RESEARCH-ARTICLE

Unified Lineage System: Tracking Data Provenance at Scale

GABRIELA JACQUES-SILVA, Meta, Menlo Park, CA, United States

EVANGELIA KALYVIANAKI, University of Cambridge, Cambridge, Cambridgeshire, U.K.

KATRIEL COHN-GORDON, Meta, Menlo Park, CA, United States

ADHAM MEGUID, Meta, Menlo Park, CA, United States

HUY NGUYEN, Meta, Menlo Park, CA, United States

DANNY BEN-DAVID, Meta, Menlo Park, CA, United States

View all

Open Access Support provided by:

Meta

University of Cambridge

Published: 22 June 2025

SIGMOD/PODS '25: International Conference on Management of Data
June 22 - 27, 2025
Berlin, Germany

Conference Sponsors:
SIGMOD

https://doi.org/10.1145/3722212.3724458
ISBN: 9798400715648

Unified Lineage System: Tracking Data Provenance at Scale

Gabriela Jacques-Silva
gabijs@meta.com
Meta Inc.
Cambridge, MA, USA

Evangelia Kalyvianaki*
ek264@cam.ac.uk
University of Cambridge
Cambridge, UK

Katriel Cohn-Gordon
katriel@meta.com
Meta Inc.
London, UK

Adham Meguid
ameguid@meta.com
Meta Inc.
Bellevue, WA, USA

Huy Nguyen
nguyenngochuy91@meta.com
Meta Inc.
Bellevue, WA, USA

Danny Ben-David
dannybd@meta.com
Meta Inc.
Cambridge, MA, USA

Carl Nayak
carlnayak@meta.com
Meta Inc.
Cambridge, MA, USA

Varun Saravagi
saravagi@meta.com
Meta Inc.
Cambridge, MA, USA

George Stasa
gstasa@meta.com
Meta Inc.
Cambridge, MA, USA

Ioannis Papagiannis
yiannis@meta.com
Meta Inc.
London, UK

David Taieb
dtaieb@meta.com
Meta Inc.
Cambridge, MA, USA

Kalkidan Tamirat
kalkidant@meta.com
Meta Inc.
Bellevue, WA, USA

Haiyang Wu
haiyangwu@meta.com
Meta Inc.
Cambridge, MA, USA

Bo Xi
xibo@meta.com
Meta Inc.
Cambridge, MA, USA

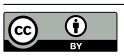
Taining Zhang
tainingzhang@meta.com
Meta Inc.
Cambridge, MA, USA

Qi Zhou
zhouqi@meta.com
Meta Inc.
Bellevue, WA, USA

Abstract

As businesses increasingly rely on large-scale data analysis to build products and features, understanding how data flows through heterogeneous storage assets has become a critical challenge. To address these challenges, we present Unified Lineage System (ULS), an end-to-end lineage aggregator designed to track data flows at scale. Our system features a general data model representing data flows between different asset types, a lineage data model that enables navigation across granularities, and techniques to stitch incomplete lineage traces. We also describe methods for processing the lineage graph to customize precision and recall. We implemented ULS at Meta, where its lineage graph contains billions of nodes and edges and supports over one hundred use cases, including routine operations, capacity management and privacy.

*Work done while author was on sabbatical at Meta Inc.



SIGMOD-Companion '25, Berlin, Germany

ACM ISBN 979-8-4007-1564-8/2025/06
https://doi.org/10.1145/3722212.3724458

CCS Concepts

• **Information systems** → **Data provenance**.

Keywords

Lineage, provenance, data discovery, data governance

ACM Reference Format:
Gabriela Jacques-Silva, Evangelia Kalyvianaki, Katriel Cohn-Gordon, Adham Meguid, Huy Nguyen, Danny Ben-David, Carl Nayak, Varun Saravagi, George Stasa, Ioannis Papagiannis, David Taieb, Kalkidan Tamirat, Haiyang Wu, Bo Xi, Taining Zhang, and Qi Zhou. 2025. Unified Lineage System: Tracking Data Provenance at Scale. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25), June 22–27, 2025, Berlin, Germany.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3722212.3724458

1 Introduction

Many businesses rely on scalable analysis of large datasets in order to build products and features for users [51]. A standard architecture for such businesses separates two main classes of database: a "production database" [9] which supports user-facing products, and a "data warehouse" used for analytics. Large data warehouses can contain millions of tables [50] and exabytes [35] of data.

Maintaining such large-scale systems requires support from *data understanding* tools. In particular, many use cases require developers to understand how data flows through heterogeneous storage assets. For example, a developer debugging corrupted data in a specific storage table may wish to understand which other tables received that data and require correction. A capacity engineer wishing to free up unused storage may need to understand whether a specific table is being used by important pipelines. And a privacy engineer wishing to understand where certain data are stored may need to track their provenance as they flow through pipelines.

Data moves between *production databases* and a *data warehouse* (and vice versa) in several ways. Data can originate in the mobile or web stack, be processed by frontend servers, and then be logged into a warehouse via a distributed queuing system [14]. These distributed queues can be processed by stream processing systems or bulk importers. Within the warehouse, the data can be transformed and aggregated before flowing into dashboards or back into production databases to improve user-facing experiences. Data flows thus form a directed graph with data storage assets as nodes and flows between them as edges. Modeling this graph requires novel techniques: lineage systems must be able to track data ingestion points, how data is read and processed, and where it is written to.

Building such a lineage graph is not as simple as it may first seem. In this paper, we discuss a number of challenges that lineage systems must overcome to support production use cases. They must integrate with a wide range of data providers, from high-volume ETL processors through to manual queries or opaque blocks of specialized code. They must support querying different views (or *granularities*) of the same data flow, from individual columns to an overall table. They must be able to infer accurate information about flows given only partial system traces. They must keep false positive or negative flows to an acceptable level, despite many potential sources of incorrect and incomplete data. And they must expose lineage information in a way that is useful to developers while still supporting bulk automated queries.

To that end, we have developed ULS, an end-to-end lineage aggregator. Our contributions are as follows. First, a system to track data flows at scale. ULS has a general data model, enabling the representation of data flows between different asset types. Flows can be represented at different *granularities*, allowing the representation of "real world" flows, in which not all metadata collection has the same accuracy (Sections 2 and 4). Second, a flexible and rich lineage data model to encapsulate the requirements to represent evidence-based lineage from diverse sources (Section 5). Our model introduces containment edges, which establish parent-child relationship for a pair of nodes to allow users to navigate across granularities and to view flows at different levels of detail (e.g., column and table). Third, different techniques to connect lineage data with flows based on common evidence metadata when information is known to be incomplete, enabling increased recall for lineage queries (Section 7). Fourth, tools and methods for processing the lineage graph that help balance false positives and negatives, allowing consumers to customize their precision and recall depending on their use case (Section 7). In Section 3 we provide a description of challenges. Finally, in Section 9 we present a quantitative view of the ULS graph, an impact analysis of lineage consumption techniques, and lessons learned from real production data (Section 9.4).

2 Unified Lineage System

Figure 1 provides a high-level overview of a standard architecture for a complex web service, and the interconnections between services as data moves through them. When a request hits a web server, it may result in reads and writes to production databases, backend services, and event logs. These logs can be processed by streaming systems and/or be ingested to a data warehouse. Warehouse data can be processed for analytics and ML training, and its results may return to production services to feed different user experiences. During these operations, data moves through different *data assets* (e.g., Hive [50] tables), being transformed and/or merged with other data (e.g., production databases or data logs). A *data flow* loosely refers to the movement of data from one asset to another. Data flows occur via automated processes such as SQL queries, other transformation primitives (e.g., Python via data frames), invocation of different services, and more.

As shown in Figure 1, ULS operates alongside these everyday operations. While many systems operate to provide day-to-day services to users, a selection of these, referred to as *providers*, also produce lineage metadata from their own operational logs. ULS takes input from all its providers, synthesizes and aggregates their lineage, and produces a new daily *data lineage graph*. The graph is used directly by developers for ad-hoc exploration, as well as programmatically to power several business use cases.

We remark that, by design, ULS allows for both false positive and false negative lineage: that is, it is possible for it to report data flow between two assets where no true flow is present, or not to report the lack of a flow despite its existence. Overall, we make no formal guarantees about the correctness of ULS results.

2.1 What is lineage used for?

Multiple use cases rely on high quality lineage. We call out three main categories:

(1) routine operations such as data pipeline debugging and data corruption recovery,
(2) efficiency and capacity management such as data removal and data collocation,
(3) privacy and data use, such as data discovery.

Data pipeline debugging: Developers routinely require lineage information during software development. For example, consider a daily batch job that reads certain datasets to create a processed output. This job is expected to complete overnight, but one day fails to finish on time. Daily jobs which depend on that output will become "stuck" waiting for their inputs, and thus the delay propagates to downstream jobs. In such a case, developers need to identify the furthest upstream piece of processing which has not yet completed and remediate any issues. In other words, given a table, they must find its transitive set of upstream tables.

Data corruption recovery: Another form of debugging is to identify and remediate incorrect or corrupt data. For instance, a logging bug might lead to incorrect data in an event logging table. Once such bugs are fixed, engineers may wish to recompute the downstream consumers' affected partitions to remove the incorrect data. To create a data recovery plan, the information on the lineage graph is leveraged to track down the set of table partitions and data pipeline tasks that need to be re-executed to remove the incorrect data.

