RESEARCH ARTICLE

*International Journal of Applied Machine Learning and Computational Intelligence*

# Tracing Coarse-Grained and Fine-Grained Data Lineage in Data Lakes: Automated Capture, Modeling, Storage, and Visualization

**Shinoy Vengaramkode Bhaskaran**

Senior Manager, Data Engineering, LogMeIn Inc..

Copyright©*2021, by Neuralslate*

*Accepted: 2021-12-05*

*Published: 2021-12-13*

**Abstract**

In contemporary data-driven organizations, data lakes have emerged as large-scale, flexible repositories that integrate heterogeneous data sources—ranging from raw transactional logs to refined analytical tables. Becuase these ecosystems grow in complexity, understanding data lineage, i.e., the end-to-end provenance and transformations that data undergo, is necessary for ensuring data quality, regulatory compliance, and stakeholder trust. This paper offers a comprehensive, technical overview of approaches to capturing, modeling, storing, and visualizing data lineage in modern data lakes, with an emphasis on distinguishing coarse-grained lineage (dataset-level traces) from fine-grained lineage (record- or cell-level provenance). We begin by examining various automated lineage capture techniques, including instrumentation of ETL and data pipeline frameworks, logical query parsing, and runtime provenance tagging. Every technique we discussed involves trade-offs in performance, accuracy, and integration complexity. We then describe strategies for modeling lineage at multiple levels of abstraction, from high-level DAG-based dependencies across datasets to detailed provenance graphs for individual records. Scalability challenges arise in storing and querying lineage at fine granularity, prompting solutions such as compression, hierarchical aggregation, and delta-based referencing. We subsequently explore state-of-the-art visualization and interaction methodologies, discussing how intuitive graph-based dashboards, hierarchical drill-down views, and interactive queries aid in quickly locating root causes of data issues, assessing impact on downstream artifacts, and supporting reproducibility.

**Keywords:** Data governance; Data lakes; Data lineage; Data provenance; ETL frameworks; Provenance visualization; Scalability challenges

## 1 Introduction

Modern data-driven organizations increasingly rely on heterogeneous data sources and large-scale processing pipelines to derive value from their data assets [1]. As the volume, variety, and velocity of data grow, data lakes have emerged as a common architectural solution, providing a flexible and scalable repository for raw and transformed datasets. Data lakes often serve as a foundation for machine learning

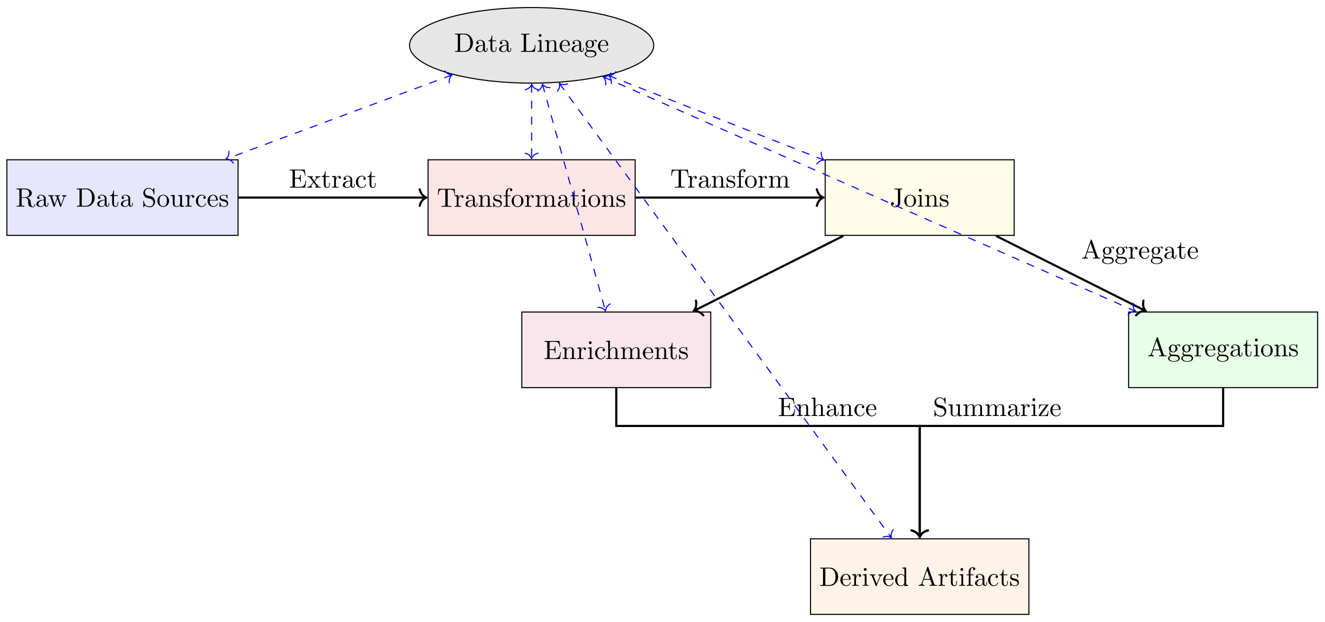

**Figure 1** Data Lineage Diagram: Visualizing data transformations, joins, aggregations, and enrichments leading to derived artifacts, with data lineage ensuring traceability and accountability.

models, real-time analytics, and business intelligence applications, enabling organizations to break down silos and support diverse computational workloads. Despite their promise, data lakes present novel challenges: as data moves through complex series of transformations, joins, aggregations, and enrichments, it becomes critical to maintain detailed knowledge about how data was produced. This knowledge is encapsulated by data lineage, a record of the "journey" data takes from raw sources to derived artifacts [2].

Data lineage is essential not only for compliance—such as ensuring adherence to GDPR or other regulatory standards—but also for assessing data quality, diagnosing pipeline issues, facilitating reproducibility of analyses, and building trust among stakeholders. At a fundamental level, lineage helps data consumers understand where a given dataset or data point came from, what transformations were applied, and what intermediate artifacts influenced its current state.

Data lineage can be viewed at different levels of abstraction. Coarse-grained lineage describes lineage at a dataset-to-dataset or table-to-table level, offering a broad overview of data flows and dependencies. Fine-grained lineage, by contrast, captures lineage at the level of individual records, cells, or even values, providing detailed traceability that can support highly nuanced debugging and compliance checks. While both forms of lineage are valuable, their capture, modeling, storage, and visualization pose distinct technical challenges. This paper offers a deep, descriptive overview of current approaches and emerging techniques in these areas, with particular emphasis on fully automated solutions that integrate into modern data lake ecosystems.

## 2 Background and Motivation

Data lakes have emerged as a critical architecture for organizations aiming to centralize and scale their data management practices. By serving as vast repositories for raw and semi-structured data, they allow enterprises to collect information from a diverse range of sources, including transactional databases, streaming platforms, sensor networks, and external data providers. Once ingested, these raw datasets are transformed, processed, and analyzed through a multitude of tools such as SQL queries, ETL (Extract, Transform, Load) pipelines, Spark jobs, Python scripts, and machine learning workflows. However, the sheer complexity of these processes and the diversity of tools involved present a unique challenge: understanding how each refined dataset, metric, or insight has been generated. This is where the concept of data lineage becomes indispensable.

Data lineage refers to the end-to-end traceability of data as it moves through an organization's ecosystem, capturing the relationships, transformations, and processes applied to it. In data lakes, where data flows are inherently nonlinear and involve a variety of interconnected systems, robust lineage tracking serves as a foundational capability for ensuring that data remains trustworthy, auditable, and operationally resilient. Below, we explore four critical aspects of why data lineage is essential in data lakes, emphasizing its role in governance, quality assurance, impact analysis, and reproducibility.

