Deep Enforcement: Policy-based Data Transformations for Data in the Cloud

Ety Khaitzin[1], Julian James Stephen[1], Maya Anderson[1], Hani Jamjoom[1], Ronen Kat[1], Arjun Natarajan[1], Roger Raphael[2], Roee Shlomo[1], and Tomer Solomon[1]

[1]IBM Research
[2]IBM Cloud

Abstract

Despite the growing collection and use of private data in the cloud, there remains a fundamental disconnect between unified data governance and the storage system enforcement techniques. On one side, high-level governance policies derived from regulations like General Data Protection Regulation (GDPR) have emerged with stricter rules dictating who, when and how data can be processed. On the other side, storage-level controls, both role- or attribute-based, continue to focus on access/deny enforcement. In this paper, we propose how to bridge this gap. We introduce *Deep Enforcement*, a system that provides unified governance and transformation policies coupled with data transformations embedded into the storage fabric to achieve policy compliance. Data transformations can vary in complexity, from simple redactions to complex differential privacy-based techniques to provide the required amount of anonymization. We show how this architecture can be implemented into two broad classes of data storage systems in the cloud: object storages and SQL databases. Depending on the complexity of the transformation, we also demonstrate how to implement them either in-line (on data access) or off-line (creating an alternate cached dataset).

1 Introduction

Managing the collection and use of private data—especially in today's cloud-centric deployments—should be effortless. Yet, the reality is far from it. Enforcing proper data privacy and security remain deeply rooted in manual governance processes and traditional enforcement techniques. Emerging governance policies, like those in GDPR, not only define who can access the data, but also what portion of the data can be accessed for a particular purpose. From a system's perspective, translating these requirements into actionable controls is not straightforward. Today's system-level controls are, for the most part, focus on managing access, defining who can read or write a dataset. This puts the burden on application and platform developers to somehow manage data governance using orthogonal solutions, often requiring careful handholding. Of course, these issues are further exacerbated as cloud adoption drives the need to be more dynamic, heterogeneous and scalable.

In this paper, we are interested in studying the fundamental disconnect between today's governance policies and low-level system enforcement controls. We focus on two storage architectures commonly used in cloud deployments: databases (e.g., PostgreSQL [19] and IBM DB2 [20]) and object stores (e.g., IBM Cloud Object Store [17] and Amazon S3 [1]). In both storage types, when it comes to defining governance policies and enforcing them, each data store provides its own mechanism, if any. To handle this pain point, two approaches to data governance have been proposed. The first is *application-based governance*, which embeds the policy enforcement point into the application directly. This allows developing specific policies and rules to handle data access. The downside is that each application needs its own enforcement module, and no governance is applied when accessing the data outside of the application. The second is *platform-based governance,* which embeds governance into the run-time platform. This approach forces all application data access through a governance layer, which—in principle—provides a unified approach to govern data. Examples include Google DLP [14], and IBM DPCM tool [16].

While platform-based governance has clear advantages for platform providers, it suffers from a serious problem: if data is accessed directly, which is not uncommon, the governance layer is simply bypassed. Also, if a specific policy requires that only compliant data leave a data store, platform-based governance would result in non-compliance because—by design—governance is performed in a separate layer.

To address this design challenge, we propose a new approach, *Deep enforcement-based governance (DE).* The main idea in DE is to embed the enforcement aspects of the governance into the data storage fabric. Enforcement here not only refers to simple allow/deny policies, but represents a rich set of transformations that can be automatically applied to datasets on access (e.g., to automatically anonymize portions

of the dataset). Embedding the policy enforcement point into the data stores guarantees that all the accesses are regulated and conform to the policies. However, it requires that the data store access protocol and the data store itself provide support for performing governance actions and transformation capabilities.[1]

We present a new *unified governance* model for the deep enforcement approach. This model has three primary components: Policy Decision Service (PDS), Transformation Generator (TG), and Deep Enforcers (DE). Briefly, the PDS captures high-level governance policies which are translated using the TG into low-level DE *transformation* primitives. While the PDS and TG are agnostic to the type of storage system, the DE's are customized for different storage systems.

