OpenDict: An Approach to Open Management of All Metadata Objects

A User-Centric Specification for Interoperable and Extensible Metadata Management

Andreas Kongstad (kong@itu.dk) & Carl Bruun (carbr@itu.dk)
Supervisor: Martin Hentschel
STADS code: KISPECI1SE

June 17, 2025

Abstract

Modern data architectures leverage open table formats (OTFs) to standardize table abstractions over flat files in data lakes, providing structure, transactional guarantees, and shared access across engines. However, OTFs are table-centric by design, leaving non-tabular metadata fragmented across proprietary metastores. As organizations increasingly adopt diverse workload-optimized engines, this fragmentation complicates interoperability, underscoring the need for a new approach to standardized metadata management.

This paper introduces the OpenDict REST catalog specification, a user-centric approach to open, extensible, and interoperable metadata management. Unlike existing OTF catalogs, which extend OTF specifications with varying fixed metadata objects, further fragmenting the ecosystem, OpenDict allows users to define custom metadata objects and mappings, linking them to platform-specific types and enabling cross-system metadata synchronization. We present a reference implementation that extends the Apache Polaris catalog, including client libraries for Spark and Snowflake. Performance experiments demonstrate strengths in storage efficiency and scan performance, supporting OpenDict's role as a flexible foundation for open, interoperable metadata management beyond tables.

1 Introduction

As modern data systems increasingly operate across diverse and cloud-native environments, managing and migrating metadata across platforms has become a key challenge. Open table formats (OTFs), such as Apache Iceberg, enable shared access to tabular data in cost-efficient and scalable object storage; however, their table-centric design leaves non-tabular metadata, including functions, users, and roles, largely unsupported. While catalog implementations, such as Apache Polaris, begin to address these gaps, operational support for broader object types remains limited and vendor-specific.

These challenges reveal gaps in current metadata management approaches, underscoring the need for a flexible and extensible solution – a direction this work explores through a, to our knowledge, novel API specification and reference implementation.

Multi-Platform Ecosystems and Increasing Complexity Enterprises increasingly adopt multi-cloud and hybrid cloud environments. For example, Flexera's 2023 annual cloud report found that 87% of enterprises had a multi-cloud strategy, rising to 89% by 2024 [1] [2]. In line with this trend, nearly all platform-as-a-service (PaaS) offerings have seen increased adoption, with warehouse usage (e.g., Snowflake, Databricks, Google BigQuery) growing from 56% in 2023 to 65% in 2024 [2]. Similarly, Accenture reported that in 2021, organizations used an average of five different cloud platforms, driven by the need to match workloads to the strengths of specific platforms: "Not all workloads perform equally on all clouds," they stated [3].

As organizations increasingly adopt multiple data platforms and multi-cloud strategies, metadata becomes more fragmented and complex to manage. According to Gartner, metadata management has evolved into a "highly distributed set of features and functionality requirements" [4]. A recent software market research report (2024) found that 74% of organizations maintain multiple metastores in parallel due to metadata fragmentation [5]. While multi-cloud approaches offer benefits such as a greater choice of features, optimized workload performance, and reduced risk of disruption, they also introduce complexity by creating isolated silos of data and applications. Metadata, often maintained in proprietary formats, further complicates migration from existing platform stacks and hinders the full potential of specialized cloud platforms [3]. To address fragmentation challenges, Gartner underscores the importance of openness: "Openness is no longer optional, says Gartner; it is mandatory" [6].

Open Table Formats as a Partial Solution While OTFs significantly reduces data fragmentation, data duplication, and vendor lock-in at the table level and makes strides toward interoperability and openness, these formats remain table-centric by design.

Some catalogs, however, extend capabilities beyond basic table pointer management and the features provided through the respective OTF specifications they implement. Apache Polaris, for instance, adds centralized role-based access control (RBAC) features applicable to data in the data lake through "credential vend-

ing" [7]. Similarly, the Apache Gravitino catalog complements Iceberg's REST specification with RBAC features and expands functionality beyond Iceberg tables. It supports managing assets from multiple sources and formats, positioning itself as a more general-purpose catalog management layer where Iceberg is just one option among many [8].

Nonetheless, these solutions represent only partial steps towards a unified metadata management solution – one that supports the essential objects and features identified in our prior research and detailed in Section 3.4. Moreover, one might argue that such extensions merely introduce new vendor-specific choices and lock-in, rather than moving toward a truly open standard. As Kyle Weller notes: "These data catalogs are the next extension of Databricks versus Snowflake competition" [9]. Moreover, this competition naturally conflicts with the principle of openness. For instance, while Databricks' Unity Catalog was open-sourced in 2024, it offered a significantly reduced feature set compared to its proprietary version [9].

Ultimately, OTFs and their catalog implementations – Polaris, Gravitino, and Unity Catalog, as the examples included here – illustrate both the progress and limitations in the broader effort toward a unified open standard for metadata management:

- Metadata is fragmented across increasingly specialized and distributed services.
- Siloed metadata across clouds hinders the full use of specialized platform capabilities.
- Open table formats promote interoperability and openness but remain limited to tabular metadata.
- Catalog implementations extend OTFs unevenly and impose vendor-specific choices.
- No unified standard for metadata exists across platforms and services.

Thesis Contributions and Structure In previous work, we conducted a study across six modern data systems to assess their support for metadata object types and management features, including benchmarking performance differences in metadata management approaches in widely used database systems. The study identified 121 distinct metadata object types, illustrating that implementing standardized schemas for all of them would be impractical and unnecessary for most use cases. These findings emphasized the need for an extensible approach – one that can adapt to diverse and evolving requirements without relying on predefined schemas.

This thesis builds on that foundation by designing and implementing a reference extension for Apache Polaris, an open-source Iceberg REST catalog. We introduce the concept of user-defined objects (UDOs) and propose the *OpenDict Specification* to enable extensible and interoperable metadata modeling, not limited to tables. Our contributions include two engine client libraries, the OpenDict Specification, and a Polaris-based reference implementation serving as a proof of concept. In addition, we benchmark metadata operations across platforms to evaluate the performance characteristics of our solution, recognizing that widespread adoption of an open standard depends not only on feature coverage but also on practical performance.

The paper is organized as follows: Section 2 addresses inconsistencies in terminology to establish a common language throughout the paper. Section 3 provides background information on metadata management approaches, investigates state-of-the-art OTFs and catalog implementations, and reviews prior research findings. Section 4 presents the design of our proposed OpenDict Specification, Polaris extension, and two client libraries to facilitate engine-to-catalog communication. Section 5 Outlines the design and results of our metadata operation performance experiments, followed by Section 6, which discusses results, limitations, and future work of our contribution. Finally, Section 7 concludes the thesis.

2 Terminology

This section presents a set of definitions used throughout this paper to establish a common language and address inconsistencies in online resources. Catalog, Database object, metastore, and DDL were initially defined in previous work by the authors[10] and are reused and expanded here.

Catalog IBM and Google define a *data catalog* as "a detailed inventory of all data assets in an organization" [11] [12]. The PostgreSQL *system catalogs* store schema and internal bookkeeping metadata [13]. ANSI uses "catalog" to refer to a database [14]. In Apache Iceberg, a *catalog* is an entity that tracks tables and their metadata pointers [15]. Unity Catalog stores hierarchical "*Catalog*" objects containing schema objects, which in turn contain tables and views [16]. This paper uses an OTF catalog definition defined as any entity that tracks table metadata for an OTF, with some providing additional governance features.

Metadata Object Multiple sources define *database objects* as a structure for storing, managing, and presenting data or metadata in a database. Some also distinguish between schema objects and non-schema objects [17]–[19]. We use the term *Metadata Object* to describe a structure that contains metadata. It can be thought of as an instantiated class in an object-oriented programming language with metadata stored in the fields. Examples are tables, views, and functions.

Metastore Hive Metastore is a service that stores metadata using a relational database [20]. BigQuery Metastore from Google is a managed Iceberg and BigQuery storage catalog [21]. In Unity Catalog, database ob-

jects at three levels are stored within a *metastore* container [22]. This paper employs a broad definition of metastore, which encompasses any storage entity that holds metadata objects. Relating our metastore to our catalog definition, Iceberg REST catalog services such as Polaris are services that contain a metastore, e.g., a Postgres backing store.

Open/Closed Metadata Management *Closed metadata management* attributes include the utilization of proprietary or vendor-specific APIs and storage formats that may not be shared or documented. Objects and features are not extendable, with limited opportunity for interoperability or migration.

Open metadata management attributes include being community-driven, offering opportunities for extending object definitions, features, and APIs, accessibility under open licenses, and utilization of open formats, enabling interoperability and facilitate easy migration.

Data Definition Language (DDL) and Metadata Operations Commands used to create or manipulate database objects: *CREATE*/*ALTER*/*DROP*. Additionally, Snowflake includes *DESCRIBE*, *SHOW*, and *USE* [23] while these are categorized as meta queries by DuckDB [24]. We use the term metadata operations, which includes DDL queries and refers to any form of management query on metadata objects.

Term	Description
Catalog	A catalog tracks and manages table metadata. Some also add additional governance features and metadata.
Metadata Object	Objects such as Table, Function, and User that store metadata and can be manipulated via metadata operations.
Metadata Operations	Any form of management query on metadata objects including DDL.
Metastore	Any entity, including relational databases, key-value stores, or similar, that stores metadata objects.
Open Metadata Management	Metadata managed in an open-source and community-driven manner with extensible and interoperable metadata objects and features.

Table 1: Terminology overview.

3 Background and Related Work

Modern data architectures are shaped by a mix of processing engines, distributed storage, and the growing need for interoperability. As data becomes increasingly fragmented across specialized systems, metadata management becomes both more important and more complex. In response, new technologies have emerged to support more interoperable and open approaches to metadata. Among these are OTFs, which offer standardized ways to access and manage tabular data across different tools and systems.

This section introduces general approaches to metadata management, OTFs, and catalog systems such as Polaris and Unity Catalog. Finally, it concludes with an overview of prior research findings, which provide an empirical foundation and contextualize the base requirements that influenced system design decisions.

3.1 Metadata Management Approaches

Database system options today are highly diverse, with specialized features fitting most use cases. As of May 2025, DB-Engines ranks 424 unique active DBMSs by popularity [25]. With system diversity comes diverse approaches to metadata storage and management. These approaches are reflected in the architectural choices and priorities of individual systems; for instance, an in-process single-user database may be best served with a simple in-memory metadata representation, whereas cloud-native data warehouses prefer independently scalable metadata services.

Previous work conducted a high-level investigation of metadata objects and management features in six widely used database systems: SQLite, DuckDB, PostgreSQL, OracleDB, Snowflake, and Databricks [10]. This section expands that work to establish three main categories of metadata object management in modern data systems. However, we limit the scope of the categorization to systems relevant to our experiments and investigation, as well as related work.

3.1.1 Table Metastore

Database systems persist and manage data. Therefore, a natural approach for databases is to store metadata objects using their own table formats, thereby benefiting from all the standard features offered by the database, including transactional safety and query optimization with minimal added complexity.

For instance, SQLite, PostgreSQL, and MySQL employ a table-based metastore model [26] [13] [27]. SQLite is an open-source, lightweight, self-contained transactional database engine in which the complete state of the database is contained within a single file on disk [28]–[30]. It stores all its metadata objects as rows in a single reserved table: *sqlite_schema* (formerly *sqlite_master*) [26] [31]. A similar approach is employed by PostgreSQL, an open-source object-relational database optimized for transactional workloads, and, according to the Stack Overflow Developer Survey 2024, the most popular database in the world among developers [32]. PostgreSQL stores metadata objects across 64 *system catalogs*, which are standard PostgreSQL tables that can be modified using SQL [13].

3.1.2 In-memory Metastore

Data locality and data access latency are key areas in software optimization. Some systems that embed the database in-process or do not require scalability may prefer to store metadata objects in-memory and rebuild on startup, allowing for low data access latency and flexibility in metastore data structure.

For instance, DuckDB, another open-source, in-process relational database, aims to adopt the simplicity in installation and in-process embedded operations known from SQLite while optimizing for analyt-

ical workloads [33]. DuckDB uses an object called *CatalogSet* for managing database object entries. The *CatalogSet* [34] uses a *CatalogEntryMap*, a key-value data structure to store objects[10]. We assume that when used in persistent mode, DuckDB serializes metadata objects to disk and reloads them into the in-memory map on startup, whereas SQLite, as an in-process database using a table metastore, reads directly from the persisted table without rebuilding an in-memory metadata structure. However, definitive information on how metadata objects are handled in persistent mode is not directly available through the documentation. Therefore, we conducted a small experiment to measure the average initialization time of DuckDB and SQLite with 1 and 100.000 stored database objects in persistent mode. We experienced a drastic increase in startup time, from 0.04 seconds to 1.18 seconds (a 29.5x increase) for DuckDB after reloading the in-memory metastore. In contrast, SQLite showed a much smaller increase of 3.3x, going from 0.004 to 0.012 seconds.

3.1.3 External Metadata Service

A common characteristic of large, highly scalable, distributed databases and data warehouse platforms is the separation of components, including a separate metadata service [35]–[37]. Separating system components into services allows systems with external metastores to scale independently at the cost of increased system complexity and communication overhead between components. The *External Metadata service* definition is arguably loose and includes any metastore that functions separately from the system using it.
For instance, Snowflake, a cloud-native data warehouse, employs a three-tier architecture with separate storage and compute tiers, supported by a control plane with various cloud services. Snowflake's metadata objects are stored in a scalable key-value store within the control plane [35], [38]. Similarly, Hive Metastore (HMS) functions externally to engines and data lake storage as a central repository for metadata, using a relational database, and provides client engines with access to metadata through a metastore service API [39]. Finally, an alternative approach, also categorized as an External Metadata service, is the catalogs in "Lakehouse" systems, providing additional metadata and governance support to the OTF metadata layer in data lakes [40].

Approach	Mechanism/Description	Examples
Table Metastore	Metadata is stored as tables within the database instance.	SQLite(sqlite_schema), Postgres(system catalogs)
In-memory Metastore	Metadata held in and in-memory data-structure	DuckDB's in-memory hash-k-v-store
External Meta-service	Metadata managed by a separate service or database.	Apache Hive Metastore, Snowflake's k-v store
External meta-service: OTF Catalogs	Tables metadata in files and metadata objects managed by external catalog services.	Iceberg/Apache Polaris, Delta Lake/Unity Catalog.

Table 2: Metadata management approach overview.

In conclusion, metadata object storage models reflect the overall architecture of data systems, of which there are many. Many traditional database systems store metadata objects in an internal metastore using their own table formats [13] [26] [27], which provides simplicity and transactional safety. Some lightweight in-process systems may prioritize low IO latency overhead obtained with in-memory data structures. Distributed and cloud-native systems are designed to be highly scalable and typically prefer external decoupled metadata services [35] [39].

3.2 Open Table Formats

OTFs are standardized table specifications designed to manage large-scale analytical datasets stored in cheap and scalable cloud object stores, such as Amazon S3 or Azure Blob Storage. By introducing a table abstraction over raw files – built on open and standard file formats like Parquet, ORC, and Avro – OTFs bring structure, versioning, and ACID compliance to data lakes. This allows data to be treated as managed tables rather than flat files, enabling both database-like features and interoperable data processing across multiple processing platforms [41], [42].

General Design Principles Internally, OTFs define a metadata layer that records schema definitions, partitioning layouts, references to underlying data files, file-level statistics, and version history, tracking all changes made to a table – including both DDL and DML operations. This metadata layer enables query optimization, consistent reads and atomic writes, schema evolution, and enforcement, and allows engines to audit or roll back to earlier table states [42], [43].
A key benefit of including metadata in data lakes is improved query performance: instead of scanning entire datasets, engines can skip irrelevant files using partition pruning and file-level statistics such as min/max bounds, which also helps to off-load some of the latency tied to external data storage [43]. Since OTFs are open specifications built on top of open standard formats, they are inherently engine-agnostic. This allows systems like Spark and Snowflake to read and write to the same table format without requiring data conversion or complex ETL synchronization [41], [43].

Purpose and Role in Modern Data Architectures The rise of OTFs can largely be attributed to the needs of modern data architectures. While cloud object stores are highly scalable and cost-efficient for storing large volumes of data – and can scale independently from compute – they remain simple key-value stores on their own with expensive operations that lack transactional properties [44]. For example, if a query were to delete all records across a table's Parquet files, concurrent readers may still access outdated data, leading to inconsistencies and a loss of data integrity. This has resulted in the adoption of the *Lakehouse Architecture* that – built on OTFs – brings features traditionally associated with warehouses to data lakes, combining the

benefits of low-cost storage over open standard formats and ACID-compliant management features [40].

The move toward OTFs began before today's data architectures were established. Data lake storage systems, such as S3 or HDFS, form the foundation for separate storage and compute, as opposed to traditional monolithic DBMSs. However, they only provide simple non-atomic interfaces. Apache Hive introduced an initial table abstraction on top of data lake storage using an external Hive Metastore with an RDBMS. However, it suffered from poor scalability due to its reliance on external metastores and limited transactional support. These limitations motivated the development of modern OTFs, such as Apache Iceberg and Delta Lake, forming a metadata layer within the data lake instead[40].

3.2.1 Iceberg

Apache Iceberg is an OTF project initiated at Netflix around 2017 and later donated to the Apache Software Foundation, becoming community-driven and fully open-sourced, designed to overcome limitations in Apache Hive's table format [45]. Iceberg brings database-like features to data lakes, including schema evolution and enforcement, hidden partitioning (where partition values are automatically derived and applied without requiring users to add filters), ACID-compliant transactions, time travel, and efficient query planning [46].