The first and perhaps most significant benefit of data lineage is its centrality to data governance and compliance efforts. Modern regulations such as GDPR, HIPAA, and the CCPA require organizations to not only safeguard sensitive data but also maintain comprehensive audit trails detailing how data is processed, transformed, and shared. Without lineage, compliance with these stringent requirements becomes both challenging and risky. For example, a healthcare provider managing sensitive patient data needs to ensure that every transformation and processing step complies with privacy regulations. Lineage allows the organization to demonstrate adherence to such regulations by clearly mapping out the sequence of operations performed on each dataset. This ability to produce a verifiable chain of custody is invaluable during audits, enabling organizations to substantiate their compliance with both internal policies and external regulations [3].

Beyond compliance, data lineage plays a pivotal role in assuring data quality and fostering trust in analytical outputs. Data quality issues often propagate through systems, with errors in upstream data sources cascading into downstream models, dashboards, and reports. Lineage facilitates the identification of these upstream dependencies, allowing data engineers to pinpoint the exact transformation or data source responsible for anomalies [3]. For instance, if a machine learning model produces unexpected predictions, lineage tracking can trace the issue back to a specific transformation step or dataset update that may have introduced inaccuracies. By enabling rapid identification and resolution of such issues, lineage not only improves the reliability of data but also bolsters confidence in the insights derived from it.

Another critical dimension of data lineage is its utility for impact analysis and change management. Data engineers and architects frequently face the challenge of making modifications to datasets, schemas, or processing workflows without inadvertently breaking downstream applications, dashboards, or analytical models. Lineage graphs offer a visual representation of these dependencies, showing precisely which downstream components will be affected by a given change. This capability allows teams to assess the potential risks associated with updates and implement safeguards to mitigate unintended consequences. For example, deprecating a rarely used dataset might appear to be a minor change, but lineage analysis could reveal that the dataset is indirectly powering a high-priority executive dashboard. By providing this visibility, lineage not only prevents disruptions but also enables more informed decision-making.

Finally, the importance of data lineage extends to the domains of reproducibility and debugging, particularly in the context of complex data science workflows. As organizations increasingly rely on machine learning and advanced analytics, the ability to reproduce experiments and trace results becomes paramount. Data lineage acts as a blueprint for these processes, detailing every step in the data preparation and analysis pipeline. For example, a data scientist investigating why a predictive model underperformed on a recent dataset can consult the lineage graph to recreate the exact environment and transformations used during the model's training phase. This ability to simulate and re-run workflows ensures that insights are reproducible and experiments are interpretable, fostering a culture of transparency and rigor within the organization [4].

To further illustrate the utility of data lineage, we present two tables highlighting key challenges in managing data lakes and the corresponding solutions enabled by robust lineage tracking. The first table provides an overview of common problems faced in data governance, quality assurance, and operational efficiency. The second table outlines practical use cases of lineage in data lakes, demonstrating its applicability across various industries.

**Table 1** Challenges in Managing Data Lakes and the Role of Data Lineage

| **Challenge** | **Description and Role of Data Lineage** |
|---|---|
| Data Governance | Difficulty in demonstrating compliance with regulatory requirements. Data lineage provides a verifiable audit trail to ensure regulatory adherence. |
| Data Quality Issues | Propagation of errors from upstream sources to downstream systems. Lineage enables identification of root causes and ensures quicker resolution. |
| Impact of Changes | Unintended disruptions caused by altering or deprecating datasets. Lineage offers a dependency map for assessing the impact of changes. |
| Reproducibility | Lack of clarity on how specific metrics or models were generated. Lineage serves as a blueprint for re-running workflows. |

**Table 2** Practical Use Cases of Data Lineage in Data Lakes

| **Use Case** | **Application and Benefits** |
|---|---|
| Regulatory Reporting | Financial institutions using lineage to validate the accuracy of reports submitted to regulators, ensuring compliance with laws such as Basel III. |
| Root Cause Analysis | Telecom companies tracing dropped call metrics back to faulty data transformations in network event logs. |
| Pipeline Optimization | E-commerce platforms identifying redundant processing steps in recommendation engine pipelines, reducing latency. |
| Model Debugging | Pharmaceutical firms recreating model training environments to understand discrepancies in clinical trial predictions. |

## 3 Coarse-Grained vs. Fine-Grained Lineage

In modern data management systems, particularly in the context of large-scale data lakes, data lineage represents a foundational mechanism for tracking the origin, transformations, and dependencies of data throughout its lifecycle. Lineage

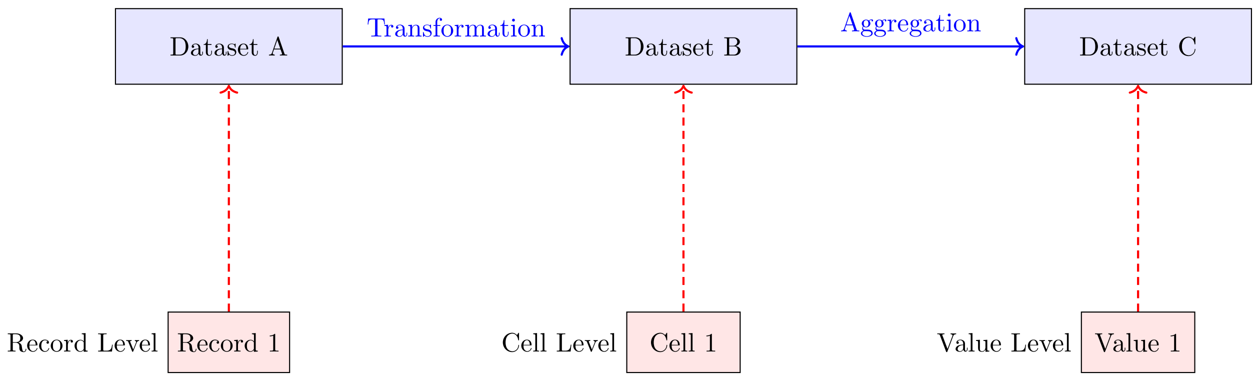

**Figure 2** Comparison of Coarse-grained and Fine-grained Data Lineage: Coarse-grained lineage tracks dataset-level dependencies, while fine-grained lineage provides granular traceability at the level of individual records, cells, or values.

provides critical traceability to support data governance, debugging, auditing, and reproducibility. The granularity of lineage tracking determines its utility for specific applications and can broadly be divided into two categories: coarse-grained lineage and fine-grained lineage. Each of these approaches captures data dependencies at different levels of detail, with coarse-grained lineage focusing on dataset-level relationships and fine-grained lineage capturing record- or attribute-level dependencies. Both forms of lineage play a vital role in ensuring that data systems are transparent, reliable, and maintainable [4].

Coarse-grained lineage refers to the tracking of lineage at the level of datasets, tables, or broader data abstractions. It captures how datasets are derived from other datasets, detailing the sequence of operations that connect input and output entities. For example, a coarse-grained lineage model might specify that Dataset_A was generated by performing a join operation between Dataset_B and Dataset_C, followed by filtering rows based on specific conditions. Coarse-grained lineage is comparatively straightforward to capture and requires minimal storage and computational resources because it does not include details about individual records, rows, or columns. This simplicity makes it particularly effective for providing a high-level overview of data dependencies and transformations, which is valuable for data governance, cataloging, and change management.

A practical application of coarse-grained lineage is in impact analysis, where it helps determine which downstream datasets, dashboards, or applications might be affected by modifications to an upstream dataset or process. For example, if Dataset_X is slated for deprecation, coarse-grained lineage can reveal that Dataset_Y and Dataset_Z depend on Dataset_X, allowing stakeholders to assess the risks and prepare for downstream impacts. Additionally, coarse-grained lineage supports compliance with regulatory and organizational data governance requirements by maintaining a clear record of the provenance and transformations applied to datasets. Organizations can use this form of lineage to demonstrate adherence to regulatory frameworks such as GDPR or CCPA, particularly when audit trails must be presented to external or internal auditors.