We implement DE's to automate policy compliant use of data by machine learning and analytic worklods in the cloud on two broad classes of data storage: object storage and SQL databases. Object storages in general have architectures that focus on scaling out, with relatively simple and coarse-grain access management mechanisms. Some object storages, such as AWS, provide transformation capabilities, but via a different api, and not as means of governance enforcement. Hence, we show in Section 3 how a DE which can support a broad-set of transformation for object storages can be implemented without sacrificing its scale-out design. SQL database, on the other hand, have many built-in support for data transformations (e.g., indexing, filtering, and fine-grained access control systems). We show in Section 4 how they can be leveraged for applying governance policies. There, we only need to *compile* policies into SQL statements or to the programming language that is supported by the database.

Transforming data to support governance policies comes with an important design tradeoff: *should we transform data (in-line) on every access or cache previously transformed datasets*. In general, the former requires more compute resources whereas the latter requires more storage. Because data can vary in size and transformations can vary in complexity, our design can balance both alternatives. Specifically, we show how to implement both approaches—in-line and transformation caching—into the storage fabric. In the in-line approach, when user accesses a data-set, a policy-compliant version of the data (e.g., filtered, masked, etc.) is immediately created and streamed to the user. In transformation catching, when user accesses a data-set, a policy-compliant replica of the data is created, and the user gets a pointer to this *alternate* dataset.

Contributions. We present the deep enforcement approach and provide an architecture for a unified data governance solution using the approach. We demonstrate implementation for relational databases and also for an object store. Additionally, we describe two methods to support data transformation poli-

[1] A variant of this approach can be deployed as a gateway on the edge of the data store at the cost of reducing the security guarantees.

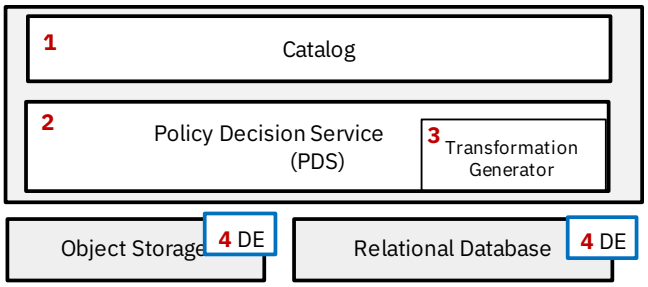

Figure 1: Enforcement Architecture

cies and discuss their trade-offs. To the best of our knowledge, we are the first to support object storage governance using the deep enforcement approach.

2 Enforcement Architecture

We now describe the components and concepts associated with our cloud based enforcement architecture, as shown in Figure 1:

1. The Catalog acts as a repository of metadata for data objects, such as files stored as data objects in object storage or tables in a SQL database. Each data object has an entry in the catalog containing its schema, business tags and metadata.
2. The Policy Decision Service (PDS) is a mechanism to define and store granular governance policies based on the metadata in the catalog. Examples of policies defined in PDS include (a) user `John` not allowed to view data objects that contain *email* or (b) user `Jane` not allowed to see credit card numbers. In the first example, access to an object can be denied entirely if it contains an email field. In the second case, object can be accessed only after credit card details are removed from it.
3. Transformation Generator (TG) is a component inside PDS responsible for creating a transformation that converts an instance of a data object governed by a policy to a compliant data object instance. The transformation is shaped according to the data object, the policy, the user that accesses the data and additional considerations. Examples for transformations are anonymizing a column and filtering out specific rows or columns.
4. The Deep Enforcers (DE) are responsible for creating a policy-compliant version of the data object by executing the transformation generated by the TG on it. The design and details of DEs are described in the following sections.

