Delta Sharing: An Open Protocol for Cross-Platform Data Sharing

Krishna Puttaswamy
Abhijit Chakankar
Tao Tao
Zaheera Valani
Ramesh Chandra
William Chau
Mengxi Chen
Databricks
San Francisco, CA, USA

Akram Chetibi
Tianyi Huang
Jonathan Keller
Celia Kung
Andy Liu
Charlene Lyu
Databricks
San Francisco, CA, USA
firstname.lastname@databricks.com

Samarth Shetty
Xiaotong Sun
Steve Weis
Lin Zhou
Ryan Zhu
Reynold Xin
Matei Zaharia
Databricks
San Francisco, CA, USA

ABSTRACT

Organizations across industries increasingly rely on sharing data to drive collaboration, innovation, and business performance. However, securely and efficiently sharing live data across diverse platforms and adhering to varying governance requirements remains a significant challenge. Traditional approaches, such as FTP and proprietary in-data-warehouse solutions, often fail to meet the demands of interoperability, cost, scalability, and low overhead. This paper introduces Delta Sharing, an open protocol we developed in collaboration with industry partners, to overcome these limitations. Delta Sharing leverages open formats like Delta Lake and Apache Parquet alongside simple HTTP APIs to enable seamless, secure, and live data sharing across heterogeneous systems. Since its launch in 2021, Delta Sharing has been adopted by over 4000 enterprises and supported by hundreds of major software and data vendors. We discuss the key challenges in developing Delta Sharing and how our design addresses them. We also present, to our knowledge, the first large-scale study of production data sharing workloads offering insights into this emerging data platform capability.

PVLDB Reference Format:
Krishna Puttaswamy, Abhijit Chakankar, Tao Tao, Zaheera Valani, Ramesh Chandra, William Chau, Mengxi Chen, Akram Chetibi, Tianyi Huang, Jonathan Keller, Celia Kung, Andy Liu, Charlene Lyu, Samarth Shetty, Xiaotong Sun, Steve Weis, Lin Zhou, Ryan Zhu, Reynold Xin, and Matei Zaharia. Delta Sharing: An Open Protocol for Cross-Platform Data Sharing. PVLDB, 18(12): 5197 - 5209, 2025.
doi:10.14778/3750601.3750637

1 INTRODUCTION

Sharing data with partners, suppliers, and customers is an increasingly common need for organizations in all industries to improve their business performance. For example, a retailer sharing data with their suppliers to improve supply chain efficiency, media companies sharing data with their partners to improve advertising. Apart from sharing data *across organizations*, sharing data *within an organization*, across business units acting as different governance domains, is a critical necessity. However, securely, efficiently, and scalably sharing and managing data is a significant challenge today. Different organizations and divisions may be using different data platforms and clouds, may be enforcing different governance rules, and may be consuming data with a diverse set of tools, all of which makes data sharing hard.

There are two main approaches our customers have used for sharing, but they have faced severe limitations in them. Delivering files via FTP is the first approach, and has been a standard in some industries for decades. Many of our customers abandoned it because it was cumbersome for both providers and recipients [19]. Providers had to invest considerable development resources to maintain ETL pipelines to create the data for each recipient, build systems to manage the recipients, and scale the servers as data and number of recipients grew. Recipients, on the other hand, had to invest resources to ingest data regularly and integrate it with their data platforms. The second approach is using proprietary in-data-warehouse sharing features available in platforms such as Snowflake [34], BigQuery [9], Redshift [32], and Azure Data Share [8]. They allow zero-copy sharing between different customers of the same data warehouse. In our customer conversations, poor interoperability stood out as a major limitation of these platforms [19, 24–27]. Data providers wish to deliver data to as many clients as possible, regardless of the clients' chosen computing platforms, and even within *one* enterprise, there are typically many data platforms in use (e.g., due to different choices in departments, corporate mergers, etc.). Furthermore, most data warehouses store data in proprietary formats, and thus do not give direct data access to external tools. This means that data consumers using other tools need to pay extra, and incur performance costs to query the data through the warehouse software.

An ideal solution would meet the following requirements:

- **Open and cross-platform.** Data sharing should work across diverse clouds and data platforms. It should not require clients to change their workflows or migrate to new tools or platforms. And it should minimize the risk of vendor lock-in.
- **Secure.** The system should provide enterprise-grade security. Shared data should be as easy to secure and govern as the rest of the data in an organization.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750637

- **Share live data.** Live data should be shared from the source without making copies to avoid replication costs and to ensure shared data is up-to-date.
- **Performant.** The end to end performance of running queries on shared assets should be comparable to doing similar operations on regular data warehouse assets, even on large datasets in the terabyte and petabyte range.

Delta Sharing is an open protocol we developed to address these requirements that now powers sharing features at Databricks and hundreds of other software vendors and data providers. Our approach builds on standard open formats (Delta Lake [4, 15] and Apache Parquet [30]) and works by sharing short-lived access to the underlying live cloud objects – such as Amazon S3 [1], Azure ADLS [6], Google GCS [21], etc. Parquet is a popular open table format. It is the underlying file format for other richer formats like Delta [15], Apache Iceberg [23] and Apache Hudi [22]. We created simple HTTP APIs that vend these access tokens to read cloud objects for just the data that a client requested (and has access to). The tokens returned from the APIs are HTTP URLs – reading shared data is just like reading any other file. So just by using a Parquet reader and HTTP APIs, clients can integrate sharing capability into the tools of their choice and achieve cross-platform data sharing. Given that the HTTP URLs are for cloud objects, the client can read them in parallel from the cloud and achieve high performance.

1.1 Challenges

To keep the core protocol simple, so integration is easy, and make it to work at scale required addressing several design challenges.

Universal sharing. Users expect to perform all read operations on shared table that they can perform on Delta Lake tables, such as partitions, time travel, change feeds (CDF), streaming, etc. In Delta Lake, these features work on top of the Delta Log, which the protocol hides in order to be cross-platform. So we propose a way to rebuild the warehouse features on simple APIs in Section 4.1. In addition, we generalized the protocol to support sharing of non-tabular assets in the warehouse – volumes, ML models, etc.

Fine-grained sharing (FGAC). Sharing different subsets of a table with different recipients – via views, row-level access policies, or column masks – are high-leverage features that drastically reduce the number of copies of data and policies the providers have to manage. Traditional data warehouses apply FGAC at query time as they are often executed on trusted query engines running within the same domain and are granted access to the full data of the underlying table. This isn't the case in Delta Sharing, where engines can be executing cross-domain, which makes enforcing FGAC challenging. Section 4.2 describes how we address this challenge with a general notion of a cross-domain trusted engine, and with physical data materialization where engines aren't trusted.

Governing shared assets. Enterprises often have a catalog, such as Unity Catalog [10], to govern their data. Both providers and recipients prefer to govern shared assets in the same catalog. For example, recipient organization's users want to configure governance policies on the shared assets (e.g., sales team has access to a shared table but analyst team does not) or discover shared assets with search. However, because they may be in different domains, managing shared data in the recipient catalog requires continuously **reconciling** shared asset metadata from providers as it changes (when tables are added/removed to/from a share) *across domains*. In addition, governance policies for sharing should be designed to be compatible with FGAC policies (for e.g., how an organization-wide ABAC policy applies to a shared asset). We describe our approach to address these challenges in Section 4.3.

Secure sharing. Large enterprises demand stronger security measures, including seamless integration with identity providers (IDPs) of providers and recipients, short-lived tokens bound to IDPs (no long-lived bearer tokens), etc. before they can share data. Designing hooks in the system for defense in depth, while keeping the core protocol simple, is a challenge that we explore in Section 4.4.

Managing cloud costs and billing. With live sharing, managing egress cost is challenging because the cost depends on where the provider storage is and the network path taken by a client query, which are not pre-determined, and replicating data does not always address the egress costs. Finally, because sharing inherently allows one organization (recipient) to access data of another organization (provider), billing the right cost to the right party is a new challenge. We describe our solutions to these challenges in Section 4.5.