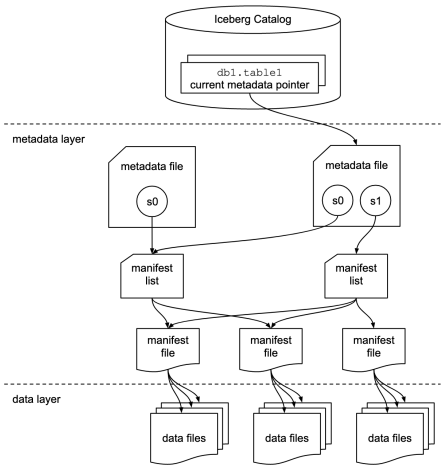

Figure 1: Overview of the Iceberg architecture [47].

Iceberg defines metadata layers to abstract over physical files. This layer is organized into a specific three-layer tree structure composed of metadata files, manifest lists, and manifest files. The internal structure can be seen in Figure 1.
Each component of the metadata layer holds a specific purpose, as described below [47]:

- **Metadata file:** Represents the whole state of the table, including its schema, partitioning configuration, custom properties (such as retention policies or write behavior), and a list of snapshots for both current and older versions – shown as pointers *s0* and *s1* in Figure 1.
- **Manifest list:** A snapshot points to a manifest list, which contains references to the manifest files included in that snapshot, acting as an index, along with metadata such as partition statistics and file counts.
- **Manifest files:** Track the actual data files of the table and include one row for each file, storing information such as partition values and file-level statistics. Manifest files are reused across snapshots to avoid rewriting metadata for files that have not changed.

All changes to the table state result in the creation of a new metadata file, which atomically replaces the previous one, providing optimistic consistency in which concurrent writers may need to retry commits if the "current" snapshot they based their write on has been updated.
In the object store, the metadata file is serialized as readable JSON and stored in a dedicated metadata folder for each table, using versioned filenames, e.g., $vxx.metadata.json$. All other components are stored in compact, open standard formats. Manifest lists and manifest files are stored in row-based Avro format, whereas data files are typically stored in columnar formats, such as Parquet or ORC [47].

3.2.2 Delta Lake

Delta Lake is another prominent OTF, developed by Databricks in 2017 and open-sourced in 2019, designed to bring ACID guarantees to data lakes, and has since then become one of the most popular OTFs in modern data architectures along Iceberg [45].

Delta Lake introduces a metadata layer in the form of a table abstraction called Delta Tables. One Delta Table wraps Parquet data files in a dedicated directory along metadata in a write-ahead transaction log. Delta Lake provides schema enforcement and evolution, partitioning, time travel, and efficient query planning [44] – the internal structure can be seen in Figure 2.

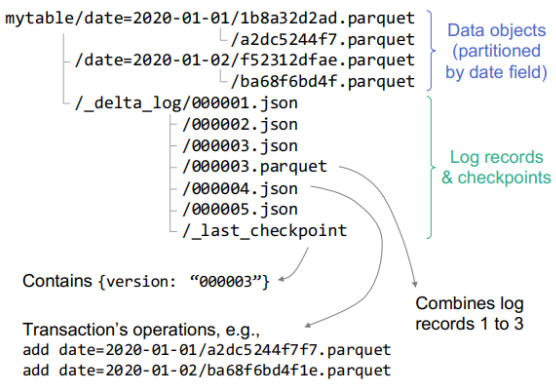

Figure 2: Sample Delta Table [44](page 3414, Figure 2).

A Delta Table is centered around its transaction log, supported by checkpoints and Parquet data files [44]:

- **Transaction log:** Stored in the *_delta_log/* subdirectory as incrementally numbered JSON files, shown as *Log Records* in Figure 2. Each record logs an atomic action: *add* (includes record count and column statistics), *remove* (marks files as deleted with a timestamp but retains the physical data – a 'tombstone'), or *metaData* (updates to schema or table configuration). The log tracks all changes and allows the reconstruction of a table's current or older states.
- **Checkpoints:** Periodically written Parquet files that compact the table state up to a given version (per default after every 10 transaction/log file). They reduce read overhead by summarizing log records and statistics and preserve 'tombstones' from removed files following retention policies.
- **Data files:** Stored in columnar Parquet format with unique, writer-defined names, referenced through the log rather than discovered through directory listings.

Delta Lake provides optimistic concurrency control when writing to the transaction log. Writers attempt to commit changes by appending a new log record but may have to retry if another transaction has already committed to that version. Since the log is append-only and versioned, Delta Lake inherently supports snapshot isolation and time travel by allowing clients to read the table as of a specific version or timestamp [44].

3.3 Catalogs

In the context of OTFs, a catalog is an external metadata service that maps table identifiers (namespace, table name) to physical metadata, acting as a centralized registry and a point of coordination for engines accessing the same dataset [48] [49].

According to Armbrust et al. (2021) [40], metadata layers were, at the time, a natural place to implement governance features such as access control. Today, governance, additional features, as well as metadata objects beyond tables are managed by catalogs, making them the most closely related systems in our work towards open interoperable metadata management.

While acknowledging the existence of many well-developed catalog options, this section focuses on Snowflake's Polaris, an Iceberg REST catalog, and Databricks' Unity Catalog, primarily a Delta Lake catalog.

3.3.1 Iceberg REST Catalog: Polaris

Iceberg supports multiple catalog types, including JDBC (Java-to-database API connector) and REST catalog implementations, each differing in how they store and expose metadata. JDBC-based catalogs use relational databases like Postgres that engines can access over a JDBC connection. REST catalogs, on the other hand, decouple the client from the underlying storage of the catalog and expose a standard HTTP API that lets engines access table metadata through a standard interface and not a database-specific connection. [15]

Apache Polaris is an open-source implementation of the Iceberg REST catalog specification. Polaris serves as a centralized catalog layer between query engines – such as Snowflake, Flink, and Spark – and Iceberg tables stored in cloud object storage. Each table is referenced through an atomic pointer to its current metadata file, allowing consistent access across engines. A high-level overview of the Polaris ecosystem can be found in Appendix A.8.
Polaris extends the Iceberg REST specification with role-based access control (RBAC) through 'credential-vending.' That is, by assigning 'principal roles' to 'principals' (entities encapsulating connection credentials), Polaris governs which operations a user is authorized to do [50]. Furthermore, Iceberg's atomic table pointers and additional management metadata objects are stored in a metastore using a relational JDBC connector with support for PostgreSQL and H2 databases[51].

3.3.2 Delta Lake Catalog: Unity Catalog

Unity Catalog is Databricks' default catalog and governance layer for managing delta tables and other assets across workspaces. It provides a centralized metastore in a three-level structure (*catalog.schema.table*), where a schema corresponds to what is typically called a database [16].

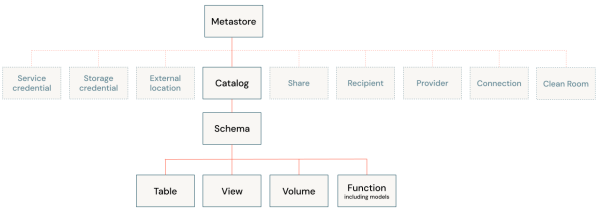

Figure 3: Unity Catalog structure [16].

As shown in Figure 3, Unity Catalog also stores configuration metadata such as storage credentials and external locations at the same level as catalogs. These are used to support temporary credential vending for external engines such as Spark, Trino, and DuckDB [52].

An inspection of the Unity Catalog Table layer shows that Unity Catalog stores governance metadata such as the table name, owner, data format, and storage path. In contrast, transactional metadata, such as schema definitions, version history, and statistics, is stored in the Delta Lake transaction log. The whole Unity Catalog Table schema can be seen in Appendix A.11. This allows Unity Catalog to act as a pointer registry to the OTF table abstractions, much like Polaris.

3.3.3 Wrap-up: Catalog Maturity and Integration Considerations

Following the investigation of Apache Polaris and Unity Catalog, it should be noted that Unity Catalog is a more mature catalog than Polaris. In addition to OTF table pointer addressing, RBAC, and credential vending, Unity Catalog offers a broader set of governance features, including a UI for data discovery, lineage tracking, auditing, and support for a broader range of object types such as files, AI models, and functions [16].

While Unity Catalog offers a broader range of management features, including support for Iceberg and Hudi tables through its UniForm abstraction, it remains tightly integrated within the Databricks environment. The proprietary version of Unity Catalog only allows external read access to managed tables via its Open APIs. In contrast, Polaris supports both read and write access through all engines that implement an Iceberg REST client. Alternatively, the less feature-rich open-source version of Unity Catalog can be self-hosted and configured to allow both read and write access with objects stored in a PostgreSQL, MySQL, or H2 metastore[53]. However, it supports only explicitly implemented integrations such as Spark, DuckDB, and Trino, with current documentation only showing examples of write support through the Spark connector [54] [9].

As a final note, as opposed to Iceberg and Polaris, Delta Lake does not hold the same functional reliance on Unity Catalog – isolation is, as earlier mentioned, handled through the transaction log. That is, while Unity Catalog provides deeper governance and management features for Delta Tables, the paths/pointers to Delta Tables could, in theory, be stored anywhere, whereas Iceberg relies on its atomic catalog pointers for ACID compliance.

3.4 Empirical Foundation: Prior Research and Motivation

This subsection revisits earlier research that investigated the requirements for a standard open metadata format. A key contribution was identifying a set of minimum requirements for a baseline specification and implementation. The underlying motivation was the assumption that, for a standard to gain adoption without being immediately diminished, it should at least align with existing system capabilities in terms of supported metadata object types, functional features, and general performance.

By analyzing six prominent data systems, the study found significant variance in both the types of metadata objects supported and management features. The study included OLTP and OLAP-optimized systems, covering in-process, traditional, and cloud-native environments. While performance considerations are revisited later in this thesis, the diversity in system structures and functionality highlights the challenge of unifying metadata models.

Supported Database Object Types The research identified a total of 121 distinct object types supported across the six systems, covering everything from common entities like tables and views to more specialized constructs such as resource monitors, foreign data wrappers, and masking policies. Even among objects serving comparable roles, differences in naming and structure were common across the systems.

Database Object	Supported by
Table	6/6
View	6/6
Function	5/6
Sequence	5/6
Materialized View	4/6
Database	3/6
Role	3/6
User	3/6
Connection	2/6
Schema	2/6

Feature	Supported by
Create	6/6
Alter	6/6
Drop	6/6
Show	5/6
Describe	4/6
Grant	4/6
Revoke	4/6
Undrop	3/6

Table 3: Overview of commonly supported objects and features [10].

The findings, as summarized in Table 3, show the 10 most widely supported object types. Out of a total of 121 objects, only two were supported across all investigated systems. Furthermore, the rapid decline in support suggests that a fixed, universal format is neither scalable nor maintainable in practice – especially considering that many of these object types are unlikely to be used regularly. Instead, the results point to the need for a metadata model that is natively extensible and capable of adapting to the varying requirements of different systems now and in the future.

Supported Database Features and DDL All systems supported standard SQL DDL operations, *CREATE*, *ALTER*, and *DROP*. However, support for additional features such as *UNDROP*, *GRANT*, and *REVOKE* varied, as shown in Table 3.

For instance, in-process systems like SQLite exclude user management completely, making access control commands irrelevant. This variation underscores a key challenge for unified standardization: the operations available for managing metadata are closely tied to specific object types and the underlying system architecture. A metadata specification cannot assume which features are relevant across all platforms, nor can it blindly include support for a union of all possible feature sets. Therefore, the research pointed to standard DDL operations – *CREATE*, *ALTER*, and *DROP* – as baseline features, while support for additional functionality remains a contextual decision depending on the specification's intended use and environment.

Altogether, these insights suggest that defining a single, unified metadata specification without further increasing fragmentation or vendor-specific choices requires a minimal baseline for core operations while allowing ex-

tensibility in object types to reflect the diverse needs of the systems that use it.

4 System Design

Previous sections outlined the characteristics of state-of-the-art OTFs, their catalogs, metadata management approaches, and findings from prior research, which form minimum requirements for a standard open metadata format. These motivate a more extensible approach. Instead of enforcing a globally unified set of object definitions, the metadata specification should allow systems to define the types they need. In this model, object types are not fixed within the specification but dynamically defined and resolved by the implementing catalog – referred to as **UDOs**. These form the foundation of the main thesis contribution, the *OpenDict Specification*, and lay the groundwork for the system design introduced in this section.

Throughout this section, we take inspiration from the C4 model for visualizing software architecture [55] to convey the complete system architecture from different viewpoints, starting with an introduction to the complete system high-level context and containers, followed by a discussion of the detailed design of individual components: The Polaris OpenDict Extension and the Client Libraries. Together, the specification, reference implementation, and client library implementations serve as a proof of concept that addresses the gaps hindering a unified standard specification for interoperable metadata objects beyond tables.

4.1 Architectural Overview

This subsection presents a high-level overview of the systems in three steps. First, we describe the operational context. Second, we detail the overall system architecture, including the containers within the system, their core components, and deployment environments. Finally, we visit the underlying design and key decisions that guided the implementation.

4.1.1 Context

The user of the Polaris Iceberg REST catalog, extended with the OpenDict REST API specification, utilizes several engines, such as Spark and Snowflake, to query Iceberg table data stored in a cloud object store. Figure 4 provides context for OpenDict Polaris, its client libraries, and the systems they interact with. The user interacts with Spark or Snowflake through Python Applications that import the opendict-spark and opendict-snowflake client libraries (1). In Spark, the user inputs our SQL-like syntax to **define** the structure of a UDO, **create** an object instance, and provide a Snowflake platform mapping that **maps** the metadata fields of the instantiated object to SQL syntax recognized by the Snowflake Parser (2,3). The object definition, instantiated object, and Snowflake platform mapping are all persisted in an S3 bucket using the Iceberg Table format, entirely managed by the user (4). Now, from Snowflake, the user inputs the **sync** command (5) into the OpenDict library, which generates an object dump using the platform mapping and executes it in Snowflake, effectively creating a Snowflake Function object (6). Finally, the user can execute the function created from Spark through any Snowflake interface (7).

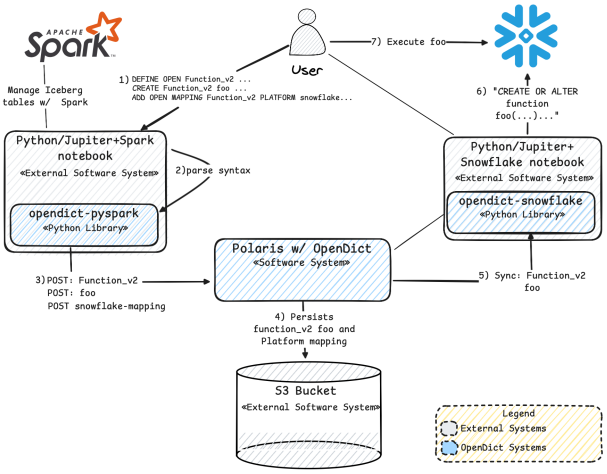

Figure 4: Context Diagram – system and external component interaction.

The legend in Figure 4 distinguishes between "external systems" and "OpenDict systems," referring to systems developed by separate parties and by us, respectively. Subsection 4.3 and 4.4 will explore the components of the "OpenDict Systems" (blue) from Figure 4 and 5 in further detail.

4.1.2 System Architecture

The OpenDict specification defines an API that any Iceberg REST catalog can implement. A significant challenge in standardizing metadata object management is adoption. The OpenDict Polaris reference implementation minimizes modification of existing codebases so that it does not depend on contributions to open-source or proprietary systems. The architecture is organized into three layers, as depicted in Figure 5, with inter-layer communication via RESTful interfaces, aside from engine and library interaction, which is handled by Python connectors.

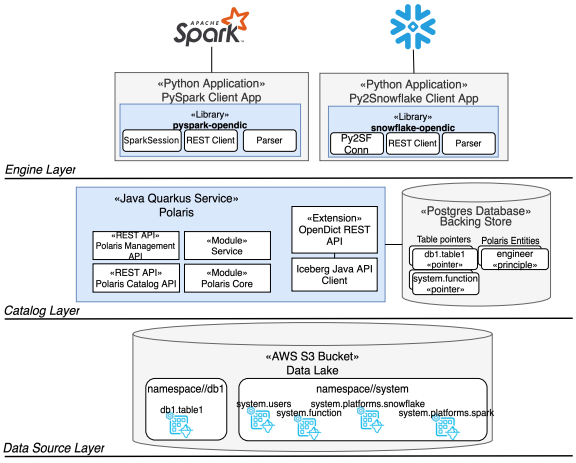

Figure 5: Layered Container Diagram – system architecture.

- **Engine Layer** This layer contains the components that a user interacts with directly. Python scripts, notebooks, or applications that import client libraries wrapping engines' Python connectors to facilitate communication with the catalog layer through defined OpenDict SQL-like syntax and query parsing.
- **Catalog Layer** This layer houses the Iceberg REST catalog instance and its backing store. The OpenDict API extends the standard Iceberg REST catalog API, defines how objects and mappings are persisted in the persistence layer and translates objects to engine-compatible syntax based on user-provided mappings.
- **Persistence Layer** This layer stores metadata objects and mappings. The reference implementation uses the Iceberg Java API client to persist objects in Iceberg tables. The storage implementation can be any Polaris-compatible file or object storage backend, including local filesystem, S3, Azure Blob storage, or Google Cloud storage.

4.1.3 Design Decisions

We present the final system design by outlining key decisions that shaped the OpenDict specification and the architecture of its reference implementation.