3 Deep Enforcement for Object Storage

Object storage provides a REST-based API for reading and writing data objects, including the ability to perform partial—

range—reads. Each object is referenced by a globally unique identifier and has an associated metadata. As the essence of deep enforcement is transforming the data to be compliant, handling certain data object metadata and API become a challenge when the returned data is different from the stored data. Specifically, *ETag* is the hash of the object data which is computed when it is uploaded. The *ETag* is returned as part of the HTTP response header for operations like `GET Object` and `HEAD Object`, and is often used for verification against tampering. Another challenge is handling *range reads*, which allow reading specified offsets within an object. The need to handle explicit ranges for a transformed object is complex, as data may be omitted or added due to the transformation. In the following subsections, we propose two patterns of deep enforcement for object storage: an inline pattern, in which the data is transformed and made complaint during transfer to the application, and an alternate asset pattern, which creates a compliant version of the data object, which can be read from the object storage. The former is ideal for transformations that can be performed in a semi-streaming manner, whereas the latter is ideal for transformations that are recurring and span wide sections of the objects.

3.1 Inline transformations

The inline approach executes the policy transformation as the data is streamed to the client. We have opted neither to support range reads nor to compute a proper ETag in order to provide an efficient solution. We note that is it possible to design metadata preserving approaches, but that would be inherently expensive. Range reads could be supported through length-preserving transformations as well.

Design: We opted to embed the transformation logic into the object storage `GET Object` code. With this approach any data that leaves the object storage is handled, and data cannot leave the object storage unless governance has been applied. We perform the transformation in the storage node, in order to avoid the communication and network overheads incurred by sending the data to a different node. The drawback is the need for additional processing and memory resources on the storage node, hence we aim for a minimal CPU and memory footprint. Thus, for the embedded transformation, we choose not to leverage an existing streaming solution like Apache Spark streaming [9] or Apache Flink [4], mainly due to the high memory footprint of such solutions. Figure 2 depicts the inline transformation flow:

1. The object storage handles a `GET Object` request.
2. The transformation engine obtains the object's schema and the transformation if needed, and prepares it for execution.
3. The object reader reads the data object and performs the needed format decoding (e.g., CSV or Parquet formats).
4. The transformation code is executed on the data.
5. The format encoder writes the data in the desired format.
6. The transformed data object is streamed to the user.

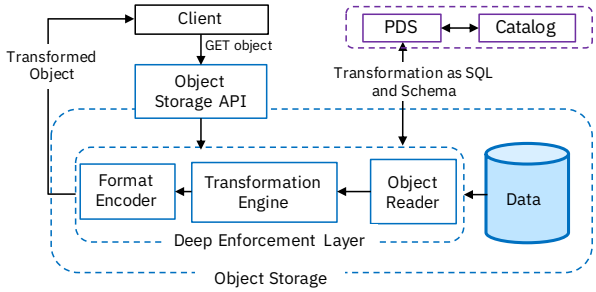

Figure 2: Inline Transformation Architecture

Transformation Engine: The PDS provides the transformation in the form of a SQL query, alongside the object's schema. The query is compiled to an executable binary by leveraging the Apache Spark SQL's Catalyst optimizer [10] to obtain a Java source code[2], which is then compiled into JVM bytecode using Janino [11]. Then, the transformation engine transforms the desired data object using this compiled query.

Object Reader: The object reader reads the data object and constructs records to be delivered to the Transformation Engine. The reader needs to understand the data object format and structure. For example, for CSV data, the reader needs to parse the CSV format and split the input into rows and columns, which is fairly simple. The second case is columnar data formats, such as parquet [6], where data is kept in blocks of columns, which are grouped into row groups. In this case, the reader assembles the row from the columns. Thus, the reader might need to load into memory an entire row group in order to assemble the rows, which in turn impacts memory usage. Another side effect when transforming a columnar data format is that the data object reading is not sequential but rather impacted by the internal format.