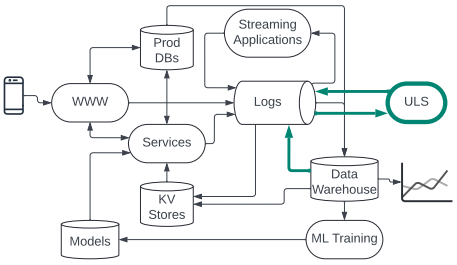


Figure 1: Unified Lineage System

Data removal: Tools such as Systematic Code and Asset Removal Framework (SCARF) [48] identify and remove unused data assets. SCARF needs lineage information to determine whether an asset is unused: if an asset is consumed by an active downstream asset, then it too must be active. ULS fulfills these lineage queries in order to enable SCARF to remove such assets.

Data collocation: Data warehouses are often sharded into namespaces, which are themselves allocated to regions and individual data centers. When joining two tables together, it is more efficient to collocate those tables to reduce cross-region traffic. To perform this optimization, lineage is needed to determine which tables are downstream dependencies of each other or joined within a query.

Data understanding: Various privacy implementation strategies rely on the ability to understand the data stored in a data asset. This may include relatively straightforward definitions such as "*this is data about users*", or metadata about complex subtyping such as "*this string-type data contains email addresses*". This kind of label can be used for determining applicability of privacy requirements such as data deletion [17], or for developer improvements such as enhanced type checking in SQL queries [37]. One technique to identify and place such data-understanding annotations on assets is via automated scanning tools as in Data Loss Prevention (DLP) systems [2, 25]. However, such tools often operate on assets independently, without knowing their relationships. In contrast, by placing semantic annotations on assets that are at the root of data pipelines or processing flows, we can use lineage information to automatically detect downstream flows of that data and propagate the labels to those downstream tables as well. For example, a lineage system can identify that a column y is downstream of a column x which is known to contain email addresses, and thus automatically propagate a semantic annotation from x to y.

Data discovery: Privacy engineers often need to identify all assets to which a given policy applies. In [30], the authors describe enforcing a commitment that a given data point is not used for a given purpose. To do so, developers must identify where this specific data is stored, in a process called asset discovery [33]. This discovery can be done via upstream or downstream asset traversals on the lineage graph. A developer can identify *root assets* containing the data in scope, and then query the lineage graph to identify other candidate assets that may contain this data and its derivations.

3 Lineage Challenges

This section describes the main practical challenges faced towards a high-quality lineage system while operating our large-scale and diverse system as shown in Figure 1. Throughout this section we use a simple SQL-based lineage example, an unstructured data flow and data movements and transformations through table manipulation that constitute common patterns of our operations. We clarify that the goal of our lineage system is to devise a lineage graph of data flows. We build a new graph every day from data coming from lineage sources referred to as providers. Such a graph can be queried by lineage-related queries. While using different examples, we discuss the practical challenges to build such a graph to capture intended data flows. We later formalize our data model and lineage models in Sections 5, 6 and 7.

```
INSERT INTO bar
     (c, p)
SELECT
     a + b as c,
     p
FROM foo
WHERE
     p = '2024-01-01'
```

Figure 2: Example SQL query.

One of the most common examples is recording lineage metadata about data flowing between relational tables, such as Hive. Figure 2 shows a simple SQL query q, where data moves between two date-partitioned Hive tables `foo` and `bar`. Recall that Hive tables are divided into partitions, according to unique values of a column [50]. The simplest flow f in q is data moving from `foo` to `bar` and can be represented as the table graph in Figure 3. In this graph the directed edge indicates the data moved from node `foo` to `bar`. q is attached to the edge to provide evidence of the flow, i.e., *data flow f is generated by the execution of query q*. Such *table-to-table* representation is straightforward and satisfies many use cases.

3.1 Data flow granularity

While straightforward, table-to-table lineage does not cover all use cases. If an engineer is trying to understand why a table has not yet received a new partition today, it is not sufficient to know that its upstream tables still exist. They must be able to identify which upstream *partitions* are delayed. If an engineer is trying to identify whether a column contains email addresses, knowing that some upstream table contains email addresses is not sufficient, as the upstream *column* containing email addresses may never be involved in a flow. Thus, the table-to-table graph in Figure 3 does not capture many use cases' needs. Figure 3 shows additional views of flow f from query q, at *table*, *partition*, and *column* granularities.

One way to support multiple granularities is to store all flows at the finest granularity, and use this data to synthesize abstractions of the lineage graph at all coarser levels. For example, in query q above, this would store a flow edge for every (column, partition) pair. There are two reasons why this approach is not straightforward. First, even a single SQL query on a large table (say with 10 years of date partitions and 300 columns) will generate over a million flow edges in the resulting graph, quickly leading to a graph size explosion. Second, not all lineage providers are able to report data at the finest granularity; we discuss this further in Section 3.3.

Challenge 1. *It is difficult to track lineage across flows of different table, partition, or column granularities.*

3.2 Data flow evidence

A lineage system which simply reports "table `foo` flows into `bar`" is not particularly useful for use cases that need to understand *why* the system believes such a flow exists. For example, an engineer

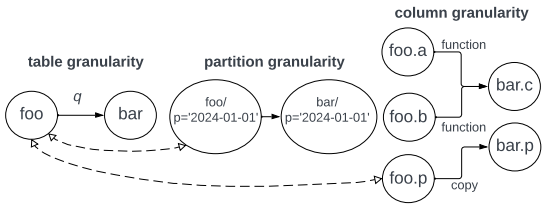


Figure 3: Lineage flows at different granularities.

debugging missing data may discover that an unrelated upstream table is blocking their pipeline, and wish to understand why there is a dependency on that upstream.

In some cases, basic evidence of flow is sufficient. For example, in Figure 3 we can store the query q identifier in the lineage graph, and report q as evidence of the flow from `foo` to `bar`.

However, simply reporting a query identifier as an evidence metadata is often insufficient. An engineer investigating q may wish to understand who ran the query, where it is stored in source control, or whether there were recent modifications to the query that could have introduced issues. Similarly, a data deprecation system [48] attempting to deprecate `foo` may wish to modify the source code of q to remove the dependency, and thus need a code pointer to the underlying repository.

Challenge 2. *It is difficult to identify and record enough evidence metadata for consumers to investigate and understand reported flows.*

3.3 Data flow stitching

Lineage providers do not always provide *flows*: sometimes they can only report reads or writes. For example, consider the Python code `script.py` in Figure 4. The code defines two strings containing SQL queries, executes them via an example Hive library binding, and writes the results to a new Hive table. Reading this code, we can see that there are only two true data flows, shown in blue.

Instrumenting the Hive library itself is a straightforward way to report lineage for such queries. For example, we can update `hive_lib.query_presto` (resp. `hive_lib.write_to_hive`) to report a "read" (resp. "write") event to ULS. For the code above, this would give us four independent events: reads via `q1` and `q2`, and writes to `bar1` and `bar2`. Since lineage users want to consume flows instead of events, ULS must therefore construct flows out of these events. One approach is to attach a common identifier to related events at execution time to capture their connections [7], and then *stitch* all reads and writes from the same identifier into flows. In the example above, if the same *id* is shared across reads and writes, this results in both the correct (blue) and incorrect (red) flows shown.

In our experience, stitching of indirectly connected assets is one of the most challenging tradeoffs when developing a lineage system. Relying too heavily on low level stitching comes with its own pitfalls. For example, an engine like Presto may read and write statistics about the query execution (e.g., processing speed) to its own internal datasets, or may do authentication and authorization operations with backend services. In this case, it is hard to differentiate automatically which assets are being read because of a user-level operation, and which are due to internal IO issued by the engine. Always stitching all assets read or written by a given processor

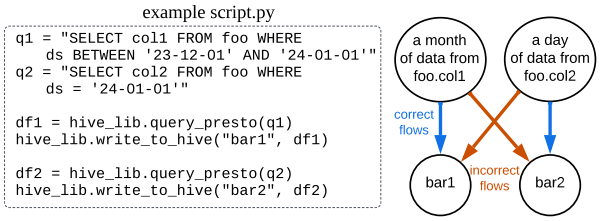


Figure 4: Lineage metadata stitching.

leads to *superconnectors*, resulting in a very noisy lineage graph with overall degraded quality of information.

Challenge 3. *It is difficult to perform meaningful data flow stitching for systems that do not report complete and accurate lineage.*

Known unknowns. While stitching is necessary to maintain end-to-end flows in the lineage graph with high coverage, it results in flows with lowered confidence as in Figure 4. In our observation, lineage users expect to have means to detect that a given flow edge may have imprecision (e.g., constructed by stitching instead of directly reported). This is so that users can distinguish which data flows are more likely to be true (i.e., higher confidence) from flows that may require further manual inspection (i.e., lower confidence).

Challenge 4. *It is difficult to establish and represent known unknowns in the graph to facilitate lineage data understanding.*

3.4 Graph traversal

Many applications require traversal of the lineage graph. While some flows are accurate (e.g., Figure 3), others are overapproximations (e.g., Figure 4). Traversing such a graph to output flows of high quality is challenging. Further, recursively traversing incorrect edges on the graph means that we are amplifying the error when a path through an incorrect edge is traversed.