Performance. In Delta Sharing, the sharing server sends metadata about files to the clients and the clients read the files directly from cloud storage. To send the right metadata, the sharing server needs to process the Delta Log (typically done by a query engine) in a service, which can be expensive, especially on large tables with millions of files. The sharing server has several optimizations to reduce the number of file metadata it sends to the clients. Furthermore, when a provider shares a large number of tables with many recipients, reconciliation traffic can be significant, which requires careful design. Section 4.6 describes how we address these challenges.

1.2 Workload Study

The sharing protocol has been in production for about four years and is now used by more than 4000 organizations across Databricks and other computing platforms. We are serving tens of millions of queries per month. And the Delta Sharing service in Databricks is powering several other Databricks products such as Marketplace [28], Clean Rooms [11], and System Tables [36]. From all this experience, we have learned a number of interesting lessons.

- **Internal sharing is prevalent.** 40% of data sharing is within an organization across business units.
- **Cardinality of sharing can be high.** Most of the data shared with *external customers* have < 10 organizations; but some providers are sharing data with 100s of organizations.
- **Sharing clients tend to be diverse.** Over half of the providers on Databricks are sharing data with recipients outside Databricks, who are consuming data using 15+ different connectors, showing that an open approach brings diverse customers into the sharing ecosystem.
- **Cross-cloud and cross-region sharing is significant.** About 40% of all sharing is between providers and recipients within Databricks; about 9% of that is cross-cloud and remaining traffic split evenly between same-region and cross-region within a cloud.

- **SaaS data sharing is an emerging use case.** SaaS providers are starting to share SaaS analytics data with all their customers by directly embedding data product into SaaS. Such sharing tends to have very high cardinality.

The key contributions of this paper are threefold. 1) We describe the key sharing use cases based on our view of sharing patterns by 4000+ organizations. We identify some novel use cases such as SaaS Data Sharing with unique patterns. 2) We present the design and implementation of Delta Sharing, the key challenges in developing it and how our design addressed them. And finally, 3) we present the first large-scale study of production sharing workload and a number of learnings from them.

2 USE CASES

We have observed five categories of data sharing use cases.

2.1 Internal Sharing

Many organizations consist of subsidiaries, zones, or divisions that are isolated and potentially use different data platforms. These divisions are different legal entities with different governance and compliance requirements. However, they still want to avoid data silos and share data with each other, effectively forming an intra-organization data mesh. Universal sharing of data and governance are critical for providers. Consuming data in diverse ways – Spark or Pandas or Excel, and in potentially different programming languages is critical for recipients.

2.2 Point-to-Point Sharing

Organizations may want to share data with external partners, vendors, customers, or suppliers for collaboration purposes, for example. Retailers often share product sales data with partners in different clouds or ad networks for marketing purposes, and share similar data with their suppliers to help them manage supply chains better. We have worked with retailers who want to share data with a large number of suppliers/vendors (in 1000s). In these cross-organization sharing cases, participants want to maintain separate governance control of data; data providers need fine-grained sharing – share a particular slice of live data with a particular supplier – and still manage shared data with low cost and overhead.

2.3 Commercial Data Sharing

Data vendors monetize their data and want to distribute data easily (low management overhead), with low cost, and at scale. Reaching customers broadly and efficiently is critical for them. They often tend to have hundreds to thousands of customers, and they want to share live data so updates are available to their customers instantly. In addition, they prefer an open platform to achieve broad customer reach. The recipients of such data want flexibility in consuming data in whatever platforms and tools they already have. Kythera Labs, AccuWeather, S&P Global, Epsilon are some examples of such data vendors in different industries using Delta Sharing to power their business. Hundreds of consumers of data from such vendors are also using Delta Sharing as recipients. The main difference between cross-organization sharing and commercial-vendor sharing is that the data is typically not customized for individual recipients in the latter case, and they often tend to publish their data on data marketplaces (such as Databricks Marketplace [28]).

2.4 SaaS (Software as a Service) Data Sharing

This is an emerging category, where a SaaS data provider wants to share first-party data through their applications with their own customers to enable them to analyze it. Databricks itself is an example of this, where we integrate Delta Sharing capabilities directly into our SaaS applications to share usage and observability data with customers via "system tables," a set of tables visible directly in their data catalog. Aveva and Amperity [2] are other examples of this pattern. The primary focus of a SaaS here is enabling their customers to use data from the SaaS application in their own analytics, which naturally requires supporting as many compute platforms as possible and benefits from using an open Delta Sharing protocol.

2.5 Privacy-safe Collaboration

Privacy-safe collaboration, such as through Clean Rooms [11], enables multiple parties to contribute data into a trusted environment where privacy-safe computations are performed. The results are then shared with the collaborators without revealing private data from any collaborator. This approach is essential for collaboration between organizations that do not want to share raw data with each other to meet regulatory, compliance, or other requirements. Such applications of privacy-safe collaboration are growing, and Delta Sharing provides a strong foundation for building such solutions. In fact, we built Databricks Clean Rooms [11] on Delta Sharing.

3 DELTA SHARING OVERVIEW

In this section, we will provide an overview of Delta Sharing. A high level architecture diagram is shown in Figure 1.

3.1 Essential Components

The main components of the system are:

- **Delta Sharing Protocol.** Delta sharing is a simple REST protocol that supports sharing live data in a Delta Lake between Providers and Recipients.
- **Data Provider.** The Data Provider is the principal who owns the data and is sharing.
- **Data Recipient.** Data Recipient is the principal receiving the data. The recipient is associated with a way to authenticate (bearer token or OAuth token) to access shared assets. Provider and recipient principals may represent an organization, a person, or a group. As a result, many users may be consuming data in the recipient organization, and when their queries come to the provider they would all be treated as the same recipient.
- **Delta Sharing Server.** The server that implements the server side of the sharing protocol. It is also responsible for enforcing governance, tracking usage, etc. of the shared assets.
- **Delta Sharing Connector.** A client that implements the sharing protocol.
- **Tables in Delta Lake.** Tables (and table-like assets such as views, materialized views, streaming tables, etc.) in Delta Lake are stored in formats such as Delta [15], Uniform [37],

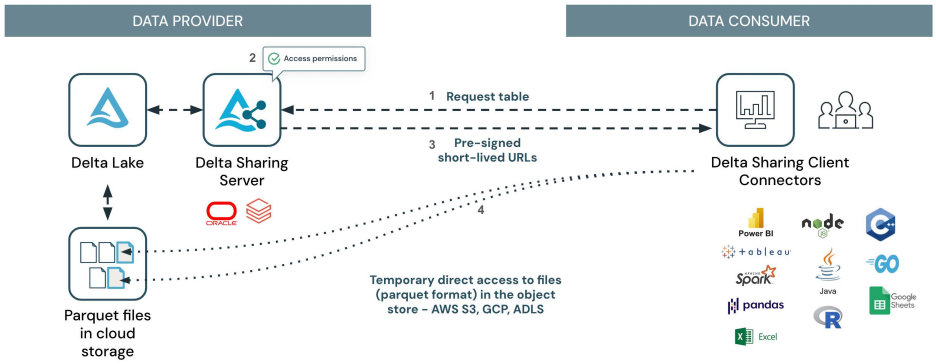

Figure 1: Delta Sharing under the hood.

Iceberg [23], etc., which essentially is a collection of Parquet files stored on cloud object stores such as AWS S3, Azure ADLS, Google GCS.

- **Short-lived access tokens.** The sharing protocol builds on top of cloud storage and allows sharing of files from providers to recipients by sending short-lived access tokens to the recipients. Using which the recipients can access the underlying cloud data object for a certain duration without any additional authentication steps.