However, coarse-grained lineage is not without its limitations. By focusing on datasets as whole entities, it does not provide insight into how individual data points are processed or transformed. For instance, if an anomaly is detected in a derived dataset, coarse-grained lineage can identify the source datasets involved but cannot trace the specific record or transformation that caused the issue. Similarly, this lack of detail makes coarse-grained lineage unsuitable for use cases that require granular traceability, such as debugging data pipelines or verifying the correctness of individual calculations in analytics workflows [5].

Fine-grained lineage addresses these limitations by offering a more detailed view of data transformations and dependencies. Unlike coarse-grained lineage, fine-grained lineage tracks data at the level of individual records, rows, columns, or even cells. This approach captures how specific data points in a target dataset were derived from corresponding data points in one or more source datasets, along with the transformations applied. For example, in a data pipeline that aggregates sales data, fine-grained lineage can trace a particular sales figure in the final report back to individual transactions in the raw input data. This level of granularity is particularly useful for tasks requiring high precision, such as debugging, compliance auditing, and validating machine learning models.

Fine-grained lineage is invaluable for root cause analysis in scenarios where data quality issues arise. Suppose a downstream dashboard reports incorrect metrics, and fine-grained lineage is available. Engineers can use the lineage information to trace the issue back to a specific row or transformation step in the upstream process, thereby isolating the error with precision. Similarly, in industries with stringent regulatory requirements, such as healthcare and finance, fine-grained lineage allows organizations to demonstrate detailed traceability for individual data points. This can be critical during audits, where regulators may require evidence of how specific records were processed and used in decision-making systems.

Despite its benefits, capturing and maintaining fine-grained lineage is a complex and resource-intensive process. Fine-grained lineage systems generate significantly more metadata than their coarse-grained counterparts, necessitating advanced storage and computational capabilities to manage the additional overhead. Additionally, the complexity of implementing fine-grained lineage increases with the scale of the data lake and the intricacy of the transformations involved. For example, in environments with millions of records and dozens of interconnected processing pipelines, maintaining fine-grained lineage requires not only sophisticated lineage tracking mechanisms but also efficient methods for querying and visualizing the resulting metadata. This complexity often necessitates trade-offs, as organizations must balance the benefits of fine-grained traceability against the costs and feasibility of implementation.

Modern data ecosystems increasingly demand both coarse-grained and fine-grained lineage, as each serves distinct yet complementary purposes. Coarse-grained lineage is ideal for understanding macro-level data flows, supporting governance, and performing high-level impact analysis. It provides an overview of dependencies and transformations without delving into operational details. Fine-grained lineage, on the other hand, is essential for use cases that require detailed traceability, such as debugging, auditing, and ensuring data correctness. By integrating both forms of lineage into their data management practices, organizations can achieve a comprehensive understanding of their data systems while addressing a wide range of operational and analytical needs [6].

The following tables provide a comparative overview of coarse-grained and fine-grained lineage, as well as examples of use cases where each approach is particularly effective. These tables illustrate how the two types of lineage complement each other and highlight their respective advantages and limitations.

**Table 3** Comparison of Coarse-Grained and Fine-Grained Lineage

| **Attribute** | **Coarse-Grained Lineage** |
|---|---|
| Granularity | Captures relationships at the dataset or table level. |
| Primary Purpose | Provides high-level insights into data dependencies and transformations. |
| Ease of Implementation | Easier to implement and store due to reduced complexity. |
| Use Cases | Impact analysis, governance, data cataloging. |
| Limitations | Insufficient for debugging or tracing individual records. |
| **Attribute** | **Fine-Grained Lineage** |
| Granularity | Tracks dependencies and transformations at the record, row, or column level. |
| Primary Purpose | Enables detailed traceability for debugging, auditing, and precision analysis. |
| Ease of Implementation | Complex and resource-intensive to implement and maintain. |
| Use Cases | Root cause analysis, regulatory compliance, model validation. |

**Table 4** Use Cases for Coarse-Grained and Fine-Grained Lineage

| **Type of Lineage** | **Example Use Cases and Benefits** |
|---|---|
| Coarse-Grained Lineage | Mapping dependencies between datasets in an enterprise data catalog for governance purposes. |
| Coarse-Grained Lineage | Identifying downstream systems affected by changes to an upstream dataset. |
| Fine-Grained Lineage | Tracing errors in a derived dataset to specific rows or columns in the source dataset for debugging. |
| Fine-Grained Lineage | Demonstrating detailed audit trails for individual records in compliance with GDPR or HIPAA. |

## 4 Automated Lineage Capture Techniques

### 4.1 Instrumentation in ETL and Data Pipeline Tools

The use of instrumentation within ETL (Extract, Transform, Load) and data pipeline tools represents a well-established approach for capturing data lineage in automated and scalable ways. Instrumentation involves configuring or extending existing frameworks, such as Apache Spark, Apache Flink, and Apache Airflow, to log metadata about data transformations and dependencies. By embedding lineage tracking capabilities directly into these execution frameworks, organizations can capture detailed information about how datasets are created, modified, and propagated through their pipelines. This section explores the mechanisms, challenges, and practical applications of instrumentation-based lineage tracking and provides examples of its implementation in modern data engineering environments.

Instrumentation in ETL and pipeline tools typically operates by integrating with the internal mechanisms of these frameworks. For instance, in Apache Spark, lineage can be captured by leveraging its logical and physical execution plans. When a Spark job performs a join operation between two datasets, Dataset_X and Dataset_Y, an instrumentation plugin can log that these datasets contributed to the generation of Dataset_Z. Similarly, filtering operations, column mappings, and aggregations can be tracked at a transformation-step level. In Apache Airflow, custom lineage plugins can record relationships between tasks and the datasets they produce or consume, creating a graph of task dependencies and their associated outputs. This

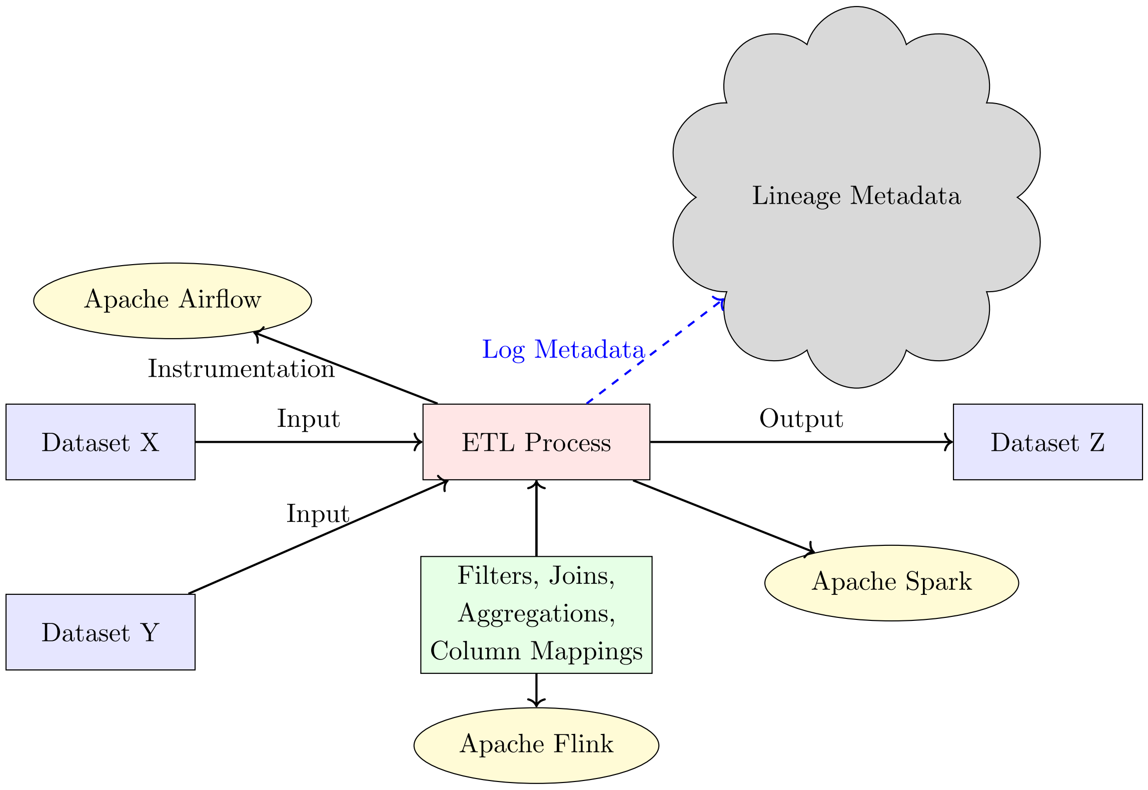

**Figure 3** Automated Lineage Capture in ETL and Data Pipelines: Illustrating the use of frameworks like Apache Spark, Flink, and Airflow to instrument and log metadata, enabling lineage tracking at both dataset and transformation levels.

process enables detailed lineage capture that is closely aligned with the underlying execution logic [7].

