# Data Guard: A Fine-grained Purpose-based Access Control System for Large Data Warehouses

Khai Tran\*, Sudarshan Vasudevan\*, Pratham Desai\*, Alex Gorelik, Mayank Ahuja, Athrey Yadatore Venkateshababu, Mohit Verma, Dichao Hu, Walaa Eldin Moustafa, Vasanth Rajamani<sup>†‡</sup>, Ankit Gupta, Issac Buenrostro, Kalinda Raina

LinkedIn Coporation, OpenAI Inc<sup>†</sup>

{khtran,suvasudevan,padesai,agorelik,mahuja,avenkateshababu,moverma,dihu,wmoustafa}@linkedin.com vasanth@openai.com,{aigupta,ibuenros,kraina}@linkedin.com

#### **ABSTRACT**

The last few years have witnessed a spate of data protection regulations in conjunction with an ever-growing appetite for data usage in large businesses, thus presenting significant challenges for businesses to maintain compliance. To address this conflict, we present Data Guard - a fine-grained, purpose-based access control system for large data warehouses. Data Guard enables authoring policies based on semantic descriptions of data and purpose of data access. Data Guard then translates these policies into SQL views that mask data from the underlying warehouse tables. At access time, Data Guard ensures compliance by transparently routing each table access to the appropriate data-masking view based on the purpose of the access, thus minimizing the effort of adopting Data Guard in existing applications. Our enforcement solution allows masking data at much finer granularities than what traditional solutions allow. In addition to row and column level data masking, Data Guard can mask data at the sub-cell level for columns with non-atomic data types such as structs, arrays, and maps. This fine-grained masking allows Data Guard to preserve data utility for consumers while ensuring compliance. We implemented a number of performance optimizations to minimize the overhead of data masking operations. We perform numerous experiments to identify the key factors that influence the data masking overhead and demonstrate the efficiency of our implementation.

#### **PVLDB Reference Format:**

Khai Tran, Sudarshan Vasudevan, Pratham Desai, Alex Gorelik, Mayank Ahuja, Athrey Yadatore Venkateshababu, Mohit Verma, Dichao Hu, Walaa Eldin Moustafa, Vasanth Rajamani, Ankit Gupta, Issac Buenrostro, Kalinda Raina. Data Guard: A Fine-grained Purpose-based Access Control System for Large Data Warehouses. PVLDB, 14(1): XXX-XXX, 2020. doi:XX.XX/XXX.XX

#### **PVLDB Artifact Availability:**

The source code, data, and/or other artifacts are available at https://github.com/linkedin/dataguard-udfs.

#### 1 INTRODUCTION

The last two decades have witnessed a tremendous growth both in the collection and usage of data to power use cases such as business analytics, recommender systems and personalization. A significant amount of data that businesses collect from end users to power these applications is sensitive or personally identifiable. Understandably, there have been concerns from regulators, privacy advocates and various other citizen groups alike around the lack of transparency in the data collection and data handling practices adopted by businesses. Regulations such as GDPR [23], CCPA [10], HIPAA [25], DMA [18] and EU AI Act [20] have been enacted into law in response to these privacy concerns and to drive transparency into the data collection and data usage practices of businesses. Specifically, these regulations require businesses to limit the usage of user data only for the purposes for which it is collected. As an example, the GDPR regulation requires businesses to seek consent from their end users (or formally, data subjects) in order to use sensitive attributes such as age and gender for advertising purposes. At the same time, the use of this data is often permitted in anonymized form to enable business use cases intended for demographic analysis.

To address these privacy concerns and ensure compliance with regulations, several solutions have been proposed [8, 13, 40] to enforce access control policies based on the purpose of data access. The systems are commonly referred to as *Purpose-based Access Control* (PBAC) systems. Depending on the granularity of enforcement, these systems can either authorize/deny access to an entire dataset (*coarse-grained* access control), or they can apply dynamic masking to hide contents of individual rows, columns or cells within the dataset (*fine-grained* access control). Enforcing policies dictated by these regulations across a large organization can be a challenging endeavor for the following reasons:

- (1) A typical modern data stack involves multiple data processing systems such as Apache Spark [33], Trino [36] and Apache Flink [21]. Thus, any access control solution is required to apply data access policies consistently across a diversity of data processing systems used in an organization.
- (2) Many data-centric organizations commonly support 10-100s of thousands of datasets, 10-100s of use cases (or *purposes*) and 100s-1000s of data pipelines accessing exabyte-scale data every day. A practical solution for access control must therefore be sufficiently scalable to meet these demands.

<sup>\*</sup>Equal contributors.

<sup>&</sup>lt;sup>‡</sup>Work done while the author was at LinkedIn Coporation.

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. 14, No. 1 ISSN 2150-8097. doi:XX.XX/XXX.XX

(3) The data models underlying these datasets can be varied and complex. For instance, it is common to model data using non-atomic data types like arrays, maps, and structs. Furthermore, it is also common for AI/ML workloads to employ application-specific descriptions of data overlaid on basic data structures such as arrays and structs. A practical solution for access control must support granular datamasking capabilities within such complex data structures to maximize data utility while ensuring compliance.

Existing solutions have limited fine-grained access control capabilities and fall short in addressing one or more of the aforementioned challenges. Furthermore, we are not aware of any prior literature describing the myriad practical challenges encountered when deploying an access-control solution across a large data warehouse.

There are additional challenges for data-centric applications (such as LinkedIn among others) that employ a consent framework that controls how end-user data is used for different use cases. For instance, LinkedIn provides granular consent settings to its over one billion users (or data subjects) giving them control on how their data is used by LinkedIn for different use cases. Thus, data present in warehouse tables need to be dynamically masked based on data subject preferences in addition to the purpose of access. In Figure 1, we illustrate how access control based on data subject preferences differs from the more common row and column-level access control used in practice. In this example, the member profiles table storing data subject information is queried for two different use cases, viz. "Ads" and "Jobs". Data subject preferences controlling usage of different attributes of their data are stored in the *member\_settings* table. Usage of data subject information for the Ads use case is controlled by two settings, allowEducationForAds and allowEmployerForAds, while that for Jobs use case is controlled by the settings allowEducationForJobs and allowEmployerForJobs. Based on the setting values for allowAgeEducationForAds and allowEmployerForAds, the education attribute associated with memberId = 123 and employer attribute corresponding to member Id = 234 in the member profiles table need to be masked for the Ads use case. Similarly, the education attribute of member 234 and the employer attribute of members 123 need to be masked for Jobs use case based on the corresponding values for jobs-related settings.

To address these challenges, we designed and implemented Data Guard, a fine-grained purpose-based access control system for large data warehouses. The system translates policies specified in a domain-specific language (DSL) into purpose-specific views that mask data at access time. This translation is facilitated by maintaining a taxonomy of data classification labels (henceforth, referred to as *policy labels*), which are used to provide semantic description of the data elements (e.g. rows, columns, and cells in a table) present in the warehouse. The policies use purpose and policy labels to define the conditions under which access to a data element described using an annotation is allowed or denied. A *policy compiler* translates these rules into a purpose-specific view SQL and registered with the warehouse. At access time, a view routing layer transparently routes accesses to the appropriate data-masking view based on the purpose attached to the access.

The key contributions of this paper are as follows:

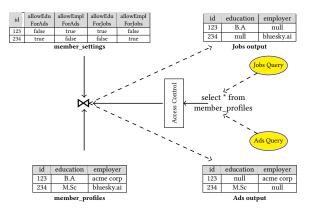


Figure 1: An example highlighting dynamic masking of data yielding different outputs based on purpose. The access control layer enforces member preferences (from the settings table) at data access time and selectively masks columns in the output.