Format encoder: The transformation filter delivers out rows that need to be encoded into a format expected by the user. For a row based format (e.g., CSV) this is relatively simple. However, in case of a columnar format like Parquet, the format encoder needs to batch together multiple rows to generate the columnar format building blocks before sending them to the user. Such aggregation can affect memory footprint and latency and should be considered when deciding which output formats to support.

3.2 Alternate Asset Transformations

An alternative approach to providing a complaint data object when the requested data object is not compliant as-is, is to "redirect" the REST API request to an alternate (and complaint) version of the data object. In our approach, whenever

[2]We note that additional options are available for compiling a SQL statement but the discussion is outside the scope of this paper.

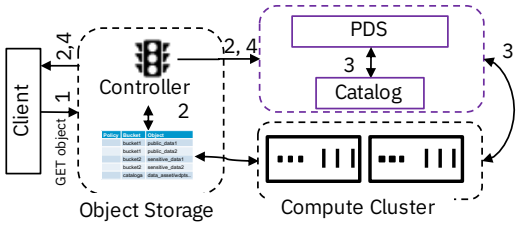

Figure 3: Alternate Asset Architecture

the object storage cannot handle a `GET Object` request due to a policy constraint, we generate a new (alternate) asset (data object) that is a complaint transformation of the original requested object. As creating the transformed object may take time, we return a deny response denoting the URL of the alternate date object. In case that the alternate data object already exists, we return it directly. The alternate asset approach uses a nearby compute cluster for running the transformation, decoupling the transformation logic run-time from the object storage.

Alternate Asset Flow: The alternate asset approach can be easily enabled for object storage through a thin layer that inspects the incoming REST requests, validates the policies and generates transformation as needed. The drawback is dependency on an additional transformation cluster. Challenges such as ETag and content-length can be addressed as the transformation is computed before the REST request is handled. Figure 3 shows the flow of a request in the alternate asset model. A detailed description is given below.

1. User *A* requests a data object from the object storage. E.g., `GET "http://hostname/foo/bar.csv`
2. The object storage queries PDS to check if user *A* can (a) access `foo/bar.csv` directly or (b) access `foo/bar.csv` after a transformation. If PDS suggests a transformation, the object storage requests PDS to initiate a transformation. Otherwise, user *A* is simply granted or denied access and the flow ends.
3. The PDS creates an entry in catalog for the transformed data object and starts the transformation logic in the compute cluster.
4. The object storage looks up the transformed data object entry and URL in the catalog.
5. The object storage returns `Deny` with URL to the transformed data object to user *A*

Alternate Asset challenges: One of the main concerns with the alternate asset approach is the need for robust life cycle management of the created alternate data objects, e.g., remove unneeded transformed objects, monitoring policy changes to trigger re-tranformations, etc.

Each alternate asset is added to the Catalog and inherits all the policies of the original data object. A lineage chain with a transformation annotating is recorded between the original data object and the transformed data object. This is the mechanism that enables reuse of transformed data object when possible.

Removing unneeded data objects, i.e., garbage collection, can be done by a simple model that takes into account the additional storage cost for the transformed data versus the repeated computation cost of creating the transformed object.

4 Databases

In this section, we describe Deep Enforcement implementations for SQL databases. As mentioned, databases have extensive built-in transformation access control mechanisms, ranging from simple allow/deny role-based access control to fine-grained access control that transforms the data using predefined rules (e.g., rows and columns access control (RCAC) for IBM DB2 database). These mechanisms by themselves do not constitute a unified governance solution. Moreover the level of support can vary across database implementations requiring that policies may have to be adapted for each one.

We introduce an implementation that can enforce unified governance policies and compatible with high range of databases.

4.1 Inline Enforcement

The inline approach follows the same pattern of object storage inline enforcement, but leverages the built-in SQL transformation capabilities that are available in databases. We demonstrate two enforcement variants based on different creation patterns, which either rewrites the incoming query according to the policy, or alternatively, redirects the query to a view that is created based on policies.