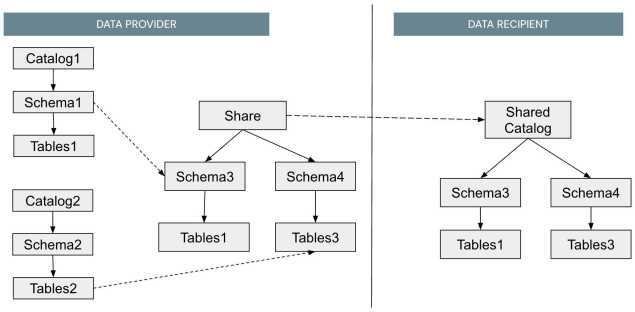

Figure 2: Delta Sharing Object Model showing Share as a unit of isolation for sharing assets from a provider to a recipient.

3.2 Object Model

The protocol is built around three core concepts below. These concepts are depicted in Figure 2 with an example.

Table: Tabular asset (table, view, etc.) is a collection of Parquet files in the cloud object store. The files of the Delta table are physically organized together in a folder or under a prefix in cloud object stores. Providers can specify table aliases when sharing individual tables, allowing recipients to access them under different names. Non-tabular assets such as volumes and ML models are leaf-level assets similar to a table. They tend to be a collection of files, too, although not necessarily in Parquet format. Most of the description in the paper is for tabular assets, but we describe how the ideas extend to non-tabular assets in Section 4.1.

Schema: helps organize tables into human-readable groups. It is a logical group of tables. The tables in a schema need not all be grouped together physically – they can continue to live in their original locations, potentially under unrelated cloud storage prefixes. Similar to individual tables, when a schema is shared, providers can share them under a different alias. When an entire schema is shared, all assets in the schema – current and future – will be shared.

Share: is a logical collection of schemas. Share is the unit of sharing. Share provides a read-only security boundary for the assets within and limits the exposure of data via sharing to only what is *explicitly added to the share*. A share can be shared with one or more recipients. A recipient with access to a share can access all the assets within the share, and newly added assets to a share will automatically be accessible to them.

3.3 Delta Format Background

Let us review the key points of how data is organized in a Delta table, so it is easy to understand how the sharing protocol works on top of Delta. A Delta table at its core is a directory (or an object key prefix) on a cloud object store that contains data objects with table content and a subdirectory containing the *log of transactions* performed on the table. The table contents are stored in Apache Parquet format objects. If the table is partitioned on a column, then the contents of the partitions are organized in sub-directories. Each table content object has a unique name.

The transaction log is stored in a sub-directory called *_delta_log*. It contains a series of JSON files named using sequentially increasing numerical IDs. Each log JSON file contains actions that were performed in one transaction on the table. Actions include operations like *add or remove files*. Each transaction increments the table version and hence performing the actions in a JSON file takes the table from previous version to the version of the JSON file. There are other actions that we do not dive into for the purpose of this paper. For efficiency, the log folder may contain checkpoints that summarize all the operations in the JSON log files up to a certain point, which are in Parquet format.

On top of this format, the Delta format includes read and write protocols for clients to read data or write data to Delta tables. These protocols provide serializable transactions on the Delta tables even if the underlying object store only provides eventual consistency guarantees. The key to this is the log JSON file. Each transaction

writes one file atomically, and a client reading this file has all the information about operations performed in this version of the table.

3.4 Delta Sharing Protocol and Server

Delta Sharing protocol builds on Delta, and works by vending access tokens to specific set of files depending on the query request sent by the client. The sharing protocol does not rewrite or copy the Delta files for sharing, instead it vends tokens to specific files the client needs, to achieve high performance. The sharing protocol [16], is a collection of REST APIs. The sharing server implements the APIs. These APIs are designed carefully to be simple, efficient, and enable clients to leverage the power of cloud storage without the server becoming a bottleneck.

The protocol is depicted in Figure 1. It works as follows:

(1) The recipient's client authenticates to the sharing server (via a bearer token or other methods) and asks to query a table (query API [16]). The client can optionally provide hints to the server to filter the data (e.g. "country=US") to just a subset. Spark Connector built on the protocol, for example, will push such filters via predicate pushdown. If a table's full history is shared, the client can send additional options such as the startingVersion/endingVersion or the startingTimestamp/endingTimestamp for further filtering the files sent back to the client to specific versions of the table.
(2) The server verifies whether the client is allowed to access the data; and then determines which subset of files to send back based on the options provided. This will be the subset of the data objects for the table on the cloud storage that actually make up the table.
(3) Then the server provides access to the data objects corresponding to the subset of files identified. To provide access, the server generates short-lived access tokens in the form of pre-signed URLs. These URLs allow the client to read the Parquet files directly from the cloud provider without coming back to the server (up to a certain configurable duration encoded in the URL).
(4) The client can then take the list of access tokens and read files in parallel at massive bandwidth, without streaming through the sharing server (and hence becoming a bottleneck). The pre-signed URL feature, supported by all the major cloud providers, makes it fast, cheap and reliable to share very large datasets with any number of clients.

4 SYSTEM DESIGN

In this section, we describe how our design addresses the challenges outlined in Section 1.

4.1 Universal sharing

This section describes how we supported all features on shared tables, and then extended sharing to non-table assets as well.

4.1.1 Short-Lived Access Tokens. Pre-signed URL is a mechanism we chose to provide short-lived access to Parquet files in cloud object stores. Pre-signed URL feature is supported by major cloud providers to grant temporary and secure access to objects *without requiring users to have direct credentials to the cloud*. So clients can interact (upload or download files) with cloud storage directly without providing them with privileged, long-term cloud credentials. Because URLs are short-lived by design, the provider does not need mechanisms to track and expire issued URLs, simplifying access management for temporary clients. AWS S3's [31] pre-signed URLs and GCP's signed URLs [20] allow time-limited access to private objects by including authentication details in the URL query string. The URL can be configured with specific permissions and expiration times. Azure offers pre-signed URLs through Shared Access Signatures (SAS) [7] for Blob Storage, allowing users to perform specific actions like reading or writing data within a defined time duration. Even though writes are possible, we limit URLs to be read-only in Delta Sharing.

A few things required care in our implementation on top of pre-signed URLs. First, the expiration time should be chosen carefully to balance usability and security – we chose default time of 1 hour to expire the URLs and we built mechanisms for the clients to renew it for a limited number of times to support long-running queries. Second, cloud platforms support monitoring and logging usage of these URLs out-of-the-box, which the provider should enable to detect abuse/misuse of these URLs.

4.1.2 Supporting various table features. The main advantage of pre-signed URL based approach is that the client gets the relevant files to read and does not need logic to process the Delta Log, and in general be agnostic to how the files are organized in the Delta Table. This simple interface is good enough to support the core features of Delta table as described next.

Time Travel. Delta Lake [4, 15] automatically versions data stored in the tables. This allows for time travel capabilities, which allows clients to access data for any version of the table. This is supported in the Delta Sharing protocol when a table is shared with entire history. Clients can use the *table version API* to discover the latest version of a table. They can lookup earlier versions by specifying the timestamp parameter in the API. Clients can also query data from a specific version of the table using the *query table API* by specifying either the version or the timestamp parameter. The open source Delta Sharing Spark and Python connectors have built-in support for time travel queries.

Change Data Feed (CDF). CDF allows Delta Lake to track row-level changes between versions of a Delta table. When enabled, Delta Lake records "change events" for all the changes in a table. This includes row data, along with metadata indicating whether the row was inserted, updated or deleted. CDF can be used to incrementally and efficiently process changed data.

Delta Sharing protocol supports CDF. Clients can use the table *changes API* [16] to query CDF by specifying parameters such as startingVersion/endingVersion or startingTimestamp/endingTimestamp. Clients can fetch changes in a batch mode, or implement CDF structured streaming. The open source Delta Sharing Spark connector supports both batch and structured streaming for CDF. The open source Delta Sharing Python connector has built-in support for batch CDF.