Creating an Open, Universally Compatible, and Extensible Format Our goal was to design a system that addresses the challenges of storing and managing metadata objects across heterogeneous data systems. Since most systems store metadata objects in proprietary or system-specific formats, we initially proposed an open universal specification for metadata within common objects as listed in Section 3.4, Table 3. The process involved crawling the documentation of individual data systems, collecting metadata object specifications, and computing their union to create a theoretically compatible schema systematically – our initial program to compute unified metadata object specifications is referenced in Appendix A.7. However, this approach had three main drawbacks: 1. The unified object specifications contained multiple fields of comparable purpose; 2. Users typically require only a subset of the crawled systems' fields; 3. Extending support to new objects or systems required considerable manual effort.

Instead, UDOs allow users to define custom objects with precisely the fields required for compatibility with target systems and engines. To accommodate syntax differences across systems, users may also specify platform mappings from UDO metadata to valid system-specific syntax.

The value of this extensibility becomes particularly clear in the context of increasingly specialized and fragmented data platforms. For example, if Snowflake introduces a new metadata type unknown to other systems and not covered by the specification, a fixed universal model would require changes to the specification itself, including client libraries, or would ignore the new type entirely. By contrast, a UDO-based specification allows such extensions to be adopted immediately.

Extension Over Modification One of our primary design goals encouraged extension over modification in integration with Iceberg REST catalogs and engines through client libraries. We reuse an argument similar to Iceberg REST catalog vs. standard catalogs: Instead of using technology-specific logic in the engines to interact with our catalog extension [15], we wrap engine connectors with reusable Python libraries. Additionally, the Polaris reference implementation extends the existing code base without modifying its core. This design principle brings modularity, which significantly reduces implementation complexity, allowing independent modifications and upgrades without conflicts.

Moving Non-Table Metadata Objects Outside Catalog-Managed Metastores The catalog options we have explored store pointers to current table versions along with proprietary management objects – e.g., Principals, Users, or Catalogs – in key-value or relational backing stores. Initially, the Polaris reference implementation stored UDOs inside the Polaris PostgreSQL backing store. Hence, we could extend the *PolarisBaseEntity* [56] and utilize the existing *MetaStoreManager* [57] abstraction to store our objects in any backing store supported by Polaris. However, this also came with several downsides. First, without modification to the core codebase, we were restricted to the current entity model and supported backing stores. Second, relying on Polaris-specific internal structures made the reference implementation less generally applicable. Instead, as OTFs already store data and tabular metadata in file storage controlled by the user, we aimed to move metadata objects into user-controlled standard Iceberg tables as well.

Method	Polaris Endpoint	Client Library SQL Syntax	Description
GET	/objects	SHOW OPEN TYPES	List all defined UDO types
POST	/objects	DEFINE OPEN <type> PROPS ...	Define a new UDO type and its schema
GET	/objects/{type}	SHOW OPEN <type>[s]	Retrieve all UDOs of a given type
POST	/objects/{type}	CREATE OPEN <type> <name> [PROPS ...]	Create a new UDO instance of a defined type
DELETE	/objects/{type}	DROP OPEN <type>	Delete all UDOs of the specified type
POST	/objects/{type}/batch	CREATE OPEN BATCH <type> OBJECT[s] <props>	Create a batch of UDOs
PUT	/objects/{type}/{name}	ALTER OPEN <type> <name> PROPS ...	Modify an existing UDO instance
GET	/objects/{type}/platforms	SHOW OPEN PLATFORMS FOR <type>	List platforms mapped to a UDO type
GET	/objects/{type}/platforms/{platform}	SHOW OPEN MAPPING <type> PLATFORM <platform>	Get mapping between a UDO type and a platform
POST	/objects/{type}/platforms/{platform}	ADD OPEN MAPPING <type> PLATFORM <platform> SYNTAX ... PROPS ...	Create a new platform-specific mapping
GET	/objects/{type}/platforms/{platform}/pull	SYNC OPEN <type> FOR <platform>	Generate SQL for all UDOs of a type for a platform
GET	/platforms	SHOW OPEN PLATFORMS	List all platforms with defined mappings
GET	/platforms/{platform}	SHOW OPEN MAPPINGS FOR <platform>	Retrieve all mappings defined for a platform
DELETE	/platforms/{platform}	DROP OPEN MAPPING FOR <platform>	Delete all mappings defined for a platform
GET	/platforms/{platform}/pull	SYNC OPEN OBJECTS FOR <platform>	Generate SQL for all mapped UDOs for a platform

Table 4: Mapping between SQL syntax and OpenDict API endpoints.

4.2 OpenDict API Specification

The OpenDict API Specification is a standalone OpenAPI 3.1.1 specification defining a standardized interface for managing user-defined metadata objects in OTF-based architectures. Just as Polaris introduces its own complimentary management specification to extend its catalog implementation with RBAC features, the OpenDict specification defines consistent metadata management beyond tables through an extensible object model and a RESTful interface.

The specification outlines a set of endpoints grouped as follows:

- **/objects:** Endpoints not tied to any specific UDO instance or platform. These are used for listing and defining UDO types in Polaris.
- **/objects/type:** Endpoints tied to a specific UDO type. Includes DDL operations for object instances and synchronization of a single UDO type.
- **/platforms:** Endpoints tied to a specific platform. This includes listing, retrieving, deleting mappings, and synchronization across all UDOs, regardless of type.

The full set of OpenDict endpoints, their corresponding SQL-like commands defined in the client libraries can be seen in Table 4.
While the endpoints listed above represent a logical grouping of operations according to the objects or platforms they target, the OpenDict Specification also defines a set of components that form the data model of the API. Central to this is the key concept of UDOs. The UDO component, as illustrated in Figure 6, is an object with a required name and type and a flexible props field containing optionally nested key-value pairs (column name, value) matching the object's schema. This structure enables UDOs to be dynamically managed, with key endpoints supporting definition, creation, mapping to platform-specific representations, and synchronization across platforms.

```
../opendict/spec/open-dict-service.yml
Components:
  Udo:
    required: [type, name]
    properties:
      type: string
      name: string
      props: object (dict[str, object])
      createdTimestamp: string
      lastUpdatedTimestamp: string
      entityVersion: integer
```

Figure 6: OpenDict OpenAPI UDO component.

In line with the prior research findings described in Section 3.4, the specification's management features include support for standard SQL DDL operations: *CREATE*, *ALTER*, and *DROP*. These were identified as the only DDL commands supported across all six investigated systems – and unlike UDO definitions, which are fully dynamic and extensible by design, DDL commands cannot be redefined or created dynamically by end users. Moreover, as the research highlighted, most of the additional commands were highly context-specific and not universally applicable. For example, implementing support for *GRANT* or *REVOKE* is unnecessary if no *USER* object exists. Thus, the specification defines only this minimal common set of operations, with the addition of *SHOW* commands – also universally applicable – to support metadata inspection. This addition is a contextual decision deemed necessary as users are to define their own mappings.

The specification aligns with API design conventions, including the outside-in and API-first models. Both emphasize consumer-first considerations and the definition of a clear API specification (or 'contract') before internal implementation [58]. Moreover, OpenDict adopts the OpenAPI standardized format for API specifications, as well as the OAuth2 protocol for authentication and authorization, aligning with both the Iceberg REST and Polaris specifications [59], ensuring consistency in the catalog implementations we aim to complement.

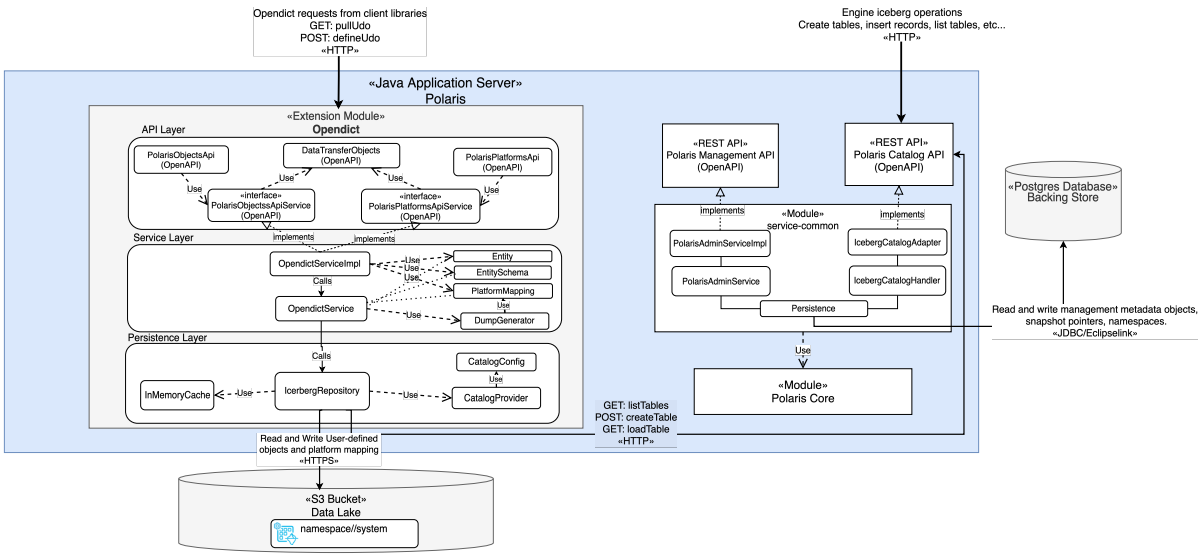

Figure 7: Component diagram – Polaris OpenDict extension.

4.3 OpenDict Polaris

OpenDict Polaris extends the Polaris catalog and serves as a reference implementation for the OpenDict specification. The extension utilizes a layered architecture with downwards pointing dependencies promoting ease of development and separation of concerns [60].

The Component Diagram, Figure 7, illustrates both the core Polaris modules and the additional OpenDict extension components. On the right-hand side, the diagram highlights Polaris' core modules:

- ***polaris-api-(iceberg/management)-service***: OpenAPI generated RESTful APIs and interfaces – one for the Polaris management specification and one for Iceberg REST.
- ***polaris-service-common***: The main components of the Polaris server including implementations of the API service interfaces.
- ***polaris-core***: Entity definitions and core business logic.

On the left-hand side, the diagram presents the OpenDict extension components, structurally inspired by Polaris' architecture and dependency flow and categorized into three layers:

- **API layer**: RESTful endpoints, service interfaces, and components generated from the OpenApi specification in Table 4.
- **Service layer:** Implementation of the API layer service interfaces, entity models, as well as object mapping and syntax expansion logic.
- **Persistence Layer:** Iceberg catalog configurations, cache, and persistence repository implementing read, write, and management operations on Iceberg tables via the Java API.

The remainder of this subsection outlines the steps required to configure an extension module in Polaris. It examines the responsibilities and components of each layer. Finally, it provides an overview of how we evaluate the correctness of the extension through a combination of unit and API integration testing.

Configuring a Polaris Extension Apache Polaris does not yet have a mature extension mechanism. Therefore, integrating the extension requires an understanding of the overarching system.

Apache Polaris is a Java project using the openapi-generator package [61] that, given an OpenAPI specification, generates JAX-RS (Jakarta RESTful Web Services) RESTful APIs. It uses Gradle for dependency management and the Quarkus framework as a build tool.

Figure 8 provides a simplified contribution tree based on our contributions to the Polaris repository according to the git log. Notice that contributions are restricted to the *polaris/extension/opendict* module except for two changes. First, *gradle/project.main.properties*, which maps folders to module names, we add: opendict-api-model, opendict-api-service, and opendict-impl. Second, in a *quarkus-server* module, we define a conditional command line property flag: *"-PincludeOpendict"* that dictates whether the OpenDict extension is included in the build assembly. With that, we can extend Polaris via the OpenDict module without modifying the remainder of the codebase. The full contribution tree can be found in Appendix Figure 19, and the contribution git log in Appendix A.10.

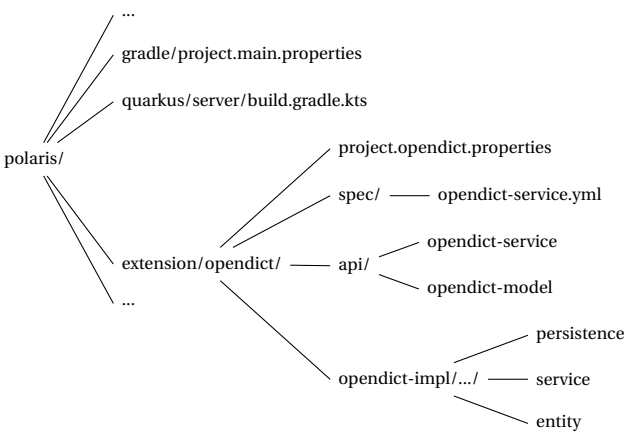

Figure 8: Polaris contribution tree visualizing OpenDict implementation.

4.3.1 API Layer

The API layer acts as the entry point for any Polaris feature, handling incoming client requests and providing standardized access to the system's functionalities.
To populate the API layer, the OpenDict module uses the OpenDict specification to generate the *objects* and *platforms* APIs – corresponding to the endpoint groups described in Section 4.2 –, their service interfaces, and data-transfer-objects (DTOs).

4.3.2 Service Layer

Service interfaces are implemented in *OpendictServiceImpl*, which transforms DTOs from the request payloads into corresponding entity models and propagates further business logic to the *OpendictService*.
By separating the service layer from both the front-facing REST protocol implementation and the Iceberg persistence implementation, the architecture allows different protocol interfaces, such as REST and gRPC, to delegate to the same service model. This separation also abstracts away details of the persistence layer, whether implemented using Iceberg, RDBMS, or other storage backends.

User-Defined Entity Models Handling dynamic user-defined objects instead of fixed definitions introduces additional complexity in entity modeling. The OpenDict extension defines three entity types outlined in Table 5.

- **UserDefinedEntity:** The standard User-defined object. It is defined in Table 5a. Besides a record name and UDO type, it stores a map of column names to values – essentially, the Java equivalent of a JSON object as provided in the API request.
- **UserDefinedEntitySchema:** The schema or signature for a user-defined object type. The definition of the Schema entity is shown in table 5b. *propDefs* instantiated from a JSON object provided in the API request, is a Map from column names to datatypes. *UserDefinedEntitySchema* defines methods for generating Iceberg schemas from the propDefs map.
- **UserDefinedPlatformMapping:** Java representation of Platform mappings for a *UserDefinedEntity* and its corresponding schema. Table 5c defines the mapping entity. It stores valid syntax for the target platform. The template syntax includes placeholders, annotated $< property-key >$, as seen in the Template Syntax from 6a. When mapped from UDO to a platform-specific object, the property key in placeholders is replaced with the corresponding values from the UDO *props* map. Additionally, the *UserDefinedPlatformMapping* stores a *complexSyntaxPropsMap* that defines how the property-key exchange should handle complex objects, such as List and Map.

UserDefinedEntity
String typeName
String objectName
Map<String, Object> props
...

(a) Object Entity.

UserDefinedEntitySchema
String typeName
Map<String, PropType> propDefs
...

(b) Schema Entity.

UserDefinedPlatformMapping	
String	typeName
String	platformName
String	templateSyntax
Map<String, ComplexSyntaxProps>	complexSyntaxPropsMap
Schema	schema
...	

(c) Mapping Entity.

Table 5: OpenDict entity definitions.

Both *UserDefinedEntity* and platform mapping objects are persisted as Iceberg Table records, while *UserDefinedEntitySchemas* define the tables used to store *UserDefinedEntities*. Therefore, Iceberg handles schema access and validation natively; no additional validation logic is required.

Object Mapping, Template Syntax, and Placeholder Exchange Broadly, the OpenDict extension uses user-defined mappings and template syntax to replace placeholder values, thus enabling universal compatibility with user-defined objects. The following illustrates how a template syntax (see 6a) and a User-defined object are transformed into valid syntax, as shown in Table 6b.
Specifically, Table 6a displays the *Template Syntax* for a Snowflake *Function* mapping. The placeholder exchange derives the syntax *Replacement List* in Table 7a from the user-provided Template Syntax. This syntax *Replacement List* contains a tuple consisting of a prefix substring, e.g., "CREATE OR ALTER," and a placeholder, e.g., "<type>." With a *Template Syntax* and a *Replacement List*, the exchange can efficiently iterate over the list, exchange placeholders with UDO properties, and build the *Expanded Syntax* in table 6b. With that, we have outlined the core functions of the placeholder exchange.

We add an additional element, *complexSyntax*, as defined in table 7b to support complex objects, e.g., List

and Map. Simply put, the *complexSyntax* defines how a collection is unfolded in the placeholder exchange. For example, *"(<args>)"* from table 6a with the format *"<key> <value>"* from table 7b becomes *"(arg1 int, arg2 int)"* in the expanded syntax in table 6b.

Template Syntax

```
CREATE OR ALTER <type> <name>(<args>)
  RETURNS <return_type>
  LANGUAGE <language>
  PACKAGES = (<packages>)
  runtime_version = <runtime>
  HANDLER = '<name>'
  AS $$
<def>
$$
```

(a) Template Syntax.

Expanded Syntax

```
CREATE OR ALTER function foo(arg2 int, arg1 int)
  RETURNS int
  LANGUAGE python
  PACKAGES = ('numpy', 'pandas')
  runtime_version = 3.10
  HANDLER = 'foo'
  AS
$$
def foo(arg1 int, arg2 int):
        return arg1 + arg2
$$;
```

(b) Expanded Syntax.

Table 6: A template syntax and the expanded syntax it produces.

Prefix	Placeholder
"CREATE OR ALTER "	"type"
" "	"name"
"("	"args"
")\n RETURNS "	"return_type"
"\n LANGUAGE "	"language"
"\n PACKAGES = ("	"packages"
")\n runtime_version = "	"runtime"
"\n HANDLER = "	"name"
"\n AS\n$$\n"	"def"
"\n$$"	""

(a) Syntax Replacement List.

complexSyntax args	
propType	"map"
format	"<key> <value>"
delimiter	", "

complexSyntax packages	
propType	"list"
format	"<item>"
delimiter	", "

(b) ComplexSyntax defs.