- To the best of our knowledge, Data Guard is the first practical implementation of a fine-grained, purpose-based access control system that has been successfully deployed across a large enterprise warehouse. Data Guard has been rolled out at scale across LinkedIn's warehouse ensuring all data accesses using Spark and Trino the data processing engines used to query the warehouse data at LinkedIn are compliant with our data usage policies.
- Data Guard introduces a number of innovations that ensure both compliance and ease of use for warehouse users and application developers such as: (i) High-level abstractions that allow policies to be defined using purposes and semantic data labels. These policies are translated by Data Guard into data-masking views that serve as the data access API. (ii) A routing layer referred to as ViewShift which transparently routes accesses to warehouse tables from user queries to the appropriate data-masking views provisioned by Data Guard. (iii) View optimizations to minimize performance overhead in existing data processing jobs, such as the use of memory-efficient bitmaps in lieu of expensive table joins. (iv) Use of engine-agnostic views which allow data-masking views to be executable across different data processing engines (Spark and Trino at LinkedIn).
- To the best of our knowledge, Data Guard is the first practical implementation of an access control system that allows data-masking at the sub-cell level. Data Guard achieves this via an expressive grammar that allows it to locate and mask deeply nested data elements inside composite attributes. These granular masking capabilities of Data Guard allows data consumers to maximize value from the underlying data while still ensuring compliance.

In the rest of the paper, we provide a detailed description of the design of Data Guard, the key architectural choices in our system along with the reasons that motivated these choices, and the system implementation details. We also present data that quantifies the performance overhead of enforcement during data access and conclude with interesting areas of future work.

#### 2 RELATED WORK

Fine-grained access control in database management systems (DBMS) has been a well-studied topic and numerous solutions [1, 30, 34, 40] have been proposed to tackle this problem. In [34], the authors propose a solution for the INGRES relational database that dynamically modifies user queries based on access restrictions defined on attributes in a relation. In [40], the authors propose a solution for Apache Spark applications that adds an access control enforcement stage in Spark's Catalyst optimizer. This stage dynamically modifies non-compliant user queries into compliant ones. Unlike [34, 40], [30] proposes using static authorization views which encode data access policies and admits a user query only if it can be mapped to an existing authorization view. A key limitation of these solutions is that they do not generalize to modern data stacks where the data models can be non-relational or can involve a diverse set of data processing engines.

The work proposed in [1] shares several advantages with Data Guard's design including: (i) its ability to define policies that use semantic descriptions of data, and (ii) the ability to apply user preferences to dynamically filter tuples and attributes in a dataset. However, unlike Data Guard, the system described in [1] modifies user queries via dynamically created views. Indeed, we made an intentional design choice in Data Guard to create static versioned views to ensure query reproducibility and debuggability.

Several modern data warehouses such as Snowflake [32], Databricks [16] and Amazon Redshift [3] provide capabilities to dynamically mask data based on semantic tags associated with attributes. However, they suffer from the same drawbacks as [1] in their inability to support query reproducibility and debuggability. Further, none of these solutions offer the scalability needed to support policies that require large joins with auxiliary tables such as the user consent tables used in our environment. In addition, these systems do not support data masking at the sub-cell level, a critical capability that Data Guard offers in order to maximize data utility.

Purpose-based access control for relational databases have been proposed in [7, 8] and further extended to non-relational data in [9]. These solutions describe a mechanism for associating an intended purpose with a data object (a relation, a tuple or an attribute) that is co-located with data. Further, they provide an algorithm to modify user query to filter data based on a comparison check between the intended purpose associated with the data object and the data accessor's purpose. However, the proposed solutions only allow for filtering at a tuple-level granularity and do not support columnar or sub-cell-level filtering of data. Further, a major disadvantage of these solutions is that they require data to be rewritten when purposes change, which is impractical in large data warehouses where such changes are common.

There have been numerous works [15, 29, 38] that provide policy language specifications for access control. The policy language used in Data Guard is similar in nature to the P3P policy language specification in [15] in terms of its ability to use semantic tags to describe data items for which a policy is being defined. However, none of these solutions provide off-the-shelf language parsers and compilers required to translate the policies into machine representations for policy enforcement. Further, it was more cost-effective

for us to define a constrained policy language and build a policy parser that fits our current needs.

Finally, there have been a number of works addressing privacy in databases e.g. [2, 11, 19]. [2] proposes a high-level system design for a privacy enforcing database and serves as a foundation for the work in [1]. [19] introduces the notion of differential privacy and provides a mathematical definition for privacy loss in the query output. [11] provides a theoretical study of different metrics for measuring privacy in statistical databases and studies candidate techniques for privatization based on these metrics. While none of these works address the problem of access control studied in this paper, we believe Data Guard's design is flexible enough to accommodate enforcement of policies that require privatization of query results.

### 3 ARCHITECTURE OVERVIEW

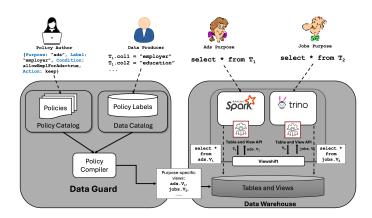


Figure 2: Data Guard System Architecture.

Figure 2 shows the overall system architecture of Data Guard. Data producers provide semantic descriptions of data in the form of policy labels that are assigned to fields in a table. Fields refer to data elements in a table and represent the unit at which data masking is applied. Thus, a field can reference a column, a row, a cell or portion of data within a cell. The mapping of policy labels to fields is stored in a Data Catalog as shown in Figure 2. Policies in Data Guard are authored in a domain-specific language (DSL) and stored in a Policy Catalog which provides APIs to query policies from the catalog. Each policy is assigned to a purpose and policy label pair. A policy definition has an associated boolean condition that determines whether a data element with the corresponding policy label is masked or preserved during data access. Figure 2 shows an example policy that is assigned to ads purpose and employer label. The Policy Compiler consumes the policy label-field mapping and policy definitions as inputs and generates purpose-specific static views which are registered with the data warehouse. User queries submitted via Spark or Trino that access tables (e.g.  $T_1$  and  $T_2$  in Figure 2) are transparently redirected by a routing module called ViewShift to the appropriate data-masking views (jobs.V1 and ads.V<sub>2</sub> respectively in Figure 2) based on the accessor's purpose (i.e jobs and ads in Figure 2). We next describe the access control model and system implementation details of Data Guard architecture.

#### 4 ACCESS CONTROL MODEL

In this section, we introduce the main abstractions that enable purpose-limitation and fine-grained access control capabilities of Data Guard: (i) data access policies, (ii) data-masking views, and (iii) field paths.

#### 4.1 Data Access Policies

A data access policy in Data Guard specifies: (i) a *purpose* under which the policy is applicable, (ii) a *label* that determines the data elements to which the policy applies, and (iii) a set of access control *rules* that specify the conditions under which access to a data element with a given label is allowed or denied. We next introduce each of these concepts followed by a formal definition of a data access policy.

4.1.1 Purpose. The access control model described in this paper is motivated by the purpose limitation principle introduced in the GDPR regulation [23]. Purpose limitation requires that data is collected from data subjects (i.e. individuals whose personal data is collected) for an explicitly declared purpose and is only processed for use cases compatible with the purpose under which it is collected. Further, usage of an individual's data for newer purposes is driven based on consents from the individual. At LinkedIn, a purpose is used to indicate the business justification for data collection or data processing. Purposes can either map to an external product such as advertising and jobs, or correspond to internal use cases such as security and business analytics.

At LinkedIn, purposes are organized as a directed acyclic graph which enables policy inheritance across purposes and minimizes policy duplication. For simplicity of discussion however, we assume henceforth that purposes are arbitrary string literals (such as *ads* and *jobs*) for the remainder of the paper.

Purpose limitation is ensured by requiring each data access to specify a valid purpose. At LinkedIn, all production workloads access warehouse data using service accounts. Similar to role-based access control systems where roles are assigned to a user, we assign a purpose to each service account. The purpose of the service account determines the access control policies that are enforced by Data Guard on the accessed tables.

4.1.2 Policy Labels. Data producers provide semantic description of data in the tables they produce using policy labels. Figure 2 shows a data producer assigning policy labels employer and education assigned to  $col_1$  and  $col_2$  of table  $T_1$ . In general, data producers can assign policy labels either to: (i) a table, (ii) column in a table (either a top-level or a nested column), (iii) cells, or (iv) sub-cell-level data elements within arrays, structs and maps. As we will see in Section 4.3, this fine-grained assignment of policy labels allows Data Guard to apply data masking at much finer granularities than existing solutions.

Figure 2 also shows that policy labels are stored as metadata inside a *Data Catalog*. This design choice is motivated by recent trends in data architecture where data catalogs such as Horizon [26], Unity [37] and DataHub [17] have emerged as the foundational infrastructure component to address data discovery and data governance needs of large organizations. These catalogs serve as the inventory of the critical data assets (e.g. warehouse tables) of an

organization and store a variety of metadata such as schemas, lineages and ownership associated with these assets. These catalogs are therefore well-suited for storing compliance metadata such as *policy labels*.