Structured Streaming. Delta Lake is deeply integrated with Spark Structured Streaming which allows incremental data reads as changes are committed to the Delta table. Delta Sharing protocol

supports Structured Streaming, too. Clients can use the query table API to get data files committed between two versions using the startingVersion/endingVersion or startingTimestamp/endingTimestamp parameters. They can get data files committed from a given version by specifying only the startingTimestamp or startingVersion. Typically, a Structured Streaming job will start from an initial version and then periodically fetch data files that are committed to newer versions of the source tables. The job maintains checkpoints of versions consumed to smoothly recover from failures. The open source Delta Sharing Spark connector has built-in support for Structured Streaming.

4.1.3 Folder Token based sharing. While the file-based approach (based on pre-signed URLs) is simple and easy to build integrations for diverse clients, it has limitations for cases where there are a large number of files (more on this in Section 4.6) or when the asset is organized such that a single file does not fully contain all the necessary information (like the .JSON file in Delta table) thereby requiring a client to have access to the asset's root folder.

Volume, for example, has features like listing files in a folder, such *list* operations are either inefficient on the table APIs or require protocol changes. As more, particularly non-tabular, assets are supported in the warehouse we want to be able to share them automatically with a generic approach. Folder-token based sharing is a form of short-lived access we designed to support such use cases. This approach provides temporary tokens with read-only permissions to an asset's root folder in cloud storage. Using this credential, the client can perform asset-specific operations on the shared asset just like it would do on the original asset – for example, the client can list specific sub-folders in a volume, or load an ML model for serving, etc. The main downside of this approach is that it requires the client to be aware of the asset type and perform appropriate operations, which is acceptable when the clients have that asset-specific logic already; for example in D2D sharing the recipient's Databricks runtime (DBR) can read shared asset from Databricks provider just like a regular asset.

We are working on bringing this generic approach to the open source implementation of the protocol and connectors.

4.2 Fine-grained sharing (FGAC)

Some sharing use cases require giving access to a subset of table's data based on the recipient accessing it. For example, a provider may have a multi-tenant table, and they want to restrict rows and columns shared with specific tenants. These fine-grained access controls are typically expressed as views with *current_recipient* [12], row level security(RLS), column masks (CM), role-based access control (RBAC) and attribute-based access control (ABAC). So the sharing server should ensure that the data shared via the protocol APIs is compliant with FGAC.

The simplest form of FGAC can be supported by physically partitioning the base table and sharing specific physical partitions of it with specific recipients. For example, a SaaS provider can create a multi-tenant table partitioned on a column 'tenant_id' which represents the tenant's unique id. Since these partitions are physically separated from each other, they can be securely shared with their respective tenants without any storage or performance overhead. However, this approach does not generalize.

To properly support views and more advanced forms of FGAC such as RBAC and ABAC, we first determine if the query is coming from a 'trusted' engine and depending on that decision, we either give access to the base table or to a materialized copy of the data. If the recipient query engine is 'trusted', i.e., isolated from the user where the user cannot bypass its security enforcement mechanisms, the sharing server sends FGAC related filters and masks to the trusted query engine and grants temporary access to the underlying base tables. The trusted engine, by definition, guarantees that the FGAC related filters and masks are always applied to the user query. If the query engine is untrusted, then the FGAC compliant subset is materialized on a trusted compute on the provider side, and access is granted on the materialized subset to the query engine.

Each sharing server needs to determine which engines it trusts. A trusted relationship between the sharing server and the query engine is typically only possible if the entity that owns the sharing server also owns the query engine. For example, if both the data provider and the data recipient are on Databricks, the Databricks sharing server and the Databricks runtime (DBR) query engine negotiate and establish this trust on every user query. If the query originates from a non-Databricks query engine, it is always considered untrusted.

4.3 Governing shared assets

In this section, we use our experience integrating Delta Sharing with Unity Catalog (UC) in Databricks as an example to describe solutions to governance challenges around Delta Sharing. Data in UC is organized in three-level hierarchical namespace as shown in Figure 2 (catalog contains schemas, which contains tables and other leaf-level assets). When a share is mounted on the recipient side, we mount it as a catalog (the top level container namespace in UC). All catalog metadata are hosted in a single database called the Metastore.

Provider Side Share Governance. Organizations have sensitive data in their catalog; accidentally sharing data outside an organization can lead to serious security incidents. As a result, the ability to share data should be guarded behind sufficiently high privileges. In Databricks UC, only users with CREATE_SHARE privilege can create shares. Each share has an owner; and we enforce the principle that the share owner must have SELECT privileges on shared assets to be able to share it with a recipient. If the owner loses SELECT permissions on a shared asset, then that shared asset becomes inaccessible to the recipient, to prevent escalation of privilege attacks where someone may gain access to an asset via sharing that they cannot directly access. This check is enforced both when an asset is added to a share as well as at query time.

FGAC brings other interesting challenges when sharing as a provider. For example, if there is an organization-wide ABAC policy to mask PII, can the owner only share after applying this policy? What if a principal responsible for disaster recovery wants to share the raw data before applying the policy with another zone/region to replicate it? We resolve such conflict by giving ABAC policy owners an option to exclude some principles from ABAC rules. Such principals can acquire CREATE_SHARE privileges and share raw data before the policy is applied.

Recipient management. As a provider, even the ability to creating recipient objects (that encapsulate the principal that receives the share) should be guarded. In Databricks UC, only users with CREATE_RECIPIENT privilege can create recipients; and users with SET_SHARE_PERMISSION can grant share access to a recipient.

Recipient Side Share Governance. In Databricks Unity Catalog, an incoming Share can be viewed as a remote catalog that can be "mounted" locally as a "shared catalog." After mounting, the shared catalog looks just like any other catalog to the user, and they can access the assets within like a regular catalog. But behind the scene it makes remote calls to the sharing server instead of accessing data locally.

Similar to the concerns on the provider side, consumers of data, especially in large enterprises, may want users with special privileges to perform this action, and govern access to these shared assets. In Databricks UC, only users with USE_PROVIDER, and CREATE_CATALOG can perform this mount. The owner who mounted the catalog can govern this data and provide access to specific principals within the recipient organization, just like any other catalog in the metastore.

Finally, Metastore administrators have the ability to turn on (or off) Delta Sharing at metastore level, and control which principals get which of the privileges described above.

4.4 Secure sharing

Delta Sharing comes with several security capabilities. Data providers are able to grant or revoke access to recipients at any time, and are able to grant access to only the slice of data the recipient needs (FGAC). The Delta Sharing server, by default, has features to track and audit access to the data by recipients. When a recipient is created on a provider, they get a link to download a recipient profile with a long-lived bearer token. We enhanced this to control the expiration time of the token on the provider side. However, these were not enough for larger enterprises who needed defense in depth. As a result, we added several other capabilities.

4.4.1 IP Restrictions. We added the ability for providers to assign IP access lists that restrict recipient access to specific IPv4 addresses ranges. These restrictions apply to REST API access, activation URLs, and credential file downloads, ensuring that only trusted network locations can interact with shared resources. Administrators can configure these lists using the Databricks Unity Catalog. This approach improves security by combining network-level restrictions with token-based authentication, making unauthorized access significantly harder.

4.4.2 OAuth-based Authentication. Security-sensitive organizations prefer to use short-term tokens with more fine grained controls. To support such customers, we added support for OpenID Connect (OIDC) based authentication for Delta Sharing as another option. This enhances security by allowing recipients to authenticate using OIDC or OAuth tokens from their trusted Identity Providers (IdPs). When a provider creates a recipient for external users, they specify an identity federation policy that dictates which external recipient's IdP and which users/principals/groups from that IdP can access the shared data. In this approach, the recipients authenticate against their own IdP, eliminating the need for shared secrets between provider's Delta Sharing server and the recipient, and then use the issued OAuth tokens to access data on the provider. As long as the token policies are met by the token (which the provider's server checks using standard OAuth protocol), the provider can grant access to data.

4.5 Managing cloud costs and billing

Managing cloud costs is already complex, and sharing makes it more complex by adding new factors. The cost of querying a shared table is primarily a function of resource type (compute and storage type), cloud provider (providers have different cost), where access happens (access from different region/cloud lead to different egress costs). The last factor is novel to sharing.

