OneProvenance: Efficient Extraction of Dynamic Coarse-Grained Provenance From Database Query Event Logs

Fotis Psallidas
Microsoft
fotis.psallidas@microsoft.com

Ashvin Agrawal
Microsoft
ashvin.agrawal@microsoft.com

Chandru Sugunan*
Snowflake
chandru.sugunan@snowflake.com

Khaled Ibrahim
Microsoft
khaled.ibrahim@microsoft.com

Konstantinos Karanasos*
Meta
kkaranasos@meta.com

Jesús Camacho-Rodríguez
Microsoft
jesusca@microsoft.com

Avrilia Floratou
Microsoft
avrilia.floratou@microsoft.com

Carlo Curino
Microsoft
carlo.curino@microsoft.com

Raghu Ramakrishnan
Microsoft
raghu@microsoft.com

ABSTRACT

Provenance encodes information that connects datasets, their generation workflows, and associated metadata (e.g., who or when executed a query). As such, it is instrumental for a wide range of critical governance applications (e.g., observability and auditing). Unfortunately, in the context of database systems, extracting coarse-grained provenance is a long-standing problem due to the complexity and sheer volume of database workflows. Provenance extraction from query event logs has been recently proposed as favorable because, in principle, can result in meaningful provenance graphs for provenance applications. Current approaches, however, (a) add substantial overhead to the database and provenance extraction workflows and (b) extract provenance that is noisy, omits query execution dependencies, and is not rich enough for upstream applications. To address these problems, we introduce OneProvenance: an efficient provenance extraction system from query event logs. OneProvenance addresses the unique challenges of log-based extraction by (a) identifying query execution dependencies through efficient log analysis, (b) extracting provenance through novel event transformations that account for query dependencies, and (c) introducing effective filtering optimizations. Our thorough experimental analysis shows that OneProvenance can improve extraction by up to ~18X compared to state-of-the-art baselines; our optimizations reduce the extraction noise and optimize performance even further. OneProvenance is deployed at scale by Microsoft Purview and actively supports customer provenance extraction needs (https://bit.ly/3N2JVGF).

PVLDB Reference Format:
Fotis Psallidas, Ashvin Agrawal, Chandru Sugunan, Khaled Ibrahim, Konstantinos Karanasos, Jesús Camacho-Rodríguez, Avrilia Floratou, Carlo Curino, Raghu Ramakrishnan. OneProvenance: Efficient Extraction of Dynamic Coarse-Grained Provenance From Database Logs. PVLDB, 16(12): 3662 - 3675, 2023. doi:10.14778/3611540.3611555

*Work done while at Microsoft.

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611555

1 INTRODUCTION

Data governance platforms aim to enable organizations to govern (e.g., catalog, overview, secure, analyze, and audit) their data estates. In this direction, Microsoft's governance platform, namely, Purview [73], makes a range of governance functionalities readily accessible to customers. (Other such platforms include Collibra [24], Alation [6], IBM Infosphere [58], or Informatica [57]—further highlighting the importance of data governance.) According to several recent business [41, 46] and academic [1] reports, and in line with our customer feedback, central to data governance is the ability to capture and use provenance (i.e., a graph encoding connections between inputs and outputs across workflows) and associated metadata (e.g., who executed a workflow) from across data systems. Unsurprisingly, since databases play a critical role in data management, capturing provenance from database systems has been one of the most requested features from Microsoft Purview customers.

Extracting provenance from database systems is challenging, however, due to the complexity and size of database workflows. Provenance extractors, such as the ones supported by the major governance platforms above, can be classified into *static* and *dynamic*. Static ones access information from database catalogs (e.g., tables, views, and stored procedures) and use static analysis to extract provenance information from them. The main advantage of these extractors is that they can be easily deployed. Unfortunately, however, such extractors can lead to incomplete or incorrect provenance graphs due to their inability to monitor the execution of queries (e.g., branches, triggers, or dynamic SQL). Hence, more recently, dynamic provenance extractors have been introduced to address these limitations. Dynamic provenance extractors operate by listening to events generated by database systems as a side effect of query execution and extracting provenance from these events.

Dynamic provenance was also one of the most highly requested features in Microsoft Purview from customers across a wide range of industries (e.g., finance, retail, healthcare, and public services). Designing an efficient and robust dynamic provenance extractor is not straightforward. In particular, based on customer interviews, we found that existing solutions have four main limitations:

Design overheads (L1): They extract provenance based on logical or physical plans carried over events generated during query execution (e.g., SAC [81], Spline [75], or OpenLineage for Spark [36]). As

we show in our experiments, this design adds significant overheads (up to ~18×) to query execution and provenance extraction. In real-world scenarios, as highlighted in our customer interviews, these overheads can be prohibitive—both performance- and cost-wise.

Limited support for complex scenarios (L2): They focus on each query in isolation, thus failing to capture important dependencies among queries (e.g., queries executed by a stored procedure). Such dependencies are ubiquitous and can be complex in practice [48]. Hence, resulting provenance graphs are largely incomplete (e.g., no provenance of stored procedures or how queries trigger one another) and, as such, hard to explore and reason upon.

Absence of drill-down/roll-up capabilities (L3): Because existing extractors fail to capture dependencies among queries, customers also pointed out that such extractors do not allow reasoning at various levels of an application. For instance, a user might want first to explore provenance at the stored procedure level and then, if needed, drill down to the provenance of the queries of this stored procedure [77]. To provide such analysis capabilities, a provenance extractor should be able to aggregate provenance information.

Disconnect with consumer applications (L4): Finally, customers also highlighted that existing extractors fail to provide mechanisms to tailor the resulting provenance information to the needs of the consumer application. For instance, they treat each query as equally important. However, queries issued by system administrators or backup processes are unlikely to be of interest to business users. For such applications, the resulting provenance graphs are often considered noisy. Similarly, existing extractors lack a rich query runtime metadata model to provide the necessary context for upstream applications [53, 61] (e.g., who executed a query, from which application, or what was the CPU and IO costs).

To this end, we introduce OnеProvenance, a novel dynamic provenance extraction system that addresses the limitations of existing extractors. In particular, our system uses a novel extraction design that collects dynamic provenance from low-volume query logs, without relying on plans (**L1**). As such, OnеProvenance avoids excessive overheads on database execution and provenance extraction to the extent that OnеProvenance extracts provenance from even sizeable transactional workloads—extending coverage beyond the traditional focus on analytical and ETL workloads.

OnеProvenance tackles complex scenarios by identifying query dependencies through efficient query log analysis (**L2**) and providing drill-down/roll-up capabilities over these query dependencies (**L3**). More specifically, query dependencies are encoded in a novel tree data structure that we refer to as QQTree. Provenance is then aggregated based on parent-child relationships of QQTrees.

To better accommodate application-specific requirements (**L4**), we introduce filtering techniques, pushed down into various points of the extraction workflow, to eliminate noisy provenance information. Moreover, OnеProvenance employs an extensible query runtime metadata model that allows capturing application-specific metadata in the provenance graph. Importantly, our data model is in compliance with open standards (Apache Atlas [11]) for better interoperability with existing metadata management systems.

To summarize, our key contributions in this paper include:

- An expressive data model encoding dynamic provenance, metadata, and dependencies among queries—while complying with open standards such as Apache Atlas (Section 3).

```
1  CREATE PROCEDURE CleanAndAppendSalesHistory
2     @trackingSystemVersion int
3  AS
4  BEGIN
5      IF @trackingSystemVersion = 1
6        BEGIN
7          INSERT SalesHistory
8          SELECT c.CustomerId, c.Region,
9                 r.Rate * c.Amount AS Amount
10         FROM   StagedSales c JOIN
11                ConversionRate r ON c.Region = r.Region
12       END
13     ELSE
14       BEGIN
15         INSERT SalesHistory SELECT * FROM StagedSales
16       END
17 END
18 CREATE PROCEDURE SyncNewSales
19    @trackingSystemVersion int
20 AS
21 BEGIN
22     IF EXISTS(SELECT * FROM INFORMATION_SCHEMA.TABLES
23               WHERE TABLE_NAME='StagedSales')
24        DELETE FROM TABLE StagedSales;
25     BULK INSERT StagedSales FROM 'newSales.csv';
26     EXECUTE CleanAndAppendSalesHistory
27             @trackingSystemVersion;
28 END
29 EXECUTE SyncNewSales 2;
```

Figure 1: Workflow of our running example.

- An efficient and extensible dynamic provenance extractor that overcomes several limitations of existing extractors (Section 4).
- Filtering techniques to optimize the extraction process depending on application requirements (Section 5).
- How we integrated OnеProvenance with Purview (Section 6).
- A thorough experimental analysis across workflow types (from transactional to analytical ones) highlighting the performance of OnеProvenance (end-to-end, in individual components, and in comparison with state-of-the-art extraction techniques), and the benefits from our proposed optimizations (Section 7).

2 RUNNING EXAMPLE

To better highlight key points in our discussion, we use the following running example throughout the rest of the paper.

Consider the T-SQL script in Figure 1. The stored procedure SyncNewSales (lines 18-28) is populating the table `StagedSales` (deleting its previous contents, if it already exists) using an external CSV file. Then, it calls the stored procedure CleanAndAppendSalesHistory (lines 1-17) that is responsible for appending the staged data from the `StagedSales` table to the `SalesHistory` table. Finally, we execute SyncNewSales to run the workflow (line 29). Note that the query that populates `SalesHistory` varies depending on the value of the input parameter `@trackingSystemVersion`. As a result, provenance information and runtime metadata for these two stored procedures may change across their runs.

Using this example, we will show how we can extract dynamic provenance, identify dependencies (e.g., different executions of a stored procedure or queries that are part of a stored procedure), attach metadata to nodes of the provenance graph (e.g., who and when executed a stored procedure or CPU time of queries of a stored procedure), and aggregate provenance across queries.

3 DATA MODEL AND PROBLEM STATEMENT

Our overall goal is to extract semantically rich, coarse-grained provenance information from executed queries. In this section, we discuss our provenance model and associated problem statement.

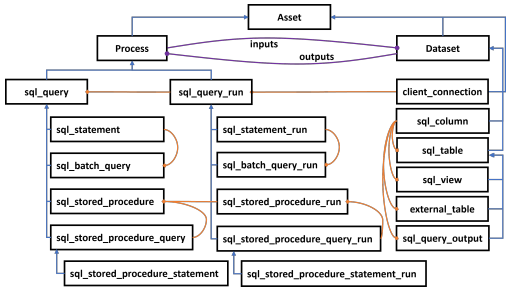


Figure 2: OnePROVENANCE Data Model. Blue lines (arrows), orange lines (diamonds), and purple lines (circles) correspond to inheritance, containment, and provenance relationships, respectively. (We omit attributes per type for brevity.)

