

[46] O. Corporation, *Oracle Database Concepts, 21c: Data Dictionary and Dynamic Performance Views*. 2021, Accessed: October 17, 2024. [Online]. Available: `https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/data-dictionary-and-dynamic-performance-views`.

[47] O. Corporation, *Oracle Database Concepts, 21c: SQL*. 2021, Accessed: October 17, 2024. [Online]. Available: `https://docs.oracle.com/en/database/oracle/oracle-database/21/cncpt/sql.html%5C#GUID-C25B548B-363A-4FE5-B4EE-784502BAAD08`.

[48] R. Jain, "Performance measurement methodology," in *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*, New York, NY: Wiley-Interscience, 1991, ch. 2.2, pp. 25–25.

[49] Heroku Support, *How can i resolve the error: "error: Out of shared memory. hint: You might need to increase* $max_locks_per_transaction$"?, Accessed: 2024-12-07, 2024. [Online]. Available: `https://help.heroku.com/EW2G2AF7/how-can-i-resolve-the-error-error-out-of-shared-memory-hint-you-might-need-to-increase-max_locks_per_transaction`.

[50] D. D. Team, *Duckdb catalog source code*, `https://github.com/duckdb/duckdb/blob/main/src/catalog/catalog.cpp`, Accessed: 2024-12-06, 2024.

[51] D. D. Team, *Duckdb duckschemaentry source code*, `https://github.com/duckdb/duckdb/blob/main/src/catalog/catalog_entry/duck_schema_entry.cpp`, Accessed: 2024-12-06, 2024.

[52] D. D. Team, *Duckdb catalogset source code*, `https://github.com/duckdb/duckdb/blob/main/src/catalog/catalog_set.cpp`, Accessed: 2024-12-06, 2024.

[53] S. Developers, *Sqlite schema table*, Accessed: 2024-12-07, 2024. [Online]. Available: `https://www.sqlite.org/schematab.html`.

[54] S. Developers, *Sqlite source code: Build.c*, Accessed: 2024-12-07, 2024. [Online]. Available: `https://github.com/sqlite/sqlite/blob/7e3e0311826a740e521ae23a232bdfad9969f3d1/src/build.c%5C#L1184C6-L1184C23`.

[55] S. Contributors, *Sqlite source code: Hash table implementation*, `https://github.com/sqlite/sqlite/blob/master/src/hash.c#L147`, Accessed: 2024-12-08, 2024.

[56] S. Contributors, *Sqlite source code: Sqliteint.h header file*, `https://github.com/sqlite/sqlite/blob/9f53d0c8179a3b69f788bd31749fc7c15092be87/src/sqliteInt.h#L1469`, Accessed: 2024-12-08, 2024.

[57] S. Contributors, *Sqlite source code: Hash.h header file*, `https://github.com/sqlite/sqlite/blob/9f53d0c8179a3b69f788bd31749fc7c15092be87/src/hash.h#L43`, Accessed: 2024-12-08, 2024.

[58] Snowflake, *Apache iceberg™ tables*, `https://docs.snowflake.com/en/user-guide/tables-iceberg`, Accessed: 2024-10-02, 2024.

A Appendix

A.1 Experiment repository

`https://github.com/Akongstad/study-dictionary-metadata`

A.2 Full object and command matrices

Feature	SQLite	PostgreSQL	DuckDB	Snowflake	Databricks	Oracle	Count
Alter	✓	✓	✓	✓	✓	✓	6
Create	✓	✓	✓	✓	✓	✓	6
Drop	✓	✓	✓	✓	✓	✓	6
Comment	✗	✓	✓	✓	✓	✓	5
Describe	✗	✓	✓	✓	✓	✗	4
Grant	✗	✓	✗	✓	✓	✓	4
Revoke	✗	✓	✗	✓	✓	✓	4
Show	✗	✓	✓	✓	✓	✓	5
Truncate	✗	✓	✗	✓	✓	✓	4
Use	✗	✗	✓	✓	✓	✓	4
Undrop	✗	✗	✗	✓	✓*	✓*	3
Refresh (Foreign)	✗	✗	✗	✗	✓	✗	1

Table 11: Feature Matrix (* indicates specific functionality, e.g., for tables or databases only)