4.5.1 Egress Cost Management. Live sharing is the default in Delta Sharing because it enables real-time data access with a single copy on the provider (low storage cost) and is simple to setup and maintain. However, it can potentially incur high and unpredictable egress costs on the provider. On cloud providers, the egress cost varies based on the region from which data is egressing as well as on the amount of data egressing, which in turn depends on the number of recipients and frequency of access by users in the recipient organization. So popular providers tend to incur higher egress costs.

However, different scenarios benefit from alternative replication options. Recipient-side replication, one option, places the onus on the recipient to maintain their data copies, resulting in egress costs as a function of the number of recipients (much smaller than number of queries). Another option is for providers to replicate data across different regions and redirect requests to provider's local-region copy of data. This results in egress costs from replicating to different regions, but results in no cost when recipients query local copies of data. These choices depend on the number of recipients, frequency of access by a recipient, who is more capable or willing to handle the associated replication costs, and operational overhead. This replication itself can be made efficient by leveraging Change Data Feed (CDF) to manage incremental updates, or by employing materialized views to auto-refresh shared tables – which reduce the amount of data transferred to updated the copies as well as keeps the tables in sync with low latency.

To further optimize and manage costs, Delta Sharing is integrated with Cloudflare R2 [13], which offers zero egress fees; thus it is most appropriate for data providers with many recipients or frequent queries. A minor downside of this approach is the cost of transferring data from their source cloud provider to R2, which is negligible compared to the cost savings.

4.5.2 Billing. Another challenge that is unique to Delta Sharing is appropriately billing the cost of cloud resources to the right participant. For FGAC (view sharing, RLS/CM, etc.) we may have to materialize the result on provider side in response to a query. Similarly, as discussed, there is egress cost on the provider in response to queries. But the recipient may be from a different organization; they may have a different account on Databricks, or they may be an open recipient with no account on Databricks. Accurately billing the cost of resources consumed on the provider is critical for providers. The providers can then use the incurred cost to either charge the

right recipient account or for open recipients allocate some budget strategically and limit resources consumed appropriately.

4.6 Performance

There are a few aspects to performance. One is the query performance – recipient user's perceived end-to-end time for running the query. The second is the performance of metadata reconciliation, which is necessary for the recipient user to see most updated metadata from the provider.

4.6.1 Efficient query processing. For the clients to achieve good end-to-end performance, Delta Sharing server should minimize the number of pre-signed URLs sent to the clients and the time it takes to do so. To do that, we allow predicate pushdown from the clients, have a number of service optimizations, and finally leverage folder-token based sharing where possible.

Predicate Pushdown. When a Delta Sharing client queries a table, it can pushdown predicates in their request. This improves efficiency both on the server (fewer files to sign) and on the client side (fewer files to read). These predicates represent filtering conditions in a structured JSON format, enabling complex logical operations such as 'and', 'or', and comparisons like 'greaterThanOrEqual' or 'lessThanOrEqual'. Each Delta table maintains metadata in its transaction log, which includes file-level statistics such as the minimum and maximum values for each column. These statistics enable Delta Sharing to determine whether a file contains data relevant to the a query's predicates. For example, if a query filters rows where 'age < 20', the server can skip files where the 'min_age' is greater than or equal to 20, as those files cannot contain matching rows. In addition to this metadata, we can also use the partition column values for filtering. The server evaluates this request against the table's metadata and partition columns, and uses Delta Lake's optimizations such as data skipping and indexing, to retrieve only the files that satisfy the predicates. As a result, it reduces the number of files, and hence the amount of data the client has to read from cloud storage.

Service Optimizations. A petabyte scale table may contain millions of data files and a large Delta Log. Generally query engines process the Delta Log, but that should be done in the Delta Sharing server to support cross-platform clients in Delta Sharing, and requests on large tables can be expensive (and non-uniform). We have a number of optimizations in the service to cope with this challenge. Autoscaling the service compute, caching state read from DB and storage, isolating the thread pools for unpacking Delta Log (I/O intensive) from the pre-signing (cpu intensive) thread pools are some such optimizations.

Folder-token based sharing. This mode of sharing is significantly more performant than pre-signed URLs because it eliminates the cost of pre-signing files within a folder, which dramatically improves the query startup time for tables with many files. However, this requires connectors to implement more APIs and build better integration with Delta Lake (like interpret the Delta Log for some operations) and hence is hard to build into many clients. Query engines that already have this capability can tell the server to use this mode instead of using pre-signed URLs.

4.6.2 Optimizing Reconciliation. Reconciliation – the problem of keeping metadata up-to-date in the recipient catalog with the provider metadata – can generate a lot of traffic. Imagine a commercial data vendor sharing data with N recipient organization each with U users on average. This introduces metadata requests from N * U sources. A simple way to reconcile is to "pull" metadata every time a user visits the shared catalog in the catalog UX. This will make the traffic a function of how often the N * U users visit the catalog page, which can be very high. Furthermore, each request can be expensive as a vendor may be sharing 1000s of assets per share. We started with this approach in Databricks. But we have now transitioned to "push" mode where the provider pushes change events when there are changes to metadata and the recipients consume it to update their metadata.

5 IMPLEMENTATION

Delta Sharing protocol, reference server, and several client connector implementations have been open-sourced [17] on GitHub since April 2021. We have over 15 open source client implementations including Spark, Pandas, Python, Go, R, Rust, Node.js clients, among others. Several open/proprietary clients are implemented by companies such as Microsoft PowerBI, Tableau, Exponam, etc. The open source project is actively used and is receiving code contributions for many organizations beyond Databricks.

5.1 Connector Implementation

Delta Sharing protocol uses restful APIs for metadata discovery and pre-signed URLs for accessing data stored in Parquet format. This combination makes connector development simple. A typical connector implementation consists of the following main steps.

- Connectors get a profile file to get access to shared assets. A profile file is sent to the recipient when a recipient is given access to a share.
- The client can use ListShare/GetShare APIs to discover shares that are available for access.
- They can then use APIs such as ListSchema/ListTables to discover shared assets in a share.
- They can use the query table API to fetch pre-signed URLs for the Parquet data files. And then use an open source Parquet reader to read data from the files. Connectors can expose time travel and streaming related parameters to their users.
- They can use the table change API to fetch pre-signed URLs for Parquet change files. Connectors can expose CDF batch and streaming related parameters to their users.
- Another strategy for connectors is to leverage the Delta Kernel [14] library to process the data files. This will allow connectors to automatically get advanced Delta Lake features such as Deletion Vectors and Column Mapping. It decouples the connector implementation from Delta Lake features and supports better interoperability with Delta Lake.

Python Connector. The Delta Sharing Python connector is a Python library that implements the Delta Sharing Protocol to read tables from a Delta Sharing Server. You can load shared tables as a Pandas DataFrame, or as an Apache Spark DataFrame if running

in PySpark with the Apache Spark Connector installed. The initial version of the connector followed the standard connector implementation described above. Once the Delta Kernel Rust library was open sourced, parts of the connector were re-written to leverage it. It now supports all advanced Delta Lake features supported by Delta Kernel [14], including Deletion Vectors and Column Mapping.

Spark Connector. The Spark connector uses the same set of APIs described above. It plugs them into the Spark query engine via Spark DataSource [35]. The connector uses the query table API's batch and stream processing capabilities. For batch, the client queries the list of files for the current version of the table and implements the DataSource primitives necessary to get a DataFrame for the shared table. For Structured Streaming, the connector queries new files using the startingTimestamp and startingVersion parameters to incrementally fetch new files and then implements the primitives needed to support a Streaming Source.

6 PRODUCTS AND COMPANIES SUPPORTING DELTA SHARING ECOSYSTEM

Beyond Databricks, a number of other companies are building on top of Delta Sharing in the broader ecosystem. In this section, we highlight a few of them.