Query Rewrite Approach: An interception point, e.g., a gateway, is added to the database, as shown in Figure 4. The gateway accepts the query requests, obtains the needed SQL transformation from PDS based on the policies and access information, performs a rewrite to accommodate the policies, and passes the rewritten SQL to the database. Note that since the SQL transformation is associated with the table, and is not query specific, queries to the same database table are associated with the same set of policies, and hence, can be cached to optimize performance and reduce communication overheads between the database and PDS. We note that "cached" rewritten queries can take the form of a database view, which will be described next.

Dynamic Views Approach: Here, we leverage the database view mechanism to apply governance and security policies. We encode policy-based transformations as *compliant views*. A table may have several compliant views for different users; users are routed to access only compliant views. In fact, access to the actual tables is denied, and access is granted only to views. One of the challenges of the view approach is a need

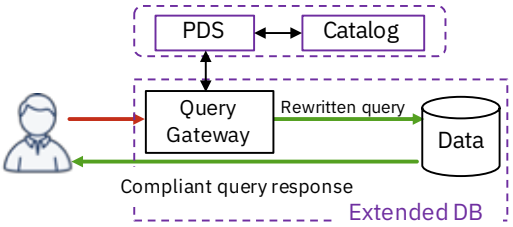

Figure 4: Query Rewrite Approach

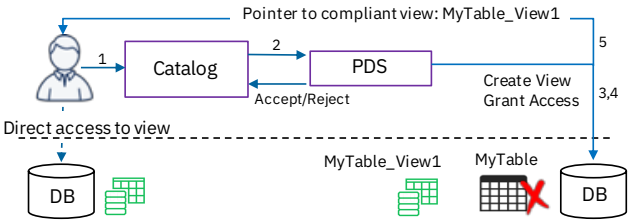

Figure 5: Compliant View Creation for *Metadata Before Data*

to manage the life cycles of the views similar to the alternate asset approach.

Views may be created by the interception point upon receipt of a query similarly to the query rewrite approach described above. Another option, most suitable for cloud use-cases, in which access to data is preceded by retrieving its information from a catalog, is that the view creation is initiated by the catalog, we called this creation model *metadata before data*. Figure 5 shows this approach, and the flow is as follows:

1. User queries the catalog for an object.
2. The Catalog calls PDS on each access to data object with the user details and object information.
3. PDS computes the policies; and if access is allowed, creates a complaint view, and grant access to it.
4. PDS responds with *allow* or *reject* to the user and the view details.
5. User can access the newly created view.

If there are performance implications to creating views on demand, a possibility is to generate in advance, all views associated with the defined policies. This could of course unnecessarily generate views that are infrequently used, further adding the overall cost of their life cycle management.

Policies that cannot be implemented as SQL (e.g., complex validation, or masking algorithms) can be supported by building UDFs, which are available in most of the common database solutions. The UDF implementation can be stored as additional metadata in the Catalog or alongside the policies and be applied when needed, e.g., when a view is created.

4.2 Alternate Asset with Materialized View

An *alternate asset* approach can be implemented for databases using materialized views. Creating a compliant replica of the original table data. We note that the same challenges described in Section 3.2 apply here as well, that is, additional storage space, life cycle management and garbage collection when policy changes.

5 Related work

An overview and research challenges for policy-based transformations for governance is described in [12]. A query review or view creation approach has been described in Stonebraker and Wong, Denning et al. and LeFevre et al. [13,18,21]. We propose a more generalized approach that applies deep enforcement and unified governance architecture across different types of data stores. Some data stores have adopted built-in fine-grained access control (e.g., HBase [2] and Accumulo [5]) and could integrate policies defined in PDS and transformed by TG. More generic solutions are available (e.g., Apache Ranger [7] and Apache Sentry [8]). They introduce a data store level agent that enables row- and column-level filtering and column-level masking, leveraging metadata and tag definitions in Apache Atlas [3]. These tools are specific for the Hadoop ecosystem and for each data store there is a different policy language and definitions, hence providing no unified governance architecture.