One of the significant advantages of instrumentation-based lineage tracking is its ability to integrate deeply with the execution frameworks. These tools often maintain detailed internal metadata, such as Spark's logical plans or Flink's dataflow graphs, which describe the transformations applied to data. By accessing and interpreting this metadata, instrumentation mechanisms can deduce lineage information efficiently and with high accuracy. For example, in Apache Spark, the logical plan provides a structured representation of the query, detailing operations like joins and filters, while the physical plan specifies the execution strategy. This dual-layered information allows for precise tracking of both coarse-grained lineage (relationships between datasets) and fine-grained lineage (dependencies at the row or column level).

Real-time lineage capture is another key feature of instrumentation-based approaches. Data pipeline frameworks, particularly those designed for streaming applications, process data continuously, making real-time lineage tracking a critical capability. Apache Flink, for example, supports lineage capture for streaming pipelines by monitoring operations such as windowed aggregations and stateful computations. In scenarios where immediate traceability is required—such as detecting and resolving errors in a streaming analytics system—real-time lineage enables prompt debugging and issue resolution.

Despite these benefits, instrumentation-based lineage tracking also presents several challenges. One major limitation is the need for deep integration with the tools being instrumented. Not all ETL or pipeline frameworks provide robust APIs or mechanisms for capturing lineage metadata, requiring significant customization or development effort. For example, while Apache Spark offers well-defined constructs for accessing its logical and physical plans, other frameworks may require invasive modifications to their internal execution logic. Moreover, capturing fine-grained lineage—such as tracing individual rows or columns through complex transformations—introduces computational overhead and can degrade pipeline performance. For instance, recording lineage for every record in a high-throughput streaming pipeline may increase latency and storage costs significantly.

Another challenge relates to the heterogeneity of modern data ecosystems. Organizations often employ multiple ETL tools and pipeline frameworks, each with its unique lineage capabilities and limitations. Integrating lineage across these disparate systems requires careful coordination and often necessitates the use of external lineage tools or centralized metadata repositories. Hybrid approaches, which combine instrumentation with external post-processing or log analysis, are frequently adopted to address these complexities.

The following subsections illustrate practical examples of instrumentation-based lineage tracking in widely used frameworks and tools. These examples, along with Tables 5 and 6, highlight the diverse applications and trade-offs of this approach.

### *4.1.1 Examples of Instrumentation-Based Lineage in Popular Frameworks*

In Apache Spark, instrumentation-based lineage tracking is typically implemented using its logical plan. For example, consider a Spark job that processes sales data by joining transaction records with customer details, followed by aggregating total sales per region. The logical plan for this job can be analyzed to deduce that the final dataset is derived from two source datasets, with details about the join conditions and the applied aggregations. This lineage metadata can then be stored in a centralized repository for downstream analysis and governance [8].

Apache Flink provides another compelling example. In a streaming application that monitors network traffic, lineage can be captured by tracking how individual network events are processed through windowed aggregations and key-based state computations. For instance, a lineage plugin might log that a specific anomaly detection alert is derived from events originating from three different network sensors, processed through a sliding window aggregation. This real-time lineage information can be critical for debugging and improving the accuracy of anomaly detection models.

In Apache Airflow, instrumentation-based lineage tracking can be achieved through the use of custom plugins or extensions. For example, in a pipeline that orchestrates ETL jobs for loading customer data into a data warehouse, a lineage plugin can record that Task_A generates Dataset_1, which is then consumed by Task_B to produce Dataset_2. This task-level lineage can provide valuable insights into the dependencies between pipeline components and help identify bottlenecks or failures.

**Table 5** Capabilities and Challenges of Instrumentation-Based Lineage Tracking

| Aspect | Description |
|---|---|
| Integration with Frameworks | Leverages the internal structures of ETL and pipeline tools (e.g., Spark's logical plans) for efficient lineage capture. |
| Real-Time Capture | Enables lineage tracking in real-time, especially in streaming pipelines such as those implemented in Apache Flink. |
| Granularity | Supports coarse-grained lineage by default; fine-grained lineage tracking may require additional logic, increasing complexity. |
| Implementation Complexity | Requires deep integration with tools and may involve custom development for frameworks lacking robust lineage APIs. |
| Performance Impact | Capturing fine-grained lineage may introduce computational overhead, especially for large-scale or streaming workloads. |

**Table 6** Examples of Instrumentation-Based Lineage Tracking in Frameworks

| Framework | Instrumentation Approach and Features |
|---|---|
| Apache Spark | Captures lineage by analyzing logical and physical plans. Example: Tracking joins, filters, and aggregations in batch pipelines. |
| Apache Flink | Logs lineage in real-time for streaming data. Example: Tracing alert generation in anomaly detection pipelines. |
| Apache Airflow | Uses custom plugins to log task-level lineage. Example: Recording dependencies between ETL jobs in data orchestration workflows. |
| Custom ETL Tools | Implements bespoke instrumentation for lineage capture. Example: Proprietary ETL pipelines logging transformation histories for audit trails. |

As demonstrated in Tables 5 and 6, this approach is best complemented with other lineage tracking strategies to achieve comprehensive coverage in diverse data ecosystems.

## 4.2 Query Parsing and Logical Plan Analysis

SQL queries, as a dominant paradigm for defining data transformations and manipulations in data lakes and relational systems, provide a rich source of information for lineage tracking. By analyzing SQL statements, it is possible to infer how datasets are transformed, which columns contribute to downstream outputs, and the relationships between source and target datasets. SQL lineage analysis can operate at multiple levels of granularity, ranging from coarse-grained tracking of table-to-table dependencies to fine-grained tracking of column-level and row-level relationships. This approach, however, is not without limitations, particularly when dealing with dynamic, user-defined, or non-declarative transformations.

At the coarse-grained level, SQL query analysis identifies dependencies between source and target tables. For example, consider the following SQL query:

```
SELECT a.id, b.value
FROM TableA a
JOIN TableB b ON a.key = b.key
WHERE b.status = 'ACTIVE';
```

This query indicates that the resulting target dataset derives its data from two source tables: `TableA` and `TableB`. Coarse-grained lineage can be extracted by noting that `TableA` and `TableB` are input dependencies for the target dataset. Such lineage provides a high-level view of the data flow, which is particularly useful for tasks such as impact analysis, governance, and cataloging. For instance, if `TableA` is modified or deprecated, lineage analysis can identify downstream datasets or reports that depend on it, allowing stakeholders to assess and mitigate the impact.

Fine-grained lineage provides more detailed insights by analyzing how specific columns and rows in source datasets contribute to target datasets. In the above query, it can be inferred that the `id` column in the target dataset maps directly to the `id` column from `TableA`, while the `value` column maps to `value` in `TableB`. Furthermore, the `JOIN` condition (`a.key = b.key`) and the filter (`b.status = 'ACTIVE'`) provide additional context, indicating that only rows satisfying these criteria are included in the target dataset.