6.1 Sharing Servers

The open source project on Github provides a reference Delta Sharing server [18] implementation in Scala. This is typically used as a starting point by other organizations to build their own servers. Several vendors such as Oracle [29], SAP [33], Amperity [3], and Aveva [5] have implemented their own Delta Sharing server. They typically add additional customizations on top, particularly for integration with internal data governance systems, integration with authentication systems, and persistent state to manage shares and recipients. These vendors act as data providers as well as data recipients, which has enriched the open Delta Sharing ecosystem and encouraged interoperability. In fact, support for OAuth authentication mechanisms was built to meet the security requirements of one of these vendors as they were building on top of the OSS sharing server to serve their customers.

6.2 Connectors

Companies such as Microsoft, Tableau, Exponam have added new connectors to the ecosystem. In fact, as we will show later, PowerBI and Microsoft PowerQuery are among the top connectors used by open recipients. Beyond these, the open source community continues to add new connectors.

7 PERFORMANCE EVALUATION

In this section, we dive into some important performance benchmarks of the protocol. To show the performance of queries on shared data, we created tables of different sizes with synthetic sales data. Specifically, we created three tables with 10, 100 and 1000 Parquet files with each file having 2 million records. These tables were shared via the Delta Sharing protocol, and were queried via the SQL Warehouse.

Table 1 shows the performance of running the exact same set of query on shared table vs directly on the source table in the SQL Warehouse. This query was on the entire table's data without any predicate pushdown. The extra overhead in the query time of shared data varies from 13 to 30% relative to querying on source data. The overhead reduces as the table size increases from 10 to 100 files and then stays nearly constant afterwards.

Table 1: Comparison of query performance directly on source data table vs query on shared data table.

Table size	Source (secs)	Shared table (secs)
Table with 10 files	2.73	3.58
Table with 100 files	14.69	16.63
Table with 1K files	53	62

Table 2: Performance on shared tables with and without predicate push down (PPD).

Table	With PPD (secs)	Without PPD (secs)
Table with 10 files	2.99	3.58
Table with 100 files	7.28	16.63
Table with 1K files	7.76	62

Table 2 shows the result of running a query with and without predicate push down on the SQL Warehouse. The predicate pushed down was the limit the query to process sales from a single region. With predicate push down, the performance of queries remains nearly constant (after an initial increase going from 10 to 100 files).

8 WORKLOAD STUDY ON THE DATABRICKS PLATFORM

We have been operating the Delta Sharing service in production at Databricks for close to 4 years on all major cloud providers. It allows customers to share data (as provider) on four different clouds: Amazon AWS, Microsoft Azure, Google GCP, and CloudFlare R2, or consume data (as recipient) anywhere on the Internet, including within the Databricks platform. In this section, we present a study of the sharing workload on how the platform is used by thousands of organizations. For the content in this section, we have removed data where Databricks is providing data as a SaaS provider and purely focus on how **external customers** are using this platform.

8.1 Cross-domain sharing adoption over time

In the month of October 2024, over 4000 companies are sharing data with other companies through 5000+ shares. The growth in the number of active (with usage in the last 30 days) sharing customers as either a provider or a recipient is shown Figure 3. Figure 4 shows the growth in providers on the platform. These graphs both exclude data related to Databricks as a provider. There were 12,000+ customers on the Databricks platform at the time; A customer is defined as an organization whose usage of Databricks is costing them at least 1000 USD in the last 30 days. We see from the graphs that companies are embracing cross-domain sharing at a rapid pace.

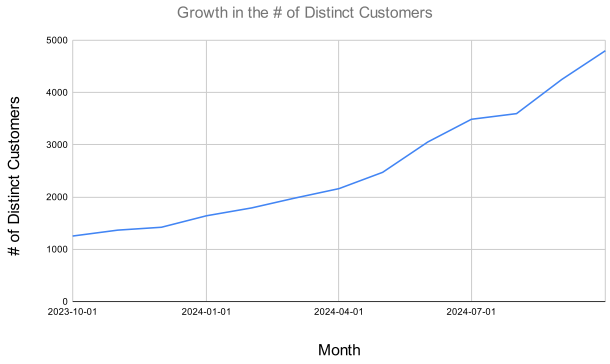

Figure 3: Number of active Delta Sharing customers (providers and recipients) on the Databricks platform. This graph excludes data related to any sharing by Databricks as a provider to only measure usage by external customers.

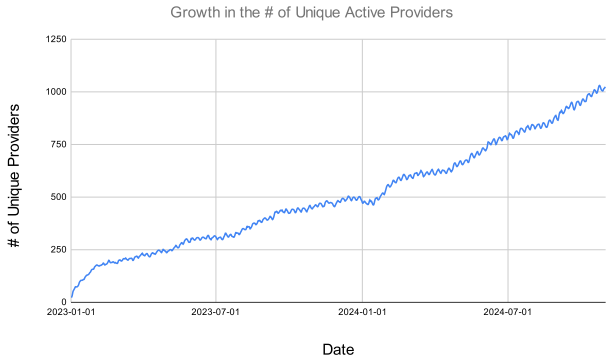

Figure 4: Number of active Delta Sharing providers sharing data with external recipients on the Databricks platform.

8.2 Sharing across domains, regions, and clouds

In terms of cross-platform sharing, *D2O sharing* where the provider is on Databricks but recipient is *outside* Databricks, is the majority. ~55% of shared data access was from D2O. D2O recipients outside the Databricks platform use a number of different connectors – Pandas, PowerBI, Microsoft PowerQuery, Apache Spark, etc., without any one tool standing out. Indicating that sharing across organizations is a necessity for modern businesses and an open approach brings more participants into the sharing ecosystem. O2D (where the provider is outside Databricks and recipient is on Databricks) accounts for less than 1% of the traffic.

After excluding the D2O and O2D traffic, the remaining ~44% of traffic is D2D (provider and recipient both on Databricks), 9% of the access is across clouds and the remaining 35% traffic is all within the same cloud. This 35% traffic is split evenly between access within the same region and access from another region in the same cloud. This shows providers and recipients are often on different regions or clouds and prefer to share in situ without migrating their platforms.

8.3 Cardinality of sharing by providers

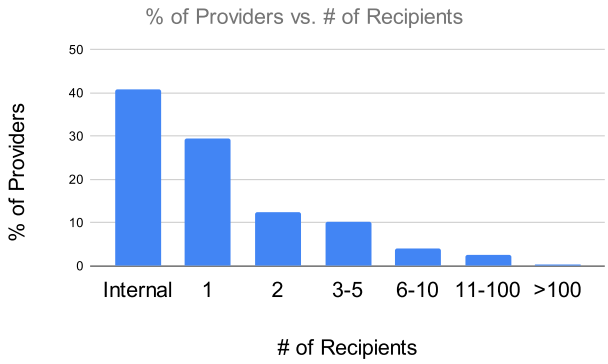

Figure 5: Distribution of % of providers with certain number of recipients. Internal refers to providers sharing data with recipients who are internal to the organization.

When we dive into the # of recipients a provider shares data with, several interesting points stand out. The graph in Figure 5 shows % of providers with a certain number of recipients out of all the active shares in Oct 2024.

- Internal sharing, where several subsidiaries or divisions within the organization share data with each other constituted the majority (approx. 40%) of the shares.
- Sharing with one outside recipient, was the next big bucket with close to 30% of the shares.
- Sharing with a small group of recipients, between 2 to 10 recipients, was close to 26% of the shares.
- Sharing with a dozens or even hundreds of recipients was close to 3%. There were several hub vendors sharing data with more than 500 other companies. Some popular hubs on the Databricks Marketplace include Acxiom, HealthVerity, LiveRamp, S&P Global, John Snow Labs, among others.

This shows that Delta Sharing is capable of supporting a wide variety of providers – those sharing data with just one recipient to large hubs sharing data with a 100s or 1000s of recipients.