Challenge 5. *It is difficult to effectively navigate the lineage graph with edges of varying confidence to produce high recall and high precision lineage traversal query results.*

3.5 Tackling These Challenges

To address Challenge I, ULS's data model allows edges of the lineage graph to be defined at different granularities which lineage providers leverage to ingest data at the appropriate column, table, or partition granularity (Sections 5 and 6). To tackle Challenge II on flow evidence, ULS's data model introduces the *processor node* for providers to declare the evidence of edges in the graph (Section 5). To address Challenge III on metadata and events stitching, ULS provides comprehensive lineage ingestion and consumption through a series of mechanisms for flow traversal of the global lineage graph including the *ULS Tailer* component at the Lineage Enrichment stage (Section 6) and the proactive stitching of *half edges* at the Lineage Consumption stage (Section 7). Finally, to tackle Challenges IV and V ULS introduces two novel techniques – *granularity sweeping* and *column-partition granularity* – that allow graph traversal even when edges are of low quality and by using available lineage metadata to reduce false flows (Section 7).

4 ULS architecture

ULS follows a streaming-based architecture as shown in Figure 5. Lineage providers (①, Section 6.1) generate daily reports of lineage data, which is combined by a lineage ingestion system (②, Section 6) to build a global lineage dataset powering global graph traversal APIs (③, Section 6.2). Users can then query these APIs in various ways (⑤, Section 7.1).

At its core, ULS uses the global, aggregated lineage dataset to dynamically build on-demand, in-memory representations of the lineage subgraphs requested by user queries. ULS considers the global dataset and the global graph, or simply the lineage dataset and graph, as equivalent representations of the same lineage and so these terms are used interchangeably hereafter. Briefly, the lineage dataset contains entries of lineage data which can be stitched together to build the user-requested subgraphs. Both the input lineage data and the derived dataset conform to the ULS data model (Section 5) designed to capture our diverse types of input lineage data, use-case requirements, and to provide the fundamental elements to address the challenges described in Section 3. Finally, ULS uses measurement-based methods in ④ to ensure the lineage quality in the lineage dataset (Section 8). This process provides quality checks, such as the measurement of coverage of specific data flows.

ULS APIs enable processing of the graph in batch fashion or via querying a service call. While users can use the lineage dataset via SQL queries, we also provide a traversal API to navigate the lineage graph accounting for the varying quality of data flows.

Most use cases operate on recent lineage data, covering perhaps a few days. To this end, we deliver a new lineage aggregated graph every day that captures the flows of the previous day. However, there are also cases, such as in interactive data pipeline debugging, where lineage data on fresh flows (e.g., past hour) are needed. To achieve that, ULS provides "real-time" lineage data on flows soon after these have been processed, generally within a few minutes.

5 Data Model

The fundamental elements of our model are *data*, *assets*, and *flows*. Each asset in our data model represents a data identifier to a physical or a logical entity in which *data moves from and/or to* (i.e., a flow). In this way, our model supports a diverse set of different asset types. For example, an asset can represent a simple storage element such as a table or a file. An asset can also represent logical entities that are associated with data movement and have a meaningful representation to a developer or represent a higher-level storage abstraction. For example, an asset can be a schema definition used for logging or a node in the Ent framework [34] that is stored in MySQL. As our model strictly relies on asset identifiers, it does not make any assumptions about the data type and format. For example, a Hive table has partitions, columns, and sub-columns (e.g., structs, maps). Its storage format does not play a role in this representation.

5.1 Lineage Graph

The lineage graph is a directed graph defined as a set $L = (V, T)$ where V is the set of vertices or nodes and T is the set of edges. A node $v \in V$ is defined as $v = (c_v, T_v, id_v, A_v)$. A category set $c_v = \{$"asset", "processor"$\}$ denotes whether node v is an asset or a *processor*. A processor provides supporting evidence of data movement from one asset to another. A node's type $T_v = \{T_v^{\text{asset}}, T_v^{\text{processor}}\}$ is defined as one element from a set of pre-defined attributes of the different types of assets T_v^{asset} or processors $T_v^{\text{processor}}$ that the lineage graph supports. An example of T_v^{asset} is a Hive table. An example of $T_v^{\text{processor}}$ is a Presto query. Finally, A_v represents an optional free-form map of attributes containing node metadata copied from a source of truth.

A directed edge $e \in E$ is defined as $e = (t_e, src_e, dst_e, proc_e, G_e, A_e)$. $t_e = \{$"$flow$", "$containment$"$\}$ denotes the edge type and can be either a *flow* or a *containment*, explained further below. src_e is the source asset node and dst_e is the target node of edge e. $proc_e$ is the processor node for edge e providing evidence of the connection from src_e to dst_e. Finally, G_e is the graph granularity and is defined as $G_e = \{$"$table$", "$partition$", "$column$"$\}$ and explained in Section 5.2. A_e is a map of edge annotations such as the lineage provider and the quality indicator (Section 5.3). While containment edges expect both src_e and dst_e to be specified, flow edges are allowed to have only one of them specified.

A *flow edge* or simply *flow* indicates a data flow between two assets. Flows must have a processor node $proc_e$ providing evidence of data flowing from src_e to dst_e. A flow edge must always have a $proc_e$ and at least a source or a target node. A *complete edge* has both source and target nodes, while a *half edge* has only a source or target node. Half edges occur when our data lineage instrumentation cannot establish the direct flow relationship between two assets, but it can establish that there was a read from src_e or a write to dst_e by the given $proc_e$. It also represents situations in which the source or destination nodes are not modeled in the lineage graph (e.g., Web UI). The same evidence node can be in multiple complete and/or half edges, such as the half edges that are later stitched in the example in Figure 4. A flow edges can also represent a cycle in the graph, where src_e and dst_e are equal. One such example where this can happen is when a query creates a new partition of a table based on the past several partitions of the same table.

A *containment edge* establishes a parent to child relationship between two nodes, be it assets or processors. Examples include a table contains columns and partitions (dashed arrows in Figure 3), or a pipeline task instance contains a query. Evidence nodes are not required for containment edges, as these often come directly from metadata systems or specific platforms (e.g., pipeline framework). Containment edges enable graph summarization and navigation across granularities. Summarization is achieved by post-processing the graph to transform nodes into their parent nodes. For example, a subgraph with columns connected by queries can be presented as a subgraph of the corresponding tables connected by the data pipeline tasks that executed the underlying queries.

5.2 Graph Granularity

The lineage graph contains information about tables, partitions, and columns; we refer to these as different *granularities*. Granularities in the ULS data model are effectively different views of the flow graph, where the same asset can be represented with different levels of precision regarding the same data flow as shown in Figure 3. We can traverse the lineage graph at any granularity, "zooming" in to capture fine details such as column-to-column flows, or out to see

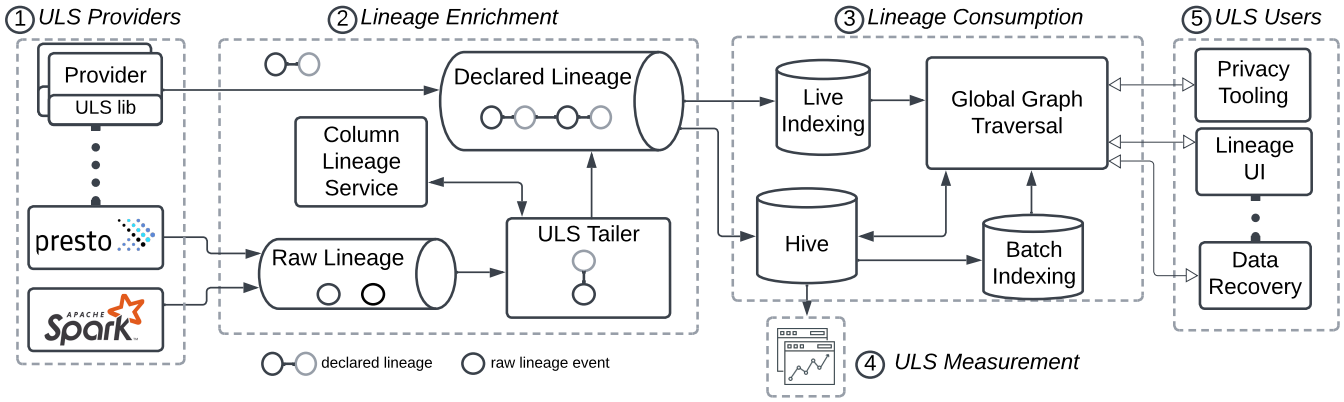


Figure 5: ULS architecture.

high-level flows between tables. Different granularities may suit better different use cases of the lineage graph (see Section 2.1).

Note that not every asset type has a meaningful representation at every granularity. In particular, this means that finer granularities (e.g., partition, column) do not necessarily have all the data from coarser granularities (e.g., table). For example, the Scribe [29] distributed queuing system, allows developers to stream data into queues that eventually flow into tables in the data warehouse. A Scribe category (i.e., a queue) is broken down into columns but not partitions, as it is a continuous queuing service. Thus, a flow from a Scribe category into a Hive table can be represented at the table and column granularities. Although one could represent flows from a Scribe category to all partitions of a Hive table, there is no gain of information when compared to the table level flow. The representation of asset types in granularities is a design decision made when ingesting new flows into ULS. Section 7 describes how ULS handles cross-granularity traversals .