At LinkedIn, policy labels are carefully curated and organized into a hierarchical structure referred to as *ontology*. An ontology-based organization not only classifies data in our ecosystem, but also captures relationships between these data classes. Ontology also supports advanced use cases at LinkedIn related to reasoning and inference on compliance metadata. A detailed discussion on the design of our ontology is beyond the scope of this paper. We note however that the access control model proposed in this paper does not require an ontology-based organization of policy labels. For simplicity, we therefore assume that policy labels are arbitrary string values for the remainder of the paper.

Policy labels link policies to the data elements (e.g. tables, columns, and cells) to which policies apply. The cardinality of policy labels is typically much smaller than the number of data assets and fields across all tables in a warehouse. Defining policies using policy labels significantly reduces policy duplication and allows policies to be consistently applied across tables in a warehouse as well as across non-warehouse data assets such as online data stores and data streams. Further, it automatically ensures compliance of newly created tables containing data elements tagged with one or more previously defined labels.

4.1.3 Policy Language. Data masking in Data Guard is controlled using rules that are assigned to a purpose and policy label pair.

DEFINITION 1. Let  $\mathcal P$  denote the set of purposes and  $\mathcal L$  denote the set of policy labels. A policy in Data Guard is a tuple  $(p,l,\langle c,a\rangle)$  where  $p\in \mathcal P$  and  $l\in \mathcal L$ .  $\langle c,a\rangle$  is a condition and action pair (or, a rule) where:

- c is a compound SQL predicate composed of simple predicates connected using the boolean operators  $\{AND, OR, NOT\}$ . Each simple predicate in c is of the form  $(x \circ y)$  where:
  - (1) where  $x \in X$  is a set of attributes (defined below),
  - (2)  $y \in dom(x)$  is the domain of values of attribute x.
  - (3) o represents a relational operator from the set of relational operators {=, ≠, <, >, ≤, ≥, BETWEEN, IN, LIKE, ISNULL}.
- a ∈ A represents an action from the set of actions A that the access control system must take when c is true.

Thus, the example policy shown in Figure 2 can be represented as a tuple:  $(ads, employer, \langle allowEmplForAds = true, KEEP \rangle)$ .

The set X of attributes over which policy conditions are defined is a union of data subject attributes (e.g. opt-in/opt-out consents, user location), data accessor attributes (e.g. accessor's role and location), and system attributes (e.g. availability/security zone where the system is located in). The set of actions  $\mathcal A$  of interest in this paper is the set  $\{KEEP, MASK\}$ , where the action KEEP allows access to the target data while MASK redacts the data. For simplicity, we only consider a specific form of data redaction in this paper, where the target data element is either replaced with NULL value or the tuple containing the target data element is filtered from the result set. Assignment of non-null replacement values to target data elements

is an area of ongoing work and will be discussed in greater detail in Section 7.

At LinkedIn, policies are managed like source code and policy changes are manually reviewed before they are committed to a code repository. Further, we run a number of validation checks on policies to avoid duplication and ensure that policies can be successfully parsed and compiled by the policy compiler. The changes committed to the repository are pushed to a *Policy Catalog* which provides APIs to query policies by purpose and policy labels.

Without loss of generality, all policies considered in the remainder of the paper are *KEEP* policies i.e. the data element referenced by a policy label is preserved if the policy condition evaluates to true and masked otherwise. We further restrict our attention to policies based on data subject consents for the remainder of this paper. In doing so, we aim to highlight:

- the scalability challenges that need to be addressed to enforce consents of a large population of data subjects during data accesses, and
- the unique challenges such policies introduce for enabling fine-grained data masking.

## 4.2 Data-Masking Views

Data Guard uses views as the interface for access control. Each data-masking view is a SQL representation of applicable data policies on a table for a given purpose. The applicable policies for a given purpose are found by matching policy labels assigned to fields in a table against the labels assigned to policies. The matched policies are then translated to SQL by the *Policy Compiler*, the implementation of which will be discussed in detail in Section 5.1.1.

We illustrate an example data-masking view created by Data Guard in Figure 2. Let us assume that the table  $T_1$  shown in the figure is keyed by an id column containing data subject identifiers. The figure also shows that  $T_1$  contains a column  $col_2$  that has education data with an applicable policy for the ads purpose. For simplicity, let us assume that all data subject attributes referenced in policy definitions are stored in the  $member\_settings$  table which is also keyed by the data subject identifier. A naively implemented policy compiler generates the following view SQL for ads purpose which masks  $col_2$  based on the value of the attribute allowGenderForAds in the  $member\_settings$  table:

```
SELECT T1.id, T1.col1, CASE WHEN allowGenderForAds = true
THEN T1.col2 ELSE NULL END AS col2
FROM T1 JOIN member_settings T2 ON T1.id = T2.id;
```

Figure 3: A data-masking view for ads purpose

The view SQL shown in Figure 3, while inefficient, has the important property that it is schema-preserving i.e. it has the same schema as the underlying table  $T_1$ . This invariant is maintained across all views created by Data Guard. The schema-preserving property ensures that applications can switch consumption from tables to views without needing to make code changes. Each view generated by Data Guard is registered within a purpose-specific database (e.g.  $ads.V_1$  in Figure 2), which facilitates search and discovery of views in the data catalog. In Section 5, we discuss a number of optimizations which will be used to rewrite the view SQL in Figure 3 in order to make it performant.

The decision to use SQL views as the interface for access control was motivated by the following reasons:

- Portability: Views are engine-agnostic and work out of the box with engines like Spark and Trino.
- Debuggability: Views make masking logic visible to end users and allow consumers to reason about what data is being filtered and why.
- Version control: Views can be versioned in the same manner as software artifacts. Thus, changes to view logic (due to policy and label changes) can be tested by data consumers before they are deployed in production.
- Agility: Implementing changes to policy compiler is much faster than making changes to the compute engine code, which caters to a much larger set of use cases beyond access control.
- Optimizations: A dedicated policy compilation layer provides the flexibility of implementing custom optimizations
  (e.g. bitmap optimization discussed in Section 5.1.2) when generating view SQL. Such use-case specific optimizations are otherwise non-trivial to implement inside general-purpose compute engines.

As shown in Figure 2, accesses to tables are routed to an appropriate masking view based on the purpose of access using a component called *ViewShift*. ViewShift along with the schema-preserving property of the masking views allows applications to seamlessly switch consumption from tables to views without incurring significant migration costs.

### 4.3 Field Paths

4.3.1 Motivation. Existing solutions [1, 30, 34, 40] support data masking at row, column and cell-level granularity. These masking operations are typically accomplished using traditional projection and selection operators. As mentioned in Section 1, Data Guard's access control model supports masking data at much finer granularities than is possible with previous solutions. This need for finer grained data masking is motivated by prevalence of non-relational data in LinkedIn's warehouse (and indeed, many modern data warehouses [32] and lakehouses [16]). It is common to model data using relation-valued attributes and collection types such as arrays and maps. Masking data selectively inside such data structures cannot be done using relational operators such as projection and selection alone.

Let us consider an example relation  $\mathcal{R}$  with the following schema:

 $col_1$ : VARCHAR

 $col_2$ : STRUCT< $field_{21}$ :BIGINT,  $field_{22}$ :VARCHAR>

 $col_3$ : ARRAY<STRUCT< $field_{31}$ :VARCHAR,  $field_{32}$ :DOUBLE»

 $col_4$ : MAP<VARCHAR, ARRAY<STRUCT< $field_{41}$ :VARCHAR,

 $field_{42}$ :BOOLEAN»>

Figure 4 shows an instance of  $\mathcal{R}$ .