3.1 Provenance Model

At a high level, we model coarse-grained provenance as a hypergraph that captures the relationships between datasets (e.g., tables or columns) and processes (e.g., queries). We now define the building blocks for modeling such a graph (i.e., our provenance model).

Since we aim to comply with open standards (recall Section 1), our provenance model is built on top of the, heavily extensible, APACHE ATLAS type system. More specifically, APACHE ATLAS introduces the generic entity types `Process` and `Dataset`. (Both types are derived from the generic `Asset` type of APACHE ATLAS, as shown in Figure 2.) Metadata on processes and datasets can be introduced as attributes (e.g., a dataset may have a name, size, and id) or through relationships (e.g., a client connection invokes a process). APACHE ATLAS supports inheritance and containment relationships that are relevant to our work, as we discuss below. Provenance is encoded as special relationships between `Process` and `Dataset` entities denoting the input/output datasets of a process. (Finally, note that APACHE ATLAS is only one target type system for provenance information. Our model can also be compiled to other standards, including OpenLineage [36] or W3C PROV-DM [64].)

With the background of APACHE ATLAS in place, we can now dive into our model which is depicted in Figure 2.

Datasets. In line with prior work on coarse-grained provenance [75, 81], datasets in our model include relations (such as tables, views, external tables, and query outputs) and their associated columns.

Processes. In our model, processes can be either *queries* or *query runs*. The former is traditionally the target of static provenance extractors, and the latter is the target of dynamic ones. Queries and their runs are then sub-typed through inheritance relationships. As shown in Figure 2, our model encodes ad-hoc statements, batches (encoded as a series of statements), queries that are part of stored procedures, and stored procedures. For every such static query type, we introduce its dynamic type by subtyping on the query run.

In addition, we use attributes to encode metadata for processes. In particular, for queries, we track the query text, and for query runs, we track the user that executed the query; CPU time; duration; rows inserted, updated, deleted, and returned. Finally, a client connection is attached to each query run (containment relationship) to encode from what application and server the query run was invoked.

Query dependencies. To support complex scenarios and enable drill-down and rollup capabilities (addressing L2-3 from Section 1), we introduce two types of dependencies: (1) query runs spawned by parent query runs and (2) runs of a query. The former allows us to encode what query runs have been triggered by other queries (e.g., queries executed as part of executing a stored procedure) and vice versa (e.g., what stored procedures a query has been part of). The latter allows us to track the runs of a particular query (e.g., the runs of a stored procedure). Both types of dependencies are encoded through containment relationships in our model (see also Figure 2).

Provenance. Finally, we model provenance as a hypergraph P connecting inputs with outputs across a workflow. Logically, each edge $i \overset{p}{\longleftrightarrow} o$ maps input i to output o, derived from i, through process p. As discussed, such processes are queries or query runs, while input and output datasets are relations and columns.

As discussed in Section 1, our main contribution to address **L3** is to aggregate provenance through query dependencies. We define the *provenance of a query run* as the set union of the provenance of the query runs that were executed as a result of executing that query (i.e., for query run Q_r we define its output $O(Q_r)$ as $O(Q_r) = \bigcup_{Q_r \leftarrow Q'_r} O(Q'_r)$, where $Q_r \leftarrow Q'_r$ denotes the set of query runs Q'_r triggered by Q_r; similarly for inputs). Furthermore, we define the *provenance of a query* as the set union of the provenance of its query runs (i.e., for query Q_s we define its output $O(Q_s)$ as $O(Q_s) = \bigcup_{Q_s \leftarrow Q_r} O(Q_r)$, where $Q_s \leftarrow Q_r$ denotes the set of query runs Q_r for the query Q_s; similarly for inputs). Note that provenance is defined recursively for both definitions. The base case is the provenance of individual statements. Similar to prior work, what constitutes an input or output of each individual statement is based on the SQL semantics of the statement type (e.g., a `CREATE TABLE X` statement has `X` as output); we discuss more on this in Section 4.4.

Note that by using the set union operator to aggregate provenance, we end up deduplicating multiple instances of an input or output dataset into a single dataset instance (e.g., in our example, if we execute SyncNewSales multiple times, each resulting in executing a query `INSERT SalesHistory...`, our set union semantics will lead to having `SalesHistory` as output of SyncNewSales only once). This design enables providing an overview of provenance with reduced noise (e.g., provenance of a stored procedure shows only single instances of inputs and outputs). At the same time, we still allow drilling down to capture all the details with respect to the alternative instances (e.g., on the provenance of the queries of the stored procedure). We discuss more on this in Section 4.5.

Finally, note that although input/output relationships in APACHE ATLAS can encode the input/output datasets of a process, they cannot encode which input contributes to which output. This would require a ternary relationship (i, p, o): input i contributes to output o through process p. Since APACHE ATLAS only supports binary relationships, a common workaround is to introduce the relationship (i, p, o) as an attribute of the process (typically serialized as a dictionary). Importantly, we still need to extract the APACHE ATLAS-based input/output relationships, despite being redundant with the information from the relationship (i, p, o), because other APACHE ATLAS features (e.g., provenance visualization or label propagation) rely on them. While not ideal, we opt for this workaround in favor of complying with the APACHE ATLAS open standard.

So far, we have introduced our provenance data model that forms the output of our system. Next, we discuss on the input: query logs.

Activity Id	Event Type	Query Text
3	sql_statement_started	EXECUTE SyncNewSales 2;
3	sp_statement_started	IF EXISTS(SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE name='StagedSales')
3	sp_statement_completed	IF EXISTS(SELECT * FROM INFORMATION_SCHEMA.TABLES WHERE name='StagedSales')
3	sp_statement_started	DROP TABLE StagedSales;
3	sp_statement_completed	DROP TABLE StagedSales;
3	sp_statement_started	BULK INSERT StagedSales FROM 'newSales.csv';
3	sp_statement_completed	BULK INSERT StagedSales FROM 'newSales.csv';
3	sp_statement_started	EXECUTE CleanAndAppendSalesHistory @trackingSystemVersion;
3	sp_statement_started	if @trackingSystemVersion = 1
3	sp_statement_completed	if @trackingSystemVersion = 1
3	sp_statement_started	INSERT SalesHistory SELECT * FROM StagedSales
3	sp_statement_completed	INSERT SalesHistory SELECT * FROM StagedSales
3	sp_statement_completed	EXECUTE CleanAndAppendSalesHistory @trackingSystemVersion;
3	sql_statement_completed	EXECUTE SyncNewSales 2;

Figure 3: Events for activity `EXECUTE SyncNewSales 2` (ordered based on the event triggered time). Each event is associated with activity id, event type, query text, and other metadata not shown for brevity (e.g., time triggered, CPU time, or on whose behalf a query was executed).

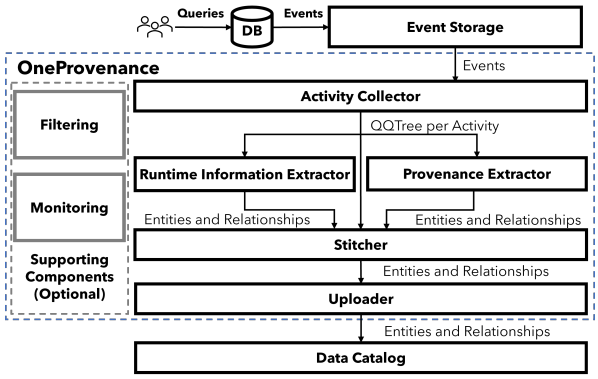


Figure 4: OneProvenance Architecture. Main components in OneProvenance's processing flow are in black boxes, and annotated with types of inputs and outputs. Grey boxes are optional components and can be used in various components of the main flow.

3.2 Query Log

We assume a database creates a query log as a side effect of query execution. We model a query log as an ordered set of events (ordered based on event trigger time). Next, we discuss the semantics we require from query logs. To ease our discussion, Figure 3 shows a query log created by Azure SQL DB for our running example.

Event types. Each event is associated with a query run, and can have one of the following two types: 1) *started* and 2) *completed*, indicating the start and completion of the corresponding query run, respectively. Furthermore, we assume each event has a type to encode whether its corresponding query is an individual or batch statement. For instance, in Figure 3, event types include `[sql|sp]_statement_[started|completed]` indicating the start and completion of ad hoc statements and statements of a stored procedure. For batches, such event types would be `sql_batch_[started|completed]` (not shown in Figure 3 for brevity).

Event schema. Each event can contain a rich set of runtime metadata (e.g., who, when, and from where executed a query run and for how much CPU time). As such, each event can be modeled as a record, where each piece of metadata is represented as an attribute. Note that events can have different schemas depending on whether they correspond to the start or completion of a query run, and individual statements or batches. For instance, only completed events contain how much CPU time was required by a query run.

Activities. Events in a log are grouped into *activities*, with each activity comprising a set of events triggered for correlated query runs. For instance, in Figure 3, all events belong to the same activity identified as 3. In general, such grouping is required to identify the set of dependent query runs of each query run, which is a critical step towards addressing **L2-3** of Section 1. Going back to our example, we use this information to deduce what queries have been run as part of SyncNewSales and CleanAndAppendSalesHistory; we discuss this in more detail in Section 4.2. (Note that in Figure 3, the activity id is provided by Azure SQL DB if causality tracking is enabled [84]. Not all databases provide such a capability, however. Yet every database needs to track such information for query execution purposes. Hence, we believe our discussion could also inform what databases need to log for effective provenance extraction purposes.)

3.3 Problems of Focus

Having defined our model and query log, our goal is twofold:

Provenance graph extraction: First, we aim to extract the provenance graph based on our model in Section 3.1. Formally, given a query log, our goal is to: (1) create the provenance graph $P(V, E)$ by extracting edges $E = \{i \overset{Q}{\longleftrightarrow} o \mid i \in I(Q) \subseteq V \text{ and } j \in O(Q) \subseteq V\}$, where Q is either a query run or its static counterpart, and types of inputs I(Q) and outputs O(Q) are relations (encoding relation-level provenance) or columns (encoding column-level provenance), and (2) associate nodes and edges in the provenance graph (i.e., inputs, outputs, and queries) with metadata from events. We discuss how OneProvenance addresses this problem in Section 4.

Noise reduction: Second, recall from Section 1 that a major limitation of existing dynamic extractors is that they treat every query in a query log as equally important, often resulting in noisy provenance graphs (**L4**). Hence, we aim to develop techniques to reduce the noise in the provenance graph and optimize the extraction process. We describe our filtering optimization techniques in Section 5.