Table 7: Components of the mapping from Table 6: A Syntax Replacement List derived from the template syntax and ComplexSyntax rules for a Map and a List.

A placeholder exchange could be performed with nothing but Java's *String.replace()*. We define the *Syntax Replacement List* in Table 7a for efficient placeholder replacements, as synchronization is a core feature of OpenDict and may be executed repeatedly in practice, mapping a significant number of objects.
The *String.replace()* approach requires no additional memory, but a search-and-replace must be performed for each placeholder.
With the *Syntax Replacement List*, we only have to linearly scan the template syntax once to create the data structure, which can be used for all the mapped objects. Therefore, locating the placeholder in the *template syntax* becomes O(1) instead of O(n), where n represents the number of characters in the *template syntax*.

4.3.3 Persistence Layer: Iceberg Storage and Execution

The implementation details of the persistence layer significantly influence the performance results presented in Section 5. Therefore, we aim to provide adequate detail in the description of relevant functions and optimization steps.

We inject a Java Iceberg *RESTCatalog* object into the *IcebergRepository*. After which, the *IcebergRepository* communicates with the *PolarisCatalogAPI* through the catalog object when performing table reading, writing, or management tasks to collect Iceberg Table pointers, as depicted in Figure 7. Essentially, we are treating Polaris as an external catalog service from the OpenDict persistence layer.
Persisting metadata objects in tables is a common approach adopted by both SQLite and Postgres. However, while storing metadata objects in Iceberg Tables enables us to use Iceberg features on top of cheap and scalable data lake storage, it presents unique challenges in table maintenance and file immutability.
The Iceberg repository performs query execution via the Iceberg Java API [62]. It enables the execution of simple read-and-write queries without bloating the Polaris instance with heavy packages or query engine instances. The OpenDict operations can be roughly mapped to operations on Iceberg tables by the persistence layer. Table 16 presents logical mappings for select write operations.

OpenDict query	→	**Iceberg Operation**
DEFINE OPEN object	→	CREATE TABLE object (...)
CREATE OPEN object obj	→	INSERT INTO object (obj)
ALTER OPEN object obj ...	→	DELETE obj, INSERT INTO object (new_obj);
SHOW OPEN object(s)	→	SELECT * FROM objects

Table 8: OpenDict operations mapped to pseudo-SQL queries by the persistence layer.

In Table 16, define and create operations are mapped to *CREATE TABLE* and *INSERT INTO* operations using the catalog instance and the Java Iceberg API to create the table, write a new data file, and add a new metadata file. Altering a record within an Iceberg table is more complex and is achieved by deleting the original record and inserting the altered version. However, since Iceberg operates on immutable data files, a row-wise deletion is impossible. Currently, we solve this by performing a transaction that includes: 1) a filtered *read*, reading records into memory except for the alter target; 2) a *deletion* of all existing records; 3) a *write* that reinserts unaltered records; 4) a write that *inserts* the altered record.

Table Maintenance Iceberg is built for high-performance ACID-compliant queries on huge analytics tables, and although the OpenDict specification allows batched create requests, creating metadata objects is often a single insertion. With each insertion, Iceberg creates a new metadata file, a new snapshot, and a new data file. Therefore, table maintenance procedures are necessary to ensure the storage backend is not flooded with fragmented data and metadata files. The Iceberg documentation recommends a series of table maintenance steps [63]. The following outlines to which degree the OpenDict extension implements them.

- *Compact data files.* Compaction is especially recommended for tables that serve streaming or small write queries [63]. Therefore, if the number of small data files exceeds 50, we perform a rewrite by deleting old files and writing data into files with a target data size.

- *Expire Snapshots*: Each write creates a new table snapshot for time-travel queries[63]. We configure the expiration and deletion of snapshots older than 1 minute when performing compactions.

- *Remove old metadata files:* Each change to a table produces a new metadata file to provide atomicity, which is kept for the sake of table lineage and history [63]. We configure automatic deletion of metadata files, retaining only the 10 most recent after each table commit.

Duplication Handling and Caching Duplicate metadata objects are generally not allowed in data systems. However, Iceberg does not support primary key or uniqueness constraints by default. Instead, it is a feature added by engines that interact with Iceberg tables, such as Spark or Flink. Therefore, for the low-level Java API, we provided two simple solutions. First, a standard mode where each record insertion triggers a table scan for a record with the same unique name (uname). Second, a cache mode is implemented using a simple in-memory hash map with object names as keys, ensuring that no two objects with the same type and name are created. The in-memory cache was built with benchmarking and demoing in mind, we note that a scalable solution would use an external caching service instead of introducing state into the persistence layer.

4.3.4 Testing

We conclude the design description of the OpenDict extension with a brief outline of the two-sided testing approach used during its development. The two-sided approach includes: 1) unit tests used to validate and implement complex functions such as placeholder replacement; 2) API integration tests defined in an external test suite that uses the *pytest*[64] framework for regression testing changes to any part of the OpenDict implementation. The integration test suite runs against the same local Polaris deployment used for metadata performance experiment in Section 5.
The OpenDict Polaris local deployment exposes the OpenDict API endpoints through a localhost URI composed as *http://localhost:8181/api/opendict/v1/...* followed by the corresponding endpoint path. The integration test suite validates these endpoints in a logical sequence (e.g., defining and UDO before creating an instance of it), issuing curl requests to each one and asserts the correctness of received responses.

4.4 Client Libraries

As described in Section 3.3.1, REST catalogs provide a unified API for accessing metadata, independent of the catalog's internal storage [15]. This allows engines to use a single client implementation to interact with any Iceberg REST catalog, including Polaris. Since we define a complementary specification, OpenDict, that adds new features to REST catalogs – Polaris in this case – engines require additional client-side logic to parse and route OpenDict-specific requests. This subsection describes two client libraries that wrap engine behavior to enable this integration – one for Spark and one for Snowflake.

4.4.1 Design and Architecture

The primary goal of the client libraries is to enable support for the OpenDict specification through specific SQL-like commands across different engines without requiring any changes to the engines themselves. Each library wraps the engine's SQL execution interface (e.g., SparkSession.sql() or Snowflake's cursor execution), intercepts incoming SQL queries, and determines whether they match the OpenDict syntax or not. If a match is found, the query is routed to the relevant Polaris endpoint through a REST API. Otherwise, a fallback mechanism passes it to the engine unchanged.

By wrapping engines in standalone Python libraries instead of extending engines internally, we reduce implementation complexity and avoid tight coupling to specific systems. This approach also aligns with the Polaris integration strategy: to extend functionality without modifying the core systems. The design goal was to create a shared wrapper library reusable across engines by adapting to different engine connectors. While our current PySpark and Snowflake libraries are implemented separately, they follow a unified structure and could be consolidated into a single engine-agnostic client library in the future.

The generalized library architecture is illustrated in Figure 9. The main components include the *OpenDictCatalog*, which wraps the Python connectors of the underlying engine (e.g., SparkSession or SnowflakeConnection) and serves as the entry point for our defined OpenDict commands. It utilizes regular expression patterns from the *OpenDictPatterns* module to match and parse OpenDict-specific syntax. Once identified, requests are validated and serialized into structured request DTOs using Pydantic models from the *OpenAPI_Models* module and passed to the *OpenDictClient*, which sends the corresponding REST request to Polaris.

Implementation Tools The client libraries are implemented in Python ≥ 3.9 for compatibility with the prebuilt *spark-iceberg* image[65] used for testing, and are packaged using the *uv* package manager [66]. They are built and published to PyPi[67] via *uv* and can be directly imported into any Python Application, such as a Jupyter Notebook – links for both libraries are provided in Appendix A.5 and A.6. API authorization is handled through OAuth with configured catalog credentials collected via an API request to the Polaris standard specification. Core packaged dependencies include:

- *pydantic*: for defining and validating OpenDict API models

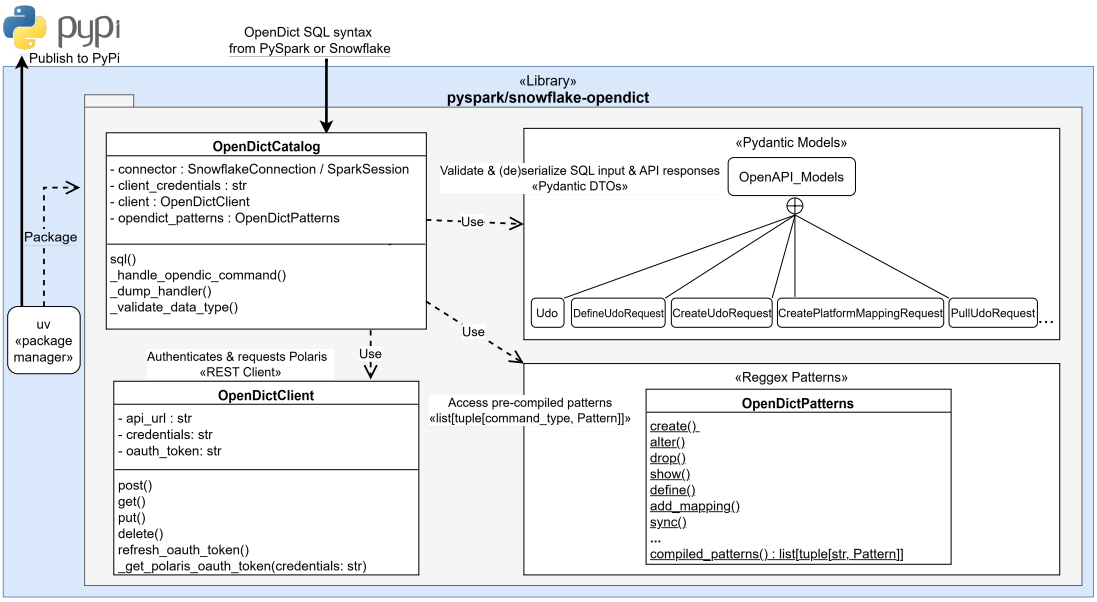

Figure 9: Code Diagram - generalized client library.

- *pandas*: for formatting and displaying response output
- *requests*: for issuing REST calls to Polaris
- *pyspark* or *snowflake-connector-python*: for fall-back query handling, depending on the target engine.

4.4.2 Syntax and Parsing Logic

The OpenDict client libraries support a fixed set of SQL-like commands, each corresponding to a specific OpenDict API endpoint. The libraries use a simple query parser that matches commands against *regular expressions*, defined as static methods in the *OpenDictPatterns* module. At initialization, these expressions are compiled into a list of *(command_type, pattern)* tuples via the *compiled_patterns()* method, as illustrated in Figure 9, enabling efficient matching without recompilation overhead for each input.
The full mapping between library-defined OpenDict-syntax and OpenDict API endpoints is shown in Table 4. For example, the HTTP *POST* endpoint with URL *.../objects/type* corresponds to the following SQL-like syntax:

CREATE OPEN <type> <name> [PROPS {<props>}]

This syntax is internally represented by a tuple that pairs the command type (*create*) with its corresponding regular expression pattern:

```
^create\s+open\s+
(?P<object_type>\w+)\s+
(?P<name>\w+)
(?:\s+props\s*
(?P<properties>\{[\s\S]*\}))?
```

All *<PROPS>* fields in OpenDict syntax are expected to be valid JSON objects. This format is both familiar to users and simplifies the parsing of nested, variable-length structures, e.g., an argument list for a UDO function definition. Moreover, it enables early structural validation before Pydantic modeling.

This list of *(command_type, pattern)* tuples is used by the *sql()* entry point of the *OpenDictCatalog* to match a query against all patterns sequentially. When a query matches one of the patterns, the corresponding *command_type* is identified, and the captured substrings for the named regex groups – such as *object_type*, *name*, and *properties* – are extracted. These extracted values are used to construct a request model (e.g., *CreateUdoRequest* or *DefineUdoRequest*), which is then validated using Pydantic, serialized, and passed to the appropriate Polaris API endpoint via the *OpenDictClient*.

4.4.3 Data Modeling and Validation

To ensure consistent communication between client libraries and the Polaris catalog, OpenDict uses *pydantic* models to define, validate, serialize, and deserialize all request and response payloads. Pydantic provides runtime validation of input parsed from our SQL-like commands, allowing invalid inputs to be caught early on the client side and prevent unnecessary API failures. Moreover, it improves code clarity and simplifies testing.

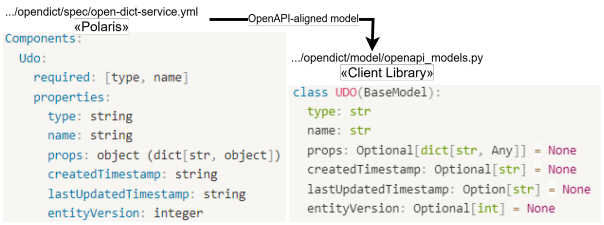

Figure 10: Client-side model of OpenAPI UDO component

Figure 10 depicts an example of how the OpenAPI *UDO* component is mirrored in Pydantic.

4.4.4 Testing

The client libraries are tested using unit tests written with the *pytest* framework, with all REST interactions, such as HTTP methods and token retrieval, mocked to isolate client behavior. The tests cover valid and invalid OpenDict SQL commands, verifying parsing logic, request model generation, routing to Polaris, fallback behavior, and error handling.

In total, the client libraries includes 22 distinct unit tests targeting the entry-point and main component, *OpenDictCatalog*, reaching 94% code coverage. While coverage alone is not a valid correctness metric, it demonstrates validation of functionality and expected edge cases.

4.5 Summary of Key Takeaways

The System Design section describes the architecture and design principles of the OpenDict specification and its Polaris reference implementation, including two client libraries to facilitate RESTful engine-to-catalog communication. OpenDict was developed to address the lack of non-table metadata support in OTFs and limitations in their catalogs, where support for metadata beyond tables is inconsistent and tied to vendor-specific implementations. OpenDict addresses these gaps by introducing User-Defined Objects (UDOs), which allow users to define their own metadata types and specify how they map to different platforms. This approach avoids fixed, predefined schemas in catalogs and reduces reliance on proprietary metastores, making it possible to synchronize any metadata object across systems without vendor lock-in.

The OpenDict Polaris architecture is organized into three layers: First, an **Engine Layer** where client libraries wrap engine connectors for Spark and Snowflake, intercepting OpenDict SQL-like commands and translating them into REST API requests to Polaris while passing standard queries through unchanged. Second, a **Catalog Layer** that extends Polaris with the OpenDict API and logic. Finally, a **Persistence Layer** where metadata objects and mappings are stored in Iceberg tables, decoupled from proprietary metastores, and maintained by the user.

This architecture supports the **extension over modification** design principle across each layer and component – including client libraries and the Polaris extension – enabling modularity, independent upgrades, and ease of integration without disrupting existing system behavior.

5 Experiments

To gain adoption as a standardized metadata management solution, the OpenDict catalog implementation must offer competitive storage efficiency and runtime performance. Previous work by the authors provided a comparative analysis of six data systems. It assessed the performance of the most common DDL features under synthetic workloads aiming to scope the performance requirements of metadata operations for an open metadata specification [10]. The performance experiments in this section reuse and extend the benchmarking driver from "Study on Dictionary Metadata: Usage, Performance, and Storage Formats" [10] with the *Function* object type and benchmarks for seven different configurations of the OpenDict implementation.

5.1 Experimental Design

The experimental design is inspired by the methodology outlined by Raj Jain in "*systems performance analysis*" [68]. The methodology presents a systematic approach designed to avoid common pitfalls that arise from an incomplete understanding of systems performance analysis.

The experiment will compare the runtime of metadata operations in OpenDict Polaris with four established and widely used database systems: SQLite, DuckDB, PostgreSQL, and Snowflake. These were specifically picked based on the following criteria. First, they incur no additional cost to the authors. Second, they are a mix of in-memory databases, a traditional relational database, and a cloud-native data platform. Third, they use a mix of table metastores, an in-memory metastore, and an external metastore, as described in Section 3.1. Finally, they represent both systems optimized for analytical and transactional processing.

5.1.1 System Specifications

The experiment includes both cloud-hosted and local hardware. All systems, system versions, and Python connector versions used by the OpenDict benchmark driver are listed in Table 9.

Data System	DB Version	Connector	Cloud
SQLite	3.43.2	sqlite3-v3.43.2	False
DuckDB	v1.2.2	duckdb-v1.2.2	False
Postgres	17.2	psycog2-v2.9.10	False
Snowflake	9.11.3	snowflake-connector-python-v3.14.1	True
Polaris+OpenDict	0.11.0-beta | 6493337	snowflake-opendict-v0.1.22	Both

Table 9: Data systems, connectors, versions [69]–[71].

All local resources in Table 9 will run on a Mac machine. Table 10 provides an overview of the system specification.

Specification	M4 Mac
Processor	Apple M4
CPU Cores	10. 4 perf(4.5 GHz), 6 eff(2.9 GHz)
RAM	16 GB LPDDR5X
Disk	256 GB SSD
Disk Speed	2671,6 MB/s read, 1830,6 MB/s Write
Operating System	MacOS 15.4.1
Instruction set	ARMv9.2a

Table 10: System specifications M4 mac.

Cloud provisioning makes certain hardware specifications inaccessible. For Snowflake in particular metadata operations modify metadata in the key-value store inside the Snowflake control plane [35]. Therefore, the size of provisioned warehouses (vCPUs) should not impact performance.

Polaris OpenDict, with a cloud storage backend, uses Azure Blob Storage in a Storage Account V2 located in Sweden Central configured with Locally Redundant Storage (LRS), the "Standard" performance tier, and the "Hot" access tier.