In order to mask data at a sub-cell granularity (e.g.  $field_{31}$ ,  $field_{42}$  shown in Figure 4), we need an operator for selecting this data. There have been prior proposals [12] which introduce recursive selection and projection operators, which can be used to select and mask data at sub-cell granularity. While there have been efforts to extend SQL language to support recursive operations [27], very few commercial and open-source systems support such extensions.

col <sub>1</sub>	cc		col <sub>3</sub>		$col_4$	
corı	field <sub>21</sub>	field <sub>22</sub>				
abc	123	foo	field <sub>31</sub> field <sub>32</sub> s1 113.2		NULL	\$.col <sub>1</sub>
def	243	bar	NULL		NULL	\$.[?(@.col <sub>1</sub> =' def')]
ghj	123	bar	field <sub>31</sub> field <sub>32</sub> s1 345.2 s3 212.0	key k1	value           field41         field42           v1         true           v2         false           NULL	$ \begin{array}{ c c c c c c c c c c c c c c c c c c c$

Figure 4: Example of a nested relation. Data Guard's field path expressions allow masking of data inside nested attributes such as structs, arrays and maps.

While introducing these extensions to existing open source systems like Spark and Trino is in theory possible, it would require a significant modification of the SQL standard adopted by these systems and a non-trivial investment of effort to drive alignment across the open-source community. On the other hand, there have been numerous path DSLs such as XMLPath [39] and JsonPath [24] that have been developed and successfully adopted to support elementwise operators. Given these trade-offs, using a DSL to evaluate field paths efficiently during data access was a cost-effective alternative. Leveraging existing DSLs such as XPath and JsonPath was not an option due to several limitations that prevent their usage for structured data handling. XPath is intended exclusively for processing XML documents. JsonPath is schema-unaware and does not distinguish between types such as maps and structs. Nevertheless, we use JsonPath to guide the design of the field path DSL described next.

4.3.2 Field Path Expressions. In this section, we provide a formal definition of *field path* that is used to select data at a sub-cell-level for non-atomic data types such as structs, arrays and maps.

DEFINITION 2. A field path is a sequence of operators  $P_1, P_2, \ldots, P_n$ , where each operator  $P_i$  operates on the result set of  $P_{i-1}$  and has one of the following types:

- (1) **Root access operator**: Denoted by a special symbol \$, this operator is an identity operator whose result set is the input relation. The root access operator is always the first operator in the sequence of operators for any given field path.
- (2) **Transform operator**: A transform operator is immediately preceded by either a root access or another transform operator and is prefixed with a . symbol. There are three types of transform operators:
  - (a) **Dereference operator**: This operator is of the form . < name > and projects a sub-attribute of a composite attribute.
  - (b) Filter operator: This operator selects a subset of a relation that satisfies a given condition. The operator is of the form [?(< condition >)], where condition is a SQL predicate string on the input relation.
  - (c) Unnest operator: a cell-wise transform operator that applies all subsequent operators on each cell of inner relations in collection-type attributes such as arrays and maps. There are three forms of unnest operators:
    - (i) [item]: an operator that applies to array types and provides access to the array items,
    - (ii) [key]: an operator that applies to map types and provides access to the map keys, and
    - (iii) [value]: an operator that applies to map types and provides access to the map values.

- 4.3.3 Examples. We next provide examples of field path expressions that demonstrate their ability to select data elements from the different attributes of the relation shown in Figure 4.
  - A dereference operator following a root access operator can be used to select an attribute from a relation. For example, \$.col<sub>1</sub> selects col<sub>1</sub>.
  - (2) A filter operator following a root access operator is referred to as a *row selector* and has the ability to select rows from a relation. For example, the field path  $.[?(@.col_1 =' def')]$  selects rows from the relation satisfying  $col_1 =' def'$ .
  - (3) A row selector followed by a dereference operator selects cell values of an attribute from the selected rows. For example, the field path  $.[?(@.col_1 =' ghj')].col_2$  selects values of  $col_2$  from rows satisfying the condition  $col_1 =' ghj'$ .
  - (4) A filter operator following an unnest operator on a collection type field (e.g. arrays and maps) filters elements of the collection. For example, the field path \$.col<sub>3</sub>.[item].[?(@.field<sub>31</sub> =' s1')] selects elements of col<sub>3</sub> such that each element satisfies the condition field<sub>31</sub> =' s1'. Similarly, the field path \$.col<sub>4</sub>.[value].[item].[?(@.field<sub>41</sub> =' v2')].field<sub>42</sub> selects values of the field field<sub>42</sub> from the collection of map values satisfying field<sub>41</sub> =' v2'.

In summary, field path expressions allow data selection at much finer granularities than is possible using traditional selection and projection operators available in commercial databases.

In order to mask data addressed by a field path expression, we need the ability to assign policy labels to them. Thus, the field paths associated with a relation are stored with the schema of the relation in the data catalog. Some field paths are automatically extracted from the schema when a schema is ingested into the data catalog. Additional field paths such as row selectors are added to the catalog by owners of the schema.

4.3.4 Data-Masking Operator. We next describe how the field path expressions are used to define data masking operations on a data element. Given an input relation  $\mathcal{R}$ , a data access policy p, and a field path f, the data-masking operator is a relational algebra function denoted by  $mask(\mathcal{R}, p, f)$  and defined as follows:

$$mask(\mathcal{R}, p, f) = \begin{cases} \sigma_{\neg pred \ OR \ p.cond}(\mathcal{R}), & \text{if } f = \$.[?(pred)] \\ \tau_{f \to m(f)}(\sigma_{\neg p.cond}(\mathcal{R}))) \cup \sigma_{p.cond}(\mathcal{R}), & \text{otherwise} \end{cases}$$
(1)

where  $\tau$  denotes a transformation operator that masks the data element at path f by applying a masking function m and reassembles the transformed attribute (potentially, a nested relation) back into its original structure. Equation 1 shows that when f does not contain a dereference operator, the masking operator reduces to a row-level mask and removes the matching rows from the result set. Otherwise, the transformation operator functions as a column-level mask applying the function m to values of f. The masking function m in Equation 1 has the following behavior:

- If f is an atomic attribute, m(f) sets the value of f to NULL.
- If f is an array element, m(f) removes the element from the array.
- If f is a map key, m(f) removes the key value pair from the map.
- If f is a map value, m(f) sets the map value to NULL.

In summary, field paths and data masking operators give us the capability of masking data at a very fine granularity. The exact implementation of the data-masking operator will be discussed in detail in Section 5.1.1.

#### 5 IMPLEMENTATION

#### 5.1 View Creation

In this section, we provide a detailed description of how Data Guard translates access control policies into data-masking views. We also describe the various optimizations in the generated views that ensure data accesses through data-masking views are performant.

5.1.1 Policy Compiler. The process of translating data policies into SQL code is referred to as policy compilation. As shown in Algorithm 1, the policy compiler incrementally constructs a relational algebra plan for a given input relation  $\mathcal R$  by iterating over a list of matching policies for  $\mathcal R$  and then converting the plan into SQL text.

**Algorithm 1** Compiling policies into SQL code where  $enforce\_policy()$  function creates a relational algebra plan representing the enforcement of a data access policy p on attribute  $\mathcal A$  of the input relation  $\mathcal R$ , and  $to\_sql()$  function returns the SQL text for a relational algebra plan

```
Input: Input relation \mathcal{R}, a list \mathcal{L} of matching pairs (field path, policy)

Output: SQL string representing a data-masking view plan \leftarrow \mathcal{R}

for (f, p) \in \mathcal{L} do

plan \leftarrow enforce\_policy(plan, f, p)

end for

return to\_sql(plan)
```

A key step in Algorithm 1 is the call to  $enforce\_policy()$  method to update the relational algebra plan. Under the hood, the  $enforce\_policy()$  method adds the data-masking operator mask introduced in Section 4.3.4 to the plan. Thus, the efficiency of the generated plan (and of the resulting view SQL) depends crucially on the implementation of the mask operator.

We focus on the implementation of the *mask* operator to perform column-level masking i.e. the data element to be masked is inside an attribute of  $\mathcal{R}$ . A naive approach to implementing the data-masking operator for a relation-valued attribute is to unnest the attribute, mask the data at a given field path within the attribute and reconstruct the attribute into its original structure. However, this approach produces complex SQL queries for relation-valued attributes, which can be both hard to read and debug especially when involving transformations on deeply nested attributes. We therefore implement a scalar version of the *mask* operator for column-level masking. Specifically, the operator employs a user-defined function (UDF)  $MASK\_FIELD()$ , which takes a field path f as an argument and sets the value at the field path to NULL.