5.3 Edge Annotations

Edges are associated with free-form annotations A_e described below. These annotations include the lineage provider it came from as well as the relevant responsible engineering team, to enable debugging in case lineage users report problems with specific edge information.

Data flow characterization. In addition to stating that there is a data flow between assets, edges can be annotated with information on the characteristics of the data transformation, further specifying how the input has influenced the output for a specific flow. For example, the column granularity provides the annotations *clause* and *category*, modeled after SQL transformations. In particular, the clause annotation tells us how information was used (in a select, filter, or group by) and can be broken down into categories to characterize how the information from source to destination happened. For example, when going to a destination column, a source column can be copied, aggregated, pass through a function, and others.

Contextual information. Edges can be annotated with other information available about the flow, depending on the provider. Examples include the name of the operator in the data pipeline used to construct the flow, or the information about which security credential was used to authorise the query.

Quality Indicator (QI). Finally, we introduce the quality indicator to capture our confidence on the flows described by edges. It is a simplified assessment of ULS's confidence that a flow edge describes a real data flow between source and destination, ranging from 1 (*high confidence*) to 4 (*low confidence*). In ULS, the indicator is generally tied to the accuracy of the flow information acquisition process. Below, we describe some guidelines to assign the indicator values. A flow indicator has value 1 when considering flows at the column granularity, and our flow collection information method is able to pinpoint with high accuracy both the source and destination columns of the flow (e.g., via SQL analysis). A value of 2 is given when we are able to point out the source or destination columns of the flow, but not both. 3 is given when we can pinpoint the source and destination tables of a flow, but not their column relationships. 4 is given when we are able to make asset connections via a processor but we do not know how they relate to each other (e.g., processor read from 5 assets and wrote to 2 assets). Quality indicators help ULS users sift through reported flows, as ULS can flag where a flow may need additional review. ULS relies on lineage providers to assign the edge quality indicators, as they are in the best position to assess the confidence of their flow data acquisition process.

6 Lineage ingestion

This section describes the overall ULS ingestion process, which starts with *lineage providers* sending data that is then validated or *enriched* with additional flow information (① and ② in Figure 5).

6.1 Lineage providers

ULS supports approximately 50 different lineage providers which can submit input lineage data to our system using ingestion APIs. These include Data Warehouse query engines, such as Presto and Spark, ETL systems, stream processing systems, and more. Data generated by providers can be either *declared* or *raw* lineage events or both. By *declared* lineage we mean metadata that follows the ULS data model i.e., full or half-edges. In contrast, we refer to *raw* lineage as metadata that must further processed to generate nodes and edges, or to enrich edge information.

Providers, such as WWW code lineage, create declared lineage data through an offline process (e.g., code static analysis, using

operational datasets) that can execute multiple times a day. At its final stage, this offline process invokes a ULS library to generate subgraphs made of nodes, flow, and containment edges. The library is responsible for data validations (e.g., validate identifiers) and publishing the subgraphs to a *declared lineage* Scribe category in ②. Other providers integrate to ULS by adding lineage instrumentation to their own code and then submitting *lineage events*, either raw or declared, in a streaming fashion as soon as the flows happen.

ULS requires lineage providers to be registered before they can start submitting lineage data. While ULS is extensible and geared towards easy on-boarding of new providers, this is still a *governed process*. This is to ensure that providers are using existing asset types, and follow the granularity and quality indicator guidance. The registration process includes the description of what asset types a given provider will submit and the responsible engineering team.

6.2 Lineage enrichment

While declared lineage from providers is used without any further processing, raw lineage events can be further enriched with information about the environment they were executed on and the transformation executed on the data.

As shown in Figure 5, enriched events go through an additional transformation step performed by the *ULS Tailer* component in ②. This component is responsible for (i) converting an event into a ULS conforming sub-graph; (ii) encoding specific attributes as part of edge metadata; (iii) creating containment edges between processors based on environment information (e.g., pipeline task contains pipeline task instance that contains a query); and (iv) creating column and subcolumn (e.g., structs, maps) level lineage for SQL queries coming from our Data Warehouse compute engines. The ULS Tailer then submits the enriched events to the *Declared Lineage* events category, which is used to construct our final results.

While compute engines could generate column lineage directly during SQL query execution, we delegate this process to a dedicated *Column Lineage Service* component that is invoked outside of the query execution path. This service receives a SQL query as input, and returns a column lineage graph describing the lineage relationships between input and output columns and subcolumns. This service also labels the edges according to the transformation made on the query, such as the *clause* and *category* described in Section 5.3. These are directly relied on by lineage users. Delegating column lineage to a service ensures that: (i) different engines use the same column lineage specification; (ii) compute engines have its lineage data enriched based on environmental information in the same way; and (iii) that the column lineage process can evolve independently of the release cycles of compute engines. This means that both bug fixes and lineage refinements can be released independently from engines. Both the ULS Tailer and the Column Lineage Service have many replicas to properly serve the required input load. At the end of the Lineage Enrichment process all declared lineage generated is given to Lineage Consumption for making the global data graph.

7 Lineage consumption

The Lineage Consumption stage is the final step in the ULS data pipeline and generates the global lineage graph for consumption by ULS users for different purposes as shown in Figure 5. Lineage Consumption can be done either on *historical* daily data, or on *fresh* data generated on the same day with minutes of latency, or in a combination of the two.

To achieve this, declared lineage data from ② is copied into two different locations. First, it enters the *Live Indexing* key-value store, which is continuously indexed with fresh declared lineage data and can be used by the *Traversal API* for answering queries that require fresh lineage data. Second, it is copied into Hive tables that are further processed, both to deduplicate the declared lineage and to *stitch* any *half edges* with input and output nodes that have common evidence nodes. Any half edges that have no matching evidence are kept as is. The result of these transformations are stored in new Hive tables and constitute the Hive interface to the global lineage graph. The Hive tables go through the *Batch Indexing* process, so that historical data is visible for the Traversal API.

Thus, ULS exposes two APIs for consumers. One is a Hive dataset, that can be used directly with SQL query engines. The other one is an Interactive Traversal API, which generates lineage subgraphs based on specified *root nodes*. One implementation of this API is pipeline-based, using the graph data processing tool from [16] to traverse the ULS Hive dataset and generate subgraphs which can be millions of nodes. The other implementation is a Thrift-based API that provides both synchronous and asynchronous calls to generate small to medium subgraphs (e.g., low hundred thousands of edges). The Thrift API builds the subgraphs using the live and batch indexes.

7.1 Granularity Sweeping Graph Traversal

Because ULS contains lineage at multiple granularities, it can combine information across granularities to fill in missing data. This relies on the existence of containment edges, which connect matching nodes in the different granularities. For example, consider the case where a provider returns only table-level lineage, instead of column-level. Querying only the column-level graph would omit any data from this provider, since its data is only present in the table graph. However, ULS can traverse from a column node to its corresponding table node, and then to any table-level downstreams. We call this operation *granularity sweeping*, since it performs a sweep through different lineage granularities when queried. Additional results from granularity sweeping are returned separately from ULS APIs, and identified as such.

The granularity sweeping graph traversal follows the principle "traverse datasets confidently where possible, and with caution otherwise". To achieve this balance ULS leverages the different flow views provided by granularities to navigate lineage in case of both *low confidence* or *missing* edges in the finer granularities. To begin, users provide a set of root assets and filtering criteria, such as graph depth. Starting from the root assets, ULS recursively traverses the high-confidence edges at the finest granularity until the user-defined stopping condition is reached (e.g., number of hops or filtering criteria); this is to *traverse datasets confidently where possible*. When a lower confidence edge (i.e., indicator value higher than the one given by the running lineage query) or no outgoing edge is found, granularity sweeping proceeds to find nodes at coarser granularities that are 1-hop away from all currently

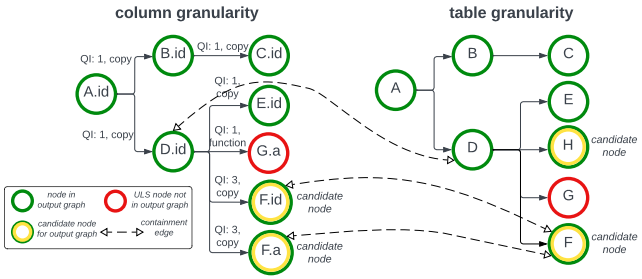


Figure 6: Granularity sweeping example traversal.

included nodes in the subgraph. In some cases, these 1-hop nodes can be found using lineage datasets that are not yet onboarded to ULS. The graph traversal operation terminates when no more edges are left to follow or traversal has reached n hops.

Nodes identified via low confidence edges or coarser granularities are curated by either manual inspection by experts such as data flow owners, or through automatic processes leveraging additional metadata not available to ULS. Curation forces users to make informed decisions about which edges with known lower precision to include in their lineage subgraph. Over the years, we have observed that allowing users to traverse all lower quality edges available recursively, produces very large graphs with very small or no added value to the output lineage graph. In fact, users would get lost when navigating lineage in such graphs as the probability of a true data flow path reduces at each hop following an edge of low confidence. The combination of granularity sweeping and curation supports our key requirement to *traverse with caution in case of low confidence* and it provides users with a powerful tool to achieve high precision and recall lineage queries.