4 ONEPROVENANCE

We are now ready to introduce OneProvenance, our provenance extraction engine. Its architecture is shown in Figure 4.

Overview. In short, events emitted by the database due to query execution are stored in an Events Log Storage (Section 4.1). Activity Collector then reads these events periodically; identifies SQL activities; and builds an internal representation for each activity, namely, QQTree, that encodes dependencies between query runs in the activity (Section 4.2). For each activity, the identified QQTree is pushed to Runtime Information Extractor (Section 4.3) and Provenance Extractor (Section 4.4) components. The former is responsible for extracting queries, query runs, and their relationships based on information encoded in the QQTree. The latter extracts provenance information from individual statements in the QQTree. The outputs of both components are encoded according to our provenance data model of Section 3.1. These outputs, along corresponding QQTrees, are then fed to the Stitcher component (Section 4.5), that (a) stitches the per-statement provenance from Provenance Extractor to the

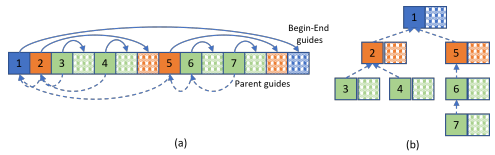


Figure 5: Example visualization of activity and QQTree construction. Solid and patterned fills indicate started and completed events, respectively. Blue, orange, and green boxes denote execution of batches, stored procedures or triggers, and statements, respectively.

queries and query runs extracted by Runtime Information Extractor, and (b) aggregates provenance to parent nodes and across runs (per **L2-3**). The end result is an instance of our data model that gets uploaded to the external Data Catalog through Uploader. Finally, we introduce hook points (Section 4.6) into OnePROVENANCE components to enable extensibility and optimizations (per **L4**).

We next discuss the role of each component and how OnePROVENANCE addresses technical challenges per component. Note that OnePROVENANCE is agnostic to the source database since we expect logs with specific semantics (Section 3.2). To ease our discussion, however, we consider Azure SQL DB as source database, since this is also the one we currently support in Microsoft Purview.

4.1 Event Logs Storage

At a high level, queries admitted to a database trigger the generation of events that are persisted by the database in a storage of event logs, as shown in Figure 4. We assume that the type of store can be provided as a configuration to the database. For instance, in Azure SQL DB, such events are known as `xEvents`, and the corresponding `xEvent` Store can be a local or remote file system, or even a finite memory buffer within the database server [84]. The choice of store is mainly influenced by the application needs and cost constraints. We have opted for Azure Storage [13] for our Event Logs Storage because this was the cheapest option among alternatives.

4.2 Activity Collector

The Activity Collector identifies activities and builds QQTrees that encode query execution dependencies in these activities. To do so, it retrieves new events from the Event Logs Storage, parses and groups events into activities, sorts activities by their trigger time, and constructs a QQTree per activity based on events of the activity.

To fetch events, the Activity Collector runs periodically (by default every 6 hours). To guarantee that activities that span multiple OnePROVENANCE runs will eventually be processed, all while processing activities only once, OnePROVENANCE uses checkpointing for runs: only events of activities that have at least one event created after the start of the last run of OnePROVENANCE will be processed.

Figure 5a illustrates an event log retrieved by the Activity Collector: each box represents an event, and the color and fill patterns are used to identify event type differences. In particular, blue, orange, and green boxes correspond to events for batches, stored procedures or triggers, and statements, respectively. Execution of each query results in the generation of two events: started (solid fill) and completed (patterned fill) events. The Activity Collector uses event attributes to build the QQTree for each activity, shown in Figure 5b.

Algorithm `QQTree Construction`

Input: Events $E_A = \{e_1, \ldots, e_n\}$ of a SQL activity A ordered by time triggered: $e_1^{time} < \ldots < e_n^{time}$ with e_i^{time} denoting the time e_i was triggered.
Output: QQTree X_A for SQL activity A.

```
1.  xStack:Stack[(event, node)] = ∅
    X_A:QQTree = ∅
    curParent:QQTreeNode=nil
2.  for each event e in E_A do
3.    if isStartedEvent(e) then
4.      node = Node(e, curParent)
5.      if xStack.isEmpty() then
6.        X_A.AddRoot(node)
7.        curParent = node
8.      else
9.        curParent.AddChild(node);
10.       if startsSubTree(e) then
11.         curParent = node
12.     xStack.Push((e, node))
13.   else
14.     (startedEvent, node) = xStack.Pop()
15.     if CheckErrors(startedEvent, node, e) then
16.       Abort()
17.     node.SetCompleted(e)
18.     if startsSubTree(startedEvent) then
19.       curParent = node.parent
20. if !xStack.Empty() then
21.   Abort()
22. return X_A
```

Figure 6: QQTree Construction Algorithm.

A QQTree is constructed by reading the events of an activity in the order that events were triggered, as shown in the algorithm of Figure 6. When a started event is encountered (lines 4-12), we generate a new node of the QQTree for this event (line 4), and push the node along the event into the stack (line 12). The new node either becomes the new root if the stack is empty (lines 5-7), or is appended to the children of the current parent (line 9). Importantly, the new node becomes the current parent (line 11) if the event corresponds to the start of a query that might result in the execution of other queries as part of it (e.g., stored procedure). If a completed event is encountered (lines 14-19), we first pop the started event and corresponding QQTree node from the stack (i.e., the completed event matches the started one at the top), and store the completed event in the popped node (line 17). Also, the parent of the popped node becomes the current parent if the popped node starts a subtree (lines 18-19). Finally, the algorithm checks for errors and aborts accordingly (lines 15-16, 20-21). Errors include malformed logs (e.g., activity starts with a completed event or a completed event matches a started one of a different type/query) or activities that have not finished execution (i.e., started events left in the stack).

Time and space complexity. The QQTree construction algorithm runs in $O(n)$ time (n being #events in an activity) by exploiting the ordering of events in the activity (i.e., a completed event always matches the started one at the top of stack). Space-wise, since only started events are pushed in the stack and generate a node in the QQTree, the extra space required is $O(m)$ (m being #started events).

Example 1. Consider the execution of SyncNewSales of our running example (Figure 1), and its corresponding query log from Azure SQL DB (Figure 3). (Recall all statements executed as part of SyncNewSales belong to the same activity.) The corresponding QQTree is shown in Figure 7. Log entries and QQTree nodes have been color-coded to denote matching of started and completed events along

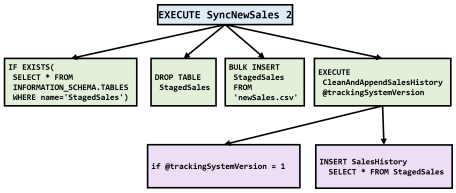


Figure 7: QQTree encoding query dependencies for the activity `EXECUTE SyncNewSales 2`. In every node, OnePROVENANCE tracks metadata available in the events that the node originated from.

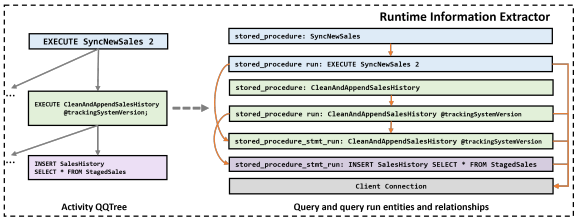


Figure 8: Runtime Information Extractor visits the nodes of QQTree of each activity to generate entities and relationships related to query and query runs per our provenance model of Section 3.1.

which QQTree nodes they contribute: The parent node denotes the execution of `EXECUTE SyncNewSales 2`, with children being all queries executed as part of `EXECUTE SyncNewSales 2` colored in green. Since SyncNewSales calls the stored procedure CleanAndAppendSalesHistory, queries executed as part of CleanAndAppendSalesHistory become children of CleanAndAppendSalesHistory, and introduce a second level in the QQTree colored in purple.

4.3 Runtime Information Extractor

Given a stream of identified activities, the goal of Runtime Information Extractor is to instantiate entities and relationships based on OnePROVENANCE's model. To do so, it analyzes events and query dependencies per activity using the corresponding QQTree.

Concretely, the extractor visits the nodes of a QQTree in a DFS traversal. If a visited node corresponds to a stored procedure execution, the extractor generates the corresponding stored procedure and stored procedure run entities (initiating their attributes, such as query text or CPU time, based on event metadata in the node), along with a relationship tying the stored procedure and its runs. Similar is the case for batches and individual statements. If a statement run X is part of a stored procedure run Y (similarly for batch), then a containment relationship is instantiated to encode that X is part of Y. Finally, a client connection is associated with run entities to encode which server and application triggered the execution.

To illustrate this process, consider again our running example. Figure 8 shows the entities and relationships generated by visiting (some of) the nodes of the QQTree of Figure 7. Visiting the root `EXECUTE SyncNewSales 2` results in generating the stored procedure and stored procedure run entities for SyncNewSales (blue boxes) along with a relationship between the stored procedure and its run. Similarly, visiting its child generates the nodes associated with CleanAndAppendSalesHistory. Note that the statement `EXECUTE CleanAndAppendSalesHistory @trackingSystemVersion`, that triggers the stored procedure CleanAndAppendSalesHistory,

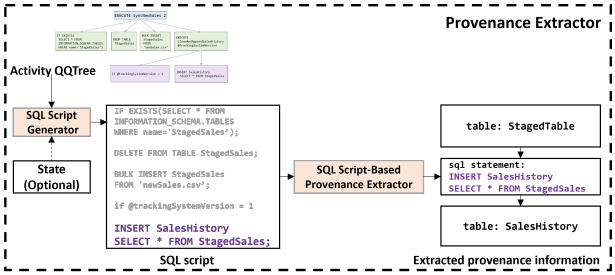


Figure 9: Provenance Extractor takes as input a QQTree and generates a script with the queries of the activity (SQL Script Generator). By analyzing the script it extracts entities (e.g., tables or columns) and their provenance relationships (SQL Script-Based Provenance Extractor). For proper identification of provenance relationships, cataloged information (e.g., table definitions) can be embedded in the script by accessing State (i.e., database catalog, catalog populated by previous runs of OnePROVENANCE, or cached copies of them).

is executed as part of SyncNewSales. Hence, besides generating the stored procedure and stored procedure run entities, OnePROVENANCE also generates the stored procedure statement run and the containment relationship with the SyncNewSales run entity. The case for the INSERT statement is similar, resulting in a stored procedure statement run and a containment relationship with its parent stored procedure run CleanAndAppendSalesHistory. Finally, all run entities are associated with the client connection that initiated the stored procedure execution.