Consider a field path f inside an attribute  $\mathcal A$  of a relation  $\mathcal R$ . We start with an implementation of the column-masking UDF with the following signature:

```
MASK\_FIELD(\mathcal{A}: ANY, f: VARCHAR)
```

The above UDF performs in-place update on attribute  $\mathcal{A}$ , instead

of a sequence of unnest, transform, and re-construct operations. The return type of the UDF is the same as the data type of  $\mathcal A$  to ensure the view schema is identical to the underlying table schema. The data-masking operator introduced in Equation 1 can thus be implemented by combining the  $MASK\_FIELD()$  UDF with the CASE statement in SQL as follows:

```
CASE
WHEN p.cond THEN A
ELSE MASK_FIELD(A, f)
END
```

Now suppose we have two field paths  $f_1$  and  $f_2$ , both accessing the same attribute  $\mathcal{A}$  with matching policies  $p_1$  and  $p_2$  respectively. Algorithm 1 yields the following relational algebra plan:

```
mask(mask(\mathcal{R}, p_1, f_1), p_2, f_2)
```

which is translated into two consecutive projection operators. Since SQL engines commonly apply the projection merge optimization rule to combine consecutive projections in a plan into a single projection, the above plan is translated into the following SQL code:

```
CASE
WHEN p2.cond THEN (CASE WHEN p1.cond THEN A ELSE
MASK_FIELD(A, f1) END)
ELSE MASK_FIELD((CASE WHEN p1.cond THEN A ELSE
MASK_FIELD(A, f1) END) , f2)
END
```

In this expression,  $MASK\_FIELD()$  is called once on  $f_2$  and twice on  $f_1$ . A straightforward analysis shows that for n fields accessing the same attribute  $\mathcal{A}$ , the number of UDF calls in the generated SQL will equal  $2^n - 1$ .

We avoid the exponential number of UDF calls in the view SQL by making the policy condition a parameter of the UDF instead of using the *CASE* expression. This results in the following modified signature for the masking UDF:

 $MASK\_FIELD\_IF(cond: BOOLEAN, \mathcal{A}: ANY, f: VARCHAR)$  where the UDF only applies data masking when the condition is true and returns the original value of  $\mathcal{A}$  otherwise.

With this optimization, the above nested CASE statement is rewritten with two UDF calls (n UDF calls instead of  $2^n$  for n enforcements) as:

```
MASK_FIELD_IF(p2.cond,
MASK_FIELD_IF(p1.cond, A, f1), f2)
```

When the policy conditions are the same  $(p_1.cond = p_2.cond)$ , the policy compiler further groups the nested UDF calls into a single UDF call:

```
MASK_FIELD_IF(p1.cond, A, CONCAT(f1, delim, f2))
```

5.1.2 Consent Bitmaps. Recall from Section 4.1.3 that the most common type of data access policies in our system are consent-based, where the policy condition involves checking preferences of data subjects stored in a separate table, referred to as the lookup table. As described in Section 4.2, a naive implementation of policy compiler can produce views that join a table containing data subject information with the lookup table using the data subject id as the join key. At LinkedIn, this table stores the consents of more than 1 billion members and any joins against this table result in expensive data shuffles. In Spark for instance, such joins are implemented as Sort Merge Joins since the size of both the tables involved in the join operation are too large to be broadcast using Broadcast

Hash Joins. Optimizing the view SQL for fetching attribute values from the lookup table is therefore critical to minimize the cost of adopting views.

In order to address this scalability challenge, we pre-compute a consent bitmap for every consent attribute referenced in a policy definition and leverage statistics associated with each consent to minimize the bitmap size. Specifically, we use a "true bitmap" (bitmap of data subject identifiers whose consent value is true) if minority of users have consent values set to true, and a false bitmap otherwise. Therefore, the number of user ids in each bitmap is at most half of the total number of users. Given that data subject identifiers at LinkedIn are numeric (a scenario we expect to be common), we use the Roaring Bitmap implementation [28] to compute the bitmaps as it achieves superior compression ratios compared to other compression methods. As an example, a Roaring bitmap of 200 million 8-byte integers that represent user ids has a size of around 134MB, which is small enough for the bitmaps to be loaded into memory on the worker nodes in Spark and Trino. In cases where identifiers are non-numeric, we can employ minimal perfect hashing functions [6] to map keys into consecutive integers and then use Roaring Bitmap on the resulting integers.

These bitmaps are then saved in a storage system like HDFS or cloud storage and are loaded at access time for fast in-memory lookup via a UDF. The bitmap implementation is much cheaper compared to a join-based solution, as we only introduce a small memory overhead for storing bitmaps in each worker's memory. In order to abstract away the storage system implementation details, we wrap the bitmap loading and lookup actions into a Bitmap-Manager interface, thus ensuring that the bitmap solution can be adopted across a range of different storage systems.

To ensure freshness of bitmaps, a bitmap computation job is scheduled periodically to reflect the latest user consents in the generated snapshot. Each snapshot of a bitmap is addressed using a combination of bitmap name and a timestamp when the bitmap was generated. The different snapshots of a bitmap are stored in a time-partitioned folder. The bitmap lookup is wrapped inside a UDF with the following signature: HAS\_USER\_CONSENT(consents: VARCHAR, user\_id: BIGINT, access\_time: TIMESTAMP), where:

- consents: a string to represent a list of consent names, which
  are used to load corresponding bitmaps into executor memory.
- *user\_id*: the identifier column of the table identifying the data subject whose consents need to be looked up,
- access\_time: the timestamp for determining the specific snapshot of a bitmap to load. This parameter gives users the ability to replay a query at a later time (aka time travel). By default, the data-masking views use the built-in SQL function CURRENT\_TIMESTAMP() for the access\_time parameter. The access\_time parameter ensures that the same snapshot of bitmap is loaded across all the worker nodes for a given query.

5.1.3 Masking deduplication. Given the scale of LinkedIn's data warehouse and the diversity of policies that need to be enforced, it is common to find tables in our warehouse containing attributes that are subject to multiple policies. Further, a single policy may apply to multiple attributes within the table. A naive implementation of the

policy compiler in Algorithm 1 can result in redundant application of the *MASK\_FIELD\_IF()* UDF.

There are two scenarios that can result in redundant masking operations on a field path:

- If a nested field needs to be masked based on one or more policy conditions, each of which is applicable to an ancestor, then a masking operation on the nested field is unnecessary. For example, masking operation on the field path in row 2 of Table 1 is redundant given an identical policy condition applicable to the root path \$. Similarly, the masking on field path in row 4 is redundant given an identical policy condition applicable to its parent in row 3.
- If a field in a table has two or more applicable policies and the set of consents used in policy  $p_1$  is a superset of the set of consents used in a different policy  $p_i$ , then the masking with policy  $p_j$  is unnecessary. For example, rows 5 and 6 of Table 1 show two different policies  $p_3$  and  $p_4$  applied to the same field path with overlapping consents. It is easy to see that policy condition check in row 6 is redundant and can be eliminated.

No	Field Path	Policy condition	Policy
1	\$	$consent_1$	$p_1$
2	$\$.col_2.field_{21}$	consent <sub>1</sub>	$p_1$
3	$\$.col_3$	consent <sub>2</sub>	$p_2$
4	$s.col_3.[item].[?(@.field_{31} = 's1')]$	consent <sub>2</sub>	$p_2$
5	$s.col_4.[value]$	consent <sub>3</sub> AND consent <sub>4</sub>	<i>p</i> <sub>3</sub>
6	$s.col_4.[value]$	consent <sub>3</sub>	$p_4$

Table 1: Example of maskings with different field paths with applicable policies. Masking operations on field paths in rows 2, 4, and 6 can be pruned without impacting the masking results.

To eliminate such redundant masking operations, we construct a schema tree from a given list of field paths and their applicable policies. A schema tree is one where each node represents an operator of a field path expression defined in Definition 2 and nodes corresponding to successive operators of a field path expression have a parent-child relationship between them. A node that is a final operator of a field path has an attribute to store the list of all policies applicable to that field path. Figure 5a shows an example schema tree constructed from field paths in Table 1.



(a) Before pruning

(b) After pruning

Figure 5: An example schema tree constructed from field paths defined in Table 1.

After constructing the schema tree, we perform a pre-order traversal to prune redundant policies as detailed in Algorithm 2. The algorithm computes the minimal set of policies required to cover the set of consents of a node that do not overlap with any of its ancestors. This problem can be mapped to the well-known minimum set cover problem, which is an NP-complete problem [22].