7.1.1 Example interactive traversal. To illustrate how interactive traversal works we use Figure 6 and the following lineage query: *for table A with column id, find all downstream assets that are a copy of A.id and only consider edges with a quality indicator value of 1 (i.e., highly probable).* To answer this query, traversal uses both the column and table granularity lineage subgraphs (Figure 6).

Traversal begins from node `A.id` in the column lineage graph and continues to recursively traverse the graph edges matching the specified quality indicator (QI) of 1 and the copy filtering constraint. As such, nodes `B.id`, `C.id`, `D.id`, and `E.id` are found and included in the output lineage subgraph (shown in green). Although there are further edges from `D.id` to each `F.id`, `F.a`, and `G.a`, these nodes are not yet included in the output graph as their corresponding edges do not satisfy the query requirements; either their quality indicator is higher than 1 or their label is not a "copy". Given `G.a` does not meet the "copy" criteria based on high confidence information, ULS excludes the node from the resulting subgraph.

For edges with low-confidence nodes, ULS pauses traversal to surface new candidate nodes to the consumer. First, since `D.id` is connected to `F.id` and `F.a` via low-confidence edges, we select both nodes as candidates (yellow). Second, since `D.id` is a child of `D` in the table-level graph (via a containment edge), and `D` is adjacent to `H`, we perform granularity sweeping and select node `H` as a candidate. It is relatively common to find flows such as `D`→`H` available only in one granularity. This may be due to a lineage provider which does

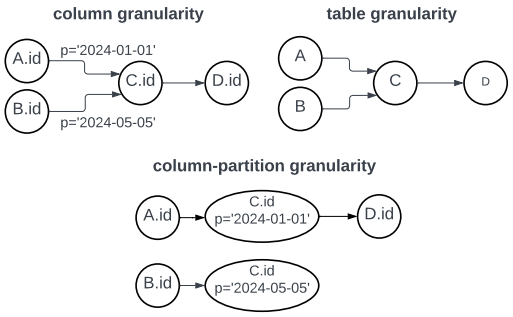


Figure 7: Example of a data warehouse flow leading to overly broad column- and table-level lineage. The column-partition granularity subgraph accurately depicts the real flows.

not provide column-level lineage, or due to not yet having ingested the column-level edge in a real-time query. ULS therefore surfaces `F.id`, `F.a` and `H` as candidate nodes.

We remark that while `D` is adjacent to `G` in the table graph, the edge `D.id`→`G.a` is explicitly excluded by the customer's filter criteria, and thus we can lift that exclusion to the table graph and do not need to select `G` as a candidate. More generally, if a flow is understood at a finer-granularity graph, it does not need to be surfaced as a candidate in a coarser-granularity graph. The consumer now can choose which of `F.id`, `F.a` and `H` to include in traversal. For any that they select, the traversal API repeats the process, selecting new high-confidence nodes and curation candidates.

Note that the traversal API does not have a direct way to specify the desired precision and recall. The API focuses on leveraging as much as possible of the lineage metadata to do a *precise* traversal (see Section 7.2). Recall can be controlled through additional API knobs. For example, a use case may decide to use strictly column-level data and not traverse flows with only table-level grain. The API also enables users to filter specific asset types, avoiding traversal of certain flows if they are not of interest for a given use case (e.g., only data warehouse flows).

7.2 Column-Partition Granularity

As discussed in Section 5.2, the ULS graph represents different views of data flows through the column, table and partition granularities. In this section we introduce the *column-partition granularity* view used to achieve more accurate graph traversal for our Data Warehouse flows. The column-partition subgraph is generated in a post-processed fashion based on both the column and partition subgraphs. It can be stored on a separate new Hive table or generated on the fly by the Global Graph Traveral APIs.

To motivate the need for column-partition granularity, consider the example in Figure 7, where data of different partitions in our data warehouse is copied across table columns as in `A.id -> C.id -> D.id` for partition '2024-01-01' and `B.id -> C.id` for partition '2024-05-05'. Figure 7 shows the two column and table ULS subgraphs for the lineage traversal query looking for *all data copies of A.id and B.id for partitions '2024-01-01' and '2024-05-05'*. The column and table subgraphs rightfully include `C.id` and `D.id` as copies of `A.id`. However, downstream of `B.id` wrongly includes `D.id` and `D` as the flow `C -> D` does not include any data from `B`. If there were more

assets downstream of D, this initial error would keep compounding with even more assets being reported as downstream of B.

This example clearly shows that *lineage flows are only transitive when the flows are modeled and metadata is available at the grain in which the real flows occur*. In this case, this is both the column and partition. For our warehouse production flows, we observe that strictly tracking flows at the column and subcolumn granularity is not sufficient to fully cover our use cases. We need to incorporate *table partition information when available* as independent queries write to and read from *different partitions of a Hive table*.

The column-partition granularity is a view of the data flow graph where data flows from a column (or subcolumn) in a source partition to another column (or subcolumn) in a destination partition. Figure 7 shows the column-partition granularity subgraph which correctly shows that in fact there are two different flows from `A.id` and `B.id` and that only one copies data to `D.id` for partition '2024-01-01'. To relate the column information with the partition flow we use the *evidence node*. For a single query, both column and partition flows have the query id as the evidence. As a result, we can join both information to achieve a more precise view of the flow. Section 9.2 illustrates the accuracy improvements achieved by combining information from the column and partition subgraphs.

7.2.1 Reducing the size of the column-partition graph. Given that the column-partition graph is a combination of the column and partition granularities, a single query can yield millions of edges. This is also exacerbated as in our current implementation, query engines only report *all partitions read* and *all partitions written* for every query, which results in ULS connecting all input partitions to all output partitions for every query, as in Figure 4.

To reduce the size of the partition graph, the ULS Tailer further analyzes partition lineage together with the SQL query column lineage extraction process. The Tailer verifies that that the SQL analysis reports that the *input and output partition columns are an exact copy of each other*. If so, the Tailer discards partition combinations in which the partition values are not equal to each other. This technique resulted in substantial reduction of the ULS partition lineage subgraph size, with the daily partition subgraph reducing about 20% in terms of number of edges and with the materialized daily column-partition size reducing up to 60%.

8 Lineage measurement

Measuring the coverage or accuracy of ULS's outputs—box ⑤ in Figure 5—is technically challenging, because it requires comparing ULS outputs against a ground truth dataset of "real" data flows, but such a ground truth is hard to build at scale. Furthermore, consumers generally do not care about *global* precision and recall, but only on precision and recall for their own datasets. For example, the lineage subgraph may have a high false positive rate on one specific type of table, but if no consumers' data flows into this type of table then the impact of this lowered data quality may be limited. To measure coverage and accuracy we rely on three high-level approaches namely unit testing, known flows and proxies of truth.
A. Unit Testing: ULS components must have unit tests to ensure correct lineage flow metadata. For example, the column lineage service in Section 4 uses an SQL parsing library to return the set of flows that a particular query creates. Our internal library has over 150 tests on different SQL queries to confirm that the expected flows for each of them and no other flows are returned. This tests the coverage and accuracy of each component independently.
B. Known-Flows: For certain datasets, such as ones which were manually curated in the past, we know the precise set of flows which should exist. We can capture these as "known flows", and compare them against ULS's outputs on the same root queries. This enables measurement of both coverage (how many of the known flows were returned) and accuracy (how many of the returned flows were known). Creating such known-flows datasets from scratch involves significant manual effort, but often they can be repurposed from previous work. Coverage and accuracy for these datasets can then be measured automatically and reported for evaluation.

One significant advantage of using known-flows datasets from previous consumers' investigations is that they are by construction relevant to their consumers. That is, they are a sample of root and downstream assets for which coverage and accuracy are known to be important to measure. One disadvantage is that, since the flows are manually curated, they do not always remain accurate over time. For instance, if a known flow detected in a previous incident is deprecated as part of the long-term incident response, ULS may correctly stop returning it in the output. Thus, discrepancies between ULS and a known-flows dataset may be errors in the dataset, and cannot always be assumed to be errors in ULS. To address this, each discrepancy can be manually reviewed by inspecting additional metadata on the assets in each flow such as code pointers, last update times, or existence in an asset catalog. Manual review can either identify an error in ULS or update the known-flows dataset.
C. Proxies of Truth: Some lineage providers have an alternative mechanism to measure data flows. For example: one lineage provider has a direct ULS integration which reports a lineage event for every query it executes. It also maintains operational metrics for its own debugging and accounting purposes, such as to enable measurement of memory footprints or query plan optimization. We compare its operational metrics with ULS's output, and ensure that at least 99% of queries from the former also appear in the latter. This ensures that ULS is not missing data. Although not all providers have internal datasets that can be used for such comparisons, many of the largest providers do, and thus these can be used to test coverage.

9 Evaluation

In this section, we provide a quantitative view of the lineage graph (Section 9.1), a performance analysis of our two key methods of granularity sweeping 9.3 and column-partition granularity (Section 9.2). Finally, we describe two case studies of using ULS in real-world development scenarios (Section 9.4).