The main focus of the Runtime Information Extractor is on extracting entities and relationships related to processes (queries and query runs). Extracting datasets and provenance relationships is the focus of Provenance Extractor and Stitcher that we discuss next.

4.4 Provenance Extractor

Provenance Extractor takes the stream of identified activities and extracts provenance relationships and datasets (e.g., tables, views, and columns) according to the data model of OnePROVENANCE.

The main components that comprise Provenance Extractor are highlighted in Figure 9: (1) SQL Script Generator and (2) SQL Script-Based Provenance Extractor. The former takes the stream of activities identified by Activity Collector and generates a SQL script per activity that includes the series of queries executed as part of the activity. The script is generated in a DFS pre-order traversal of the QQTree that appends the query text of each visited node to the script. Then this script is analyzed statically by the latter component, extracting dataset entities and provenance relationships from it. Importantly, SQL Script Generator is responsible for location tracking (i.e., mapping queries back to QQTree nodes they come from). This is necessary for stitching extracted provenance information with runtime information, as we will see in Section 4.5.

To ensure the correctness of the output entities and provenance information, one key technical challenge that provenance extractors need to address is binding objects in a query to catalog objects. For instance, for the query `INSERT SalesHistory SELECT * FROM StagedSales`, we need to know the schema of `SalesHistory` and `StagedSales` to provide column-level provenance. There are many

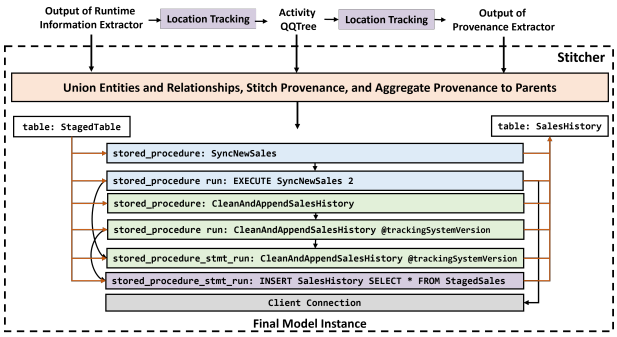


Figure 10: Stitcher operates on the outputs of Runtime Information and Provenance Extractors: unions entities and relationships from their outputs, stitches provenance from Provenance Extractor to statements extracted by Runtime Information Extractor, and aggregates provenance from statements to parent queries and query runs (e.g., stored procedures and their runs) based on the QQTree.

ways binding information can become available to a provenance extractor. OnePROVENANCE supports the following three cases.

First, queries appearing in a query log may already be bound (e.g., * in our example bound to the `StagedSales` columns). This is an ideal situation because it allows provenance extractors to operate in a stateless fashion. To be more precise, the ideal situation would be if objects in the query are mapped to their name *and* id in the source database catalog. This is because names alone may not be unique over time (e.g., if we create, drop, and then create a table with name X, then X may refer to the table before or after the drop).

If binding information is unavailable, OnePROVENANCE needs to infer it. To do so, Provenance Extractor maintains state to mirror the source database catalog. Hence, a second case supported by OnePROVENANCE is to infer binding based on available state. However, it is possible that state is either unavailable or not in sync with the database state (e.g., Provenance Extractor runs on logs post-mortem when the database catalog is unavailable). Under this third case, OnePROVENANCE extracts provenance under ambiguity, resorting to all-inputs-affect-all-outputs in the worst case.

Overall, we believe provenance extractors need to support all three cases, and customize behavior based on query workloads and log semantics. If a database catalog does not change, mirroring is a sensible option. Otherwise, either the log ensures strict serializability [14] for catalog replay or queries in the log need to be bound. If such approaches are infeasible, provenance extractors should run under best-effort semantics, making suggestions under ambiguity.

4.5 Stitcher

The goal of the Stitcher is two-fold: (1) union the set of entities and relationships from Runtime Information and Provenance Extractors, and (2) attach and aggregate provenance from Provenance Extractor to the process entities from Runtime Information Extractor.

Back to our running example, consider the inputs to Stitcher from Runtime Information and Provenance Extractors (Figure 10). For individual statements, the Stitcher attaches the provenance extracted by Provenance Extractor to the corresponding query and query run entities from Runtime Information Extractor. To do so, SQL Script Generator performs location tracking to map statements in the SQL script to QQTree nodes they came from, as discussed in Section 4.4. Similarly, Runtime Information Extractor maps output entities to QQTree nodes they come from. Stitching is then performed by going from statements in the SQL script to QQTree nodes they come from and, from there, to corresponding query and query run entities.

Finally, note that Provenance Extractor extracts provenance at the statement level. Stitcher is also responsible for aggregating the provenance information across runs and to parent entities following the QQTree structure. (Aggregation adheres to the set union semantics of Section 3.1.) For instance, `StagedSales` is aggregated as input to (a) the parent SyncNewSales run from the statement `INSERT SalesHistory ...` and (b) the SyncNewSales stored procedure. The end result for our example is shown in Figure 10.

The last component in the main flow of OnePROVENANCE is Uploader that uploads the instantiated provenance model to external Data Catalogs. This step is trivial, and we defer a discussion in [71].

Figure 11: Hooks in the logic of the Activity Collector.

4.6 Hook Points and Code Injection

OnePROVENANCE introduces hook points in its components to enable custom logic injection. These hook points are central to the extensibility and optimizations of OnePROVENANCE. In particular, hooks are used extensively for filtering optimizations (Section 5) but also for monitoring and debugging. Next, we discuss on hooks in Activity Collector; principles are similar across components.

The extraction logic in Activity Collector consists of four subcomponents (see Figure 11): (1) download batch of events from Event Logs Storage, (2) parse and deserialize events into activities, (3) sort activities temporally, and (4) construct a QQTree per activity. Activity Collector then exposes a set of hook points for code injection. ●s correspond to points at the start of (sub-)components while ★s are points at the end of (sub-)components. ■s correspond to points right before a component sends its results to other components, while ♦s correspond to points after sending these results (i.e., when the control flow returns to the component). Finally, ▲s correspond to points right before the end of loops over data items of focus in (sub-)components. For instance, we can use the ▲ point when constructing QQTrees to filter QQTree nodes that are not interesting for provenance applications. For a discussion on programming interfaces and state exposed on hook points, see [71].

5 OPTIMIZATIONS

So far, we have presented the workflow of OnePROVENANCE, assuming that every event in a log is equally important. Based on customer feedback, however, this assumption is rarely true in real-world applications. This is because event logs contain either uninteresting (e.g., system maintenance or statistic generation queries) or redundant (e.g., queries executed in loops) information that is overall not useful for business purposes. As such, extraction over all such events results in performance overheads only to extract noise.

In this section, we present simple yet powerful application-aware filtering optimizations for reducing the emitted noise. These optimizations have yielded up to four orders of magnitude performance improvements in customer workloads. Provenance applications of such customers include end-of-fiscal-year audits, data observability (e.g., are outputs of a stored procedure updated?), data estate understanding (e.g., which datasets are produced by a stored procedure?), context-based analytics [53], or impact and root cause analysis.

As shown in Figure 4, the main items in the flow of OneProvenance are events, queries, activities (each modeled as a QQTree or series of events), entities, and relationships. Our filters can be applied on such items throughout the flow of OneProvenance:

Loop compression. Loops executing queries iteratively introduce redundancies in the extracted provenance information and can overwhelm a provenance extraction system. To prevent this issue, OneProvenance introduces a filter that groups QQTree nodes originating from multiple iterations of the same loop. Then, OneProvenance can be configured to keep only the latest k iterations of the loop. Using this filter, we avoid calling Provenance and Runtime Information Extractors with a sheer volume of queries and, thus, overloading the data catalog and applications with noise. Additionally, this filter reduces the working memory per activity.

Drop uninteresting queries. OneProvenance can also avoid capturing the provenance of uninteresting queries—to better meet application needs. In particular, queries can be matched based on several factors, including their type (e.g., `SELECT` or CRUD statements), syntax tree (e.g., queries to compute statistics in Azure SQL DB are structured as `SELECT STATMAN(...) FROM T`), access or not to tables and columns (e.g., `SET @a=2`), or their query text itself (e.g., queries not matching a regular expression). For each matched query, we can either (a) drop the QQTree node corresponding to the query altogether or (b) consider the node in Runtime Information Extractor and ignore it in Provenance Extractor (e.g., filter out `SET @a=2` from Provenance Extractor since it has no provenance, but consider it in Runtime Information Extractor for metadata extraction).

Drop uninteresting activities. OneProvenance can also drop activities if they contain uninteresting queries. For instance, filtering out activities that do not contain DDL queries or stored procedure executions is common in OneProvenance deployments.

Filters on event metadata. OneProvenance can also drop events and activities by filtering on event metadata. For instance, the condition `client_app_name='SSMS' or username='sa'` can filter out events for queries coming from `SSMS` or by the system admin `sa`. Furthermore, recall that activities can be modeled as an ordered set of events (Section 3). As such, conditions on event metadata can be used to drop activities as opposed to individual events (e.g., to focus only on activities with long-running queries, we can filter out any activity with `duration` < X seconds for all completed events).

Filters on activities from uninteresting connections. Based on the above filters on event metadata and activities, OneProvenance can filter out an activity if it is coming from an uninteresting connection. A connection is considered uninteresting if (a) it has no interesting queries or (b) has interesting queries, but it does not include any of their last K executions (e.g., last K executions of interesting stored procedure X). Note that this optimization guarantees extraction from at least the last K executions of an interesting query, but OneProvenance can still extract provenance from more than K executions. This is because another interesting query (e.g., stored procedure Y) may be in the connection, rendering the connection interesting (e.g., the execution of X is in the last K executions of Y). Under such a case, OneProvenance can be configured to either consider all queries interesting or keep only the ones matching the last K semantics (e.g., keep Y and filter out X). The former is important to provide execution context (e.g., stored procedure Y was called after executing X), while the latter is preferable for further noise reduction when no such context is necessary. As we will see in the experiments, the overall effect is a substantial noise reduction and a significant improvement in extraction latency.

Drop levels of aggregation. OneProvenance can also be configured to emit provenance at different levels of aggregation to account for different application needs. For instance, an application may not be interested in provenance at the SQL statement level but rather at coarser levels (e.g., stored procedure level), or vice versa, and OneProvenance can emit provenance at the requested level.