This level of detail is critical for debugging and auditing. For example, if a downstream report contains anomalies in the `value` column, fine-grained lineage can trace the issue back to specific records or transformations in `TableB`. Fine-grained lineage also supports reproducibility by providing a precise mapping of how each target column or row is derived from its inputs, allowing for accurate reconstruction of transformations on new data.

Achieving fine-grained lineage from SQL queries often requires the construction of a logical query plan, which represents the sequence of operations (e.g., joins, filters, projections) needed to execute the query. Logical plans serve as a blueprint for understanding data transformations and are essential for lineage inference at a granular level. Tools such as Apache Calcite, which parses SQL queries into logical plans, or query engines like Spark SQL, which produce detailed execution plans, can be used to extract and analyze lineage metadata.

While SQL query analysis provides a structured and deterministic way to infer lineage, it faces several challenges, particularly when dealing with advanced or non-standard transformations. One notable challenge is the use of dynamic SQL or transformations defined at runtime. For example, queries generated by applications or scripts with dynamic parameters may not have a static structure, complicating lineage extraction. Another challenge arises from user-defined functions (UDFs) and procedural extensions like PL/SQL, where data transformations are encapsulated within custom code. These black-box transformations obscure the logic, requiring additional effort to parse or reverse-engineer lineage.

Further complexity arises in cases involving intermediate datasets or nested queries. For example:

```
WITH Filtered AS (
    SELECT id, key FROM TableA WHERE status = 'ACTIVE'
)
SELECT f.id, b.value
FROM Filtered f
JOIN TableB b ON f.key = b.key;
```

Here, the intermediate dataset `Filtered` must first be analyzed to trace its lineage back to `TableA`, before mapping the final output to both `TableA` and `TableB`. Similarly, handling complex aggregations, window functions, and recursive queries requires sophisticated lineage extraction techniques to ensure completeness and accuracy. certain transformations defy traditional logical analysis. For example, machine learning feature generation often involves embedding SQL queries within workflows that apply additional, non-SQL transformations. In such cases, logical analysis of SQL alone is insufficient, and integration with external tools or lineage capture mechanisms is necessary.

SQL-based lineage analysis is most effective when combined with other lineage tracking techniques, such as instrumentation or log analysis. For instance, instrumentation can provide runtime lineage metadata for black-box transformations, while SQL analysis captures declarative relationships. Table 7 illustrates several examples of SQL queries and their corresponding lineage outputs, while Table 8 summarizes common challenges and strategies for overcoming them.

**Table 7** Examples of SQL Queries and Corresponding Lineage Outputs

| SQL Query | Lineage Output |
|---|---|
| `SELECT id, value`<br>`FROM TableA`<br>`WHERE status = 'ACTIVE';` | Target columns (`id, value`) map to source columns in `TableA`. Rows are filtered based on the condition `status = 'ACTIVE'`. |
| `SELECT a.id, b.value`<br>`FROM TableA a`<br>`JOIN TableB b`<br>`ON a.key = b.key;` | Target column `id` derives from `TableA.id`, and `value` derives from `TableB.value`. `JOIN` condition establishes row-level dependencies. |
| `WITH Aggregated AS (`<br>`    SELECT region, SUM(sales) \\ AS total_sales`<br>`    FROM Transactions`<br>`    GROUP BY region`<br>`)`<br>`SELECT * FROM Aggregated;` | Target dataset columns (`region, total_sales`) derive from source columns in `Transactions`. Grouping creates new derived relationships at the row level. |

**Table 8** Challenges in SQL Lineage Analysis and Mitigation Strategies

| Challenge | Mitigation Strategy |
|---|---|
| Dynamic SQL Queries | Capture and log runtime query structures for analysis. Leverage query templates for consistent lineage extraction. |
| User-Defined Functions | Analyze UDF logic separately or instrument UDF execution for lineage metadata capture. |
| Intermediate Datasets | Parse query plans to trace lineage through temporary views or subqueries. |
| Non-SQL Transformations | Combine SQL-based analysis with external lineage tracking tools or instrumentation. |

#### *4.2.1 Runtime Data Tracking for Lineage Capture*

Runtime data tracking represents a sophisticated and highly granular approach to capturing data lineage. Unlike static methods such as SQL analysis or instrumentation, runtime tracking propagates unique identifiers or provenance tags through the data processing pipeline at execution time. By annotating individual records as they traverse the pipeline, this technique enables precise traceability, allowing data engineers to backtrack the final output dataset to its exact source records and transformations. While this approach offers unmatched accuracy, it also introduces computational and storage overhead, requiring careful consideration of trade-offs in large-scale data environments.

The core concept of runtime tracking is to attach a unique provenance tag to each record in the source dataset. These tags, often implemented as metadata attributes, are propagated through the processing pipeline as transformations such as joins, filters, and aggregations are applied. For example, consider a dataset `CustomerTransactions` in which each transaction record is annotated with a unique tag at ingestion. When this dataset is joined with a `CustomerProfiles` dataset, the lineage system combines the tags from both datasets to produce a new composite tag for the resulting records in the output. Similarly, filters preserve the tags of records that pass the specified condition, ensuring that the lineage of the filtered output can be traced back to the original source.

This approach effectively embeds lineage information directly into the data itself. At the conclusion of the pipeline, the output dataset contains a "lineage fingerprint," enabling engineers to reconstruct the entire data flow for each record. For example, if a report generated from the output dataset contains anomalies, the lineage fingerprint can be used to identify the exact source records and transformations that contributed to the problematic rows, facilitating targeted debugging and correction.

One of the most significant advantages of runtime data tracking is its ability to produce fine-grained lineage with exceptional accuracy. Unlike coarse-grained techniques that capture lineage at the table or dataset level, runtime tracking operates at the level of individual records, rows, or even specific data points. This level of granularity is particularly valuable in industries with stringent compliance and audit requirements, such as healthcare or finance, where organizations must provide detailed traceability for sensitive or regulated data.

Moreover, runtime tracking is inherently dynamic, capturing lineage in real time as data is processed. This ensures that lineage information reflects the actual execution of the pipeline, accounting for runtime conditions such as dynamic filters, data-dependent branching, or variations in data distributions. For example, in a streaming pipeline for fraud detection, runtime tracking can dynamically propagate tags through sliding window aggregations, enabling precise identification of which transaction records contributed to a flagged anomaly.

The most notable of these is the computational and memory overhead associated with propagating tags through the pipeline. For large-scale data lakes containing billions of records, annotating and tracking tags for each record can result in significant increases in storage requirements and processing times. For example, in a pipeline that processes petabytes of log data, the additional metadata generated by runtime tracking could exceed the size of the original data itself, straining system resources and increasing costs.

Another limitation is the complexity of handling operations that aggregate or transform data in ways that obscure individual record-level provenance. Aggregations, for instance, combine multiple input records into a single output record, requiring the system to encode the lineage of all contributing records within the resulting tag. This can lead to metadata "explosion," where the size of the tags grows disproportionately with the complexity of the transformations. Joins and nested transformations further exacerbate this issue, as they often involve combining multiple datasets with independent provenance tags [9].

### 4.3 Optimizations for Runtime Tracking

To address the computational and storage overheads associated with runtime tracking, several optimization techniques have been proposed:

1 Instead of maintaining exact tags for every record, probabilistic techniques encode lineage information using compact data structures such as Bloom filters. While this approach sacrifices some accuracy, it significantly reduces the size of the metadata, making runtime tracking feasible for large-scale systems.

2 Sampling involves tracking lineage for a representative subset of records rather than the entire dataset. By analyzing the lineage of the sampled records, engineers can infer patterns and dependencies without incurring the overhead of

full-scale tracking. This approach is particularly useful for exploratory analysis or scenarios where approximate lineage suffices.