We note that results for cloud-provision systems will include a network latency overhead. Finally, to make OpenDict Polaris available to the benchmark driver, we must expose its APIs locally. Therefore, it is containerized and deployed on the experiment machine in Table 10, along with a containerized Postgres instance that stores table pointers. The *opendict-snowflake* library (App A.5) facilitates communication with the OpenDict Polaris deployment.
Building, containerization, bootstrapping credentials, and deployment are handled by the *polaris-boot* bootstrapping repository, as described in Appendix A.4.

5.1.2 Metrics

The experiment exposes three metrics that may be used to argue about the performance viability of OpenDict Polaris against metadata management solutions in widely used data systems.

Query Runtime (seconds). The number of elapsed seconds from a query is executed until a response is received. Query Runtime provides insight into the performance of DDL operations across multiple systems.
Total Benchmark runtime (hours). The benchmark suite contains 111.165 operations in total for each system. Total Benchmark runtime is the sum of all operation runtimes with repetitions taken as an average. It provides a holistic perspective on DDL query runtime over all commands.
Disk space used by file(s) (GB). The difference in disk space consumed by files before and after running the experiment. Disk usage is useful for discussing storage efficiency, a significant metric for systems that store a large number of objects.

For Snowflake and OpenDict Polaris with Azure blob storage, *Query Runtime* is influenced by network latency between the client and data center servers. However, we note that the networking should add a constant performance overhead, which will not affect the overall runtime complexity of operations. Similarly, we note that *disk usage* is directly affected by system-specific features such as time travel, file immutability, and redundancy. *Disk usage* is recorded by the MACOS file system.

5.1.3 Parameters and Levels

To capture performance differences of both scan and point query operations over a varying number of objects, we record the runtime performance of individual metadata operations: CREATE, ALTER, COMMENT, and SHOW. For systems without a *SHOW* command, namely SQLite and PostgreSQL, the *sqlite_schema* table and Postgres system catalogs were queried instead. We assume this to be equivalent to a show operation as we target the metadata directly in the metastores.
The experiment can be categorized into two separate sub-experiments: "Standard," targeting the four established data systems, and "OpenDict," targeting different configurations of the OpenDict Polaris implementation. Table 11 provides an overview of the included systems and parameters in the 2 experiment types. The *standard* and the *OpenDict* experiments use the *Table* object type because all the target systems support it. Additionally, we run the benchmark with the *Function* type, which is supported by all systems except SQLite, to investigate whether any table-specific optimizations are implemented. Table 12 outlines the parameters and corresponding levels used in the experiment.

Experiment	Included Parameters	Systems
Standard	Granularity, DDL command, Object Type	Sqlite, Duckdb, Postgres, Snowflake
OpenDict	Granularity, DDL command, Storage location, is_batched	(local), (local, cache), (local, batch), (local, cache, batch), (cloud, cache), (cloud, batch), (cloud, cache, batch)

Table 11: Experiment types, included parameters, and systems.

Parameters	Description	Levels
Systems	The data system or OpenDict configuration. (OpenDict(...) encapsulates all the configurations in Table 11)	Sqlite, Duckdb, Postgres, Snowflake, OpenDict(...)
Granularity	Number of CREATE statements before ALTER, COMMENT, SHOW	1, 10, 100, 1.000, 10.000, 100.000
DDL Command	Metadata management command executed	CREATE, ALTER, COMMENT, SHOW
Object	The metadata object.	Table, Function
Storage Location	Storage backend.	File, Cloud
Batch	Whether CREATEs are executed in batches	Yes, No

Table 12: Levels of Parameters.

5.1.4 Workloads

The experiment uses simple synthetic workloads that perform common DDL queries on a varying number of objects. Workloads are generated using a Python script that takes the system (–db) and experiment (–exp) as command-line arguments. Given a set of command line arguments, e.g., *–db duckdb –exp –standard_table*, the script initializes a Python connector for the set database to execute the queries and run the experiment. The Python connectors simplify the benchmark driver with Python timing functionality and allow for the reuse of logic by passing the connectors as function arguments.

Both experiment types follow a targeted sequence of runs. The sequence is structured as follows:

1. Execute a sequence of *{#granularity} CREATE {Object Type}*
2. Execute the following sequence of DDL commands: *ALTER → COMMENT → SHOW*.
3. Repeat the DDL command sequence a total of 3 times with a random target for *ALTER* and *COMMENT* point queries.
4. Drop the database/schema, increase *granularity* level, and repeat 1-4.

The DDL execution sequence (2) is repeated three times to ensure consistent results and minimize performance measurement deviations, such as cold starts, cache optimizations, and the influence of external processes on the benchmarking machine, including memory load and network interference.
Algorithm *create_tables()* shows how the benchmark driver creates standard tables or OpenDict tables by looping over *num_objects* (Granularity).

create_tables(*conn, system, num_objects*)

```
if system == "opendict" then
    define_query ← "DEFINE OPEN table PROPS {...}"
    _ ← execute(conn, define_query, system)
end if

for i = 0 to num_objects − 1 do
    if system == "opendict" then
        query ← "CREATE OPEN table PROPS {...}"
    else
        query ← "CREATE TABLE t_{i}(id INT PRIMARY KEY, value TEXT);"
    end if

    start, end, elapsed ← execute(conn, query, system)

    recorder.record(system, "CREATE", query, "TABLE", i, elapsed, ...)
end for
```

The batched create configuration in the OpenDict experiment modifies the experiment loop by adding all *num_objects* queries to a list and defers to a single execution that uses the OpenDict batch syntax and endpoint, as shown in Table 4. Furthermore, the batched experiment loop executes multiple smaller batches if the number of objects to create exceeds 10.000 due to a request packet size limit on the Polaris REST API.

*ALTER*s target a randomly chosen table and adds a column. Defining reusable methods for executing queries and recording results allows us to lessen the work required to add commands.

alter_tables(*conn, system, granularity, num_exp*)

```
table_num ← random(0, granularity.value - 1)

if system == "opendict" then
    query ← "ALTER OPEN table t_table_num PROPS {...}"
else
    query ← "ALTER TABLE t_{table_num} ADD COLUMN {num_exp} TEXT;"
end if

start, end, elapsed ← execute(conn, query, system)

recorder.record(system, "ALTER", query, "TABLE", granularity, num_exp,
elapsed, start, end)
```

The data recorder records runtime results as illustrated by the schema in Table 13. It persists entries in a DuckDB database to export results as Parquet files for visualization – see Appendix A.1 for the full results dashboard.

Column name	Column type
system_name	VARCHAR
ddl_command	VARCHAR
query_text	VARCHAR
target_object	VARCHAR
granularity	INTEGER
repetition_nr	INTEGER
query_runtime	DOUBLE
start_time	TIMESTAMP
end_time	TIMESTAMP

Table 13: Recorder record schema.

Finally, the benchmark driver repository, Appendix A.2, contains a *README*.md file with instructions on how to run the benchmark and export the results.

5.2 Results

This section presents the results from the performance experiment conducted across various database systems and OpenDict Polaris configurations. The results are used for evaluating the efficiency, performance, and scalability of the supported DDL operations *CREATE*, *ALTER*, and *SHOW* in OpenDict Polaris against widely used data systems.

The findings presented in this section provide the foundation for the discussion in Section 6, which will discuss the viability of the OpenDict reference implementation.

5.2.1 Total Workload Runtime

Figure 11 shows the cumulative runtime of all 111.165 benchmark queries across the experiment's systems and OpenDict configurations. While the total runtime does not provide insight into the performance of specific operations, it does offer a general indication of performance, provided that significant outliers are taken

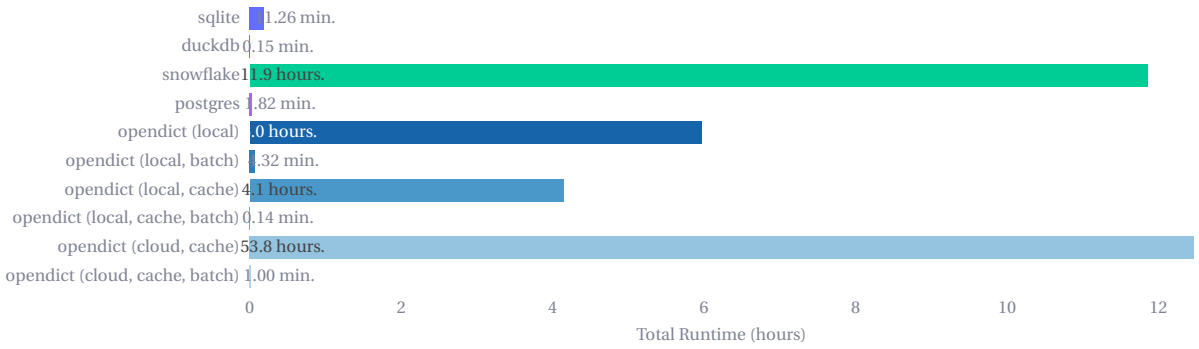

Figure 11: Cumulative query runtime across all systems & configurations.

into consideration.

OpenDict (cloud, cache) has the largest total runtime, followed by Snowflake. Unsurprisingly, these were the only cloud-based systems in the experiment, and both had prominent outliers – likely due to network inconsistencies during the experiments. For example, the 64.953rd *CREATE* in OpenDict (cloud, cache) took 3172,54 seconds. Furthermore, Snowflake had 5 *CREATE*s with a runtime larger than 300 seconds.

The standard database systems – SQLite, DuckDB, and Postgres – finished the experiment in less than 15 minutes. DuckDB was the fastest overall, 75x and 12x faster than SQLite and PostgreSQL, respectively.
OpenDict (local) took 6 hours to complete. Enabling the name-conflict cache allowed OpenDict (local, cache) to reduce the total runtime by 33%.

Finally, batching CREATE statements significantly reduced the total runtime for all three OpenDict configurations (local, local cache, and cloud cache). The most significant improvement was observed in configurations with cache enabled. OpenDict (local, batch) and OpenDict (local cache, batch) demonstrated a decrease in total runtime of 98.7% and 99.9%, respectively, compared to their non-batched counterparts.

5.2.2 Storage Efficiency

Figure 12 shows the total disk usage before and after creating 100.000 table objects for SQLite, DuckDB, Postgres, OpenDict(local), and OpenDict(local, batch).
OpenDict (local) outperforms widely used database systems in table storage efficiency, with 100.000 tables occupying 26%, 9%, and 3% of the storage used by SQLite, DuckDB, and Postgres, respectively. Furthermore, the total disk space used by OpenDict decreases to 0,01 GB when creation is batched into groups of 10.000 tables. Thus suggesting that we can further improve the storage efficiency via background batching or more comprehensive compaction.

Note that creating 100.000 *Function* objects in DuckDB and PostgreSQL used 28.1 MB and 406.5 MB, respectively. The difference in total disk usage from the one observed when creating tables suggests that table creation triggers an allocation of additional structures.

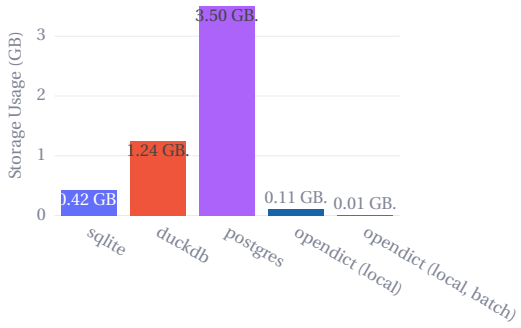

Figure 12: Total disk used after creating 100.000 tables.

5.2.3 DDL: Create

Figure 13 shows the average runtime (seconds) of *CREATE* statements for the standard database systems and the best-performing local and cloud-based OpenDict configurations. Additional OpenDict configurations are omitted to avoid overfilling the figure. A base-10 logarithmic scale is applied to the y-axis to allow comparison despite large differences in runtime.

Unsurprisingly, the cloud-based systems – Snowflake and OpenDict (cloud, cache) – have the highest average runtime for *CREATE* operations regardless of the number of objects stored. This can be attributed to networking inconsistencies and cloud latency overheads, as was made evident in Section 5.2.1. Excluding outliers, Snowflake had average runtimes between ~0.35 s and ~0.45 s, while OpenDict (cloud, cache) ranged between ~1.8 s and ~2.0 s.

Among the local systems, OpenDict (local, cache), DuckDB, and Postgres maintained a constant runtime not affected by the number of objects in the metastore.

DuckDB was the fastest overall, with an average runtime ranging from 0.000076 s to 0.000082 s. In contrast, OpenDict (local, cache) ranged from 0,14 s to 0,16 s per operation. Finally, Postgres landed between the two and showed the most inconsistency in average runtime. Notably, SQLite is the only system in Figure 13 with a linear increase in runtime as the number of stored objects increases.

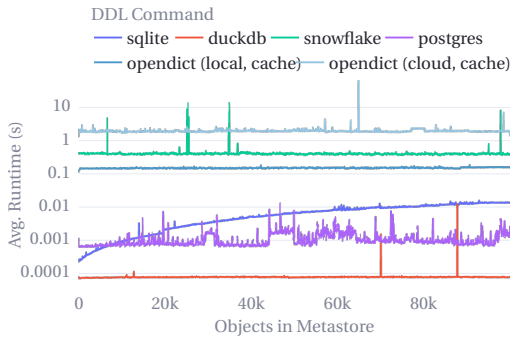

Figure 13: Runtime of CREATE as stored objects increase.

5.2.4 DDL: Alter

ALTER and *COMMENT* may be categorized as point query operations – they fetch a specific metadata object, update it, and write it back to the metastore. We will only present *ALTER*, as the observed difference in their results was negligible.

Figure 14 presents the average runtime of ALTER statements across systems. A base-10 logarithmic scale is applied to both axes to enable fair comparison and because the number of objects increases by a factor of 10 with each measurement.

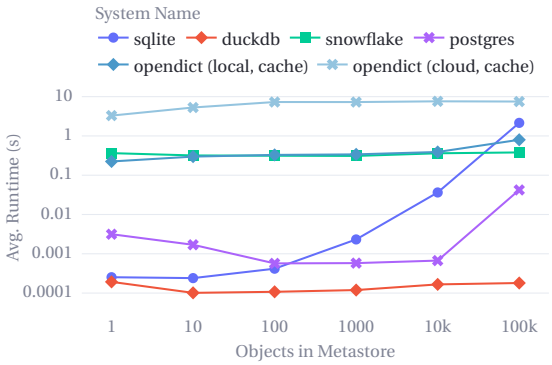

Figure 14: Runtime of ALTER as stored objects increase.

For the cloud-based systems, Snowflake has consistent but high runtimes, largely due to network latency. OpenDict's (cloud, cache) average runtime per operation increases as stored objects increase and reaches 7.5 seconds at 100 objects, after which it remains constant.

Among the local systems, DuckDB maintains a constant and low average runtime, while SQLite grows with the number of stored objects.

OpenDict (local, cache) increased in runtime as objects increased: from 0.22 seconds at 100 objects to 0.79 seconds at 100.000. This contrasts with the stable performance of the cloud version after 100 objects, as highlighted in Figure 15, which isolates OpenDict (local, cache) and Opendict (cloud, cache).

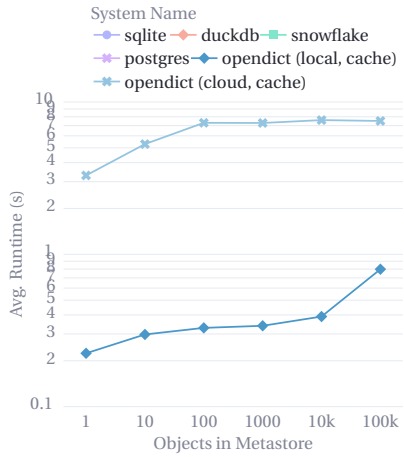

Figure 15: Expanded runtime of ALTER in OpenDict (local and cloud) as stored objects increase.

Table 14 shows the runtime of individual *ALTER* repetitions for OpenDict (local, cache) and OpenDict (cloud, cache). Investigating the results from individual repetitions reveals that the first repetition is consistently slower. In OpenDict (local, cache), the percent-wise decrease after the first repetition was between 23% and 13%. In OpenDict (cloud, cache), the drop is much more pronounced – between 67–75%.

Objects	Repetition	Runtime
10k	0	0.44s
10k	1	0.34s
10k	2	0.33s
100k	0	0.86s
100k	1	0.75s
100k	2	0.74s

(a) OpenDict(local, cache).

Objects	Repetition	Runtime
10k	0	15.11s
10k	1	3.93s
10k	2	3.79s
100k	0	13.72s
100k	1	4.35s
100k	2	4.47s

(b) OpenDict(cloud, cache).

Table 14: ALTER: runtime per repetition.

5.2.5 DDL: Show

SHOW commands are scan queries that collect all instances of an object in the metastore. Figure 16 presents the average runtime of *SHOW* operations across systems as stored objects increase. A base-10 logarithmic scale is applied to both axes to enable fair comparison and logical increments. OpenDict (cloud, cache) outpaces Snowflake at 10.000 objects and scales more efficiently than OpenDict (local, cache) as object count increases.

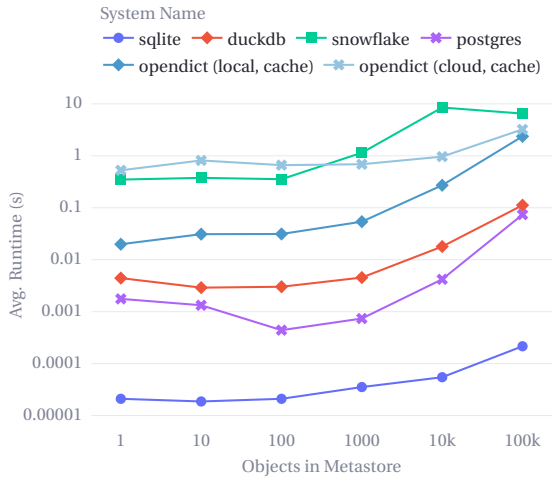

Figure 16: Runtime of SHOW as stored objects increase.