We therefore use a greedy approximation algorithm that chooses policies with largest number of non-overlapping consents first. Figure 5b demonstrates the schema tree produced by applying Algorithm 2. Maskings in rows 1, 3, and 5 are chosen to be retained by the algorithm.

#### Algorithm 2 Prune redundant policies

```
Input: A schema tree with root r.
Output: A new schema tree with redundant policies pruned.
function PrunePolicies(n, C)
    P \leftarrow n.policies
                                                 //Get node n's policy set
    C_n \leftarrow \emptyset
                                        //Initialize node n's consent set
    for p_i \in P do
       C \leftarrow C_n \cup p_i.consents
    end for
    \Delta \leftarrow C_n \setminus C //consents non-overlapping with n's ancestors
    R \leftarrow \emptyset
                                     // Initialize the retained policy set
    while \Delta \neq \emptyset
       select p \in P that maximizes |p.consents \cap \Delta|
       \Delta \leftarrow \Delta \setminus p.consents
       R \leftarrow R \cup \{p\}
       P \leftarrow P \setminus \{p\}
    end while
    n.policies \leftarrow R
                                            //Update node n's policy set
    for c \in n.children do
       PRUNEPOLICIES(c, C \cup C_n)
    end for
end function
PRUNEPOLICIES(r, \emptyset)
```

Suppose  $col_1$  of  $\mathcal{R}$  is labeled as a *user id* column, the resulting data-masking view SQL produced by Algorithm 1 is as follows:

- 5.1.4 View Maintenance. We implement a system for creating and updating of data-masking views across tables in our warehouse. At a high-level, this system performs the following steps:
  - (1) Generates a candidate list of warehouse tables along with a map of field paths to matching policies for each candidate table. This is accomplished by joining: (i) an inventory of warehouse tables, (ii) a metadata table that contains information about policy labels assigned to each field path of a table, and (iii) a table containing policy labels and policy definitions.
  - (2) For each candidate table *T* computed in the previous step and each candidate purpose *P*:
    - (a) the system checks if it needs to create or update a masking view for T. Specifically, view update for T is triggered due to:

- schema changes to the underlying tables (e.g. new column additions),
- changes to policy labels assigned to field paths, or
- new policy additions or changes to existing policy definitions for purpose *P*.
- (b) If a view update is deemed necessary, the system invokes Algorithm 1 to generate a new view definition, and
- (c) registers the updated view definition with the view catalog (a Hive-based [5] catalog in our ecosystem).

To ensure smooth rollout of views and enable rollbacks in case of errors, each data-masking view is versioned according to the semantic versioning scheme [31]. End user pipelines are routed to the latest view versions by default. We also provide users with the option to pin their pipelines to specific versions of views and be notified when newer versions of views become available. In this mode, users can test their pipelines against the new view versions before deploying the change to production. Older versions of views are garbage collected periodically to minimize resource consumption due to stale views.

## 5.2 View Consumption

At LinkedIn, warehouse users write data processing applications using Apache Spark and Trino as the underlying compute engines. To ensure ease-of-use of data-masking views in user applications, we implemented two important optimizations: (i) dynamic table-to-view routing, and (ii) multi-engine support, each of which will be discussed in detail in this section.

5.2.1 Dynamic Table-to-View Routing. A key design choice of our system was to ensure access control policies are automatically enforced during data accesses by redirecting table accesses to the appropriate data-masking views determined by the purpose of access. This minimizes overhead for existing warehouse users to ensure their accesses remain compliant. Recall from Section 3 that the table to view routing at access time is accomplished using a component called ViewShift. At LinkedIn, we implement this component as a plugin inside existing engine catalog implementations. ViewShift intercepts loadTable() API calls to the catalog from applications and returns the corresponding data-masking view instead. The View-Shift plugin exposes the following API:

ViewIdentifier getView(TableIdentifier tableName, Map<String, String>contextMap)

where *contextMap* parameter is query context map that contains attributes such as the purpose of the access (obtained from the identity of the user that submits the query), the optional pinned version of the data masking view, and the environment where the application is running.

Every invocation of the getView() API is published to a logging system as a log entry containing the contextual information inside the contextMap. These logs are consumed by downstream monitoring applications that monitor data accesses across the warehouse.

5.2.2 Multi-engine support. In large organizations, it is common for infrastructure teams to support a diversity of data processing systems, each optimized for a specific set of use cases. For instance,

Apache Spark is commonly used by data scientists for data exploration, while Trino is commonly used for ad hoc data analysis. There are variances in the SQL dialects supported by each system. Further, each system has a different mechanism and type system to define UDFs. A key requirement that Data Guard needed to address was to ensure that the data masking views are executable across different data processing systems and avoid view duplication by creating engine-specific views.

To ensure portability of views across engines, we leverage two in-house technologies, which are independently available as open-source projects: (i) a SQL translation tool called *Coral* [14] that translates SQL between different dialects, and (ii) a framework called *Transport UDF* [35] to author UDFs that can be executed on multiple engines.

The data-masking views generated by Data Guard are stored in a view catalog that supports the Hive SQL dialect and are translated by Coral to Spark/Trino SQL at access time. Coral uses Apache Calcite [4] under-the-hood to parse SQL from a source dialect into a relational algebra representation, applies transformations to the representation to address variations in the SQL dialect between the source and target systems, and finally converts the relational algebra representation back to the SQL dialect of the target system.

The Transport framework provides interfaces for common data types supported by each engine (e.g. *int*, *float*, *double*, *boolean*, *string*, *struct*, *array*, and *map*). We provide engine-specific implementations of Transport's interfaces as wrappers over native types supported by the engine and define masking UDFs using Transport's type system. The Transport framework for authoring UDFs provides multiple benefits:

- Saves development time as developers need to author the UDF only once.
- Simplifies code maintenance as we have a single source of truth for the UDF logic.
- Ensures UDFs remain performant as Transport accesses native data types directly without needing conversion between different type systems.

## **6 PERFORMANCE EVALUATION**

In this section, we evaluate the performance of the UDF implementing the data-masking operation. Specifically, we study the impact of factors such as: (1) data type of a column being masked, and (2) size and depth of the column being masked and (3) the number of columns being masked on the overall performance of the data-masking view.

#### 6.1 Methodology

We first set up a micro-benchmark to evaluate the performance of the <code>MASK\_FIELD\_IF()</code> UDF in a controlled and isolated environment. We analyze trends in the UDF's performance as a function of the above-mentioned factors and provide insights obtained through CPU profiles. We confirm the trends observed from the micro-benchmarks with results obtained by running Spark SQL queries that use the <code>MASK\_FIELD\_IF()</code> UDF on a large compute cluster running Apache Spark 3.1.1. Each query uses 50 executors, each provisioned with 8GB memory and 4 CPU cores.

We use a homegrown synthetic data generation tool to generate datasets that include the following types of columns common in our ecosystem:

- Primitive Fields: columns of string type to simulate simple flat data structures,
- Flat Structs: structs with varying number of primitive fields (from single-field structs to structs containing five primitive fields, e.g. STRUCT<field\_1: STRING, ..., field\_5:STRING>).
- Nested Structs: structs containing nested fields of varying depths. For example, a three-level deep struct produced by our tool has the following shape:
  - nfield\_11: STRUCT<pfield: STRING, nfield\_12:
    STRUCT<pfield: STRING, nfield\_13: STRUCT<pfield:
    STRING»>.
- Arrays: columns of array type, with varying entries per array (e.g. 10, 50, 100). We generate datasets containing oneand two-dimensional arrays, to simulate schemas that are commonly used in AI/ML training pipelines.

For the cluster benchmarks, we generate a synthetic dataset containing 10 million rows. The size of the dataset is chosen so that the task startup overheads for benchmarking queries get amortized over a sufficiently large partition of rows processed by each task. The data is written in ORC storage format with ZLIB compression, which is the storage format adopted for LinkedIn's warehouse data. We note however that the performance numbers reported in this section should generalize to other commonly used formats like Parquet.

The primary performance metric is the per-row CPU time difference between queries using the masking operator (via the  $MASK\_FIELD\_IF()$  UDF) and those using an "identity" operator (a no-op UDF that returns the input as-is) on the same field. This approach isolates the computational overhead of the masking operation, eliminating the influence of external factors like storage latency, network performance, and disk I/O.