9.1 Lineage Graph measurements

9.1.1 Graph sizes across granularities: ULS's daily run constructs a lineage graph containing approximately a billion nodes and edges from nearly 50 providers, varying by nearly 2x day-to-day. We depict a sample of two weeks of daily graph sizes in Figure 8, starting on an arbitrary date. All numbers are normalized to the number of edges on the table granularity on the first day of measurement. We give sample data in Table 1; similar results appear in other periods.

# daily nodes	approx 1 billion
# daily edges	approx 12 billion
% of column edges of quality 1,2,3,4	65% / 21% / 14% / 0%
# providers	approx 50
# monthly users of ULS tooling	approx 4000

Table 1: ULS data and usage statistics.

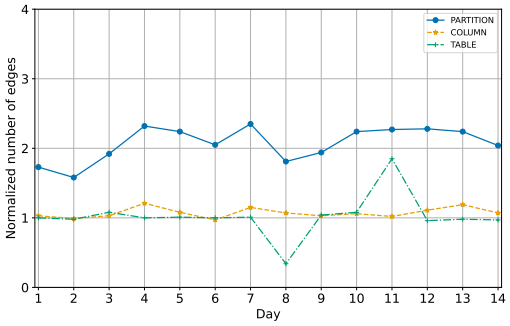


Figure 8: Lineage graph edges across granularities.

We first note that the partition subgraphs has consistently more edges than both the table and column subgraphs. This is as expected as data warehouse flows typically read data from multiple partitions of a table and write it to multiple partitions of another table. This results in several partition-level edges but a single edge at the table granularity. Further, note that while the column granularity also has finer grained nodes (and therefore more edges), table granularity has 12 additional providers than column, as it is generally harder to obtain flow metadata at this level of fidelity. In this case, the column subgraph has a similar number of edges as the table subgraph.

Second, there is a significant daily variation in the size of the graph, ranging from −66% to +85% of the initial data point. This is mainly due to changing workloads and missing data. While some workloads are daily, there are many weekly or monthly operations. These will report extra lineage edges only on the days that they run. This is the cause of the spike on day 11: one provider's workload increased by 6x on that specific day. For the latter, in a production environment operational factors mean that data are often delayed, and ULS cannot wait for all providers to submit their data. This is the cause of the drop on day 8: one of the largest lineage providers failed to submit its data until the following day. This variability poses a challenge to ULS users, who often add lookback periods into their queries to smooth out day-over-day issues.

Note that while the overall graph is very large, not all data is used for all use cases. Some scenarios only care about table and column lineage, while others need the more detailed information from partition lineage. ULS is a general purpose system, in which other special purpose applications can be built on top of consuming only a subset of the data. We continuously re-evaluate where data representation can be simplified or removed, so that we can reduce the overall dataset volume while still providing sufficient metadata to meet the needs of the different use cases.

9.1.2 Input and output edge distribution. In Figure 9 we depict the Cumulative Distribution Function (CDF) of the table, column and partition vertex degrees. All three CDFs are skewed with a long tail of high-degree vertices, but the majority of nodes have an input (41-93%) or output (58-71%) degree of 1. The table graph has the largest variation in degree, with a $p95$ output degree of 252. This is in part because table data is the most frequently provided data type, and with the most low-confidence data providers.

To enable traversal of high-degree nodes, users can configure the Global Graph Traversal API to pause traversal once it reaches a node with a degree meeting a specified user threshold. This gives users the opportunity to curate the node first before continuing traversal. Our distribution graphs shows that high degree nodes are common, with a few nodes reaching more than a million distinct neighbors. Often, such nodes reflect flows from core infrastructure components, such as a deletion system.

9.1.3 Longest Path distribution. We measure the distribution of downstream paths from each node. To do so, we first postprocess the graph to increase the data quality: we filter only to high-quality column edges to avoid long spurious paths connected by low-quality edges, and aggregate over seven days to include flows that span more than one day. We then filter down to only edge types known to copy data. The resulting CDF is depicted in Figure 10. We also include the CDF of the path length distribution when including *function* and *aggregation* edges as well as copies.

The $p95$ path length is 17 (copy-only: 7), and the longest path contains 193 nodes. We confirmed that this is a true data flow, but one whose data pipeline contains a large number of intermediate tables before generating its final output. This is a fairly common pattern to increase development speed at the cost of additional storage. We also note that several libraries themselves create intermediate data notes, which are nevertheless reported to ULS. Having long paths highlights the importance of our traversal tooling doing recursive traversal of high-confidence lineage where possible.

9.2 Column-Partition impact

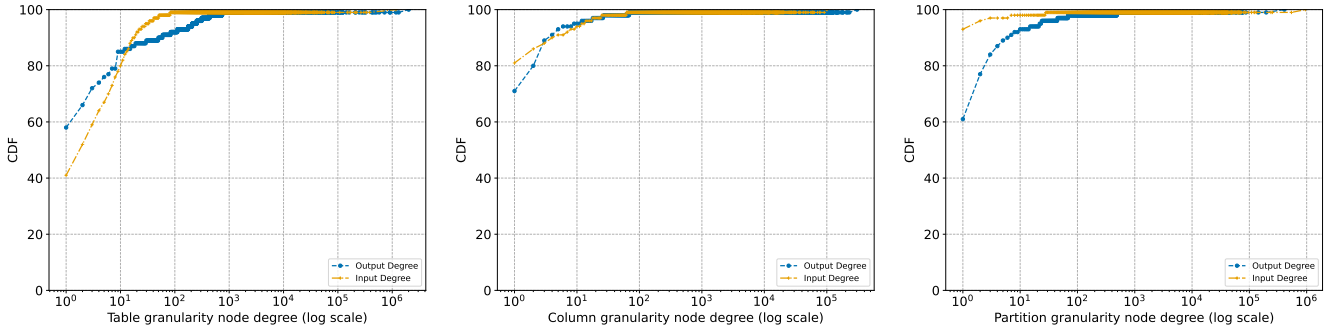
We conducted an experiment to evaluate the effect of including partition information in lineage discovery. We select random sets of 10, 50 and 100 root nodes in the lineage graph from our production use cases, traverse 10 hops into the lineage graph, and measure the number of nodes of the resulting subgraph. This simulates a consumer using ULS to discover the assets receiving data from a fixed set of known root nodes. The results are depicted in Figure 11.

In all cases, using only column-level lineage results in including additional nodes in the output set. These nodes do not contain flows from source partitions, but instead flows from source *columns* and thus are detected as false positives. Up to 20% of output nodes fall into this case. In other words, using ULS's column-partition lineage can reduce false positive rates by up to 20%.

9.3 Granularity Sweeping Impact

We compare two methods of performing one-shot lineage queries: (a) a naïve traversal that includes all low-quality edges, and (b) excluding low-quality edges and relying instead on granularity sweeping as in Section 7.1 to ensure sufficient coverage.

We used three different discovery setups of 10/50/100 root nodes sampled from real scenarios used in production, and traversed from each set to 10 levels deep. Figure 12 shows the sizes of the different subgraphs. Granularity sweeping splits the output subgraph into



(a) Table granularity **(b) Column granularity** **(c) Partition granularity**

Figure 9: Input and output degree CDF of a ULS graph nodes across granularities on a single day.

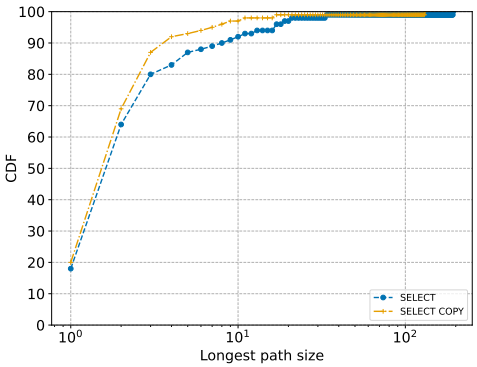


Figure 10: CDF of maximum path length, column subgraph.

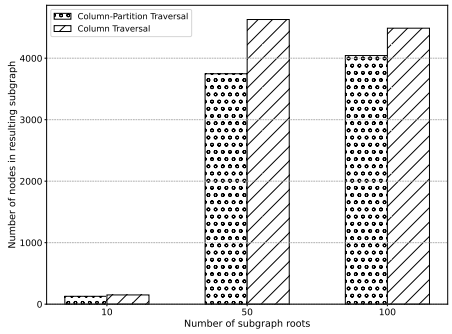


Figure 11: Impact of column partition on subgraph size.

included nodes and nodes which *need review* and additional curation. The granularity sweeping traversal can produce significantly smaller subgraphs (up to a 59.1% reduction in the 100 roots traversal) with 321 nodes to review. The size reduction increases with the number of root nodes, likely because adding more roots increases the chance that traversal will intersect with low-confidence lineage nodes that become superspreaders. This shows that granularity sweeping indeed raises low-confidence lineage for review rather than traversing further, allowing use cases to review lower-quality flows early, and remove any false positive inclusions stemming from them, before further downstream traversal occurs.