There is a general trend of average runtime increasing with the number of objects. Specifically – when going from 10.000 to 100.000 objects – OpenDict (local, cache) shows an 8,4x increase in runtime, from 0,27 to 2,27 seconds, while OpenDict (cloud, cache) grows by only 3,3x, from 0,96 to 3,2 seconds. Snowflake, however, shows a slight improvement in runtime from 10.000 to 100.000 objects. This observation is due to a limitation in the Python connector, which caps *SHOW* results at 10.000 records, reducing the effective load.

As expected, among the remaining systems, the average runtime of SHOW operations increases with the number of objects. Furthermore, SQLite notably outperforms local alternatives. At 100.000 objects, it is approximately 500x faster than DuckDB and 10.136x faster than OpenDict (local, cache).

5.2.6 OpenDict Polaris Maintenance and Optimizations

There are several alleys for optimizing the OpenDict Polaris implementation, including caching and batching requests. Figure 17 compared the runtime of *CREATE* operations as the number of stored objects increase for OpenDict (local) with and without an in-memory cache for name-conflict-checking.

Figure 17 clearly shows the effects of name-conflict caching. As expected, for the non-cache configuration, the *CREATE* operation runtime increases linearly with the number of objects. In contrast, the version that employs the in-memory cache remains constant in runtime, regardless of the number of stored objects.

The effect of data compaction on performance can be derived from the runtime of individual *CREATE*s. Data compaction occurs when more than 50 individual small data files are detected. Since the benchmark runs single insertions, compaction should happen once every 50 inserts. For OpenDict (local), the runtime of the first *CREATE* with compaction is 4.4 times larger than that of the subsequent CREATE. Note, however, that this does not imply that compactions are not worth it. The last create before that compaction had a 1.48x larger runtime than the *CREATE* directly following the compaction.

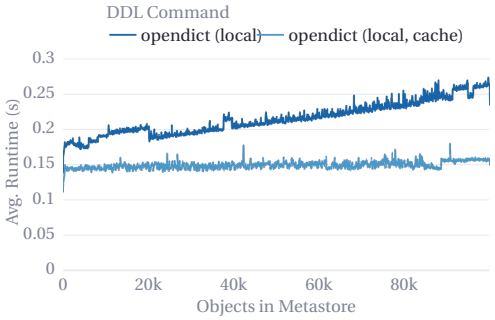

Figure 17: CREATE performance as the number of stored objects increases, with and without in-memory cache optimization.

6 Discussion

This section uses insights from research into metadata management approaches, OTFs, and related work in data lake catalogs with extended metadata support as the basis for reflection on key findings from the design of the OpenDict specification, the design and implementation of OpenDict Polaris, and the results of the metadata operation performance experiment.
Specifically, this discussion: (1) Aims to position OpenDict among modern metadata storage approaches by performance characteristics; (2) Evaluates the viability of OpenDict Polaris by comparing it to widely used database systems, highlighting areas for improvement and presenting significant findings from the metadata operation performance results presented in Section 5.2; (3) Discusses the strengths and weaknesses of integration options for OpenDict with existing systems, and how users of these systems can interface with it; (4) Revisits the motivation and value proposition of the OpenDict extension for OTF catalogs, and explores an alternative design that shifts persistence and mapping logic for user-defined objects to the client libraries, leaving Polaris unchanged; (5) Highlights limitations in the current regex-based parsing layer in the OpenDict client libraries; (6) Uses adoption as a measure of success for open standards, discussing the viability of the user-centric approach taken in the OpenDict Specification. Finally, it addresses the limitations of the current design, implementation, and experimental approach and identifies priorities for future improvements.

6.1 Discussion of Results

A key part of software development is evaluating and iterating on solution design and implementation through experimentation. This section reflects on the results from the metadata operations experiment and

evaluates metadata management approaches across the benchmarked systems and OpenDict configurations. It uses these results to derive insights into system bottlenecks that may be used as a point of reference for future work.

6.1.1 Categorizing the Metadata Storage Approach of OpenDict

In this section we evaluate the performance impact and characteristics of different metadata storage approaches, with the goal of positioning OpenDict within the landscape of modern data architectures.

Table Metastores: SQLite and Postgres had drastic differences in their general runtime across operations. SQLite had linear increases in runtime for both *CREATE* and *ALTER* operations as the number of objects grew. At the same time, PostgreSQL maintained constant *CREATE* performance but showed variability in *ALTER* operation runtime. We attributed Postgres's inconsistency to resource contention at 100.000 objects due to the significant variance between repetitions. Previous work attempted to locate the root cause of SQLite's linear increase in runtime complexity and attributed it to inefficiencies in duplication checks [10]. Both SQLite and Postgres excelled at *SHOW* operations, with SQLite performing notably better than any other system. These results suggest that in-table metastores are particularly efficient for metadata scans. Specifically, for SQLite and PostgreSQL, we can attribute this to their nature as row stores reading metadata from tables, allowing them to gain scan performance from optimizations such as sequential prefetching.

Key-value Metastores: like those used in DuckDB and Snowflake – albeit DuckDB uses an in-memory metastore and Snowflake uses an external metastore – show the opposite trend.

Both systems demonstrate constant runtime for point query operations, including *CREATE* and *ALTER*. DuckDB achieves high efficiency by maintaining metadata in memory, which minimizes access latency. Snowflake, despite being constrained by network latency, still outperformed the other remote systems (OpenDict (cloud)). However, both systems experienced performance degradation compared to others on *SHOW* operations, though the runtime remained constant. These findings confirm that key-value stores like metastores excel at lookups rather than large scans, and their categorization as in-memory or external mainly influences metadata access latency.

OpenDict: Table Metastore? The underlying design of OpenDict draws inspiration from table metastores; as such, the UDOs are persisted into Iceberg Tables. This aligns with experimental results, revealing that OpenDict shares some attributes with the standard table metastore systems, e.g., Postgres and SQLite.

When it comes to runtime trends, OpenDict generally performed worse on point query operations than the systems using key-value metastores. However, both the cloud and local versions outperformed Snowflake on scan operations. This pattern is consistent with its categorization as a table metastore system.

By using Iceberg tables, OpenDict – in theory – gains transactional guarantees, query optimization, data lineage, and time-travel features with minimal added complexity – all useful in metadata management. However, based on results for *ALTER* and *CREATE* operations, we admit that it does not achieve the same simplicity in metadata management as the standard systems using table metastores. Storing objects in Iceberg tables introduces additional complexity in file immutability and increased overhead for small writes as a result of using data lake storage.

In the end, it is challenging to fit OpenDict strictly into the table Metastore category. We defined the category around Postgres and SQLite – both OLTP row stores – which excel at simple operations and retrieval of full records. However, Iceberg uses a column store and is optimized for complex analytical queries and retrieval of partial records. The performance characteristics of table metastores unsurprisingly depend on the features of the underlying table format. So, while we categorize OpenDict as a table metastore system, its metadata management approach cannot be equated to that of the other systems using table metastores.

In conclusion, OpenDict appears to embody a whole new, or hybrid, category, inheriting elements of both traditional table metastore systems and analytical storage systems. As such, to use these categorizations as a framework for understanding, comparing, and positioning OpenDict in modern data architectures – which we ultimately aim for –, a redefinition of the categories is required.

6.1.2 Evaluating the Viability of OpenDict Polaris

Performance results provide a straightforward indication of a system's viability. If it is slow or storage is inefficient, users will likely find an alternative or continue to manage their metadata in closed, vendor-specific metadata stores. For the sake of this discussion, viability is defined in terms of two aspects: (1) does it compare in terms of storage efficiency? (2) does it compare to existing systems in query runtime and complexity?

In terms of storage efficiency – when storing 100.000 tables – OpenDict proved to be significantly more efficient than the alternatives. While the table object for the OpenDict configurations was stored in an Iceberg table – like any other OpenDict UDO –, other systems appeared to allocate additional structures when creating tables. As a result, OpenDict's advantage was smaller when it came to *Function* objects; however, it still maintained better overall efficiency.

As a disclaimer, however, OpenDict may store objects directly in cloud object stores, which, in most cases, would be cheaper and more scalable than local or man-

aged instances of the alternative systems from the experiment. That is, while OpenDict presents better overall storage efficiency, these findings are not a reflection of an inefficient implementation of other systems but rather design choices based on practical use cases. Nevertheless, we conclude that OpenDict is viable in terms of storage efficiency as is.

Performance results varied between operations. In general, with the exception of *ALTER*s, OpenDict maintained similar runtime complexities to other systems.
OpenDict (both local and cloud) was consistently slower than its counterparts in *CREATE* operations, with the cached OpenDict configuration showing constant runtime performance. In contrast, the runtime complexity of *ALTER* operations increased with the number of stored objects, reaching an average of approximately 8 seconds for the cloud-based configuration.
These observations may be tied to the implementation of *ALTER*, which involves deleting all records, rewriting them, and adding the altered record. This is done because files are immutable, and row-wise updates are not an option. This approach is simple and can be improved with additional table maintenance, limiting the number of files to delete and rewrite. However, to change the runtime complexity, a different approach is required. OpenDict could store new and old versions of objects and only return the ones that were updated last, thereby improving the runtime of alter commands while increasing implementation complexity and filtering requirements for scan queries.

In conclusion, using our definition of viability as the basis, the current reference implementation is viable in terms of storage efficiency and performance of *SHOW* and *CREATE* operations. While *SHOW* presents competitive performance and complexity, the *CREATE* operations of the cached OpenDict configuration showed comparable performance in runtime complexity. Finally, a general definition of viability also encompasses the ability to integrate with existing systems and the ease of adoption for users. The discussion explores these aspects further in Section 6.2 and 6.5.

6.1.3 Batching for Performance

Batching *CREATE* operations significantly improved OpenDict performance and storage efficiency. It reduced the cumulative runtime of all operations by 98.7% for Opendict(local) and 99.9% for OpenDict(local, cache) while also reducing disk usage from 0.11 GB to 0.1 GB. Beyond improving the performance of *CREATE* operations themselves, batching had a ripple effect on subsequent operations, as it produced only a few small files without requiring additional maintenance steps.
To provide further detail, several factors influence the performance gained from batching. First, Iceberg files are immutable, and every write operation creates both data and metadata files. Many small files result in a larger search space, more file accesses, increased disk usage, and, as such, decreased query performance.
To gain batching benefits for single insertions, we may silently cache requests and persist them in batches. However, this introduces additional complexity, and we must ensure that requests are persisted in the event of outages. This may require replication, further increasing complexity and potentially negatively affecting performance. Another way to maintain performance and storage efficiency is via table maintenance. OpenDict Polaris performs data file compaction when 50 small files exist. However, that is not a free lunch either. The performance experiments *CREATE*Ss showed that while the operations immediately following compaction were faster, compactions themselves were expensive.

An optimal file count trigger for compaction can be determined through further experimentation. However, a better, although more complex, approach would be to have a separate service running asynchronously, performing tasks such as table maintenance and compaction.
Finally, the viability of this improvement is based on the assumption that we are limited to the current Java Iceberg API. The Spark engine already has efficient implementations for maintenance functions, including data and metadata file rewrites, that we can leverage. However, it is a heavy dependency to add to the catalog codebase, considering that the queries we perform are relatively simple and can be implemented with the low-level Java API.

In summary, choosing the level of aggressiveness in table maintenance and selecting an appropriate engine involves trade-offs in performance, storage, features, and implementation complexity. Moreover, we note that excessive file cleaning or batching limits some of Iceberg's core features, such as table lineage and rollback.

6.1.4 Bottleneck Differences: Cloud and Local OpenDict

A key difference in performance bottlenecks between OpenDict (Cloud) and OpenDict (Local) can be derived from the results.
The average runtime of the cloud configuration's *ALTER* operations increases until 100 stored objects. Afterwards, it remains constant at approximately 8 seconds. In contrast, the local configuration increases linearly with the number of objects.

Additional networking overhead is expected in the cloud-based configuration. However, it should add a consistent delay, not alter the overall pattern or runtime complexity. The runtime of individual repetitions for the *ALTER* operations in Table 14 showed that, for both OpenDict configurations, the first iteration was slower; although, this was much more pronounced

for the cloud configuration. We have established that rewriting records to perform *ALTER* operations is an inefficient approach. However, the impact is more significant when operating on files in the cloud because we incur network latency with every file access. Since ALTER accesses up to 50 data files, it incurs a significant networking overhead.

Coincidentally, the performance impact from file accesses also explains why runtimes increase until there are 100 stored objects and then stagnate; at 100 objects, we reach the maximum number of small data files allowed by our compaction policy. The local configuration has significantly lower file access latencies, and therefore, the impact of additional records has a more pronounced effect on performance.
In conclusion, the primary bottleneck for the cloud-based configuration is the number of file accesses. For local configurations, file access latencies add minimal overhead, while the number of stored objects has a more significant impact on performance. This indicates that levels of caching, compaction, and batching may impact configurations differently and should be tailored to the storage model.

6.2 Integration with Current Systems

A logical metric for evaluating a system that manages universally compatible metadata is how well it integrates with the platforms it supports.

OpenDict connects to engines like Spark and Snowflake through Python client libraries. As long as an engine has a Python connector, integration should be possible with minimal development effort. Consequently, this means that a Python runtime with an OpenDict library is required. This requirement may create friction for users interacting with engines through other interfaces, such as Spark SQL for Spark or Snowflake's web UI.
For instance, a Snowflake user wishing to synchronize an OpenDict-managed UDO must open a Jupyter Notebook, configure a Python connecter, and instantiate the OpenDict library. In other words, we may not be able to integrate with some engines at a level that warrants adoption.
Implementing Python client libraries was a deliberate choice that reduced complexity, particularly considering the proprietary nature of systems like Snowflake. However, as OpenDict matures, a module integrated directly into engine cores might be desirable. Nevertheless, this poses a significant increase in development burden and may require contributions to closed platforms.

Given the constraints of the project and implementation, a more realistic approach is to offer an external syncing service. This service could issue scheduled *SYNC* requests to OpenDict Polaris and push metadata into platforms, provided valid object mappings exist. This would enable users, including those relying solely on other interfaces like Snowflake's web UI, to access UDOs managed by OpenDict.

In conclusion, although OpenDict's management features are limited to access via its Python libraries. The implementation already makes integration with a range of systems feasible and provides a foundation for further development.
Nevertheless, as a proof of concept, OpenDict currently puts interfacing with many existing systems and engines within reach through its Python client libraries.

6.3 Shifting Data Lake Writes to the Catalog

In standard Iceberg setups, engines like Spark, Flink, Hive, or Snowflake use dedicated modules [72] to handle the persistence of data files (e.g., Parquet, ORC) and metadata files (e.g., metadata.json, manifest files, and lists), as well as other table operations as per the Iceberg Specification. From this, the catalog maintains only atomic pointers to the current metadata files for each table, ensuring a consistent and up-to-date view of the table state across engines, excluding implementation-specific objects and features.

Our PoC design shifts all data writes, including UDOs and mappings, into Polaris. This centralizes persistence and logic for defining, managing, and synchronizing these dynamically defined objects, eliminating duplication across systems and significantly reducing implementation complexity and potential inconsistencies – regardless of whether a system is proprietary or not. That is, our design follows the principle of extension over modification: engines do not need to modify their core logic or query parsing. They connect to OpenDict Polaris through client libraries that wrap the execution interfaces, keeping integration simple while Polaris manages the complexity.

An alternative would be to centralize our logic within the client libraries themselves. This would free the reference implementation from Polaris and would arguably be simpler from a PoC perspective, avoiding the need to learn and integrate with Polaris' codebase. This alternative effectively reduces our contribution to a library of client-side parsers that execute commands as local methods, retrieve pointers from any REST Iceberg catalog, and read and write directly to the data lake. That is, this approach would eliminate the specification, as we would no longer need an API – shifting our service, or in this case, a very large library, away from the goal of an open standard specification aligned with the broader ecosystem we aim to improve.

On a different note, one might ask: *If engines already support Iceberg, couldn't they just create table abstractions for these objects themselves?* Technically, they could. However, this is much like asking: Why use UI libraries when you can write custom css? Or why use a package manager when we can manually handle dependencies?
For example, engines like Spark or Snowflake could

define an Iceberg table named "Function" with persistence and management facilitated by their integrated modules. However, this would only cover persistence and table operations as defined in the Iceberg specification. OpenDict complements this specification by introducing an abstraction for any metadata object, specifically, a consistent structure for dynamically defined object types, which form the basis for management and interoperability features, including mappings and cross-platform synchronization. Without this abstraction, every platform would have to coordinate how to structure, manage, and interact with said data, creating significant complexity and effectively defeating the purpose of an open standard.

6.4 External Regex-Based Parsing

As a result of the design, wrapping with client libraries introduce an additional parsing layer, keeping the engines untouched and externalizing all OpenDict logic. However, since the engines' native parsers neither recognize OpenDict-specific syntax nor contain any internal logic for rerouting such commands, the client libraries must serve as entry points that first determine whether a query is intended for OpenDict or not. This is done entirely through regular expressions: if a match is found, the command is routed to a catalog endpoint; otherwise, it is forwarded to the engine, resulting in yet another round of parsing.

While this approach cleanly separates OpenDict logic from engine internals, it also causes duplicated parsing efforts and added computational overhead. This is mainly due to the use of regular expressions, which simply match patterns within raw text input without recognizing the syntactic units (and reusable units), also called tokens, that compose valid commands. By contrast, grammar-based parsers like those generated by YACC – one of the most widely used parser generators, as noted by DuckDB [73] – tokenize the input, build a tree structure and use it to process the entire language syntax in a single pass [74]. The regex-based method in our client libraries, on the other hand, matches flat input sequentially against each regex pattern, which results in increased computational cost, especially for common units that could have been tokenized.