8.4 Queried table size distribution

Out of the tables queried in the month of Oct 2024, we present the distribution of data read by those queries when there was no filter pushed down in Table 3. In general, we see smaller shared tables queried much more frequently compared to larger tables. Over 95% of the queries on shared table were of size less than 10GB. But there are a small % of queries for shared tables of size over 1TB. This shows that even though smaller data is more frequently queried, some providers do need to share tables 10s of TBs in size. The Delta Sharing platform is capable of scaling to sharing large tables.

8.5 Heterogeneity of clients using the platform

Figure 6 shows the usage of different connectors in terms of % of recipients using them in the year 2024 until October against Databricks server. Spark connector running within Databricks,

Table 3: Distribution of table sizes read by queries on shared tables. Most queries are for smaller tables. But we do see large shared tables queried.

Table Size Category	% of Overall Queries
<1GB	83.485000
<10GB	12.450000
<1TB	3.900000
<10TB	0.10
>10TB	0.14

which accounts for 60% of usage by all recipients, is excluded from this graph to focus on usage from outside Databricks. Tableau is missing in the graphs due to instrumentation issues.

The open source connectors are used by about 40% of the recipients consuming data from the Databricks server. Out of these recipients, each of PowerBI, PowerQuery and Pandas Open source connectors are used by a little over 25% of the recipients. The rest of the connectors in aggregate are used by about 21% of the recipients. We see 18 non-Databricks clients making calls to our server. Clients even in languages often not used for data analysis – Go and Ruby – are consuming data from sharing server. This shows that our decision to keep the protocol simple to make it easy for diverse connectors to consume from the sharing server is helping the client ecosystem thrive.

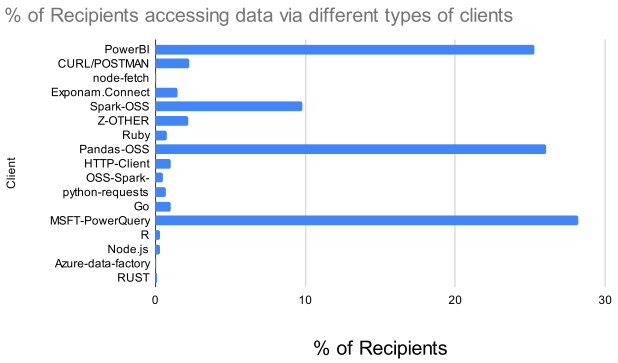

Figure 6: Distribution of recipients using different types of client connectors to connect to the Databricks sharing server.

8.6 Types of shared assets queried

Figure 7 shows the distribution of queries to different shared assets using token-based sharing within the Databricks platform in January 2025. Queries to shared tables dominate usage, but there is growing demand to share other assets, even non-tabular assets such as volumes and functions. We also see that fine-grained sharing via views (which is a relatively new feature we added in 2024) is gaining rapid adoption.

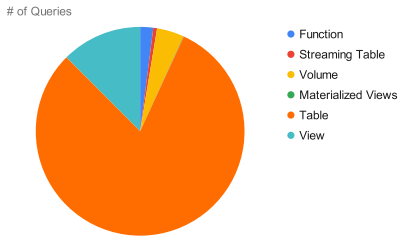

Figure 7: % of queries by different asset types. Queries on tables dominate, queries on other assets are gaining usage.

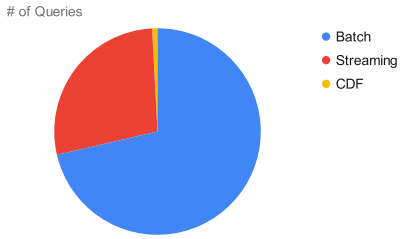

Figure 8: % of queries by type. Batch and Streaming queries are quite common while CDF queries are a small fraction.

8.7 Types of queries on shared data

Figure 8 shows the distribution of different types of queries on shared assets in January 2025. Customers are using various warehouse features on shared tables. Batch queries are close to 70% of all queries. However, a significant number of customers are running Streaming queries, close to 25% of all queries, on shared data. CDF queries are run on shared data too, but it is a small fraction.

9 CASE STUDY: DELIVERING SYSTEM TABLES WITH DELTA SHARING

System tables in how Databricks shares various analytical data about a customer's usage of Databricks back with the customer [36]. It allows customers to do centralized monitoring and analysis of platform usage, billing, and access activities, providing a robust foundation for operational intelligence within the Databricks Lakehouse environment. In this way, customers can get deep observability into their usage of Databricks out-of-the-box.

System tables is internally built as a platform that allows different product teams to surface relevant operational data to customers in a unified way. The operational data tables are stored in a Databricks-managed storage account and are securely shared with customers using Delta Sharing. System tables data is partitioned by account ID, and individual partitions are shared with the customers owning that account ID, thus ensuring they can only access data relevant to their own accounts. Delta Sharing allows customers to access their live operational data without the need for complex ETL processes, thereby reducing latency, cost, and complexity.

System table and all Databricks internal usage was removed from the data we reported in Section 8. If we include them, System tables

is by far the largest user of Delta Sharing – we are sharing analytical data with almost every 12,000+ customer of Databricks (the biggest external data provider is almost 10X smaller in cardinality in comparison).

10 RELATED WORK

There are two broad categories of systems related to Delta Sharing: a) FTP and data dumps are traditional methods for sharing data, and b) using sharing features in modern data warehouses, which share between instances of that same data warehouse. We describe how Delta Sharing differs from them in detail in this section.

Sharing via FTP/SFTP. Sharing data via FTP is common, perhaps not as prevalent as it used to be. It is a versatile solution supported by most operating systems, with low barrier to use. However, it has a number of limitations. As a recipient, one will have to build and maintain ETL pipelines to copy data from the server and keep it up to date, which introduces operational overhead. Providers face even more limitations. They will have to maintain ETL pipelines to create different filtered datasets for different recipients, and cannot easily scale the system to serve a large number of recipients. Furthermore, it can be cumbersome to manage recipients, requiring manual account creation and password/token resets by administrators, which increases security risks. To address these limitations, providers often need to invest significant resources to develop a custom infrastructure and integrate it with their other systems.

Dumping data into cloud storage such as S3/GCS/ADLS. While this is more modern and scalable than FTP, it comes with many of the same challenges. Providers still have to maintain ETL pipelines to create filtered datasets per recipient and push it to the recipient's storage bucket. Access controls and audit trails are coarse-grained, at the object storage level. Managing recipients and their permissions at the object storage level remains cumbersome. Recipients will have to integrate their tools and workflows with these data dumps. If a recipient consumes data from different providers, these integrations will have to adapt to different formats from the different providers.

Unlike FTP or data dumps, Delta Sharing allows live access to shared datasets without requiring complex ETLs. Providers can share structured tables, at fine grain, with a large number of recipients, with low management overhead. Table management facilities such as time travel, change data feeds and streaming consumption are easily supported. In addition, Delta Sharing supports cross-platform compatibility by enabling recipients to consume shared data directly in standard format in their tools of choice. Because the bulk of the data is in Parquet format, tools that integrate with Parquet can fairly easily be adapted to support Delta Sharing (they mostly need to request some data URLs from the server and read Parquet from a HTTPS URL).

In-data-warehouse sharing features. Snowflake Sharing [34], Azure Data Share [8], Google BigQuery Analytics Hub [9], and Amazon Redshift Data Sharing [32] are other options available for customers in this category.

Snowflake Sharing focuses on seamless real-time data sharing within the Snowflake ecosystem. It allows live access to shared data and provides robust security features. However, it is limited to sharing within Snowflake. When data is shared with external users, they are still required to create "reader accounts" on Snowflake and access it there. Azure Data Share offers flexibility with both snapshot-based and in-place data sharing, but it requires Azure subscriptions for both providers and consumers – as in both parties have to be on Azure. Google BigQuery Analytics Hub emphasizes privacy-safe collaboration through features like data clean rooms but is restricted to only Google Cloud ecosystem. Amazon Redshift Data Sharing enabling live in-place sharing across Redshift clusters within AWS; it supports workload isolation by allowing separate clusters for different teams but sharing same data. However, data sharing is still limited to AWS Redshift users.