Object	SQLite	Postgres	DuckDb	Snowflake	Databricks	Oracle	Count
Table	✓	✓	✓	✓	✓	✓	6
View	✓	✓	✓	✓	✓	✓	6
Function	✗	✓	✓*	✓	✓	✓	5
Index	✓	✓	✓	✗	✗	✓	4
Materialized View	✗	✓	✗	✓	✓	✓	4
Schema	✗	✓	✓	✓	✓	✗	4
Sequence	✗	✓	✓	✓	✗	✓	4
Procedure	✗	✓	✗	✓	✗	✓	3
Role	✗	✓	✗	✓	✗	✓	3
Trigger	✓	✓	✗	✗	✗	✓	3
User	✗	✓	✗	✓	✗	✓	3
Connection	✗	✗	✗	✓	✓	✗	2
Database	✗	✓	✗	✓	✓*	✗	3
Operator	✗	✓	✗	✗	✗	✓	2
Secret	✗	✗	✓	✓	✗	✗	2
Share	✗	✗	✗	✓	✓	✗	2
Type	✗	✓	✓	✗	✗	✗	2
Access Method	✗	✓	✗	✗	✗	✗	1
Aggregate	✗	✓	✗	✗	✗	✗	1
Alert	✗	✗	✗	✓	✗	✗	1
API Integration	✗	✗	✗	✓	✗	✗	1
Application	✗	✗	✗	✓	✗	✗	1
Application Package	✗	✗	✗	✓	✗	✗	1
Authentication Policy	✗	✗	✗	✓	✗	✗	1
Budget	✗	✗	✗	✓	✗	✗	1
Cast	✗	✓	✗	✗	✗	✗	1
Context	✗	✗	✗	✗	✗	✓	1
Cluster	✗	✗	✗	✗	✗	✓	1
Catalog Integration	✗	✗	✗	✓	✗	✗	1
Catalog	✗	✗	✗	✗	✓	✗	1
Collation	✗	✓	✗	✗	✗	✗	1
Conversion	✗	✓	✗	✗	✗	✗	1
Cortex Search Service	✗	✗	✗	✓	✗	✗	1
Compute Pool	✗	✗	✗	✓	✗	✗	1
Custom Classifier	✗	✗	✗	✓	✗	✗	1
Data Metric Function	✗	✗	✗	✓	✗	✗	1
Directory	✗	✗	✗	✗	✗	✓	1
Database Role	✗	✗	✗	✓	✗	✗	1
Domain	✗	✓	✗	✗	✗	✗	1
Dynamic Table	✗	✗	✗	✓	✗	✗	1
Dimension	✗	✗	✗	✗	✗	✓	1
Event Table	✗	✗	✗	✓	✗	✗	1
Event Trigger	✗	✓	✗	✗	✗	✗	1
External Access Integration	✗	✗	✗	✓	✗	✗	1
External Function	✗	✗	✗	✓	✗	✗	1
External Table	✗	✗	✗	✓	✗	✗	1
External Location	✗	✗	✗	✗	✓	✗	1
External Volume	✗	✗	✗	✓	✗	✗	1
Library	✗	✗	✗	✗	✗	✓	1
Extension	✗	✓	✗	✗	✗	✗	1
Failover Group	✗	✗	✗	✓	✗	✗	1
File Format	✗	✗	✗	✓	✗	✗	1
Foreign Data Wrapper	✗	✓	✗	✗	✗	✗	1
Foreign Table	✗	✓	✗	✗	✗	✗	1
Git Repository	✗	✗	✗	✓	✗	✗	1
Group	✗	✓	✗	✗	✗	✗	1
Hybrid Table	✗	✗	✗	✓	✗	✗	1
Iceberg Table	✗	✗	✗	✓	✗	✗	1
Index Type	✗	✗	✗	✗	✗	✓	1

Table 12: Part 1: Database Object Matrix (*, explained in comments as Database = Schema, Function = Macro)