Drop events. So far, we have assumed that logs emitted by database engines are complete and filters on events preserve the semantics of Section 3.2. However, for certain workloads (e.g., heavy transactional) database engines emit a significant amount of events. Even though OneProvenance optimizes the extraction process by relying on logs with query text (as opposed to query plans), such heavy workloads can introduce high overhead to both the database and provenance extractor. For such loads, OneProvenance can continue operating by dropping events from the input query log. In our experiments, we will show how event retention options supported by `xEvents` (dropping events based on event buffer availability) can allow OneProvenance to process such high transactional loads.

6 INTEGRATION WITH PURVIEW

Microsoft Purview is a governance platform that allows organizations to govern (e.g., catalog, overview, secure, analyze, and audit) their data estate. To extract metadata and provenance, Purview provides a rich collection of extractors. Each extractor can connect to an underlying data system to extract metadata and provenance. Extraction can be scheduled either as one-off or recurring. The output of extraction is metadata and provenance modeled based on Apache Atlas-based data models. Purview then ingests and stores instantiated data models in its underlying Data Catalog, on top of which it exposes data governance functionalities.

Based on this design, the integration with OneProvenance is straightforward. Recall OneProvenance currently supports extraction from Azure SQL databases. When a customer requires dynamic provenance, Purview sets up an `xEvent` session in the corresponding Azure SQL database, and the database starts emitting `xEvents` in Azure Storage. (Note that for security purposes the blob storage is owned by customers and managed by Purview.) Then, Purview schedules OneProvenance to run periodically (currently, every 6 hours). When executed, OneProvenance analyzes the underlying logs, as discussed in Section 4, and pushes the extracted metadata and provenance to Purview. Regarding optimizations, Purview employs all optimizations discussed in Section 5 to decrease the noise of the extraction as much as possible. (We defer a discussion to [71] on default filter configurations.) Finally, note that users can also alter filters, as we also perform in our experiments.

7 EXPERIMENTS

We now present our thorough evaluation of OnePROVENANCE with the goal to (a) compare OnePROVENANCE with state-of-the-art extraction techniques and (b) demonstrate the benefits of our optimizations on performance improvement and noise reduction.

We begin by briefly describing our experimental setup.

Workloads and databases. For our experiments, we generate workloads using (1) SQL-ProcBench [48], (2) TPC-H, (3) TPC-DS, and (4) TPC-C benchmarks. These workloads provide a mix of both real-world and realistic OLAP and OLTP use cases to help us show the application of OnePROVENANCE and its performance across a wide spectrum of database workloads. Note that Microsoft Purview does not log customer workloads internally—to comply with privacy requirements. Moreover, the extracted provenance is only accessible by authorized customers. Hence, it is infeasible to run experiments using real customer workloads. Through the workloads of our experiments, however, we have reproduced the key insights that we observed in production workloads, drove the design of OnePROVENANCE, and discussed throughout the paper.

SQL-ProcBench is designed using insights derived from an analysis of SQL queries, UDFs, triggers, and stored procedures in 6500 real-world applications [48]. The workload uses the TPC-DS dataset and consists of 63 stored procedures, out of which we selected 35 that can be run multiple times. In our experiments, we generate the TPC-DS database with scale factor set to 1. For a database of this size, we observed an average of 207 SQL statement runs for the selected stored procedures, including (1) 8 statements originating from nested triggers, (2) 48 statements originating from UDFs, (3) 33 total loop iterations, and (4) up to 6 levels of nested dependencies.

TPC-H and TPC-DS are standard benchmarks for performance evaluation of decision support systems. They consist of 22 and 99 ad hoc analytical queries, respectively. For workloads using these queries, we generated the corresponding TPC-H and TPC-DS databases with scale factors 1 and 10. (Insights on TPC-H and TPC-DS are similar for both scale factors. As such, we report results mainly on TPC-H with scale factor 1 to avoid redundant insights.)

TPC-C is a standard benchmark for performance evaluation of OLTP systems. In contrast to prior analytical workloads, which are the traditional focus of dynamic provenance extraction systems, the transactional load of TPC-C serves as a stress test that can help us identify the extent of our coverage over high-load workloads. It involves a mix of 5 concurrent transactions of different types and complexity. We generated a database with 40 warehouses, and observed 109 statement runs per transaction on average. The workload mainly consists of many low-latency simple statements, but also includes `IF` conditions and up to 16 iterations of `WHILE` loops.

Workload Generator. We used HammerDB [50, 51], an open-source benchmarking tool hosted by TPC, to generate and run workloads based on TPC-C, TPC-H, TPC-DS, and SQL-ProcBench. For TPC-C and TPC-H workloads, HammerDB builds a configurable #client threads to concurrently run a configurable #transactions (for TPC-C) or #queries (for TPC-H). Also, we extended HammerDB to generate workloads from SQL-ProcBench and TPC-DS, with the same configurations (i.e., #client threads, #transactions, #queries).

Platform. We run all workloads against a serverless Azure SQL DB instance [79], with 8 cores and 24 GB memory, which emits logs to an Azure Storage account. HammerDB and OnePROVENANCE are installed on a Standard D8s v3 Azure VM (8 cores, 32 GB memory). All resources and services are deployed in the same Azure region.

Outline. We start our discussion by breaking down the performance of OnePROVENANCE in comparison with state-of-the-art prior work (Section 7.1), followed by experiments highlighting the benefits of our optimizations (Section 7.2), and concluding with a discussion comparing OnePROVENANCE with SAC [81], Spline [75], and the OpenLineage Spark extractor [36] (Section 7.3). Settings and compared techniques are outlined inline, per experiment.

7.1 OnePROVENANCE Performance Breakdown

We start our experiments with a performance breakdown of the different extraction components in the OnePROVENANCE architecture.

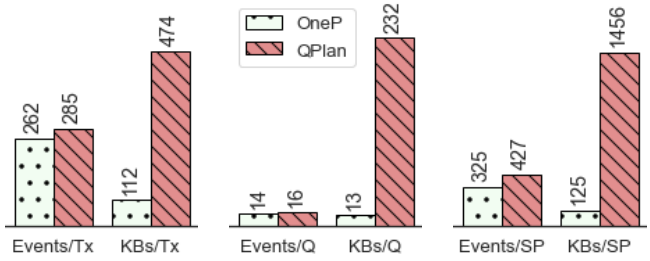
Note that our goal is to compare OnePROVENANCE against state-of-the-art techniques for the provenance capture problem of our focus. Such techniques are employed by dynamic provenance extraction systems (e.g., SAC [81], Spline [75], and OpenLineage for Spark [36]). These systems, however, target Spark as their source of query event logs. In contrast, OnePROVENANCE targets Azure SQL DB. Differences in logging by Spark and Azure SQL DB, and overall network topology of systems (e.g., SAC and Spline run in the master node of Spark, OpenLineage uses Azure functions, and OnePROVENANCE runs outside of Azure SQL DB) are the main reasons why performance comparisons do not reveal meaningful insights on the core provenance capture problem of our focus.

At their core, however, these systems operate on query plans. As such, we alter OnePROVENANCE to operate on plans to mimic the behavior of these extractors in Azure SQL DB, and perform meaningful comparisons. We denote this system (OnePROVENANCE with physical plans as input) as `QPlan` and compare it with OnePROVENANCE. (A discussion on SAC, Spline, and OpenLineage is included in Section 7.3.) To better understand overheads on source databases, we also compare OnePROVENANCE and `QPlan` with query execution without provenance capture (denoted as `Baseline`).

7.1.1 Logging. The first major difference between OnePROVENANCE and prior work is on logging events. Prior work uses events with query plans, whereas OnePROVENANCE uses only events with query text. Hence, next, we compare the database overheads between logging query plans (prior work) and text (OnePROVENANCE).

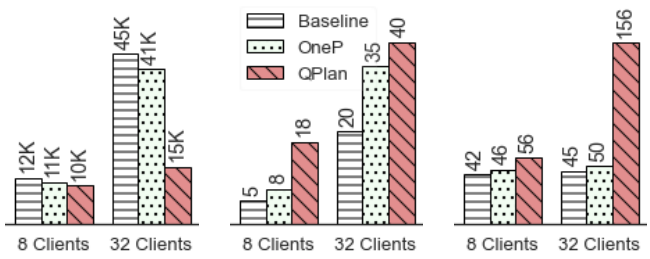
In comparison. More specifically, recall from Section 3.2 that Azure SQL DB provides a configurable query log architecture that allows us to log events of different types. OnePROVENANCE logs `started` and `completed` events for executions of `sql_statement`, `sp_statement`, and `sql_batch`. To mimic the behavior of state-of-the-art dynamic provenance extractors that rely their extraction on plans, we introduce QPlan that logs the `query_post_execution_plan_profile` event type that carries query plans. Note that QPlan logs the `query_post_execution_plan_profile` event type on top of the rest of the event types because the query text and runtime metadata (that are carried only in the rest event types) need to be in the output provenance models (of both ours and prior work) anyways. To compute the overhead of each technique, we use the performance of the database with logging turned off as `Baseline`.

Metrics. We compare OnePROVENANCE and QPlan on logging based on their I/O requirements and database overheads. For the



(a) TPC-C (b) TPC-H (c) SQL-ProcBench

Figure 12: Characteristics of query text (OneP) vs. query plan (`QPlan`) logging across different workloads. Characteristics include #events and required storage (in KBs) per transaction, query, or stored procedure. The results show that `QPlan` events are significantly larger.



(a) Avg. TPM (b) Avg. DB CPU load (c) Avg. Tx Latency

Figure 13: Comparison of Tx/min (TPM), CPU load, and Tx latency overheads of query text (OneP) vs. query plan (QPlan) logging over no logging (Baseline) for 8 and 32 clients running TPC-C workloads.

former, we report the size of logs (in KBs) and number of events per granularity of interest (i.e., KBs and #Events per transaction for TPC-C, query for TPC-H, and stored procedure for SQL-ProcBench). For the latter, we rely on transaction rate (TPM) metric, average execution latency (in ms), and database CPU utilization (in % used).

Figure 12 shows the results of our comparison on I/O requirements for the different workloads of our experiments.

I/O requirements. Across workloads, we observe that QPlan logs a few more events per granularity of interest. For instance, TPC-C transactions run 109 SQL statements on average—hence, the database emits at least 218 (started, completed) event pairs. QPlan captures 263 events due to the extra logging of `query_post_execution_plan_profile`. As query plans are intrinsically large, `QPlan` logs require up to ~18X larger storage compared to OnePROVENANCE logs. This means query plan logging exhibits a very high demand for critical event buffer and I/O resources, that put prohibitive pressure on the database engine and event storage (esp. for high-load workloads). (Note that allocating a large event buffer is not recommended [25], and, indeed, in our experiments, increasing the buffer did not yield better results for plan-based logging.)