All data warehouses discussed above restrict sharing to be **only within** their system/cloud. Furthermore, many clouds store data in proprietary formats. Together, it severely sacrifices interoperability across tools and platforms. This restricts data providers who want to serve a large number of customers in different clouds. To overcome these problems the providers are forced to, again, maintain ETL pipelines to copy and update data to different clouds. And then they have to manage their recipients in each cloud, perform audit/tracking across clouds, etc. creating a lot of management overhead. This restricts recipients, too: recipients have to adapt or migrate to new clouds/tools. If they want to consume data from different data providers, especially if they are not on the tools/clouds the recipient uses, they are forced to use different platforms and tools, replicate data, etc. These limitations create artificial barriers for organization to collaborate and share data.

In contrast, Delta Sharing's open interface allows cross-platform interoperability and removes barriers for data collaboration. This approach leverages simple REST APIs and standard open formats like Delta Lake and Parquet. This open approach avoids vendor lock-in, works with existing tables in the Parquet and Delta Lake formats, and supports diverse workflows across cloud platforms and enables interoperability with even non-Databricks tools like Tableau, PowerBI, Pandas, and others. Many customers have cited these reasons for using Delta Sharing [24–27].

11 CONCLUSION

This paper introduces Delta Sharing, an open protocol for secure, efficient, and cost-effective sharing of data within and across organizations. Delta Sharing builds on open formats like Delta Lake and Apache Parquet, with simple HTTP APIs on top to vend access to cloud objects, thus making it easy to consume shared data live in heterogeneous client tools and platforms. We outlined the key challenges in building Delta Sharing and how our solutions addressed them. We also presented what we believe to be the first large-scale study of production sharing workload based on usage by over 4000 organizations since its launch in 2021. The study shows that: 1) Our open approach and simplicity of the sharing protocol have enabled interoperability across a wide set of tools and platforms. 2) Many organizations, beyond Databricks, have adopted and are participating in advancing the ecosystem. 3) Delta Sharing has empowered organizations to seamlessly collaborate with tabular and non-tabular assets both within and outside their organizations. We hope that the insights from our study will help researchers and practitioners understand this emerging data platform capability.

REFERENCES

[1] Amazon S3. https://aws.amazon.com/s3/.

[2] Amperity builds an innovative consumer data platform with Delta Sharing. https://www.databricks.com/blog/delta-sharing-and-emergence-lakehouse-customer-data-platform-cdp.

[3] Amperity Bridge. https://docs.amperity.com/datagrid/bridge_databricks.html.

[4] M. Armbrust, T. Das, S. Paranjpye, R. Xin, S. Zhu, A. Ghodsi, B. Yavuz, M. Murthy, J. Torres, L. Sun, P. A. Boncz, M. Mokhtar, H. V. Hovell, A. Ionescu, A. Luszczak, M. Switakowski, T. Ueshin, X. Li, M. Szafranski, P. Senster, and M. Zaharia. Delta lake: High-performance acid table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13:3411–3424, 2020.

[5] AVEVA and Databricks Forge Strategic Collaboration to Accelerate Industrial AI Outcomes and Enable a Connected Industrial Ecosystem. https://www.aveva.com/en/about/news/press-releases/2024/aveva-and-databricks-forge-strategic-collaboration-to-accelerate-industrial-ai-outcomes-and-enable-a-connected-industrial-ecosystem/.

[6] Azure Blob Storage. https://azure.microsoft.com/en-us/products/storage/blobs.

[7] Azure Shared Access Signatures. https://learn.microsoft.com/en-us/azure/storage/common/storage-sas-overview.

[8] Azure Data Share. https://azure.microsoft.com/en-us/products/data-share.

[9] BigQuery Analytics Hub. https://cloud.google.com/analytics-hub/?hl=en.

[10] R. Chandra, H. Chen, R. Matharu, S. Cai, J. Chen, P. Dutta, B. Ghita, T. Greenstein, G. Holla, P. Huang, Y. Huo, A. Ionescu, A. Ispas, T. Januschowski, V. Karajgaonkar, S. Leone, D. Lewis, A. Li, N. Li, C. Lian, S. Link, Q. Lu, Y. Ma, C. Pettit, V. Prabhakaran, B. Raducanu, K. Rong, P. Roome, S. Shetty, S. Smith, X. Sun, Y. Tang, W. Wen, L. Xia, J. Zeng, B. Zhang, R. Xin, and M. Zaharia. Unity catalog: Open and universal governance for the lakehouse and beyond. In *Proceedings of the 2025 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. ACM, 2025.

[11] Databricks Clean Rooms. https://www.databricks.com/product/clean-room.

[12] current_recipient function. https://docs.databricks.com/en/sql/language-manual/functions/current_recipient.html.

[13] Announcing Public Preview of Delta Sharing with Cloudflare R2 Integration. https://www.databricks.com/blog/announcing-public-preview-delta-sharing-cloudflare-r2-integration.

[14] Delta Kernel. https://delta.io/blog/delta-kernel/.

[15] Delta Lake. https://delta.io/.

[16] Delta Sharing Protocol. https://github.com/delta-io/delta-sharing/blob/main/PROTOCOL.md.

[17] Delta Sharing. https://github.com/delta-io/delta-sharing.

[18] Delta Sharing Server. https://github.com/delta-io/delta-sharing/tree/main/server.

[19] Top Three Data Sharing Use Cases With Delta Sharing. https://www.databricks.com/blog/2022/01/14/top-three-data-sharing-use-cases-with-delta-sharing.html?t.

[20] GCP Signed URLs. https://cloud.google.com/storage/docs/access-control/signed-urls.

[21] Google Cloud Storage. https://cloud.google.com/storage.

[22] Apache Hudi. https://hudi.apache.org/.

[23] Apache Iceberg. https://iceberg.apache.org/.

[24] How Delta Sharing Enables Secure End-to-End Collaboration. https://www.databricks.com/blog/how-delta-sharing-enables-secure-end-end-collaboration?t.

[25] ActionIQ Adds Databricks Delta Sharing: Flexibility and Control in Data Integration. https://www.actioniq.com/blog/databricks-delta-sharing-integration/?t.

[26] Exploring Premium Data Sharing Solutions: Databricks Delta Sharing vs. Snowflake Data Sharing. https://www.linkedin.com/pulse/exploring-premium-data-sharing-solutions-databricks-delta-ben-poole/.

[27] Streamline data collaboration with Databricks Delta Sharing and Microsoft Power BI. https://techcommunity.microsoft.com/blog/analyticsonazure/streamline-data-collaboration-with-databricks-delta-sharing-and-microsoft-power-/3707837?t.

[28] Databricks Marketplace. https://www.databricks.com/product/marketplace.

[29] Unlimited data-driven collaboration with Data Sharing of Oracle Autonomous Database. https://blogs.oracle.com/datawarehousing/post/share-data-with-oracle-autonomous-database-data-sharing.

[30] Apache Parquet. https://parquet.apache.org/.

[31] Pre-Signed URLs. https://docs.aws.amazon.com/AmazonS3/latest/userguide/ShareObjectPreSignedURL.html.

[32] Amazon Redshift Data Sharing. https://aws.amazon.com/redshift/features/data-sharing/.

[33] SAP and Databricks Open a Bold New Era of Data and AI. https://news.sap.com/2025/02/sap-databricks-open-bold-new-era-data-ai/.

[34] Snowflake for Collaboration. https://www.snowflake.com/en/data-cloud/workloads/collaboration/.

[35] Data Sources. https://spark.apache.org/docs/latest/sql-data-sources.html.

[36] Monitor account activity with system tables. https://docs.databricks.com/en/admin/system-tables/index.html.

[37] Universal Format (UniForm). https://docs.delta.io/latest/delta-uniform.html/.