3 Compression techniques can be applied to reduce the storage footprint of provenance tags. For example, hierarchical encoding can compactly represent tags for aggregated records by grouping them according to their source datasets.

4 In many cases, fine-grained lineage is only needed for specific portions of the pipeline or for particular datasets. By selectively enabling runtime tracking for critical transformations or datasets, organizations can balance the trade-off between granularity and overhead.

Runtime tracking has been implemented in various systems to support fine-grained lineage for complex data workflows. Table 9 provides illustrative examples of runtime tracking mechanisms and their applications in different frameworks.

**Table 9** Examples of Runtime Data Tracking for Lineage

| System/Framework | Runtime Tracking Mechanism and Application |
|---|---|
| Apache Spark | Propagates provenance tags through RDD transformations, enabling record-level traceability in batch and streaming pipelines. |
| Apache Flink | Supports custom tagging for records in streaming pipelines, with tags propagating through windowed aggregations and stateful computations. |
| Scientific Workflow Systems | Uses runtime tracking to annotate and trace intermediate datasets in workflows for computational biology or climate modeling. |
| Proprietary Data Lakes | Implements hierarchical tagging schemes for tracking lineage in multi-stage ETL pipelines with aggregations and joins. |

- Runtime data tracking is particularly effective for use cases requiring high levels of granularity and precision. For example:
- Organizations in regulated industries can use runtime tracking to provide detailed audit trails that map specific outputs to their source records, ensuring compliance with regulations like GDPR or HIPAA.
- When anomalies or errors are detected in downstream systems, runtime tracking enables engineers to trace issues back to the exact records or transformations responsible.
- In environments with rapidly changing data or runtime conditions, real-time runtime tracking provides immediate visibility into how data flows through the pipeline.

Table 10 summarizes the main challenges associated with runtime tracking and the strategies employed to address them.

**Table 10** Challenges and Optimizations in Runtime Data Tracking

| Challenge | Optimization Strategy |
|---|---|
| High Memory and Storage Overhead | Use probabilistic or sampling-based lineage to reduce metadata size. |
| Metadata Explosion in Aggregations | Compress provenance tags using hierarchical or compact encoding techniques. |
| Complex Joins and Nested Transformations | Track composite tags and selectively enable tracking for critical transformations. |
| Performance Impact on Large Pipelines | Implement selective tracking or optimize metadata propagation algorithms. |

Runtime data tracking offers a highly accurate and fine-grained method for capturing data lineage by propagating provenance tags through the data pipeline during execution. While it provides unmatched precision for traceability and auditing, the associated computational and storage costs necessitate careful consideration of optimizations such as probabilistic techniques, sampling, and metadata compression. As illustrated in Tables 9 and 10, runtime tracking is used in applications requiring detailed record-level lineage, such as regulatory compliance, debugging, and dynamic data environments.

### 4.4 Hybrid Approaches

In many real-world scenarios, organizations adopt hybrid approaches that combine coarse-grained lineage derived from query parsing with selective, on-demand fine-grained lineage capture using runtime tracking. This hybrid model can be orchestrated by lineage management systems that determine the required level of granularity for each dataset and adapt accordingly. Such an adaptive approach can balance performance, storage requirements, and accuracy.

## 5 Modeling and Storing Lineage Information

Efficiently modeling and storing data lineage is critical for enabling traceability, governance, and reproducibility in modern data systems. Given the diversity in granularity requirements—ranging from coarse-grained (dataset-level) to fine-grained (record- or cell-level)—designing metadata models and storage systems involves addressing challenges related to scalability, performance, and flexibility. This section examines common approaches for modeling lineage information, focusing on metadata models for coarse-grained and fine-grained lineage, and explores techniques to address the scalability and performance challenges associated with storing lineage in large-scale data lakes.

Coarse-grained lineage models represent dependencies and transformations at the dataset or table level, offering a high-level view of data flows within a system. A common representation for coarse-grained lineage is the directed acyclic graph (DAG), where nodes correspond to datasets, and edges denote transformations or data dependencies. For example, an edge from Dataset_Y and Dataset_Z to Dataset_X indicates that Dataset_X was derived from the union or transformation of Dataset_Y and Dataset_Z. Such models provide clear visualizations and allow users to trace upstream or downstream dependencies efficiently.

Graph databases such as Neo4j, JanusGraph, or TigerGraph are well-suited for storing coarse-grained lineage due to their natural alignment with DAG structures. These systems allow lineage information to be queried flexibly, enabling users to trace the lineage of a specific dataset or transformation step. For instance, a user querying the lineage of Dataset_X might retrieve all upstream datasets contributing to it, along with the specific transformation types (e.g., joins, filters, aggregations). Additionally, graph databases can integrate with other metadata sources, such as schema details or governance tags, enabling multi-dimensional queries. For example, a query might return not only the upstream datasets but also their associated schema versions and compliance labels.

To add granularity to coarse-grained lineage models, column-level lineage metadata maps individual columns in target datasets to their source columns. For instance, if Dataset_X.column_1 is derived from Dataset_Y.column_A and Dataset_Z.column_B, this mapping can be recorded in structured formats such as JSON, Avro, or Parquet. Metadata catalogs such as Apache Hive Metastore and Apache Atlas support this type of lineage modeling by extending the schema definitions to include column-level mappings. Column-level lineage metadata is particularly valuable for identifying the specific input columns that contribute to an erroneous output column, enabling more focused debugging and impact analysis.

### 5.1 Models for Fine-Grained Lineage

Fine-grained lineage models capture dependencies at the level of individual records or cells, providing precise traceability for high-resolution debugging, compliance, and reproducibility. However, modeling and storing fine-grained lineage present significant challenges due to the sheer volume of metadata generated in large-scale systems. Several approaches have been proposed to address these challenges [10].

Fine-grained lineage can theoretically be modeled as a provenance graph where each record is a node, and edges represent the relationships between records in source and target datasets. For example, if a record in Dataset_X is derived from two records in Dataset_Y and Dataset_Z, the provenance graph would include nodes for each of these records and edges connecting them. While this approach provides unparalleled granularity, it results in extremely large graphs that are impractical to store or query directly. Segmentation or compaction strategies can alleviate this challenge by grouping similar records or storing transformations separately from the data references.

To reduce storage overhead, compressed representations of fine-grained lineage summarize dependencies for groups of records sharing similar provenance. For instance, if multiple records in the target dataset are derived from the same set of source records, a single lineage reference can be stored for the entire group. Alternatively, hashes or signatures of source records can be used instead of storing full references, enabling partial reconstruction of lineage when needed. These techniques significantly reduce metadata volume while preserving the ability to trace dependencies at a record level.

Fine-grained lineage can also be coupled with data versioning systems such as Delta Lake or LakeFS, which track changes at a granular level. By referencing the exact snapshot of a source dataset that contributed to a downstream record, delta-based storage systems enable precise lineage reconstruction. For example, if a record in Dataset_X is derived from a specific version of Dataset_Y, the lineage metadata would include a pointer to the relevant snapshot in Dataset_Y. This approach tightly integrates lineage tracking with version control, facilitating reproducibility and debugging but introducing additional complexity in terms of metadata management [11].

### 5.2 Scalability and Performance Considerations

Storing lineage information for large-scale data lakes poses significant scalability challenges, particularly when dealing with fine-grained lineage. Efficient storage systems must balance the trade-off between metadata granularity, storage costs, and query performance.

Distributed graph databases or metadata repositories can partition lineage data by dataset, partition, or time, enabling horizontal scaling. For example, lineage for a dataset processed in daily partitions can be stored separately for each partition, allowing lineage queries to target specific subsets of data. Partitioning by transformation type or lineage depth (e.g., immediate parents versus full ancestry) can further improve query performance by narrowing the scope of metadata retrieval [12].