Next, we analyze the overheads of logging on database performance (Figure 13) while increasing the client threads to collect additional data points. For this experiment, we focus only on TPC-C: due to the high-load and low-latency requirements of transactional workloads we consider TPC-C a stress test in our setup.

Impact of logging on database performance. Figures 13a to 13c show that logging query plans (QPlan) leads to significantly lower TPM (up to 3× lower), higher CPU utilization (up to 2× increase), and slower transaction execution (up to 3× increase in transaction execution latency). Finally, Figures 13a to 13c also highlight that the performance numbers (TPM, CPU, transaction latency) of OnePROVENANCE are negligible wrt. those of the baseline, indicating that OnePROVENANCE can still operate on such transactional workloads without regressing the database performance considerably.

Takeaways: Our results highlight that relying on query text, as opposed to query plan, significantly improves logging performance (18× less storage) and database overheads (negligible overheads, avoiding 2-3× lower TPM, slower transaction latency, and higher CPU of query plans). We conclude that provenance extractors relying on query text are low-cost and more practical solutions.

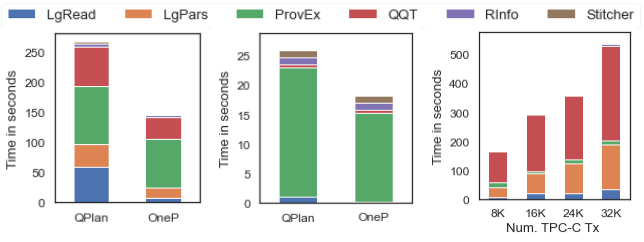
7.1.2 OnePROVENANCE Components. We now focus on the performance evaluation of the main OnePROVENANCE components (both in isolation and end-to-end). Our analysis focuses on understanding the performance of (1) OnePROVENANCE and its components, (2) OnePROVENANCE compared to QPlan on provenance extraction, and (3) aggregating provenance information.

Benchmarks. For this analysis, we used HammerDB to run 25 iterations of TPC-H and SQL-ProcBench query sets. These workloads can better reveal the performance of OnePROVENANCE components and the overheads of QPlan, since they contain complex analytical queries. Furthermore, we run the TPC-C workload with #transactions ranging between 8K and 32K. With this workload, we aim to test how OnePROVENANCE scales as query loads increase.

In comparison. In this experiment, we run OnePROVENANCE with two optimizations on. More specifically, we enable the filtering out of activities from uninteresting connections and loop compression optimizations. For both, we enable their most aggressive filtering out options (i.e., admit only activities with the latest runs of SPs and last loop iterations). These optimizations and their configurations are on by default in production deployments and, as such, better reflect OnePROVENANCE's performance. (We discuss the performance of OnePROVENANCE with and without these optimizations in Section 7.2.) Finally, we compare OnePROVENANCE with QPlan. Note that, to gain meaningful insights on overheads of processing query plans, we enable the same set of optimizations for QPlan.

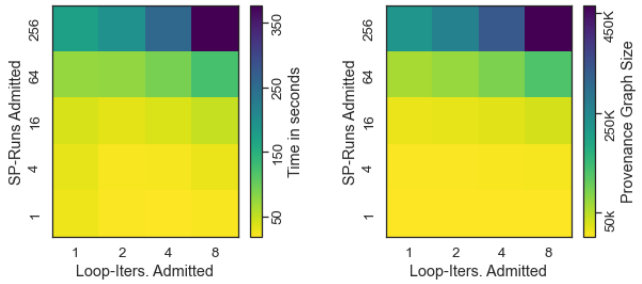
Components. We break down the latency of OnePROVENANCE per component: (1) `ProvEx` (Provenance Extractor), (2) Stitcher, and (3) RInfo (Runtime Information Extractor). For Activity Collector, we drill down into its sub-components for more meaningful insights: (4) `LgRead` (download logs from event storage), (5) `LgPars` (deserialize events from logs), and (6) `QQT` (analyze events and build QQTrees).

Query plan processing overheads. As seen in Figure 14a for SQL-ProcBench, processing plans (QPlan) is 2× slower compared to OnePROVENANCE's processing query text. This is because `LgRead`, `LgPars`, and `QQT` for QPlan have to process 10× additional log bytes—dominating the end-to-end latency. Extraction from TPC-H logs witnesses similar overheads, shown in Figure 14b. Recall that the TPC-H query logs are rather small (e.g., logs corresponding to 100 iterations of the 22 queries are less than 30 MB). Consequently, `LgRead`, `LgPars`, and `QQT` (Section 4.4) terminate rather quickly. Finally, for both workloads `ProvEx` has sizeable latency requirements;



(a) SQL-ProcBench (b) TPC-H (c) TPC-C

Figure 14: Latency (end-to-end and by component; measured in seconds) of OneProvenance and QPlan on (a) SQL-ProcBench and (b) TPC-H, and OneProvenance latency while increasing #Tx in (c) TPC-C. Main results include: 1) processing query plans (QPlan) is costlier than query text (OneProvenance), 2) latency of aggregating provenance is negligible compared to other components, and 3) latency of OneProvenance increases linearly with query load.



(a) Total Execution Time (b) Provenance Graph Size

Figure 15: Time taken and output size for processing logs of 4K TPC-C Txs, while varying admissibility settings of SP-Runs Admitted and Loop-Iters Admitted. The most admissible setting (256,8) is ~20× slower than (1,1) and the generated graph is ~450× larger.

this is expected since TPC-H and SQL-ProcBench contain complex analytical queries. Interestingly, we observed that the latency of ProvEx for QPlan is a bit higher than the one of OneProvenance primarily due to the difference in the sizes of plans and query texts.

Provenance aggregation. As also seen in Figure 14a, the latency required for aggregating provenance information (Stitcher) is negligible in comparison to the ones required by other components. As such, OneProvenance addresses the limitations **L2-3** of prior work originating from lack of aggregations (as discussed in Section 1) by incurring a negligible overhead in provenance extraction.

Stress test-Latency under query load increase. Figure 14c illustrates that OneProvenance remains stable when subjected to higher loads, and the end-to-end latency increases linearly with the load. (QPlan incurs prohibitive costs for TPC-C as we discussed in Section 7.1.1 and we omit its performance). Furthermore, an interesting insight from this experiment is that the extraction engine spends the majority of its time in LgPars and QQT tasks. This is because TPC-C consists of many low latency queries which are executed repeatedly, thousands of times. In such a workload, our default optimizations manage to filter out repetitive queries. These optimizations are applied right after QQT (i.e., after the Activity Collector)—leading to reduced load for ProvEx, RInfo, and Stitcher, and explaining why LgPars and QQT dominate the latency.

Takeaways: Overall, our results highlight (1) the importance of avoiding query plans for provenance extraction, (2) that aggregating provenance has negligible overhead, and (3) OneProvenance is an efficient provenance extraction system to the extent that it can support even considerably high transactional workloads.

7.2 Optimizations

As discussed in Section 5, filtering techniques we introduced in OneProvenance can speed up the processing of query logs, and reduce the noise of traditional dynamic provenance extraction systems. We evaluate the strengths of these techniques next.

Benchmark. For these experiments, we run TPC-C for a total of 4K TPC-C transactions (16 clients), of which new_order and payment transactions are invoked 1707 and 1738 times, respectively. The database emits ~250 events/transaction (~1M events overall).

Optimizations. We compare OneProvenance with and without optimizations. Our comparisons are one optimization at a time to better understand the value of the corresponding optimization. More specifically, recall that the TPC-C statements are executed 100s of times, and some of the executions result in 100s of loop iterations. As such, (1) loop compression, (2) dropping uninteresting activities, and (3) filters on activities from uninteresting connections are valuable. We denote the loop compression optimization as **Loop-Iters Admitted** and vary the #admitted iterations. The other two filters (i.e., uninteresting activities and filters on activities from uninteresting connections) interoperate in OneProvenance, as we discussed in Section 5. Thus, we group them, denote the technique as **SP-Runs Admitted**, and vary how many of the latest runs are admitted (per the last-K semantics of Section 5). Finally, we aim to see the effect of **dropping levels of aggregation** and **events**. (We omit experiments filtering on event metadata and queries for brevity, noting that their impact varies based on their selectivity.)

Metrics. We evaluate OneProvenance with and without optimizations based on end-to-end latency and size of provenance graphs emitted. We define the graph size to be the total #nodes and #edges.

Loop-Iters Admitted and SP-Runs Admitted. The results of our evaluation for these two optimizations are shown in Figure 15. The most aggressive setting is (1,1), bottom-left corner tile in Figure 15, which prompts **SP-Runs Admitted** to remove all but the latest run of the 5 TPC-C SPs, and **Loop-Iters Admitted** to identify unique control flows in an activity and retain the latest iteration. For instance, with the setting (1,1), 1706 (out of 1707) QQTrees corresponding to new_order runs would be dropped, and the latest QQTree would have 75% fewer SQL statements. Consequently, Provenance Extractor analyzes only 191 SQL statements, thereby reducing end-to-end latency. Setting the optimization to (256, 8), shown in the top-right corner tile in Figure 15a, means an admission of at most 256 runs of SPs and at most 8 loop iterations. This leads to 110K SQL statements being analyzed by Provenance Extractor, which results in a ~20× slowdown over processing 191 statements.

Finally, the top-right tile in Figure 15b corresponds to the worst-case scenario for the provenance graph sizes (i.e., the output of OneProvenance without optimizations). In absolute terms, the output has more than 450K nodes and edges. The large size of the

graph is due to modeling of recurring SPs and SQL statement runs. Such a large graph can be overwhelming for upstream applications. In contrast, the aggressive optimization setting (1,1) leads to a concise, noise-free output model with just over 1K nodes and edges.

Drop levels of aggregation. Using this optimization, we configure OnеProvenance to report provenance only at aggregate levels (i.e., at stored procedures), and we drop statement runs and their provenance for noise reduction purposes. This leads to an ~80% drop over the provenance graph size of OnеProvenance without the optimization. Note that this optimization is applied in Stitcher (i.e., we do not push down aggregation to earlier components because the provenance of individual statements needs to be extracted anyway) and, as such, does not improve end-to-end latency.

`Drop Events`. Recall that this optimization drops events based on database event buffer availability, leading to extracting reduced provenance graphs. OnеProvenance with this optimization drops only 3% of #nodes and #edges for the TPC-C workload (over OnеProvenance without the optimization). To better understand this result, we also run QPlan with `Drop Events` on. The drop was ~50%—further highlighting the prohibitive overheads of query plans and OnеProvenance's ability to process higher TPM workloads.