Since the goal is to analyze performance trends rather than absolute values, this metric effectively illustrates how the masking operator overhead changes across different scenarios. Both UDFs are implemented using Spark's Expression APIs to ensure consistency.

#### 6.2 Experiment Results

The results shown in this section are from cluster benchmarks unless mentioned otherwise.

6.2.1 Impact of Column Size. We measure the overhead of applying the  $MASK\_FIELD\_IF()$  UDF on a column as a function of its size. In Figure 6, we show the CPU overhead of masking a struct column containing varying number of string fields. From Figure 6, we see that the overhead for masking a primitive field (string type in this case) is slightly lower than that for complex fields such as structs. But increase in the column size does not have any impact on the overhead. We confirm this behavior through a CPU profile of the  $MASK\_FIELD\_IF()$  UDF which shows that evaluating the input column is slightly more expensive for a struct compared to

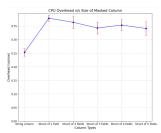
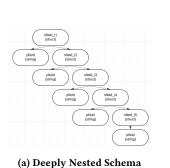
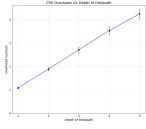


Figure 6: Impact of field size on CPU overhead (cluster-benchmark)

primitive types such as string, in the generic masking implementa-

6.2.2 Impact of Field Depth. We analyze the performance of the  $MASK\_FIELD\_IF()$  UDF as a function of the depth of the field being masked. The synthetic data generation tool generates random records conforming to the deeply nested schema shown in Figure 7a, where the struct at each level contains a string field and another nested struct. Figure 7b shows the overhead for masking the string fields at varying depths, starting with  $nfield\_l1.pfield$  (represented by the x-axis label d=1) and moving deeper down the hierarchy (e.g.,  $nfield\_l1.nfield\_l2.pfield$ ). As seen from Figure 7b, the



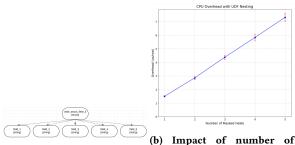


(b) Impact of field path depth on CPU overhead (clusterbenchmark)

Figure 7: Impact of field path depth

overhead grows linearly with increasing depth of the string fields. CPU profiling reveals that this behavior stems from how nested struct objects are represented internally in Spark. The struct objects are organized in a tree-like hierarchy, so the <code>MASK\_FIELD\_IF()</code> UDF must traverse this tree to locate and transform the target field. Upon transformation of the target field, each parent struct in the tree needs to be reconstructed with the modified field due to immutability of objects passed to the UDF.

6.2.3 Impact of number of fields masked. In this section, we evaluate the masking overhead as a function of the number of fields that need to be masked. For simplicity, we consider a "flat" struct consisting of five string fields (Figure 8a) and vary the number of members of the struct that are masked. As seen in Figure 8b, the overhead increases linearly with the number of fields being masked. The schema considered in Figure 8a does however point to an optimization opportunity, where instead of invoking the masking operation separately for each member of the struct, the masking operations across members of the struct can be performed together at once. We expect this optimization to reduce the per-field overhead incurred



(a) A flat struct containing masked fields on CPU overprimitive fields head (cluster-benchmark)

Figure 8: Impact of number of maskings

due to schema traversal and struct object transformation and reconstruction. A special case of this optimization is implemented in the policy compiler, where the UDF performs struct-level masking when all its members need to be masked. We plan to incorporate the generalization of this optimization in our UDF implementation to further exploit the UDF batching feature of the policy compiler.

6.2.4 Impact of consent rates. The % of rows from which one or more fields need to be masked depends on the fraction of data subjects that consent to allow access to those fields. We refer to this fraction as the consent rate. Figure 9 contains the micro-

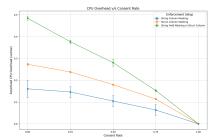


Figure 9: Impact of consent rates on CPU overhead (microbenchmark)

benchmark results showing the CPU overhead for different data types as a function of the consent rate. A consent rate of 1 reduces the *MASK\_FIELD\_IF()* UDF to a no-op action. As expected, the average CPU overhead increases as the consent rate decreases for each data type. As discussed in the previous sections, struct column masking has higher overhead than string column masking for all consent rates. Similarly, masking a field inside a struct has a higher overhead than struct column masking due to the additional overhead of schema traversal and object transformation.

- 6.2.5 Data Masking in Arrays. We next evaluate the overhead of the masking operation on array-typed columns where each element is of type struct. Specifically, the column has the following data type:  $acol: array(struct < field_1: string, field_2: string >)$ . We consider masking overhead on two different field paths:
  - (1)  $\$.acol.[item].field_1$ : which indicates unconditional masking of  $field_1$  from all elements in the acol, and
  - (2) \$.acol[?(@.field<sub>1</sub> ==' jackfruit')].[item].field<sub>2</sub>: which indicates conditional masking of field<sub>2</sub> based on the value of field<sub>1</sub>.



Figure 10: Impact of array sizes on CPU overhead (clusterbenchmark)

Figure 10 shows that the masking overhead grows linearly with array size for each field path. For a given array size, the overhead of unconditional masking is smaller than conditional masking due to the additional cost of comparing element values. In our implementation, this comparison requires conversion between Spark's UTF-8 strings to Java UTF-16 strings, which is the dominant contributor to the overhead.

#### 7 PRACTICAL CONSIDERATIONS

In this section, we discuss practical issues that needed to be considered when deploying Data Guard at-scale in LinkedIn.

- 1. **View Schema Evolution:** In a large warehouse, tables often undergo changes to schema and new field additions (either top-level columns or nested fields) to tables are very common. As Data Guard data-masking views are schema-preserving, the schema of the view at creation time may be different from the schema of the view at the consumption time due to newly added fields. This means that newly added fields may accidentally be consumed in user queries, resulting in non-compliant accesses. To prevent this data leakage, we dynamically rewrite the data-masking view at the view consumption time via the ViewShift getView() API to set all newly added field values to NULL.
- 2. Handling non-nullable fields: The data-masking function described thus far replaces fields to be masked with *NULL* values and thus, assumes that fields are nullable. In large warehouses, it is common to define schemas containing non-nullable fields. To ensure compliance, our approach when rolling out Data Guard at LinkedIn, has been to over-filter data and filter out the entire row from the result set when a non-nullable field needs to be masked. It is possible to extend this behavior to provide non-NULL replacement values which are defined system-wide. This would however require changes by data consumers to handle such values in their applications. An area of ongoing work is to enhance Data Guard's APIs to provide data consumer-specified replacement values for non-nullable fields in order to avoid over-filtering data and maximize data utility.
- 3. Enforcement Verification: In addition to proactive enforcement of policies at the time of data access, Data Guard also monitors access logs for all data accesses to the warehouse data. Data Guard's monitoring sub-system relies on raw access logs from underlying storage system (i.e. HDFS in our case) along with query execution logs from data processing engines like Spark and Trino to detect any inadvertent access to warehouse data that bypasses Data Guard enforcement. Data Guard also detects accesses to stale versions of Data Guard views from the access logs. In each scenario, Data Guard notifies consumers about potential access violations and provides

them with steps to remediate their accesses.

- 4. **Result Caching:** Commercial warehouses commonly support result set caching, even though this functionality is not available currently in LinkedIn's warehouse. One of the conditions for result caching is that queries should not include non-deterministic SQL functions such as *CURRENT\_TIMESTAMP()*. Thus, the data-masking views described in this paper are not eligible for result set caching. However, caching results of data-masking views has limited value in our environment because the view evaluation depends on data subject consents which are frequently updated. In scenarios where such updates are infrequent, result set caching can indeed be effective. To enable reuse of cached results, alternatives such as defining data-masking views as parametrized views that use timestamp as a parameter passed to the view or defining the masking views as materialized views appear viable.
- 5. Engine optimizations: An important limitation of implementing the data-masking operator using UDFs is that it prevents engines from applying optimizations such as predicate pushdown or nested column pruning. Our primary focus in implementing and rolling out Data Guard at-scale inside LinkedIn has been on the APIs for enforcing policies and ensuring a friction-free developer experience. While we implemented a number of performance optimizations to ensure that the cost of adopting data-masking views remains acceptable, we recognize the opportunity for further optimizations in the future.