Object	SQLite	Postgres	DuckDb	Snowflake	Databricks	Oracle	Count
Image Repository	✗	✗	✗	✓	✗	✗	1
Java Class	✗	✗	✗	✗	✗	✓	1
Java Resource	✗	✗	✗	✗	✗	✓	1
Java Source	✗	✗	✗	✗	✗	✓	1
Language	✗	✓	✗	✗	✗	✗	1
Link	✗	✗	✗	✗	✗	✓	1
Listing	✗	✗	✗	✓	✗	✗	1
Macro	✗	✗	✓	✗	✗	✗	1
Masking Policy	✗	✗	✗	✓	✗	✗	1
Materialized View Log	✗	✗	✗	✗	✗	✓	1
Model	✗	✗	✗	✓	✗	✗	1
Network Policy	✗	✗	✗	✓	✗	✗	1
Network Rule	✗	✗	✗	✓	✗	✗	1
Notebook	✗	✗	✗	✓	✗	✗	1
Notification Integration	✗	✗	✗	✓	✗	✗	1
Operator Class	✗	✓	✗	✗	✗	✗	1
Operator Family	✗	✓	✗	✗	✗	✗	1
Object Type	✗	✗	✗	✗	✗	✓	1
Object Table	✗	✗	✗	✗	✗	✓	1
Object View	✗	✗	✗	✗	✗	✓	1
Packages Policy	✗	✗	✗	✓	✗	✗	1
Package	✗	✗	✗	✗	✗	✓	1
Password Policy	✗	✗	✗	✓	✗	✗	1
Pipe	✗	✗	✗	✓	✗	✗	1
Policy	✗	✓	✗	✗	✗	✗	1
Profile	✗	✗	✗	✗	✗	✓	1
Privacy Policy	✗	✗	✗	✓	✗	✗	1
Provider	✗	✗	✗	✗	✓	✗	1
Projection Policy	✗	✗	✗	✓	✗	✗	1
Publication	✗	✓	✗	✗	✗	✗	1
Replication Group	✗	✗	✗	✓	✗	✗	1
Resource Monitor	✗	✗	✗	✓	✗	✗	1
Row Access Policy	✗	✗	✗	✓	✗	✗	1
Rule	✗	✓	✗	✗	✗	✗	1
Recipient	✗	✗	✗	✗	✓	✗	1
Server	✗	✓	✗	✗	✗	✗	1
Service	✗	✗	✗	✓	✗	✗	1
Storage Credential	✗	✗	✗	✗	✓	✗	1
Session Policy	✗	✗	✗	✓	✗	✗	1
Snapshot	✗	✗	✗	✓	✗	✗	1
Snowflake.ml.anomaly_detection	✗	✗	✗	✓	✗	✗	1
Snowflake.ml.classification	✗	✗	✗	✓	✗	✗	1
Snowflake.ml.forecast	✗	✗	✗	✓	✗	✗	1
Stage	✗	✗	✗	✓	✗	✗	1
Statistics	✗	✓	✗	✗	✗	✗	1
Storage Integration	✗	✗	✗	✓	✗	✗	1
Stream	✗	✗	✗	✓	✗	✗	1
Streamlit	✗	✗	✗	✓	✗	✗	1
Subscription	✗	✓	✗	✗	✗	✗	1
Synonym	✗	✗	✗	✗	✗	✓	1
Table Space	✗	✓	✗	✗	✗	✗	1
Tag	✗	✗	✗	✓	✗	✗	1
Text Search	✗	✓	✗	✗	✗	✗	1
Transform	✗	✓	✗	✗	✗	✗	1
Task	✗	✗	✗	✓	✗	✗	1
Tablespace	✗	✗	✗	✗	✗	✓	1
User Mapping	✗	✓	✗	✗	✗	✗	1
Virtual Table	✓	✗	✗	✗	✗	✗	1
Volume	✗	✗	✗	✗	✓	✗	1
Warehouse	✗	✗	✗	✓	✗	✗	1

Table 13: Part 2: Database Object Matrix (*, explained in comments as Database = Schema, Function = Macro)

A.3 Additional results

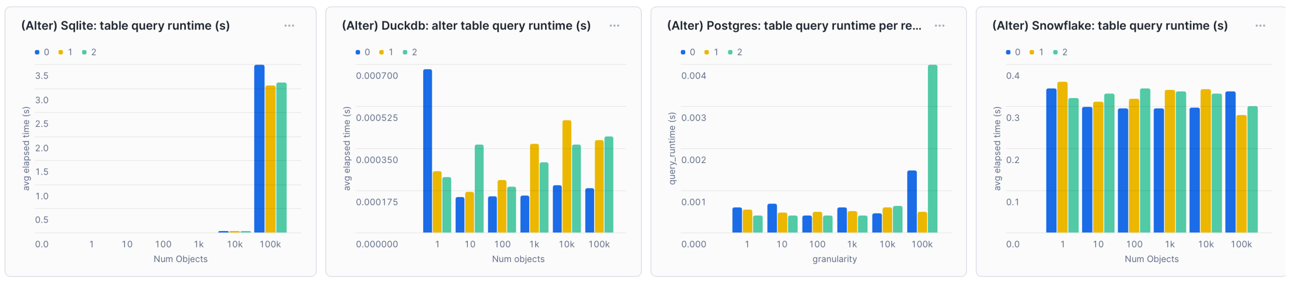

Figure 10: Alter results from all systems with repetitions.