Initially, this regex approach was a quick way to support basic syntax in the PoC, with nested constructs represented as JSON objects within OpenDict syntax. However, as OpenDict has expanded to include 15 distinct commands with defined SQL-like syntax, many of which share similar patterns, this approach has become increasingly inefficient and challenging to maintain.

6.5 Achieving Adoption: User-Centric vs. Vendor-Centric Design

In aiming to establish a unified open standard for metadata management that is interoperable across systems, the OpenDict approach distinguishes itself by adopting a user-centric design that ensures all metadata, including non-table objects, is fully user-defined and managed within their own data lakes.

However, this user-centric design shifts responsibility onto users, introducing two main drawbacks:

- Defining the schema for each new metadata type and specifying explicit mappings to valid syntax for each platform in their ecosystem may be complex and bothersome.

- No predefined or shared types and mappings are available, meaning each new catalog instance requires a full, manual configuration effort.

This additional complexity compared to other catalog specifications with predefined schemas may hinder adoption, particularly for users with limited technical resources or experience in metadata management.

One possible approach to addressing potential adoption challenges is to establish a community-driven UDO Hub, where users can share and reuse object definitions and mappings. Much like DockerHub enables the sharing and reuse of container images, this hub would serve as a central repository for UDO schemas and mappings, lowering the entry barrier for individual users and reducing the setup effort required. Additionally, it represents the logical extension of the user-centric design approach: By defining metadata types and platform mappings as a shared, community-driven process, the UDO Hub would ensure that the standard evolves in response to real-world needs. Instead of vendors competing to become the de facto catalog through fixed extensions to the Iceberg REST implementation, the UDO Hub would let the community define the standard itself. This would create a common ground of widely used metadata types and mappings, enabling true interoperability and openness while remaining fully extensible to new platforms and use cases.

6.6 Limitations and Future Work

This thesis focused on researching the OTF ecosystem and metadata layer, designing the OpenDict specification, implementing it within the existing Polaris system, developing client libraries with SQL-like syntax and parsing capabilities, and conducting benchmarks. We did not aim to create a scalable, production-ready system but rather a proof-of-concept reference implementation to demonstrate and benchmark. As such, certain system aspects are simplified and remain areas for future work.

Limitations The following outlines the key limitations identified in our work:

- **Consolidated client libraries**: We maintained two separate client libraries, which could have been merged into a shared core library for maintainability, reduced duplication, and easier integration.
- **Query parsing**: The implemented parser is functional but limited in its capabilities. A more robust and extensible parsing system would reduce ambiguity and support more advanced syntax scenarios as OpenDict commands grow.
- **Manual synchronization**: OpenDict objects are not immediately usable post-creation, and synchronization is not automatic. This feature is designed to allow flexible push-based syncing controlled by the user. However, it may lead to unfairness in experiment results compared to systems where objects are immediately ready.
- **Benchmark scope**: Additional cloud-based benchmarks and targeted evaluations against other catalog implementations, such as Unity Catalog and base Polaris, would have provided a more thorough comparison.
- **Table Maintenance**: The Java Iceberg API has limited built-in compaction logic compared to fully featured engines, such as Spark. While the Iceberg documentation recommends data file compaction and metadata rewrites for workloads that include small writes, these require manual implementation. Our current solution, built to support a working prototype and benchmark, can be considered naive.
- **Integration & Interfacing Options**: We provide Python client libraries for Snowflake and Spark. We can easily extend our libraries with additional engine support. However, interfacing with OpenDict Polaris requires running a Python application that imports these libraries, which may be constraining for users.

Future Work Looking forward, the main focus is on driving the adoption of OpenDict, with the ultimate goal of becoming an open standard or part of an open standard. To achieve this, we must transition from the current proof-of-concept reference implementation to a robust, production-ready module.
While all the points listed in the Limitations paragraph above represent areas for improvement, we especially recommend consolidating the client libraries and improving persistence logic as the first steps for future work.

Consolidating the libraries is not just about reducing duplication, maintenance overhead, and possible inconsistencies. It is a fundamental improvement for integration, specifically by enabling the library to wrap any Python connector provided by any engine, making it easier to support multiple systems with minimal changes.

On the persistence side, a key priority is improving table maintenance operations, in line with Iceberg recommendations, to reduce file fragmentation – particularly by replacing the current naive approach to compaction. Another major persistence challenge is the current implementation of the *ALTER* operation, which involves deleting and rewriting entire datasets. This approach is simple but inefficient, especially in cloud environments where networking and I/O costs are high. These essentially boil down to performance inefficiencies and are significant enough to discourage the use of OpenDict Polaris. Resolving these should be prioritized as an initial target for improvements.

7 Conclusion

This thesis addresses the growing challenge of metadata fragmentation as organizations increasingly adopt specialized engines and multi-cloud environments to optimize for diverse workloads, resulting in metadata becoming distributed across isolated, proprietary metadata stores.

Our background research categorized metadata management approaches across modern, in-process, traditional, cloud-native, and data lake systems. We highlighted how OTFs form metadata layers on top of object stores, enabling shared access to tabular data, while non-tabular metadata – such as functions, users, and roles – remains siloed in proprietary metastores, hindering interoperability in a growing ecosystem of specialized engines.

Based on the requirements and insights drawn from our investigation of state-of-the-art metadata management solutions and prior research of metadata object and feature support across widely used data systems, this thesis introduced the OpenDict Specification: a minimal yet extensible model for representing and managing any metadata object in open formats. We demonstrated how user-defined objects can extend OTF catalogs to support non-tabular metadata and synchronize it across engines, using user-defined mappings to bridge system-specific differences. In practice, we achieved this through a reference implementation of the OpenDict specification within Apache Polaris, complemented by two Python-based client libraries for PySpark and Snowflake, which facilitate communication between the engine and the catalog.

We experimented and compared the performance of metadata operations across systems with different metadata management approaches to that of OpenDict Polaris as the number of stored objects increased. The results highlighted strengths in storage efficiency and scan operations, inefficiencies in point query manipulations, and insights into optimization through

batching and compaction.

Following the experiment, we used the results, combined with the current system's integration options, as a simple indication of viability. We concluded that the storage efficiency and operation runtime, except for *ALTER* operations, are comparable to those of other systems benchmarked, meeting our criteria for success. Furthermore, we deemed that OpenDict provided sufficient integration options for a PoC system.

Beyond technical contributions and performance characteristics, this thesis emphasized adoption in terms of usability. OpenDict's user-centric model gives organizations control over metadata support, but it also introduces the responsibility of defining these objects themselves and ensuring compatibility through valid platform-specific mappings. To address this, we proposed the community-driven UDO Hub as a natural extension of OpenDict's approach, lowering entry barriers and shifting the pursuit of an open standard from vendor competition to a shared community effort guided by industry needs.

In conclusion, this thesis addressed a standing gap in metadata management by proposing a fundamentally new approach: OpenDict. Unlike existing catalog implementations that extend Iceberg REST with vendor-specific features, often reinforcing fragmentation and lock-in, the OpenDict specification introduces a user-centric model that empowers organizations to define and manage any metadata type in a standardized and interoperable manner. Our design and implementation demonstrated the feasibility of extending OTF catalogs beyond tabular objects, achieving the proof-of-concept goals set for this thesis. While unifying all metadata types under a single fixed standard is not realistic, OpenDict represents a significant shift toward a more open and extensible approach, laying the groundwork for interoperable metadata management that is adaptable to diverse platform needs now and in the future.

Finally, we acknowledge opportunities for extension of our proposed design and implementation. Therefore, we encourage future work to build upon our experiments, design, and implementation and to challenge the feasibility of our proposed improvements for library consolidation, table maintenance, system integration, and query parsing.

References

[1] T. Luxner, *Cloud computing trends and statistics: Flexera 2023 state of the cloud report*, Accessed: 2025-05-07, 2023. [Online]. Available: https://www.flexera.com/blog/finops/cloud-computing-trends-flexera-2023-state-of-the-cloud-report/.

[2] T. Luxner, *Cloud computing trends: Flexera 2024 state of the cloud report*, Accessed: 2025-05-07, 2024. [Online]. Available: https://www.flexera.com/blog/finops/cloud-computing-trends-flexera-2024-state-of-the-cloud-report/.

[3] Accenture, "To the multi-cloud and beyond," Accenture, Tech. Rep., 2021, Accessed: 2025-05-07. [Online]. Available: https://www.accenture.com/content/dam/accenture/final/a-com-migration/r3-3/pdf/pdf-180/accenture-to-the-multi-cloud-and-beyond.pdf.

[4] D. Dunne, *The future of data management: A metadata-centric approach*, Accessed: 2025-05-07, 2025. [Online]. Available: https://www.informatica.com/blogs/the-future-of-data-management-a-metadata-centric-approach.html.

[5] PW Consulting, *Data observability software market*, Accessed: 2025-05-08, 2025. [Online]. Available: https://pmarketresearch.com/it/data-observability-software-market/.

[6] Team Atlan, *Gartner magic quadrant for metadata management: What changed in 2025?* Accessed: 2025-05-07, 2024. [Online]. Available: https://atlan.com/gartner-magic-quadrant-for-metadata-management/.

[7] Apache Software Foundation, *Entities - apache polaris documentation (unreleased)*, Accessed: 2025-05-08, 2024. [Online]. Available: https://polaris.apache.org/in-dev/unreleased/entities/.

[8] Apache Gravitino, *Apache gravitino overview*, Accessed: 2025-05-08, 2024. [Online]. Available: https://gravitino.apache.org/docs/0.6.0-incubating/.

[9] K. Kywe, *Unity catalog vs. apache polaris*, Accessed: 2025-01-29, 2025. [Online]. Available: https://medium.com/@kywe665/unity-catalog-vs-apache-polaris-522b69a4d7df.

[10] MSC Open Metadata, *Study on dictionary metadata usage, performance and storage formats*, Accessed: 2025-05-09, 2023. [Online]. Available: https://github.com/msc-open-metadata/study-on-dictionary-metadata/blob/main/study-on-dictionary-metadata-usage-performance-and-storage-formats.pdf.

[11] IBM Corporation, *What is a data catalog?* Accessed: 2025-12-05, 2024. [Online]. Available: https://www.ibm.com/topics/data-catalog.

[12] Google Cloud, *Data catalog: Overview*, Accessed: 2025-12-05, 2024. [Online]. Available: https://cloud.google.com/data-catalog/docs/concepts/overview.

[13] PostgreSQL, *Chapter 51. system catalogs*, Accessed: 2025-12-05, 2024. [Online]. Available: https://www.postgresql.org/docs/current/catalogs.html.

[14] Snowflake Inc., *Snowflake information schema*, Accessed: 2025-12-05, 2024. [Online]. Available: https://docs.snowflake.com/en/sql-reference/info-schema.

[15] Apache Iceberg, *Terms*, Accessed: 2025-05-20, 2025. [Online]. Available: https://iceberg.apache.org/terms/.

[16] Databricks Documentation. "Object hierarchy in the metastore." Accessed: 2025-05-18. (2025), [Online]. Available: https://docs.databricks.com/aws/en/data-governance/unity-catalog#metastore (visited on 05/18/2025).

[17] Wikipedia contributors, *Database object*, Accessed: 2025-12-05, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Database_object.

[18] Snowflake Inc., *Databases, tables & views*, Accessed: 2025-12-05, 2024. [Online]. Available: https://docs.snowflake.com/en/user-guide/databases.

[19] Oracle Corporation, *Create table*, Accessed: 2025-12-05, 2024. [Online]. Available: https://docs.oracle.com/en/database/oracle/oracle-database/21/sqlrf/CREATE-TABLE.html.

[20] IBM, *Hive metastore overview*, Accessed: 2025-12-05, 2024. [Online]. Available: https://www.ibm.com/docs/en/watsonx/watsonxdata/1.0.x?topic=components-hive-metastore-overview.

[21] Google Cloud. "Introducing bigquery metastore, a unified metadata service with apache iceberg support." (2025), [Online]. Available: https://cloud.google.com/blog/products/data-analytics/introducing-bigquery-metastore-fully-managed-metadata-service (visited on 05/29/2025).

[22] Databricks, *What is unity catalog?* Accessed: 2025-13-05, 2024. [Online]. Available: https://docs.databricks.com/en/data-governance/unity-catalog/index.html.

[23] Snowflake, *Data definition language (ddl) commands*, Accessed: 2024-10-02, 2024. [Online]. Available: https://docs.snowflake.com/en/sql-reference/sql-ddl-summary.

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

[25] DB-Engines, *Db-engines ranking*, Accessed: 2025-05-16, May 2025. [Online]. Available: https://db-engines.com/en/ranking.

[26] SQLite Development Team. "The schema table," SQLite Consortium. (), [Online]. Available: https://www.sqlite.org/schematab.html (visited on 05/16/2025).

[27] Oracle Corporation, *The mysql system schema*, version 8.4, Accessed: 2025-05-30, 2024. [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/system-schema.html.

[28] R. Hipp, *Sqlite home page*, Accessed: 2025-05-16, D. Richard Hipp, 2025. [Online]. Available: https://www.sqlite.org.

[29] R. Hipp, *What is sqlite*, Accessed: 2025-05-16, D. Richard Hipp. [Online]. Available: https://www.sqlite.org/fileformat2.html.

[30] SQLite Consortium, *Most widely deployed and used database engine*, Accessed: 2025-05-16, 2025. [Online]. Available: https://www.sqlite.org/mostdeployed.html.

[31] SQLite Development Team. "Database file format," SQLite Consortium. (), [Online]. Available: https://www.sqlite.org/fileformat2.html (visited on 05/16/2025).

[32] Stack Overflow. "Stack overflow developer survey 2024: Most popular technologies – database professional." Accessed: 2025-05-16. (2024), [Online]. Available: https://survey.stackoverflow.co/2024/technology#most-popular-technologies-database-prof.

[33] DuckDB Development Team, *Why duckdb?* Accessed: 2025-05-17, 2025. [Online]. Available: https://duckdb.org/why_duckdb.

[34] DuckDB Contributors, *Catalog_set.cpp — catalog implementation in duckdb*, Accessed: 2025-05-17, 2024. [Online]. Available: https://github.com/duckdb/duckdb/blob/main/src/catalog/catalog_set.cpp.

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

[36] Databricks. "Data warehouse." Accessed: 2025-05-18. (May 2025), [Online]. Available: https://www.databricks.com/discover/data-warehouse.

[37] Apache Software Foundation, *Design*, Accessed: 2025-05-18, 2025. [Online]. Available: https://cwiki.apache.org/confluence/display/hive/design.

[38] Snowflake. "How foundationdb powers snowflake metadata forward," Snowflake. (Mar. 2018), [Online]. Available: https://www.snowflake.com/en/blog/how-foundationdb-powers-snowflake-metadata-forward/ (visited on 05/17/2025).

[39] Apache Software Foundation, *Hive metastore server (hms)*, Accessed: 2025-05-16, 2025. [Online]. Available: https://hive.apache.org/#:~:text=Hive%20Metastore%20Server%20(HMS).

[40] M. Zaharia, A. Ghodsi, R. Xin, and M. Armbrust, "Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics," in *CIDR 2021*, 2021. [Online]. Available: http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf.

[41] Teradata, *What are open table formats (otfs)?* Accessed: 2025-05-16, 2024. [Online]. Available: https://www.teradata.com/insights/data-platform/what-are-open-table-formats.

[42] S. D. Engineering, *What is an open table format? & why to use one?* Accessed: 2025-05-16, 2022. [Online]. Available: https://www.startdataengineering.com/post/what_why_table_format/.

[43] P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, and M. Zaharia, "Analyzing and comparing lakehouse storage systems," in *Conference on Innovative Data Systems Research (CIDR)*, 2023. [Online]. Available: https://www.cidrdb.org/cidr2023/papers/p92-jain.pdf.

[44] 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.

[45] A. Sadeghi. "The history and evolution of open table formats." Accessed: 2025-05-18. (2024), [Online]. Available: https://alirezasadeghi1.medium.com/the-history-and-evolution-of-open-table-formats-0f1b9ea10e1e.

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

[47] Apache Software Foundation, *Iceberg table spec*, Accessed: 2025-05-16, 2024. [Online]. Available: https://iceberg.apache.org/spec/#overview.

[48] Snowflake Inc. "Apache iceberg™ tables." Accessed: 2025-05-20. (2025), [Online]. Available: https://docs.snowflake.com/en/user-guide/tables-iceberg#catalog.

[49] A. Aysha, *Using unity catalog with apache spark and delta lake*, Accessed: 2025-05-20, 2024. [Online]. Available: https://www.unitycatalog.io/blogs/unity-catalog-spark-delta-lake.

[50] Apache Polaris, *Overview*, Accessed: 2025-05-20, 2024. [Online]. Available: https://polaris.apache.org/in-dev/0.9.0/overview/.

[51] Apache Software Foundation, *Welcome to the apache polaris™ (incubating) web site!* Accessed: 2025-05-31, 2025. [Online]. Available: https://polaris.apache.org/in-dev/unreleased/metastores/.

[52] Databricks, *Unity catalog credential vending for external system access*, Accessed: 2025-05-20, 2025. [Online]. Available: https://docs.databricks.com/aws/en/external-access/credential-vending.

[53] Deployment, *Unity catalog deployment*, Accessed: 2025-05-31, 2025. [Online]. Available: https://docs.unitycatalog.io/deployment/.

[54] Unity Catalog Contributors, *Unity catalog: Open, multimodal catalog for data & ai¶*, Accessed: 2025-05-20, 2025. [Online]. Available: https://docs.unitycatalog.io/.