#### 8 CONCLUSIONS AND FUTURE WORK

We have presented Data Guard - a fine-grained access control system for large data warehouses. Data Guard's policy language supports specification of access-control rules based on semantic labels used to describe data present in the warehouse tables. Data Guard's policy compiler translates these policies into SQL views which encode the data-masking logic. Further, Data Guard transparently routes table accesses to the appropriate data-masking views based on the accessor's purpose, thus supporting purpose-based access control. Data Guard has been deployed at-scale in LinkedIn and is the primary policy enforcement mechanism for accesses to warehouse tables. There are a number of avenues for further optimizing the performance of Data Guard's data-masking operator as highlighted in the paper. We also plan to enhance Data Guard to support additional capabilities to enable privacy-enhancing technologies such as anonymization and differential privacy, and encryption for data storage.

#### **ACKNOWLEDGMENTS**

We would like to thank Kapil Surlaker and Raghu Hiremagalur for providing numerous critical inputs on the design of Data Guard. We would like to thank Yong Li, Manasa Subramanian, Divya Singh, Yanwen Lin, Justin Heaton, Vishwaa Patel, Bryan Ji, Qiang Fu, Steve Cao, Wenning Ding, and Carleen Li for their significant contributions to the development of Data Guard. We also thank Kiran Shivaram, Souvik Ghosh, Chris Harris, David Leung and their team members for being early adopters and providing valuable customer feedback that helped improve the product. Finally, we would like to thank Maneesh Varshney and Chris Harris for their helpful feedback in improving the quality of the paper.

#### REFERENCES

- R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi. 2005. Extending relational database systems to automatically enforce privacy policies. In 21st International Conference on Data Engineering (ICDE'05). 1013–1022. https://doi.org/10.1109/ICDE.2005.64
- [2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. 2002. Hippocratic databases. In Proceedings of the 28th International Conference on Very Large Data Bases (Hong Kong, China) (VLDB '02). VLDB Endowment, 143–154.
- [3] Amazon Redshift 2024. Dynamic data masking. https://docs.aws.amazon.com/ redshift/latest/dg/t\_ddm.html
- [4] Apache Calcite 2014. Apache Calcite a dynamic Data Management Framework. https://calcite.apache.org/
- [5] Apache Hive 2010. Hive Metastore a central repository of tables and views. https://hive.apache.org/
- [6] D. Belazzougui, F. Botelho, and M. Dietzfelbinger. 2009. Hash, Displace, and Compress. In Algorithms - ESA 2009, Amos Fiat and Peter Sanders (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 682–693.
- [7] E. Bertino. 2005. Purpose Based Access Control for Privacy Protection in Database Systems. In *Database Systems for Advanced Applications*, Lizhu Zhou, B. Ooi, and X. Meng (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–2.
- [8] J. Byun, E. Bertino, and N. Li. 2005. Purpose based access control of complex data for privacy protection. In Proceedings of the Tenth ACM Symposium on Access Control Models and Technologies (Stockholm, Sweden) (SACMAT '05). Association for Computing Machinery, New York, NY, USA, 102–110. https://doi.org/10.1145/1063979.1063998
- [9] J. Byun and N. Li. 2008. Purpose based access control for privacy protection in relational database systems. *The VLDB Journal* 17, 4 (July 2008), 603–619. https://doi.org/10.1007/s00778-006-0023-0
- [10] CCPA 2018. California Consumer Privacy Act. https://leginfo.legislature.ca.gov/faces/codes\_displayText.xhtml?division=3.&part=4.&lawCode=CIV&title=1.81.
- [11] S. Chawla, C. Dwork, F. McSherry, A. Smith, and H. Wee. 2005. Toward privacy in public databases. In Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2. Springer, 363–385.
- [12] Latha S. Colby. 1990. A recursive algebra for nested relations. *Information Systems* 15, 5 (1990), 567–582. https://doi.org/10.1016/0306-4379(90)90029-O
- [13] P. Colombo and E. Ferrari. 2017. Enhancing MongoDB with Purpose-Based Access Control. IEEE Transactions on Dependable and Secure Computing 14, 6 (2017), 591–604. https://doi.org/10.1109/TDSC.2015.2497680
- [14] Coral 2020. Coral SQL translation, analysis, and rewrite engine. https://github.com/linkedin/coral
- [15] L. Cranor, M. Langheinrich, M. Marchiori, and J. Reagle. 2002. The Platform for Privacy Preferences 1.0 (P3P1.0) Specification. (01 2002).
- [16] Databricks 2024. Filter sensitive table data using row filters and column masks. https://docs.databricks.com/en/tables/row-and-column-filters.html
- [17] Datahub 2019. DataHub: A generalized metadata search and discovery tool. https://www.linkedin.com/blog/engineering/archive/data-hub
- [18] DMA 2024. Digital Markets Act. https://digital-markets-act.ec.europa.eu/index\_en
- [19] C. Dwork. 2006. Differential privacy. In International colloquium on automata, languages, and programming. Springer, 1–12.
- [20] EU AI Act 2024. The EU Artificial Intelligence Act. https://artificialintelligenceact.
- [21] Flink [n.d.]. Apache Flink. https://flink.apache.org/
- [22] M. Garey and D. Johnson. 1990. Computers and Intractability; A Guide to the Theory of NP-Completeness. W. H. Freeman & Co., USA.
- [23] GDPR 2016. General Data Protection Regulation. https://gdpr-info.eu
- [24] Stefan Gössner, Glyn Normington, and Carsten Bormann. 2024. JSONPath: Query Expressions for JSON. RFC 9535. https://doi.org/10.17487/RFC9535
- [25] HIPAA 2003. Health Insurance Portability and Accountability Act. https://www. hhs.gov/hipaa/index.html
- [26] Horizon 2024. Snowflake Horizon Catalog. https://docs.snowflake.com/en/user-guide/snowflake-horizon
- [27] ISO-ANSI. 1999. Database languages SQL Part 2: Foundation (SQL/Foundation). https://www.iso.org/standard/32432.html ISO/IEC 9075-2:1999 (SQL:1999).
- [28] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. 2016. Consistently faster and smaller compressed bitmaps with Roaring. Softw. Pract. Exper. 46, 11 (Nov. 2016), 1547–1569. https://doi.org/10.1002/spe.2402
- [29] Q. Ni, E. Bertino, and J. Lobo. 2008. An obligation model bridging access control policies and privacy policies. In Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (Estes Park, CO, USA) (SACMAT '08). Association for Computing Machinery, New York, NY, USA, 133–142. https: //doi.org/10.1145/1377836.1377857
- [30] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. 2004. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the 2004 ACM*

- SIGMOD International Conference on Management of Data (Paris, France) (SIG-MOD '04). Association for Computing Machinery, New York, NY, USA, 551–562. https://doi.org/10.1145/1007568.1007631
- [31] Semantic Versioning 2013. Semantic Versioning 2.0.0. https://semver.org/
- [32] Snowflake 2024. Using Dynamic Data Masking. https://docs.snowflake.com/en/ user-guide/security-column-ddm-use
- [33] Spark [n.d.]. Apache Spark. https://spark.apache.org/
- 34] M. Stonebraker and E. Wong. 1974. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 Annual Conference - Volume 1 (ACM '74)*. Association for Computing Machinery, New York, NY, USA, 180–186. https://doi.org/10.1145/800182.810400
- [35] Transport 2018. Transport a portable UDF Framework. https://github.com/ linkedin/transport
- [36] Trino [n.d.]. Trino. https://trino.io/
- [37] Unity 2024. Unity Catalog. https://www.databricks.com/product/unity-catalog
- [38] XACML 2013. Oasis Extensible Access Control Markup Language (XACML) 3.0. https://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.html
- [39] XPath 3.1 2017. XML Path Language (XPath) 3.1. https://www.w3.org/TR/xpath-31/
- [40] T. Xue, Y. Wen, B. Luo, G. Li, Y. Li, B. Zhang, Y. Zheng, Y. Hu, and D. Meng. 2023. SparkAC: Fine-Grained Access Control in Spark for Secure Data Sharing and Analytics. *IEEE Transactions on Dependable and Secure Computing* 20, 2 (2023), 1104–1123. https://doi.org/10.1109/TDSC.2022.3149544