Hamlen et al. [15] describe a high-level policy enforcement framework for cloud data management at the platform level. They mention that query rewrite and materialized views as a possible approaches. However, a platform level solution could be bypassed as opposed to a deep enforcement approach.

6 Discussion

Providing unified governance across the data in the cloud is an important challenge that need to be resolved; we present an approach that can be deployed on multiple types of data stores. Moreover, we present two designs—the inline and the alternate asset approach—to address different deployment tradeoffs. The inline approach is easier to consume from an application perspective, as existing applications can be supported as-is. On the other hand, it adds latency and not all transformations (e.g., differential privacy) can be easily implemented. Furthermore, repeated queries will result in repetition of the same transformation logic, and in inefficient use of resources. The alternate asset approach provides an optimization for repeated queries, as the transformation is to be run once. The cost is the need to increase the storage footprint of the data store to keep transformed copies of the data, and the need to perform life cycle maintenance of the transformed copies either due to policy changes or capacity maintenance. Properties of these two approaches can be combined, for example, performing transformation in-line and keeping cached copies of the transformed data.

References

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

[2] Apache. Apache Accumulo. https://hbase.apache.org/.

[3] Apache. Apache Atlas. https://atlas.apache.org/.

[4] Apache. Apache Flink® - Stateful Computations over Data Streams. https://flink.apache.org/.

[5] Apache. Apache HBase. https://accumulo.apache.org/.

[6] Apache. Apache Parquet columnar storage format. https://parquet.apache.org/documentation/latest/.

[7] Apache. Apache Ranger. https://ranger.apache.org/.

[8] Apache. Apache Sentry. https://sentry.apache.org/.

[9] Apache. Spark Streaming makes it easy to build scalable fault-tolerant streaming applications. https://spark.apache.org/streaming/.

[10] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1383–1394. ACM, 2015.

[11] TIBCO Software Inc. Arno Unkrig. Janino - Super-small, super-fast Java compiler. https://janino-compiler.github.io/janino/.

[12] Elisa Bertino and Elena Ferrari. Big data security and privacy. In *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years.*, pages 425–439. 2018.

[13] D. E. Denning, S. G. Akl, M. Heckman, T. F. Lunt, M. Morgenstern, P. G. Neumann, and R. R. Schell. Views for multilevel database security. *IEEE Transactions on Software Engineering*, SE-13(2):129–140, Feb 1987.

[14] Google. Cloud Data Loss Prevention. https://cloud.google.com/dlp/.

[15] Kevin W Hamlen, Lalana Kagal, and Murat Kantarcioglu. Policy enforcement framework for cloud data management. 2012.

[16] IBM. Data Policy and Consent Management. http://www.research.ibm.com/haifa/projects/imt/consent/index.shtml.

[17] IBM. IBM Cloud Object Storage. https://www.ibm.com/cloud/object-storage.

[18] Kristen LeFevre, Rakesh Agrawal, Vuk Ercegovac, Raghu Ramakrishnan, Yirong Xu, and David J. DeWitt. Limiting disclosure in hippocratic databases. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases*, pages 108–119, 2004.

[19] PostgreSQL Community Association of Canada. PostgreSQL Database Management System - a powerful, open source object-relational database system. https://www.postgresql.org/about/.

[20] C. M. Saracco and D. J. Haderle. The history and growth of ibm's db2. *IEEE Annals of the History of Computing*, 35(02):54–66, apr 2013.

[21] Michael Stonebraker and Eugene Wong. Access control in a relational data base management system by query modification. In *Proceedings of the 1974 Annual Conference - Volume 1*, ACM '74, pages 180–186. ACM, 1974.