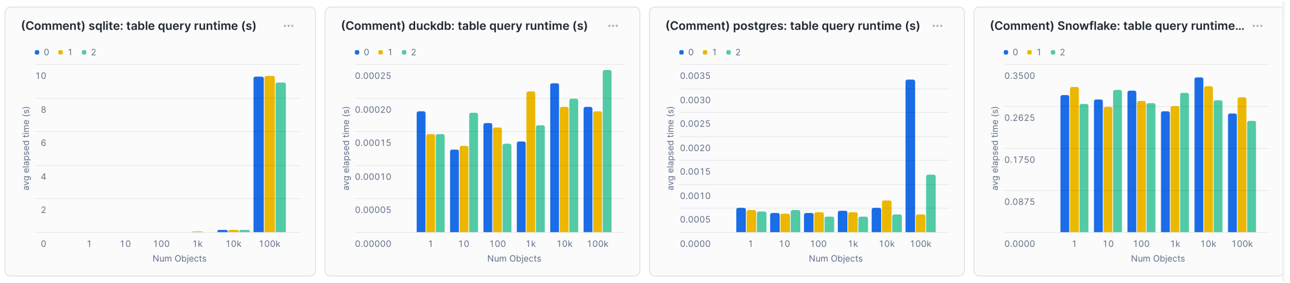

Figure 11: Comment results from all systems with repetitions.

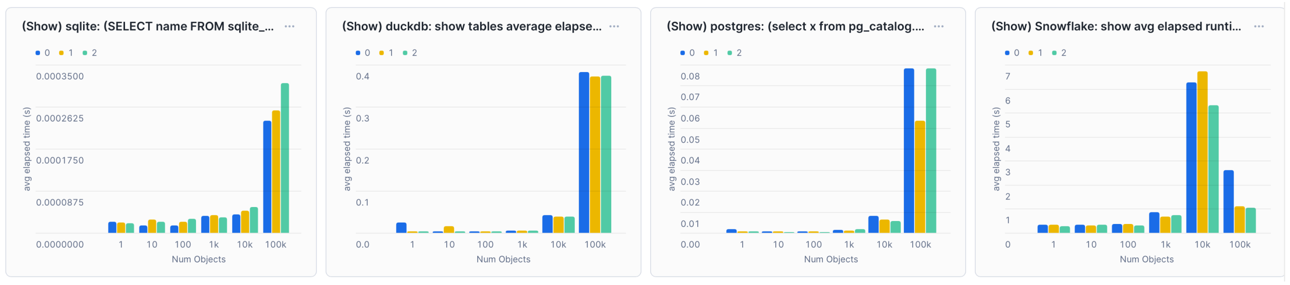

Figure 12: Show results from all systems with repetitions.

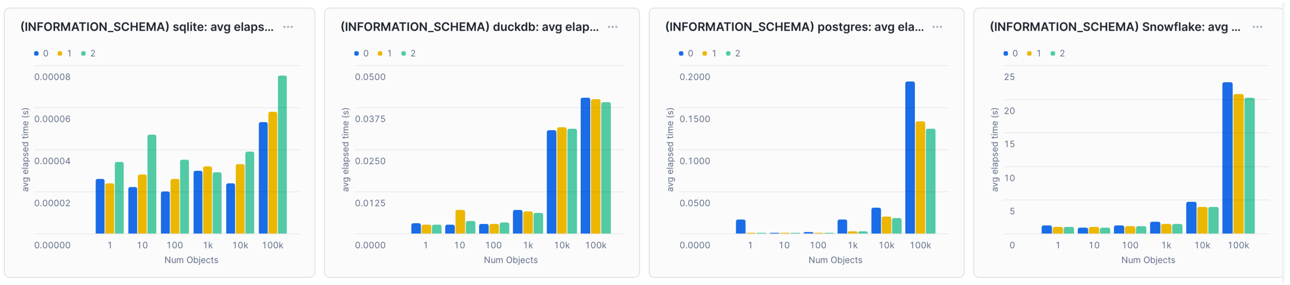

Figure 13: Information schema select tables results from all systems with repetition.