Frequently accessed lineage queries, such as tracing all inputs to a specific dataset, can be precomputed and stored as materialized views. For example, a materialized view might store the full upstream dependency graph for a high-priority dataset, enabling rapid retrieval without recomputation. Similarly, caching lineage information for recent or commonly queried datasets can reduce the latency of lineage queries in interactive systems.

For less frequently accessed lineage queries, systems can defer detailed computation until the query is issued. For instance, coarse-grained lineage may be stored upfront, while fine-grained lineage is reconstructed on demand using compressed or summarized metadata. This approach minimizes storage costs but trades off higher query latency for infrequently accessed lineage details [13].

### 5.3 Examples of Metadata Storage Models

Table 11 summarizes common models for storing coarse-grained and fine-grained lineage, while Table 12 outlines techniques for scaling lineage storage to meet the demands of large-scale data systems.

**Table 11** Lineage Metadata Models and Storage Approaches

| Model | Description and Example Use Cases |
|---|---|
| Coarse-Grained Lineage | DAG representation of dataset-level dependencies. Example: Neo4j storing dataset derivation paths for impact analysis. |
| Column-Level Lineage | Maps target columns to source columns using JSON or Avro. Example: Debugging column-level transformations in Apache Atlas. |
| Fine-Grained Provenance Graphs | Record-level dependency graphs for precise traceability. Example: Research workflows tracking intermediate results. |
| Compressed Lineage Summaries | Groups records with shared lineage to reduce storage. Example: Summarized metadata for aggregations in large-scale pipelines. |
| Delta-Based Storage | Integrates lineage with version control. Example: Delta Lake referencing specific dataset snapshots for reproducibility. |

**Table 12** Techniques for Scaling Lineage Storage

| Technique | Description and Benefits |
|---|---|
| Sharding and Partitioning | Splits lineage metadata by dataset, partition, or time to enable horizontal scaling. Example: Partitioning by date for daily ETL jobs. |
| Caching and Materialized Views | Precomputes frequently accessed lineage queries to reduce retrieval latency. Example: Materialized dependency graph for critical datasets. |
| On-Demand Lineage Reconstruction | Stores compressed metadata for deferred computation of fine-grained lineage. Example: Reconstructing lineage from hash-based summaries. |
| Metadata Compression | Uses compact representations to minimize storage overhead. Example: Hierarchical encoding for aggregated lineage data. |

## 6 Visualization and Interaction with Lineage Data

The visualization and interaction with lineage data play a pivotal role in enabling users to understand complex data dependencies, transformations, and their impacts. By presenting lineage data in a user-friendly manner, organizations can enhance transparency, support troubleshooting, and facilitate governance. Visualizing coarse-grained and fine-grained lineage requires different techniques tailored to their respective granularity, while integrations with business intelligence (BI) and governance systems ensure actionable insights. This section explores visualization techniques for coarse-grained and fine-grained lineage and discusses their integration into operational and decision-making workflows [14].

## 6.1 Coarse-Grained Lineage Visualization

Coarse-grained lineage visualization often involves rendering directed acyclic graphs (DAGs) where datasets or tables are represented as nodes, and edges represent data dependencies. These visualizations provide a clear, high-level understanding of data flows within a system and are instrumental for tasks such as governance, impact analysis, and pipeline auditing.

Hierarchical graph views are one of the most common methods for visualizing coarse-grained lineage. These layered graphs position upstream datasets at the top and downstream consumers at the bottom. The hierarchical structure allows users to trace data dependencies by expanding or collapsing specific nodes to control the level of detail. For example, a dataset node can be expanded to reveal associated metadata, such as schema details or transformation types, aiding in impact analysis and debugging.

Dependency matrices or heatmaps provide an alternative to graph-based representations. In this approach, datasets are represented along the rows and columns of a matrix, with cells indicating the presence or strength of a dependency. Heatmaps can visually highlight frequently accessed datasets or critical data paths. For example, heavily referenced datasets may appear as highly saturated cells, signaling their importance in downstream workflows. This visualization is particularly effective for organizations managing large numbers of datasets and requiring a condensed, tabular representation of lineage [15].

Time-based lineage views add a temporal dimension to lineage visualization. These views allow users to explore how lineage evolves over time, tracking changes to dataset dependencies and transformations across pipeline runs. For instance, users can "rewind" lineage graphs to examine historical states, which is particularly useful for auditing or debugging issues that occurred in previous versions of a pipeline. This temporal context enables a more comprehensive understanding of how data dependencies and processes have evolved over time.

## 6.2 Fine-Grained Lineage Visualization

Visualizing fine-grained lineage presents unique challenges due to the scale and granularity of the data. Directly representing record- or cell-level dependencies in a node-link diagram is often impractical, as the resulting graph would be too dense and overwhelming. Instead, fine-grained lineage visualizations use techniques designed to simplify and contextualize the data.

Interactive filtering and sampling allow users to focus on specific subsets of the lineage graph. For example, a user investigating a particular anomaly can select a single record or value in a target dataset and request its lineage. The system dynamically generates a subgraph that traces the selected record's dependencies across upstream datasets and transformations. This focused approach avoids overwhelming users with excessive detail while providing the precise information needed for debugging and analysis.

Aggregation and summarization are also commonly employed to manage the complexity of fine-grained lineage. Rather than displaying every individual record, these visualizations present aggregated statistics. For instance, a visualization might indicate that "90% of records in Dataset Z originate from Dataset Y, and 10% come from Dataset X." Users can drill down further if more detail is required, allowing them to maintain high-level oversight while still having access to granular insights when needed.

Multi-level drill-down combines the strengths of coarse- and fine-grained visualizations by enabling users to start with a high-level view and progressively explore deeper levels of detail. For example, users might begin by viewing dataset-level dependencies, then drill into specific transformations, and finally examine record-level lineage within those transformations. This hierarchical navigation aligns with user workflows and minimizes cognitive overload, ensuring a more intuitive exploration experience.

### 6.3 Tooling and Integration with BI and Governance Systems

To maximize the utility of lineage data, it must be integrated into existing BI tools and governance systems. Such integration ensures that lineage insights are accessible to both technical and non-technical users and can be seamlessly applied to decision-making and operational workflows.

Lineage overlays in BI dashboards are a practical example of integration. Business analysts using a BI tool can hover over a specific metric to view its upstream datasets, transformations, and filtering criteria. For instance, a dashboard visualizing sales metrics might display lineage information that identifies the source tables and filters used in its computation. This transparency builds trust in data insights and reduces the need for technical users to manually verify data origins.

Collaboration and annotation features enhance the usability of lineage tools by enabling users to document insights, add context, and facilitate knowledge sharing. For example, a data engineer investigating a dataset deprecation can annotate the lineage graph with notes explaining the reasons for the deprecation and its potential downstream impacts. These annotations enrich lineage data with human context, improving knowledge transfer across teams and fostering a shared understanding of data workflows.

Programmatic access through APIs is critical for automating lineage-related tasks and integrating lineage data into broader systems. Lineage tools often expose metadata through REST APIs, GraphQL endpoints, or specialized query languages. For example, a CI/CD pipeline might query the lineage system to determine whether a schema change affects critical downstream datasets, triggering additional validation steps if necessary. Similarly, governance workflows can use APIs to monitor lineage for compliance purposes, such as ensuring that sensitive data does not propagate into unauthorized datasets [2].

### 6.4 Examples of Lineage Visualization and Integration Tools

Table 13 summarizes common visualization techniques for lineage data, while Table 14 outlines strategies for integrating lineage tools with BI and governance systems.