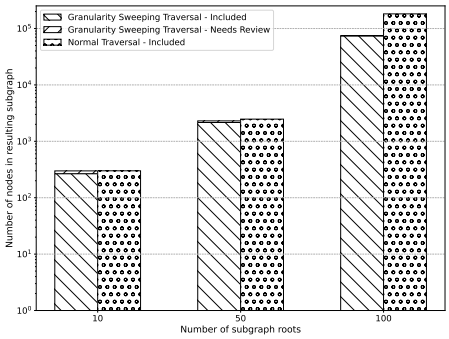


Figure 12: Impact of granularity sweeping on subgraph size.

9.4 Case Studies

9.4.1 Graph Explosion. ULS is under continuous development, as developers add more lineage providers and increase the accuracy of existing ones. We refer to data changes that substantially increase or decrease the size of lineage subgraphs as *graph explosions* and *implosions* respectively. Although some explosions and implosions represent step-change improvements in ULS data quality, we have observed that in practice they are generally due to data issues.

For example, in one case an existing lineage provider added a new set of edges which connected a previously disconnected subgraph to the main cluster of the graph. This subgraph had incorrect annotations on its edges that were marked as "high-quality". ULS therefore started including incorrect lineage flows via this service in its outputs. The incident was resolved by improving the quality annotations on the newly connected subgraph.

To avoid such incidents, we ensure that new lineage events are tagged with an experiment identifier. When data is added to ULS with a new experiment identifier tag, ULS creates two versions of the lineage graph: one containing just production events, and the other including also the experimental ones. Use case owners can then write data quality checks (e.g., subgraph size change thresholds or known flows that should be detected) which are verified in the experimental version of the graph. If a check fails, ULS engineers are alerted to investigate and resolve. Conversely, if the checks pass consistently, the experimental data can be safely added.

9.4.2 Data Age. In this section, we describe a use of ULS for privacy. Companies may want to ensure that data is deleted within n days of a user request [17]. One mechanism to implement this data deletion is to limit table retention periods to n days, for instance by setting an upper limit on the age of date partitions. This ensures that any specific user data deletion request will be satisfied within n days, by removing *all* data from the table within the required period.

Unfortunately, relying solely on partition retention limits is insufficient: a query can create a new partition that reads *previous* partitions of its input tables. For example, consider a partition `bar.2024-01-02` defined as `SELECT * FROM foo WHERE date='2024-01-01'`: `bar`'s partition today is a copy of `foo`'s partition *yesterday*. This means that n days after a user's deletion request, one day of that user's data could still remain in `bar`. We say that `bar` introduces "one day of age lag". Age lag can be introduced by any query reading earlier date partitions and writing to later ones.

There are several strategies to handle age lag. One approach is to permit it as long as the total amount of age lag, plus the total table retention, is less than n. Another is to filter out deleted users when copying data from earlier partitions, for instance by performing an `INNER JOIN` with a table of current users. We note that remediating one table may fix age lag issues in all of its downstreams, and thus choosing where to apply filtering is important to reduce total effort.

We deployed ULS to identify instances of age lag to ensure that user data deletion requests are fulfilled in time. Using the lineage graph, we applied a constraint optimizer to select a minimum set of approximately 1,700 core tables to apply deleted data filtering, and approximately 2,400 core tables on which to reduce the total table retention, resolving approximately 21,000 tables downstream.

10 Related Work

Over the years, data lineage (or provenance) has been instrumental in tracking and understanding data movements in database management systems [10, 11, 41, 53]. Data provenance is now key to many different data processing systems beyond traditional databases such as graph processing [3], data stream processing systems [38], log analysis [23, 43], non-SQL databases [12], Spark-based systems [27, 28, 49], ML systems [42], scientific workflows [26], distributed systems [52], and blockchains [46]. Tailored approaches also exist for a variety of domain applications such as in data science [12, 47, 54], security and privacy [39], and debugging [4, 52]. Compared to such bespoke approaches tailored for specific data processing systems and applications ULS uniquely aggregates lineage data from many different providers and systems into a single global graph for consumption across use cases.

Whole-system provenance [40] refers to capturing lineage events and tracking data flows across the kernel and the application stack. To stitch events such systems require either to taint lineage data or to exploit application semantics and are evaluated on small and medium-size network of nodes [8, 22, 36, 47]. ULS is a large-scale lineage system designed to support our user-level data pipelines composed of generic processing frameworks. ULS's stitching approaches are application-agnostic.

Microsoft Purview [1] and Databricks Unity Catalog [21] are data governance solutions also providing lineage functionalities. Both aggregate data from multiple providers and may have lineage data at multiple granularities. In this paper, we describe how ULS handles lineage metadata that has granularity differences, both in the data model and when building consumption experiences.

Several approaches focus on optimizing lineage capture overheads and lineage query processing costs [24, 44]. ULS provides different APIs for consuming lineage (sync and async) that have different query processing costs, operating with reasonable overheads that are acceptable for our use cases. In cases where near real-time access to lineage data is required [44], ULS provides a traversal API that combines historical and fresh data to answer lineage queries.

There are tailored lineage approaches for different data management systems. For example, [18–20, 31] track data in data warehouses using relational views over relational sources and general transformations. Others devise coarse-grain lineage at column- and table-level in databases [5, 15]. ULS has a dedicated component for SQL query analysis and generates column and subcolumn lineage edges. These results are aggregated and connected with lineage metadata coming from other sources.

There is related work for scalable provenance [6, 13, 32, 45]. ULS produces a daily graph of billions of nodes and edges, and a scalable platform for users to query the graph. In our experience, in addition to storing the data, we must provide users with consumption APIs so that users can properly understand the available lineage data.

Finally, Amsterdamer et al. [5] propose zoom in and out transformation operations for workflow provenance analysis to provide fine- and coarse-grain views of the provenance graph to reveal or hide respectively intermediate computations. This work has similarities to our notion of column- and table-lineage granularities and containment edges. However, ULS leverages the connection between different granularities through the granularity sweeping method to traverse the subgraphs at different granularities in cases where high quality lineage is missing from fine-grain subgraphs.

11 Conclusion

In this work, we presented ULS, a lineage system used to track data flows. ULS addresses key challenges in data lineage management, such as enabling representation of data flows at different granularities and with different confidence levels, and providing traversal tooling to enable high precision and high recall lineage subgraph queries. While ULS is used in several internal applications, providing high quality answers to lineage queries requires continuous iteration and refinement of the lineage metadata. This is because patterns of our internal workloads change over time, and ULS must evolve to represent them well and continue to fulfill our use cases. In the future, we plan to continue improving our lineage consumption experiences, such as facilitating precise label propagation and further scale our traversal tooling.

Acknowledgments

We would like to thank (in alphabetical order): Adrian Zgorzalek, Alex Lambert, Brani Stojkovic, Calvin Li, Dzmitry Charnahalau, Francesco Logozzo, Gayathri Ayer, Joe Hu, John Roll, Kai Li, Komal Mangtani, Lucas Waye, Ning Yang, Rajesh Nishtala, Rishab Mangla, Sandy Yen, Seth Silverman, Tuncay Tekle, and Wenlong Dong.

References

[1] Shafi Ahmad, Dillidorai Arumugam, Srdan Bozovic, Elnata Degefa, Sailesh Duvvuri, Steven Gott, Nitish Gupta, Joachim Hammer, Nivedita Kaluskar, Raghav Kaushik, Rakesh Khanduja, Prasad Mujumdar, Gaurav Malhotra, Pankaj Naik, Nikolas Ogg, Krishna Kumar Parthasarthy, Raghu Ramakrishnan, Vlad Rodriguez, Rahul Sharma, Jakub Szymaszek, and Andreas Wolter. 2023. Microsoft Purview: A System for Central Governance of Data. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3624–3635.

[2] Sultan Alneyadi, Elankayer Sithirasenan, and Vallipuram Muthukkumarasamy. 2016. A survey on data leakage prevention systems. *J. Netw. Comput. Appl.* 62 (Feb. 2016), 137–152.

[3] Enes Altinisik, Fatih Deniz, and Hüsrev Taha Sencar. 2023. ProvG-Searcher: A Graph Representation Learning Approach for Efficient Provenance Graph Search. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2247–2261.

[4] Peter Alvaro, Joshua Rosen, and Joseph M. Hellerstein. 2015. Lineage-driven Fault Injection. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 331–346.

[5] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. 2011. Putting lipstick on pig: enabling database-style workflow provenance. *Proc. VLDB Endow.* 5, 4 (Dec. 2011), 346–357.

[6] Manish Kumar Anand, Shawn Bowers, Timothy McPhillips, and Bertram Ludäscher. 2009. Efficient provenance storage over nested data collections. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Saint Petersburg, Russia) *(EDBT '09)*. Association for Computing Machinery, 958–969.

[7] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: Online Modelling and Performance-aware Systems. In *9th Workshop on Hot Topics in Operating Systems (HotOS IX)*. USENIX Association, Lihue, HI. https://www.usenix.org/conference/hotos-ix/magpie-online-modelling-and-performance-aware-systems

[8] Adam Bates, Dave (Jing) Tian, Kevin R.B. Butler, and Thomas Moyer. 2015. Trustworthy Whole-System Provenance for the Linux Kernel. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, Washington, D.C., 319–334.