Provenance querying. Finally, note that our experiments focus on the problem of provenance capture of Section 3.3. We omit analysis on provenance querying since OnеProvenance is external to the data catalog that serves provenance queries. We note, however, that in production workloads, we have not observed non-interactive query response times (>.5s) over OnеProvenance's output.

Takeaways: Our results demonstrate that our application-aware optimizations lead to substantial noise reduction over provenance graphs (more than two orders of magnitude graph size reduction) and improve extraction performance significantly (~20× speedup).

7.3 Discussion on other systems

We conclude our discussion with a comparison between OnеProvenance and SAC [81], Spline [75], and OpenLineage for Spark [36]. For these experiments, we used the TPC-DS benchmark (25 iterations of 99 queries) since these systems target analytical workloads, and we focus our analysis only on the output graph. (As discussed in Section 7.1, performance comparisons are not meaningful. This is why we introduced QPlan in Section 7.1, to compare OnеProvenance with state-of-the-art techniques in a principled way.)

Overall, OnеProvenance supports provenance aggregation, query runtime metadata (CPU and #records), user and client connection details, inference of static queries, and column-level provenance that SAC, Spline, and OpenLineage for Spark lack. In contrast, these systems embed Spark plans in their output model that OnеProvenance does not. Such plans can be large in size. As such, while OnеProvenance outputs more metadata, its output is often on par or smaller than the one of such systems. Furthermore, customer feedback highlighted a disconnect: data governance teams that consume the output of provenance extractors are not DBAs to understand or make use of such plans. Finally, we note that our optimizations further reduce the output size in contrast to SAC, Spline, and OpenLineage which do not provide noise reduction optimizations during extraction (e.g., in our TPC-DS workload, the output of OnеProvenance is ~10× smaller compared to SAC).

8 RELATED WORK

We describe related work in the areas of metadata and provenance management. To highlight the importance of these domains, we note that several techniques and applications of metadata and provenance management have been included in many high-profile products and open source systems. Adaptive [5], Alation [6], Acryl Data [4], Alex [7], ASG [10], Collibra [24], Data Advantage Group [29], Datakin [32], data.world [33], Amazon [12], erwin [37], Global IDs [45], Google [42], IBM [58], Informatica [57], Microsoft [73], Precisely [70], Semantic Web Company [76], Smartlogic [78], and Syniti [80] are only a few companies with offerings that include metadata and provenance capabilities. Open source systems in the same domains include Apache Atlas [11], Egeria [35], OpenLineage [36], DataHub [31], and Spline [75], among others.

Metadata management is a sub-field of data management with over five decades of research and practice [15, 65]. Problems of focus in this space include discovery [34, 40, 63], classification [19, 66], extraction [39, 86], or storage and querying of metadata [49, 53, 72, 82]. Furthermore, metadata management systems are central for many metadata-driven applications and problems, including: data integration and exchange [9, 20, 30, 38, 52], schema (and general metadata) evolution [27, 28], lifecycle management and versioning [17, 18, 62], profiling [2, 3], data cleansing [39, 67, 85], reproducibility [21, 74], enterprise search [59], and auditing [22, 43]. Our work is closely related and largely orthogonal to these lines of work. In particular, our proposed techniques can extract a rich provenance model with a multitude of metadata on queries, query runs, client connections, and datasets (e.g., tables or outputs of ad hoc queries). With such rich information, we can better assist metadata management applications to better drive their logic.

Provenance management is a subfield of data and metadata management with a focus on capturing, modeling, and querying the connections between input and output data elements across a workflow. Traditionally, in the context of databases, provenance is classified into coarse-grained [8, 23, 54, 55, 60, 62, 69, 75, 81] and fine-grained [16, 23, 26, 44, 47, 54, 56, 68, 72, 82, 83]. The latter encodes the relationships between input and output records or cells, while the former focuses on modeling relationships at a coarse level (e.g., tables and columns). Our proposal is related to capturing dynamic coarse-grained provenance information in the context of database systems. In contrast to prior work, however, we highlighted unique challenges and proposed corresponding techniques to extract semantically rich provenance information from event logs efficiently.

9 CONCLUSION

In this paper, we presented OnеProvenance, our provenance extraction engine over database logs that currently powers dynamic provenance extraction in Microsoft Purview. OnеProvenance improves over prior work by processing database query execution event logs carrying query text, aggregating provenance information, and filtering noise during extraction. We believe our work is a step towards optimized provenance extraction systems, and a pointer towards important future work (e.g., introduce more complicated filtering techniques, extract provenance by processing logs in a distributed fashion to cope with even higher loads, or push down application logic during dynamic provenance extraction).

REFERENCES

[1] Daniel Abadi, Anastasia Ailamaki, David Andersen, Peter Bailis, Magdalena Balazinska, Philip A. Bernstein, Peter Boncz, Surajit Chaudhuri, Alvin Cheung, Anhai Doan, Luna Dong, Michael J. Franklin, Juliana Freire, Alon Halevy, Joseph M. Hellerstein, Stratos Idreos, Donald Kossmann, Tim Kraska, Sailesh Krishnamurthy, Volker Markl, Sergey Melnik, Tova Milo, C. Mohan, Thomas Neumann, Beng Chin Ooi, Fatma Ozcan, Jignesh Patel, Andrew Pavlo, Raluca Popa, Raghu Ramakrishnan, Christopher Re, Michael Stonebraker, and Dan Suciu. 2022. The Seattle Report on Database Research. *Commun. ACM* 65, 8 (jul 2022), 72–79. https://doi.org/10.1145/3524284

[2] Ziawasch Abedjan, Lukasz Golab, and Felix Naumann. 2015. Profiling relational data: a survey. *VLDB J.* 24, 4 (2015), 557–581. https://doi.org/10.1007/s00778-015-0389-y

[3] Ziawasch Abedjan, Toni Grütze, Anja Jentzsch, and Felix Naumann. 2014. Profiling and mining RDF data with ProLOD++. In *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, Isabel F. Cruz, Elena Ferrari, Yufei Tao, Elisa Bertino, and Goce Trajcevski (Eds.). IEEE Computer Society, 1198–1201. https://doi.org/10.1109/ICDE.2014.6816740

[4] acryldata 2023. Acryl Data. https://www.acryldata.io/.

[5] adaptive 2022. Adaptive. https://adaptive.com.

[6] alation 2022. Alation. https://alation.com.

[7] alex 2022. Alex Solutions. https://alexsolutions.com.au.

[8] Yael Amsterdamer, Susan B Davidson, Daniel Deutch, Tova Milo, Julia Stoyanovich, and Val Tannen. 2011. Putting lipstick on pig: Enabling database-style workflow provenance. *arXiv preprint arXiv:1201.0231* (2011).

[9] Angelos Christos Anadiotis, Oana Balalau, Catarina Conceição, Helena Galhardas, Mhd Yamen Haddad, Ioana Manolescu, Tayeb Merabti, and Jingmao You. 2022. Graph integration of structured, semistructured and unstructured data for data journalism. *Information Systems* 104 (2022), 101846. https://doi.org/10.1016/j.is.2021.101846

[10] asg 2022. ASG. https://www.asg.com.

[11] atlas 2019. Apache Atlas - Type System. https://atlas.apache.org/#/TypeSystem.

[12] awsdatazone 2022. AWS DataZone. https://aws.amazon.com/datazone/.

[13] azurestorage 2022. Azure Storage. https://azure.microsoft.com/services/storage.

[14] P.A. Bernstein, D.W. Shipman, and W.S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering* SE-5, 3 (1979), 203–216.

[15] Phillip A. Bernstein, Alon Y. Halevy, and Rachel A. Pottinger. 2000. A Vision for Management of Complex Models. *SIGMOD Rec.* 29, 4 (Dec. 2000), 55–63. https://doi.org/10.1145/369275.369289

[16] Deepavali Bhagwat, Laura Chiticariu, Wang Chiew Tan, and Gaurav Vijayvargiya. 2004. An Annotation Management System for Relational Databases. In *VLDB*. 900–911.

[17] Anant P. Bhardwaj, Souvik Bhattacherjee, Amit Chavan, Amol Deshpande, Aaron J. Elmore, Samuel Madden, and Aditya G. Parameswaran. 2015. DataHub: Collaborative Data Science & Dataset Version Management at Scale. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper18.pdf

[18] Souvik Bhattacherjee, Amit Chavan, Silu Huang, Amol Deshpande, and Aditya Parameswaran. 2015. Principles of Dataset Versioning: Exploring the Recreation/Storage Tradeoff. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1346–1357. https://doi.org/10.14778/2824032.2824035

[19] Will Brackenbury, Rui Liu, Mainack Mondal, Aaron J. Elmore, Blase Ur, Kyle Chard, and Michael J. Franklin. 2018. Draining the Data Swamp: A Similarity-Based Approach. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (Houston, TX, USA) *(HILDA'18)*. Association for Computing Machinery, New York, NY, USA, Article 13, 7 pages. https://doi.org/10.1145/3209900.3209911

[20] Michael J. Cafarella, Alon Halevy, and Nodira Khoussainova. 2009. Data Integration for the Relational Web. *Proc. VLDB Endow.* 2, 1 (aug 2009), 1090–1101. https://doi.org/10.14778/1687627.1687750

[21] Steven P. Callahan, Juliana Freire, Emanuele Santos, Carlos E. Scheidegger, Cláudio T. Silva, and Huy T. Vo. 2006. VisTrails: Visualization Meets Data Management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data* (Chicago, IL, USA) *(SIGMOD '06)*. Association for Computing Machinery, New York, NY, USA, 745–747. https://doi.org/10.1145/1142473.1142574

[22] ccpa 2022. California Consumer Privacy Act (CCPA). https://oag.ca.gov/privacy/ccpa.

[23] James Cheney, Laura Chiticariu, and Wang Chiew Tan. 2009. Provenance in databases: Why, how, and where. *Foundations and Trends® in Databases* 1, 4 (2009), 379–474.

[24] colibra 2022. Colibra. https://colibra.com.

[25] createeventsession 2022. Create Event Session. https://docs.microsoft.com/en-us/sql/t-sql/statements/create-event-session-transact-sql?view=sql-server-ver15.

[26] Yingwei Cui. 2001. *Lineage tracing in data warehouses.* Ph.D. Dissertation. Stanford University.

[27] Carlo Curino, Hyun Jin Moon, Letizia Tanca, and Carlo Zaniolo. 2008. Schema Evolution in Wikipedia - Toward a Web Information System Benchmark. In *ICEIS 2008 - Proceedings of the Tenth International Conference on Enterprise Information Systems, Volume DISI, Barcelona, Spain, June 12-16, 2008*, José Cordeiro and Joaquim Filipe (Eds.). 323–332.