[55] S. Brown, *The c4 model for visualising software architecture*, Accessed: 2025-05-21, 2025. [Online]. Available: https://c4model.com.

[56] Apache Software Foundation, *Apache polaris (incubating)*, Accessed: 2025-05-31, 2025. [Online]. Available: https://github.com/apache/polaris.

[57] Apache Software Foundation, *Basemetastoremanager.java*, Accessed: 2025-05-31, 2025. [Online]. Available: https://github.com/apache/polaris/blob/main/polaris-core/src/main/java/org/apache/polaris/core/persistence/BaseMetaStoreManager.java.

[58] Postman, *Api design guide documentation*, Accessed: 2025-05-24, 2021. [Online]. Available: https://www.postman.com/postman/postman-team-collections/documentation/cvg32o0/api-design-guide.

[59] Apache Software Foundation, *Apache polaris and apache iceberg rest catalog api*, Accessed: 2025-05-25, 2024. [Online]. Available: https://editor-next.swagger.io/?url=https://raw.githubusercontent.com/apache/polaris/refs/heads/main/spec/generated/bundled-polaris-catalog-service.yaml.

[60] M. Richards and N. Ford, *Software architecture patterns*, 2015. [Online]. Available: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/.

[61] OpenAPITools, *Openapi generator*, Accessed: 2025-05-23, 2018. [Online]. Available: https://github.com/OpenAPITools/openapi-generator.

[62] Apache Software Foundation, *Iceberg java api*, Accessed: 2025-05-24, 2025. [Online]. Available: https://iceberg.apache.org/docs/1.9.0/api/.

[63] A. Contributors, *Maintenance*, Version 1.9.0 documentation, accessed 2025-05-24. [Online]. Available: https://iceberg.apache.org/docs/1.9.0/maintenance/.

[64] H. Krekel and pytest Development Team, *Pytest: Simple, powerful testing with python*, version stable, Accessed: 2025-06-01, 2025. [Online]. Available: https://docs.pytest.org/en/stable/.

[65] tabulario, *Apache spark with apache iceberg support [docker image]*, Accessed: 2025-05-24, 2025. [Online]. Available: https://hub.docker.com/r/tabulario/spark-iceberg.

[66] Astral, *Uv*, Accessed: 2025-06-01, 2025. [Online]. Available: https://docs.astral.sh/uv/.

[67] PyPy Project. "Pypy – a fast, compliant alternative implementation of the python language." Accessed: 2025-05-25. (2025), [Online]. Available: https://pypy.org.

[68] 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.

[69] msc-open-metadata, *uv.lock in "opendic-benchmark"*, Accessed: 2025-05-14, 2025. [Online]. Available: https://github.com/msc-open-metadata/opendic-benchmark/blob/main/uv.lock.

[70] MSC Open Metadata, *Polaris: Version.txt*, Accessed: 2025-05-14, 2025. [Online]. Available: https://github.com/msc-open-metadata/polaris/blob/main/version.txt.

[71] msc-open-metadata. "Commits on main · msc-open-metadata/polaris." Accessed: 2025-05-25. (2025), [Online]. Available: https://github.com/msc-open-metadata/polaris/commits/main/.

[72] Apache Software Foundation, *Iceberg*, Accessed: 2025-05-27, 2024. [Online]. Available: https://github.com/apache/iceberg/tree/main#:~:text=Iceberg%20also%20has,with%20Apache%20Hive.

[73] H. Mühleisen and M. Raasveldt, *Runtime-extensible sql parsers using peg*, Accessed: 2025-05-27, 2024. [Online]. Available: https://duckdb.org/2024/11/22/runtime-extensible-parsers.html.

[74] GeeksforGeeks, *Introduction to yacc*, Accessed: 2025-05-27, 2024. [Online]. Available: https://www.geeksforgeeks.org/introduction-to-yacc/.

[75] S. Shah, J. Malone, M. Lee, and S. Teal, *Introducing polaris catalog: An open source catalog for apache iceberg*, Accessed: 2025-05-20, 2024. [Online]. Available: https://www.snowflake.com/en/blog/introducing-polaris-catalog/.

A Appendix

A.1 Opendict Benchmark Dashboard Deployment

Deployment: https://opendic-benchmark-dashboard.streamlit.app
Purpose: Dashboard for visualizing the results our metadata operation performance experiment.

A.2 Opendict Benchmark Driver Code

Code: https://github.com/msc-open-metadata/opendic-benchmark
Purpose: Benchmark suite for running the metadata operation performance experiment against widely used database systems and different OpenDict configurations.

A.3 Opendict Polaris Fork Code

Code: https://github.com/msc-open-metadata/polaris
Purpose: An extension to the Polaris catalog that implements the OpenDict specification allowing storage and management of user-defined metadata objects in data lake storage.

A.4 Opendict Polaris Development Environment Infrastructure Bootstrapping Code

Code: https://github.com/msc-open-metadata/polaris-boot **Purpose**: This repository supports OpenDict Polaris development by providing:

- Docker-Compose infrastructure and utility code for building and deploying a local Apache Polaris with the OpenDict extension.
- Tasks to streamline Polaris bootstrap and infrastructure setup.
- Integration tests for a local OpenDict Polaris deployment.
- Sample Jupyter notebooks demonstrating and validating OpenDict Polaris functionality.

A.5 Opendict Snowflake Client Library

Code: https://github.com/msc-open-metadata/snowflake-opendic
PyPi: https://pypi.org/project/snowflake-opendic/
Purpose: A REST client library for OpenDict Polaris and Snowflake enabling user-defined metatadata management via SQL-like syntax.

A.6 Opendict PySpark Client Library

Code: https://github.com/msc-open-metadata/pyspark-opendic
PyPi: https://pypi.org/project/pyspark-opendic/
Purpose: A REST client library for OpenDict Polaris and Spark enabling user-defined metatadata management via SQL-like syntax.

A.7 Python Program: Union of Metadata Structures

Code: https://github.com/msc-open-metadata/open-metadata-spec
Purpose: This was used for early brainstorming in creating a universally compatible metadata object specification. The approach was later abandoned for a user-centric approach.

A.8 Overview of Polaris Ecosystem

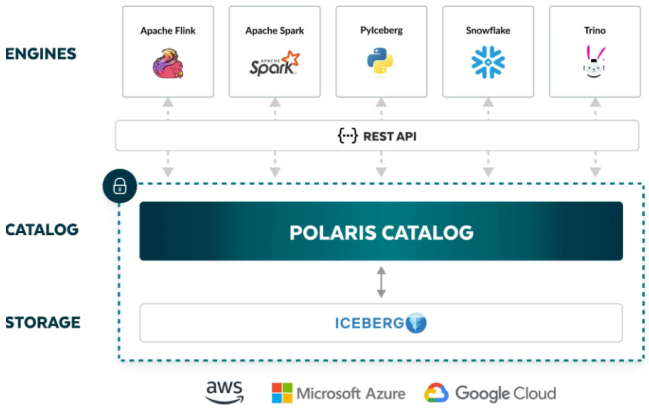

Figure 18: High-level overview of Polaris ecosystem [75].

A.9 Polaris contribution overview tree

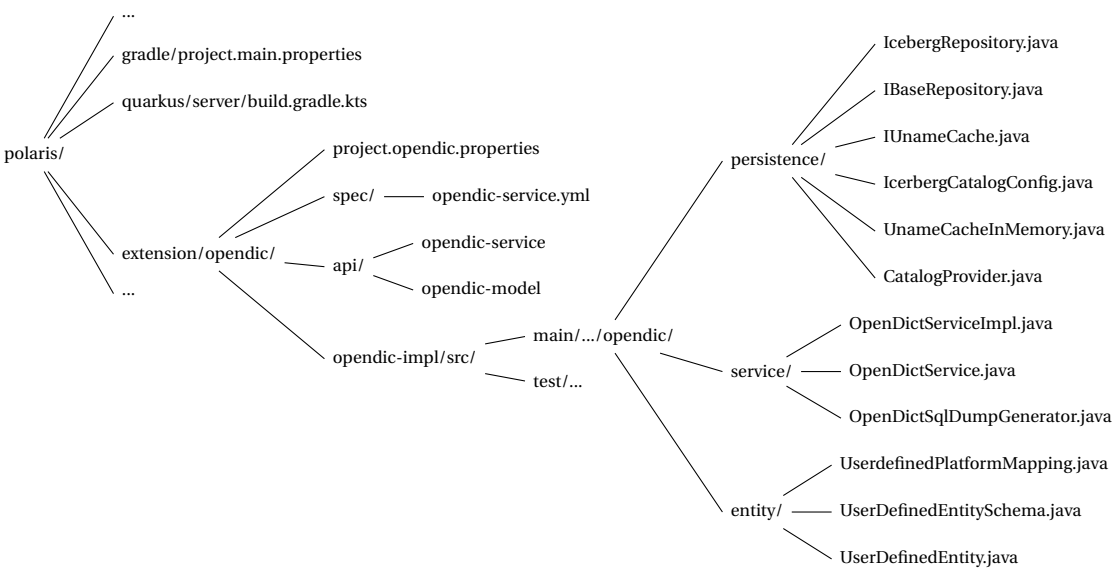

Figure 19: Opendict extension for Polaris. Implementation Overview based on file contributions in A.10.

A.10 Our contributions to the Polaris codebase (git log)

Count	File path
32	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/IcebergRepository.java
20	extension/opendic/spec/open-dic-service.yml
20	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/PolarisOpenDictServiceImpl.java
18	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/PolarisOpenDictService.java
18	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/IBaseRepository.java
13	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/UserDefinedPlatformMapping.java
13	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/UserDefinedEntity.java
12	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/service/OpenDictServiceImpl.java
12	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/service/OpenDictService.java
10	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/IcebergConfig.java
10	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/UserDefinedEntitySchema.java
9	extension/opendic/opendic-impl/build.gradle.kts
8	extension/opendic/opendic-impl/src/test/java/org/apache/polaris/extension/opendic/persistence/IcebergRepositoryTest.java
7	extension/opendic/opendic-impl/src/test/java/org/apache/polaris/extension/opendic/entity/UserDefinedPlatformMappingTest.java
4	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/service/OpenDictSqlDumpGenerator.java
4	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/service/IOpenDictDumpGenerator.java
4	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/UnameCacheInMemory.java
4	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/UserDefinedEntityType.java
3	spec/polaris-management-service.yml
3	settings.gradle.kts
3	service/common/src/main/java/org/apache/polaris/service/admin/PolarisServiceImpl.java
3	service/common/src/main/java/org/apache/polaris/service/admin/PolarisAdminService.java
3	extension/opendic/opendic-impl/src/test/java/org/apache/polaris/extension/opendic/entity/UserDefinedEntityTypeTest.java
3	extension/opendic/opendic-impl/src/test/java/org/apache/polaris/extension/opendic/entity/UserDefinedEntitySchemaTest.java
3	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/IUnameCache.java
3	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/SchemaRegistryService.java
3	extension/opendic/api/opendic-service/build.gradle.kts
3	extension/opendic/api/opendic-model/build.gradle.kts
2	extension/opendic/project.opendic.properties
2	extension/opendic/opendic-impl/src/test/java/org/example/LibraryTest.java
2	extension/opendic/opendic-impl/src/test/java/org/apache/polaris/extension/opendic/service/OpenDictSqlDumpGeneratorTest.java
2	extension/opendic/opendic-impl/src/main/java/org/example/Library.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/PolarisObjectsServiceImpl.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/PolarisObjectsService.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/CatalogProvider.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/ISchemaRegistryService.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/DeprecatedUserDefinedEntity.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/entity/DeprecatedPolarisUserDefinedEntity.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/deprecated_persistence/IRepositoryBase.java
2	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/deprecated_persistence/IcebergRepository.java
1	spec/open-dic-service.yml
1	spec/generated/bundled-polaris-catalog-service.yaml
1	quarkus/server/build.gradle.kts
1	gradle/projects.main.properties
1	extension/opendic/opendic-impl/src/test/java/org/apache/polaris/extension/opendic/persistence/UnameCacheInMemoryTest.java
1	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/IRepositoryBase.java
1	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/persistence/IcebergCatalogConfig.java
1	extension/opendic/opendic-impl/src/main/java/org/apache/polaris/extension/opendic/deprecated_persistence/IcebergConfig.java
1	.gitignore

Table 15: Contributions per file Polaris

A.11 Unity Catalog Table

ID	COLUMN_COUNT	COMMENT	CREATED_AT	DATA_SOURCE_FORMAT	NAME	SCHEMA_ID
c389adfa-5c8f-497b-8f70-26c2cca4976d	null	Managed table	2024-07-17 18:40:05.595	DELTA	marksheet	b08dfd57-a939-46cf-b102-9b906b884fae
9a73eb46-adf0-4457-9bd8-9ab491865e0d	null	Uniform table	2024-07-17 18:40:05.611	DELTA	marksheet_uniform	b08dfd57-a939-46cf-b102-9b906b884fae
32025924-be53-4d67-ac39-501a86046c01	null	External table	2024-07-17 18:40:05.617	DELTA	numbers	b08dfd57-a939-46cf-b102-9b906b884fae
26ed93b5-9a18-4726-8ae8-c89dfcfea069	null	Partitioned table	2024-07-17 18:40:05.622	DELTA	user_countries	b08dfd57-a939-46cf-b102-9b906b884fae

TYPE	UNIFORM_ICEBERG_METADATA_LOCATION	UPDATED_AT	URL	CREATED_BY	OWNER
MANAGED	null	2024-07-17 18:40:05.595	etc/data/managed/unity/default/tables/marksheet	null	null
EXTERNAL	file:///tmp/marksheet_uniform/metadata/00002-5b7aa739-d074-4764-b49d-ad6c63419576.metadata.json	2024-07-17 18:40:05.611	file:///tmp/marksheet_uniform	null	null
EXTERNAL	null	2024-07-17 18:40:05.617	etc/data/external/unity/default/tables/numbers	null	null
EXTERNAL	null	2024-07-17 18:40:05.622	etc/data/external/unity/default/tables/user_countries	null	null

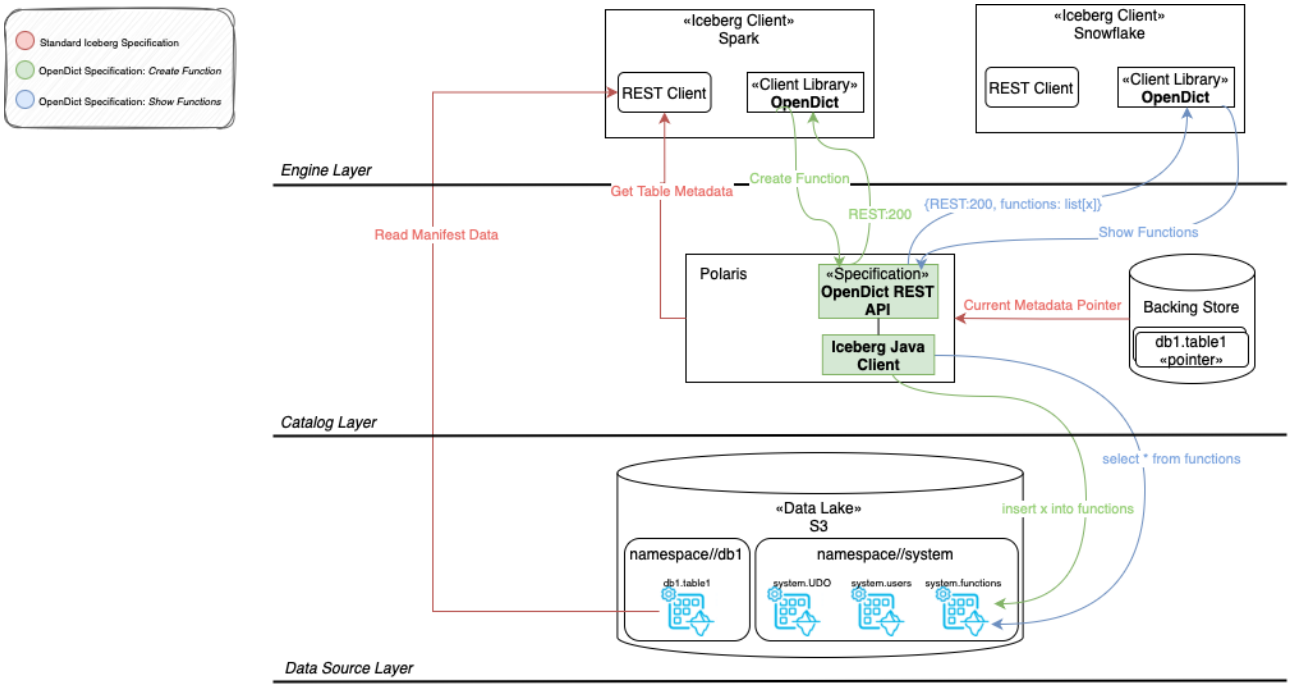

Figure 20: Proof of concept diagram

System	Operation	Complexity
OpenDict (local)	*CREATE*	O(1)*
	ALTER	O(n)
	SHOW	O(n)
OpenDict (cloud)	*CREATE*	O(1)*
	ALTER	O(n)
	SHOW	O(n)
SQLite	*CREATE*	O(n)
	ALTER	O(n)
	SHOW	O(n)
Postgres	*CREATE*	O(1)
	ALTER	O(1)
	SHOW	O(n)
DuckDB	*CREATE*	O(1)
	ALTER	O(1)
	SHOW	O(n)
Snowflake	*CREATE*	O(1)
	ALTER	O(1)
	SHOW	O(n)

Table 16: Simplified Runtime complexity for each system and operation, with n being the number of stored objects.