[9] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[10] Peter Buneman, Adriane Chapman, and James Cheney. 2006. Provenance management in curated databases. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, 539–550.

[11] Peter Buneman and Wang-Chiew Tan. 2019. Data Provenance: What next? *SIGMOD Rec.* 47, 3 (February 2019), 5–16.

[12] Adriane Chapman, Paolo Missier, Giulia Simonelli, and Riccardo Torlone. 2020. Capturing and querying fine-grained provenance of preprocessing pipelines in data science. *Proc. VLDB Endow.* 14, 4 (2020), 507–520.

[13] Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. 2008. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 993–1006.

[14] Guoqiang Jerry Chen, Janet L. Wiener, Shridhar Iyer, Anshul Jaiswal, Ran Lei, Nikhil Simha, Wei Wang, Kevin Wilfong, Tim Williamson, and Serhat Yilmaz. 2016. Realtime Data Processing at Facebook. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1087–1098.

[15] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends Databases* 1, 4 (April 2009), 379–474.

[16] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. 2015. One trillion edges: Graph processing at facebook-scale. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1804–1815.

[17] Katriel Cohn-Gordon, Georgios Damaskinos, Divino Neto, Joshi Cordova, Benoît Reitz, Benjamin Strahs, Daniel Obenshain, Paul Pearce, and Ioannis Papagiannis. 2020. DelF: Safeguarding deletion correctness in Online Social Networks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/cohn-gordon

[18] Yingwei Cui and Jennifer Widom. 2001. Lineage Tracing for General Data Warehouse Transformations. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 471–480.

[19] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.* 25, 2 (June 2000), 179–227.

[20] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the lineage of view data in a warehousing environment. 25, 2 (2000), 179–227.

[21] Databricks. 2024. Capture and view data lineage using Unity Catalog. https://docs.databricks.com/en/data-governance/unity-catalog/data-lineage.html

[22] Ashish Gehani and Dawood Tariq. 2012. SPADE: support for provenance auditing in distributed environments. In *Proceedings of the 13th International Middleware Conference* (ontreal, Quebec, Canada) *(Middleware '12)*. Springer-Verlag, 101–120.

[23] Devarshi Ghoshal and Beth Plale. 2013. Provenance from log files: a BigData problem. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops* (Genoa, Italy) *(EDBT '13)*. Association for Computing Machinery, New York, NY, USA, 290–297.

[24] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing Provenance and Data on the Same Data Model through Query Rewriting. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE '09)*. IEEE Computer Society, USA, 174–185.

[25] Michael Hart, Pratyusa Manadhata, and Rob Johnson. 2011. Text Classification for Data Loss Prevention. In *Privacy Enhancing Technologies*, Simone Fischer-Hübner and Nicholas Hopper (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 18–37.

[26] Thomas Heinis and Gustavo Alonso. 2008. Efficient lineage tracking for scientific workflows. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 1007–1018.

[27] Daniel Hernández, Luis Galárraga, and Katja Hose. 2021. Computing how-provenance for SPARQL queries via query rewriting. *Proc. VLDB Endow.* 14, 13 (Sept. 2021), 3389–3401.

[28] Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson Condie. 2015. Titian: data provenance support in Spark. *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 216–227.

[29] Manolis Karpathiotakis, Dino Wernli, and Milos Stojanovic. 2019. *Scribe: Transporting petabytes per hour via a distributed, buffered queueing system.* Meta. https://engineering.fb.com/2019/10/07/core-infra/scribe/

[30] Rituraj Kirti and Diana Marsala. 2024. Approaches and Challenges to Purpose Limitation across Diverse Data Uses. USENIX Association, Santa Clara, CA.

[31] SDF Labs. 2024. Lineage. https://docs.sdf.com/guide/basics/lineage_metadata

[32] Seokki Lee, Bertram Ludäscher, and Boris Glavic. 2020. Approximate summaries for why and why-not provenance. *Proc. VLDB Endow.* 13, 6 (Feb. 2020), 912–924.

[33] Rishab Mangla, David Taieb, Wenlong Dong, Gabriela Jacques-Silva, Brani Stojkovic, Slobodan Predolac, Alex Lamber, Francesco Logozzo, and Taha Bekir Eren. 2025. How Meta discovers data flows via lineage at scale. https://engineering.fb.com/2025/01/22/security/how-meta-discovers-data-flows-via-lineage-at-scale/ Accessed: 2025-04-03.

[34] Patricia McKenzie and Rohit Ahuja. 2022. *The Ent Framework: Meta's Object-Relational Mapping.* Meta. https://www.facebook.com/watch/?v=444784850790246

[35] Meta. 2023. Data Engineering at meta: High-level overview of the Internal Tech Stack. https://medium.com/@AnalyticsAtMeta/data-engineering-at-meta-high-level-overview-of-the-internal-tech-stack-a200460a44fe

[36] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer. 2006. Provenance-aware storage systems. In *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference* (Boston, MA) *(ATEC '06)*. USENIX Association, USA, 4.

[37] Daniel Ohayon. 2022. Using static analysis for automated SQL query insights. https://engineering.fb.com/2022/11/30/data-infrastructure/static-analysis-sql-queries/ Accessed: 2024-04-30.

[38] Dimitris Palyvos-Giannas, Bastian Havers, Marina Papatriantafilou, and Vincenzo Gulisano. 2020. Ananke: a streaming framework for live forward provenance. *Proc. VLDB Endow.* 14, 3 (nov 2020), 391–403.

[39] Bofeng Pan, Natalia Stakhanova, and Suprio Ray. 2023. Data provenance in security and privacy. *Comput. Surveys* (2023).

[40] Thomas Pasquier, Xueyuan Han, Mark Goldstein, Thomas Moyer, David Eyers, Margo Seltzer, and Jean Bacon. 2017. Practical whole-system provenance capture. In *Proceedings of the 2017 Symposium on Cloud Computing (SoCC '17)*. ACM.

[41] Buneman Peter, Khanna Sanjeev, and Wang-Chiew Tan. 2001. Why and Where: A Characterization of Data Provenance. In *Database Theory — ICDT 2001*. Springer Berlin Heidelberg, Berlin, Heidelberg, 316–330.

[42] Arnab Phani, Benjamin Rath, and Matthias Boehm. 2021. LIMA: Fine-grained Lineage Tracing and Reuse in Machine Learning Systems. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, 1426–1439.

[43] Fotis Psallidas, Ashvin Agrawal, Chandru Sugunan, Khaled Ibrahim, Konstantinos Karanasos, Jesús Camacho-Rodríguez, Avrilia Floratou, Carlo Curino, and Raghu Ramakrishnan. 2023. OneProvenance: Efficient Extraction of Dynamic Coarse-Grained Provenance from Database Query Event Logs. *Proc. VLDB Endow.* 16, 12 (aug 2023), 3662–3675.

[44] Fotis Psallidas and Eugene Wu. 2018. Smoke: fine-grained lineage at interactive speed. *Proc. VLDB Endow.* 11, 6 (Feb. 2018), 719–732.

[45] Christopher Ré and Dan Suciu. 2008. Approximate lineage for probabilistic databases. *Proc. VLDB Endow.* 1, 1 (Aug. 2008), 797–808.

[46] Pingcheng Ruan, Gang Chen, Tien Tuan Anh Dinh, Qian Lin, Beng Chin Ooi, and Meihui Zhang. 2019. Fine-grained, secure and efficient data provenance on blockchain systems. *Proc. VLDB Endow.* 12, 9 (May 2019), 975–988.

[47] Lukas Rupprecht, James C. Davis, Constantine Arnold, Yaniv Gur, and Deepavali Bhagwat. 2020. Improving reproducibility of data science pipelines through transparent provenance capture. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3354–3368.

[48] Will Shackleton, Katriel Cohn-Gordon, Peter C. Rigby, Rui Abreu, James Gill, Nachiappan Nagappan, Karim Nakad, Ioannis Papagiannis, Luke Petre, Giorgi Megreli, Patrick Riggs, and James Saindon. 2023. Dead Code Removal at Meta: Automatically Deleting Millions of Lines of Code and Petabytes of Deprecated Data. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '23)*. ACM. https://doi.org/10.1145/3611643.3613871

[49] Mingjie Tang, Saisai Shao, Weiqing Yang, Yanbo Liang, Yongyang Yu, Bikas Saha, and Dongjoon Hyun. 2019. SAC: A System for Big Data Lineage Tracking. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1964–1967.

[50] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Ning Zhang, Suresh Antony, Hao Liu, and Raghotham Murthy. 2010. Hive - a petabyte scale data warehouse using Hadoop. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. 996–1005. https://doi.org/10.1109/ICDE.2010.5447738

[51] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. 2010. Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 1013–1020.

[52] Michael Whittaker, Cristina Teodoropol, Peter Alvaro, and Joseph M. Hellerstein. 2018. Debugging Distributed Systems with Why-Across-Time Provenance. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 333–346.

[53] A. Woodruff and M. Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In *Proceedings 13th International Conference on Data Engineering*. 91–102.

[54] Nan Zheng and Zachary G. Ives. 2020. Compact, tamper-resistant archival of fine-grained provenance. *Proc. VLDB Endow.* 14, 4 (Dec. 2020), 485–497. https://doi.org/10.14778/3436905.3436909