**Table 13** Lineage Visualization Techniques and Applications

| **Technique** | **Description and Use Cases** |
|---|---|
| Hierarchical Graph Views | Layered graphs showing dataset dependencies. Useful for impact analysis and governance workflows. Example: Expanding nodes to reveal upstream transformations. |
| Dependency Matrices or Heatmaps | Tabular representations of lineage with visual emphasis on critical datasets. Effective for managing large numbers of datasets. |
| Time-Based Lineage Views | Temporal graphs showing the evolution of data dependencies over time. Useful for auditing and debugging historical pipeline runs. |
| Interactive Filtering and Sampling | Subgraph generation for specific records or values. Useful for debugging anomalies in downstream data. |
| Aggregation and Summarization | Summarized views of lineage with drill-down options. Useful for maintaining high-level oversight while supporting granular analysis. |

**Table 14** Integration Strategies for Lineage Tools with BI and Governance Systems

| **Integration Approach** | **Description and Benefits** |
|---|---|
| Lineage Overlays in BI Dashboards | Embeds lineage information into BI tools, providing transparency for non-technical users. Example: Hovering over a metric to view its upstream datasets and transformations. |
| Collaboration and Annotation | Allows users to document lineage graphs with notes and comments. Useful for facilitating knowledge transfer and providing context for decisions. |
| Programmatic Access via APIs | Provides REST or GraphQL endpoints for integrating lineage data with CI/CD pipelines or governance workflows. Useful for automated impact analysis and compliance monitoring. |

## 7 Conclusions

This paper has examined the methodologies and challenges involved in capturing and utilizing data lineage in modern data lakes, with a particular focus on both coarse-grained and fine-grained approaches. We have reviewed key techniques for lineage capture, including the instrumentation of ETL frameworks, the parsing of declarative transformations such as SQL queries, and runtime data tracking. Each method has distinct strengths and limitations in terms of granularity, computational overhead, and feasibility. Furthermore, we analyzed approaches for modeling and storing lineage at varying levels of granularity, exploring solutions that balance scalability, performance, and usability. Finally, we discussed state-of-the-art techniques for visualizing and interacting with lineage data, as well as their integration into business intelligence (BI) and governance tools to enhance accessibility and utility.

Both coarse-grained and fine-grained lineage play complementary roles in data management. Coarse-grained lineage offers a high-level overview of dataset dependencies and transformations, making it indispensable for governance, impact analysis, and regulatory compliance. In contrast, fine-grained lineage provides the detailed traceability necessary for debugging, root cause analysis, and precise auditing. The ability to navigate between these perspectives ensures that organizations can address a diverse range of use cases effectively, from high-level oversight to granular troubleshooting.

Capturing lineage data involves inherent trade-offs depending on the method employed. Logical plan analysis and instrumentation are efficient for coarse-grained lineage but may lack the depth required for fine-grained traceability. Runtime tracking, while capable of capturing detailed lineage at the record or cell level, introduces considerable computational and storage overhead, particularly in large-scale data systems. The choice of an appropriate method often depends on the specific requirements of the organization, including its need for granularity, the scale of its data lake, and operational constraints.

Modeling and storing fine-grained lineage present significant technical challenges due to the sheer volume of metadata generated. Naïve approaches, such as storing record-level provenance for every transformation, are typically infeasible for large-scale systems. Techniques like lineage compression, grouping of similar records, and delta-based storage can help mitigate these challenges. By optimizing storage and retrieval methods, these approaches enable scalable lineage tracking while preserving the necessary level of detail.

Visualization is central to the usability of lineage data. Coarse-grained lineage visualizations, such as directed acyclic graphs and dependency matrices, provide intuitive representations of dataset-level dependencies. For fine-grained lineage, interactive tools that allow users to filter, aggregate, and drill down into specific details are essential for managing complexity. Integration with BI platforms and governance systems further enhances the usability of lineage data by embedding it into operational workflows, enabling users to access actionable insights directly within familiar tools.

Standardized formats and APIs for lineage metadata would improve interoperability across tools and platforms, reducing implementation complexity in heterogeneous environments. Furthermore, adaptive lineage systems that dynamically adjust granularity based on data sensitivity or compliance requirements could optimize resource allocation while ensuring that critical datasets are appropriately tracked.

The integration of lineage systems with data contracts and semantic metadata layers presents an opportunity to enhance governance frameworks. Combining lineage with contractual guarantees about data quality and compliance could enable automatic validation of transformations and early detection of discrepancies. As lineage graphs grow increasingly complex, machine learning and automated reasoning could play an instrumental role in deriving actionable insights, detecting anomalies, and predicting downstream impacts of changes in data pipelines.

Data lineage is a cornerstone of modern data management, providing essential transparency and traceability across the data lifecycle. As organizations continue to scale their data operations, advancements in lineage capture, modeling, visualization, and integration will be critical to addressing emerging challenges. By pursuing the research directions outlined here, the next generation of lineage systems can achieve greater scalability, usability, and impact, empowering organizations to manage their data ecosystems with confidence and precision.

**Author details**
Senior Manager, Data Engineering, LogMeIn Inc. https://orcid.org/0009-0008-0726-5403.

**References**

1. Backes, M., Grimm, N., Kate, A.: Data lineage in malicious environments. IEEE Transactions on Dependable and Secure Computing **13**(2), 178–191 (2015)

2. Woodruff, A., Stonebraker, M.: Supporting fine-grained data lineage in a database visualization environment. In: Proceedings 13th International Conference on Data Engineering, pp. 91–102 (1997). IEEE
3. Palyvos-Giannas, D., Gulisano, V., Papatriantafilou, M.: Genealog: Fine-grained data streaming provenance at the edge. In: Proceedings of the 19th International Middleware Conference, pp. 227–238 (2018)
4. Missier, P., Paton, N.W., Belhajjame, K.: Fine-grained and efficient lineage querying of collection-based workflow provenance. In: Proceedings of the 13th International Conference on Extending Database Technology, pp. 299–310 (2010)
5. Ikeda, R., Widom, J.: Data lineage: A survey. Technical report, Stanford InfoLab (2009)
6. Huo, Y., Lu, Y., Niu, Y., Lu, Z., Wen, J.-R.: Coarse-to-fine grained classification. In: Proceedings of the 42nd International ACM SIGIR Conference on Research and Development in Information Retrieval, pp. 1033–1036 (2019)
7. Qin, Z., Chen, J., Jiang, Z., Yu, X., Hu, C., Ma, Y., Miao, S., Zhou, R.: Learning fine-grained estimation of physiological states from coarse-grained labels by distribution restoration. Scientific Reports **10**(1), 21947 (2020)
8. Zhang, Z., Sparks, E.R., Franklin, M.J.: Diagnosing machine learning pipelines with fine-grained lineage. In: Proceedings of the 26th International Symposium on High-performance Parallel and Distributed Computing, pp. 143–153 (2017)
9. Dasgupta, R., Ganesan, B., Kannan, A., Reinwald, B., Kumar, A.: Fine grained classification of personal data entities. arXiv preprint arXiv:1811.09368 (2018)
10. Ruan, P., Chen, G., Dinh, T.T.A., Lin, Q., Ooi, B.C., Zhang, M.: Fine-grained, secure and efficient data provenance on blockchain systems. Proceedings of the VLDB Endowment **12**(9), 975–988 (2019)
11. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. The VLDB journal **12**(1), 41–58 (2003)
12. Soedarmadji, E., Stein, H.S., Suram, S.K., Guevarra, D., Gregoire, J.M.: Tracking materials science data lineage to manage millions of materials experiments and analyses. npj Computational Materials **5**(1), 79 (2019)
13. Aggarwal, C.C.: Trio a system for data uncertainty and lineage. In: Managing and Mining Uncertain Data, pp. 1–35. Springer, ??? (2009)
14. Bose, R.: A conceptual framework for composing and managing scientific data lineage. In: Proceedings 14th International Conference on Scientific and Statistical Database Management, pp. 15–19 (2002). IEEE
15. Tang, M., Shao, S., Yang, W., Liang, Y., Yu, Y., Saha, B., Hyun, D.: Sac: A system for big data lineage tracking. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE), pp. 1964–1967 (2019). IEEE