[28] Carlo A. Curino, Hyun J. Moon, and Carlo Zaniolo. 2008. Graceful Database Schema Evolution: The PRISM Workbench. *Proc. VLDB Endow.* 1, 1 (aug 2008), 761–772. https://doi.org/10.14778/1453856.1453939

[29] dag 2022. Data Advantage Group. https://www.dag.com.

[30] Anish Das Sarma, Xin Dong, and Alon Halevy. 2008. Bootstrapping Pay-as-You-Go Data Integration Systems. 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, 861–874. https://doi.org/10.1145/1376616.1376702

[31] datahub 2023. DataHub. https://datahubproject.io/.

[32] datakin 2022. Datakin. https://datakin.com.

[33] dataworld 2022. data.world. https://data.world.

[34] Dong Deng, Raul Castro Fernandez, Ziawasch Abedjan, Sibo Wang, Michael Stonebraker, Ahmed K. Elmagarmid, Ihab F. Ilyas, Samuel Madden, Mourad Ouzzani, and Nan Tang. 2017. The Data Civilizer System. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p44-deng-cidr17.pdf

[35] egeria-lineage 2022. Egeria - Lineage Management. https://egeria-project.org/features/lineage-management/overview/#lineage-styles.

[36] egeria-lineage 2022. OpenLineage - Lineage Management. https://openlineage.io/.

[37] erwin 2022. erwin. https://www.erwin.com.

[38] Ronald Fagin, Phokion G. Kolaitis, Renée J. Miller, and Lucian Popa. 2005. Data exchange: semantics and query answering. *Theor. Comput. Sci.* 336, 1 (2005), 89–124. https://doi.org/10.1016/j.tcs.2004.10.033

[39] Mina Farid, Alexandra Roatis, Ihab F. Ilyas, Hella-Franziska Hoffmann, and Xu Chu. 2016. CLAMS: Bringing Quality to Data Lakes. 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, 2089–2092. https://doi.org/10.1145/2882903.2899391

[40] Raul Castro Fernandez, Ziawasch Abedjan, Famien Koko, Gina Yuan, Samuel Madden, and Michael Stonebraker. 2018. Aurum: A Data Discovery System. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. IEEE Computer Society, 1001–1012. https://doi.org/10.1109/ICDE.2018.00094

[41] gartner-governance 2020. Gartner Report on Metadata Management Solutions. https://www.gartner.com/en/documents/3993025.

[42] gdc 2022. Google Data Catalog. https://cloud.google.com/data-catalog.

[43] gdpr 2022. General Data Protection Regulation (EU GDPR). https://gdpr-info.eu.

[44] Boris Glavic and Gustavo Alonso. 2009. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*.

[45] globalids 2022. Global IDs. https://www.globalids.com.

[46] Achim Granzen. 2021. Data Governance Solutions. https://www.forrester.com/report/the-forrester-wave-tm-data-governance-solutions-q3-2021/RES161533.

[47] Todd J. Green, Grigoris Karvounarakis, Zachary G. Ives, and Val Tannen. 2007. Update Exchange with Mappings and Provenance. In *VLDB*.

[48] Surabhi Gupta and Karthik Ramachandra. 2021. Procedural Extensions of SQL: Understanding Their Usage in the Wild. *Proc. VLDB Endow.* 14, 8 (apr 2021), 1378–1391. https://doi.org/10.14778/3457390.3457402

[49] Alon Halevy, Flip Korn, Natalya F. Noy, Christopher Olston, Neoklis Polyzotis, Sudip Roy, and Steven Euijong Whang. 2016. Goods: Organizing Google's Datasets. 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, 795–806. https://doi.org/10.1145/2882903.2903730

[50] hammerdb 2022. HammerDB. https://www.hammerdb.com.

[51] hammerdbgithub 2022. HammerDB on GitHub. https://github.com/TPC-Council/HammerDB.

[52] Dennis Heimbigner and Dennis McLeod. 1985. A Federated Architecture for Information Management. *ACM Trans. Inf. Syst.* 3, 3 (jul 1985), 253–278. https://doi.org/10.1145/4229.4233

[53] Joseph M. Hellerstein, Vikram Sreekanti, Joseph E. Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, Mark Donsky, Gabriel Fierro, Chang She, Carl Steinbach, Venkat Subramanian, and Eric Sun. 2017. Ground: A Data Context Service. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017,*

Chaminade, CA, USA, January 8-11, 2017, Online Proceedings. www.cidrdb.org. http://cidrdb.org/cidr2017/papers/p111-hellerstein-cidr17.pdf
[54] Melanie Herschel, Ralf Diestelkämper, and Houssem Ben Lahmar. 2017. A survey on provenance: What for? What form? What from? *The VLDB Journal* 26, 6 (2017), 881–906.
[55] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R Pocock, Peter Li, and Tom Oinn. 2006. Taverna: a tool for building and running workflows of services. *Nucleic acids research* 34, suppl_2 (2006), W729–W732.
[56] Robert Ikeda. 2012. *Provenance In Data-Oriented Workflows*. Ph.D. Dissertation. Stanford University.
[57] informatica 2022. Informatica. https://www.informatica.com.
[58] infosphere 2022. IBM Infosphere. https://www.ibm.com/analytics/information-server.
[59] Udo Kruschwitz, Charlie Hull, et al. 2017. *Searching the enterprise*. Vol. 11. Now Publishers.
[60] Bertram Ludäscher, Ilkay Altintas, Chad Berkley, Dan Higgins, Efrat Jaeger, Matthew Jones, Edward A Lee, Jing Tao, and Yang Zhao. 2006. Scientific workflow management and the Kepler system. *Concurrency and computation: Practice and experience* 18, 10 (2006), 1039–1065.
[61] Ruslan Mavlyutov, Carlo Curino, Boris Asipov, and Philippe Cudré-Mauroux. 2017. Dependency-Driven Analytics: A Compass for Uncharted Data Oceans.. In *CIDR*.
[62] Hui Miao, Amit Chavan, and Amol Deshpande. 2017. ProvDB: Lifecycle Management of Collaborative Analysis Workflows. In *Proceedings of the 2nd Workshop on Human-In-the-Loop Data Analytics* (Chicago, IL, USA) *(HILDA'17)*. Association for Computing Machinery, New York, NY, USA, Article 7, 6 pages. https://doi.org/10.1145/3077257.3077267
[63] Renée J. Miller, Fatemeh Nargesian, Erkang Zhu, Christina Christodoulakis, Ken Q. Pu, and Periklis Andritsos. 2018. Making Open Data Transparent: Data Discovery on Open Data. *IEEE Data Eng. Bull.* 41, 2 (2018), 59–70. http://sites.computer.org/debull/A18june/p59.pdf
[64] Paolo Missier, Khalid Belhajjame, and James Cheney. 2013. The W3C PROV family of specifications for modelling provenance metadata. In *EDBT '13*. 773–776.
[65] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. 2019. Data Lake Management: Challenges and Opportunities. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 1986–1989. https://doi.org/10.14778/3352063.3352116
[66] Fatemeh Nargesian, Erkang Zhu, Ken Q. Pu, and Renée J. Miller. 2018. Table Union Search on Open Data. *Proc. VLDB Endow.* 11, 7 (mar 2018), 813–825. https://doi.org/10.14778/3192965.3192973
[67] Felix Naumann and Melanie Herschel. 2010. *An Introduction to Duplicate Detection*. Morgan and Claypool Publishers.
[68] Xing Niu, Raghav Kapoor, Boris Glavic, Dieter Gawlick, Zhen Hua Liu, Vasudha Krishnaswamy, and Venkatesh Radhakrishnan. 2017. Provenance-aware Query Optimization. In *ICDE*.
[69] Christopher Olston and Anish Das Sarma. 2011. Ibis: A Provenance Manager for Multi-Layer Systems.. In *CIDR*. 152–159.
[70] precisely 2022. Precisely. https://www.precisely.com.
[71] Fotis Psallidas, Ashvin Agrawal, Chandru Sugunan, Khaled Ibrahim, Konstantinos Karanasos, Jesús Camacho-Rodríguez, Avrilia Floratou, Carlo Curino, and Raghu Ramakrishnan. 2022. OneProvenance: Efficient Extraction of Dynamic Coarse-Grained Provenance from Database Logs [Technical Report]. *arXiv preprint arXiv:2210.14047* (2022).
[72] Fotis Psallidas and Eugene Wu. 2018. Smoke: Fine-grained lineage at interactive speed. *arXiv preprint arXiv:1801.07237* (2018).
[73] purview 2022. Microsoft Purview. https://azure.microsoft.com/en-us/services/purview.
[74] 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. https://doi.org/10.14778/3415478.3415556
[75] Jan Scherbaum, Marek Novotny, and Oleksandr Vayda. 2018. Spline: Spark lineage, not only for the banking industry. In *2018 IEEE International Conference on Big Data and Smart Computing (BigComp)*. IEEE, 495–498.
[76] semanticweb 2022. Semantic Web Company. https://semantic-web.com.
[77] Ben Shneiderman. 1996. The eyes have it: A task by data type taxonomy for information visualizations. In *Symposium on Visual Languages*. 336–343.
[78] smartlogic 2022. Smartlogic. https://www.smartlogic.com.
[79] sqldb 2022. Azure SQL Managed Instance. https://azure.microsoft.com/en-us/products/azure-sql/managed-instance.
[80] syniti 2022. Syniti. https://www.syniti.com.
[81] 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. https://doi.org/10.1109/ICDE.2019.00215
[82] Jennifer Widom. 2005. Trio: a system for integrated management of data, accuracy, and lineage. In *CIDR*.
[83] Allison Woodruff and Michael Stonebraker. 1997. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*.
[84] xevents 2019. Extended Events Overview. https://docs.microsoft.com/en-us/sql/relational-databases/extended-events/extended-events.
[85] Mohamed Yakout, Ahmed K. Elmagarmid, Jennifer Neville, Mourad Ouzzani, and Ihab F. Ilyas. 2011. Guided Data Repair. *Proc. VLDB Endow.* 4, 5 (feb 2011), 279–289. https://doi.org/10.14778/1952376.1952378
[86] Erkang Zhu, Fatemeh Nargesian, Ken Q. Pu, and Renée J. Miller. 2016. LSH Ensemble: Internet-Scale Domain Search. *Proc. VLDB Endow.* 9, 12 (aug 2016), 1185–1196. https://doi.org/10.14778/2994509.2994534