Microsoft Purview: A System for Central Governance of Data

Shafi Ahmad, Dillidorai Arumugam, Srdan Bozovic, Elnata Degefa, Sailesh Duvvuri, Steven Gott, Nitish Gupta, Joachim Hammer, Nivedita Kaluskar, Raghav Kaushik, Rakesh Khanduja, Prasad Mujumdar, Gaurav Malhotra, Pankaj Naik, Nikolas Ogg, Krishna Kumar Parthasarthy, Raghu Ramakrishnan, Vlad Rodriguez, Rahul Sharma, Jakub Szymaszek, Andreas Wolter
Microsoft Corporation
[shahmad,diminnal,srbozovi,elnatad,saduvvuri,stevengo,nitgup,johammer,nikal,skaushi,rakesh.khanduja,prmujumd,gamal,pankajn,niogg,krishkp,raghu,vlrodrig,sharmarahul,jaszymas,anwolter]@microsoft.com

ABSTRACT

Modern data estates are spread across data located on premises, on the edge and in one or more public clouds, spread across various sources like multiple relational databases, file and storage systems, and no-SQL systems, both operational and analytic; this phenomenon is referred to as *data sprawl*. Data administrators who wish to enforce compliance across the entire organization have to inventory their data, identify what parts of it are sensitive, and govern the sensitive data appropriately — across the entirety of their sprawling data estate. Today, governance of data is completely siloed; each of the data subsystems has its own (and varied) governance features. Policies applied to sensitive data are applied piece-meal by iterating over all the data sources in a custom language specific to each source. This makes data governance cumbersome, error-prone (because a given policy must be manually enforced across different subsystems, inconsistencies can easily arise), and expensive.

This paper presents *Microsoft Purview*, a service for unified governance of the entire data estate of an organization from a single central pane of glass. The Purview service consists of three parts: (1) a *Data Map* or *metadata catalog* that is populated by automated scanning of data sources in the organization, (2) a system to store and manage sensitivity *classification* of data, and (3) a *policy* system that enables data security officers to author and implement policies that span the entire organization, e.g., a policy that says, "Non-full-time employees should be denied access to data classified as PII (Personally Identifiable Information.")

Purview transforms data governance across a complex data estate by offering the ability to govern centrally and automating data discovery, classification and policy enforcement. While other commercial catalog systems also build a global catalog, Purview is unique in its support for policies. It is also distinguished by covering both structured and unstructured data, thanks to its deep integration with Office 365 and its governance framework; indeed, "Microsoft Purview" represents a new unified offering that combines Office 365 governance and what was formerly a service for governing structured data called "Azure Purview".

By integrating with Office 365's Rights Management Service, Purview offers central governance over structured data stored in databases and stores, reports in systems such as Power BI, as well as document data stored in Office 365. The Purview vision is to make the metadata in the Data Map increasingly richer through further automation and curation support and to use this 360 degree view of the data estate to support a wide range of governance policies, ranging from access control to lifecycle management (e.g., retention, deletion, restricting data movement). This paper covers the design and implementation challenges in building the Purview service for Attribute-Based Access Control (ABAC) policies, focusing specifically on a detailed description of its integration with Azure SQL Database. We illustrate the power of unifying Office 365 governance with structured data governance through Purview policies that enforce consistent access control even as data flows between Office 365 and structured data engines like Azure SQL Database. We also describe the results of our empirical evaluation of the performance overheads imposed by Purview.

PVLDB Reference Format:
Shafi Ahmad, Dillidorai Arumugam, Srdan Bozovic, Elnata Degefa, Sailesh Duvvuri, Steven Gott, Nitish Gupta, Joachim Hammer, Nivedita Kaluskar, Raghav Kaushik, Rakesh Khanduja, Prasad Mujumdar, Gaurav Malhotra, Pankaj Naik, Nikolas Ogg, Krishna Kumar Parthasarthy, Raghu Ramakrishnan, Vlad Rodriguez, Rahul Sharma, Jakub Szymaszek, Andreas Wolter. Microsoft Purview: A System for Central Governance of Data. PVLDB, 16(12): 3624 - 3635, 2023.
doi:10.14778/3611540.3611552

1 INTRODUCTION

Organizations have complex data estates, spread across many data subsystems. First, data is split between on-prem, edge and cloud. Second is the division between operational stores and analytic stores. Third, there are various kinds of each, e.g., operational stores can be relational systems or no-SQL systems, and analytic stores can be data lakes or relational warehouses. Big organizations typically use some combination of all of the above.

Let us consider the role of a data security officer who is entrusted with enforcing compliance across the entire organization. The officer would first need to conduct an inventory of the data across the entire organization, which is expensive given the complexity of the data estate. She would then need to identify which portions of the data are sensitive, i.e., need to be protected in some way. This is also a challenging task since not all data is sensitive and identifying the subset that is requires iterating over the entire inventory, one subsystem at a time. Third, if the officer wishes to enforce any policy across the organization, then she would be faced with a further challenge in that the data is scattered across not only

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611552

a large number of sources, but also a heterogeneous set of sources, each with its own language for expressing security policies. Hence, the officer would need to iterate over all data subsystems, and translate her policy into a specific "local" policy for each subsystem in the data estate. The policy language for relational databases uses grant-revoke based on ANSI SQL, but the one for data lakes is based on file-system access control lists. In short, data governance is completely siloed across a large set of heterogeneous sources, and hence laborious, cumbersome and error-prone. To address this situation, we introduce the approach of unified data governance. While other global catalog systems exist, we are the first to support fine-grained policy enforcement out of the box.

1.1 Microsoft Purview: Central Governance

This paper presents *Microsoft Purview* [8], a service for unified governance of the entire data estate from a single pane of glass. While there are related offerings from Amazon Web Services [2], Databricks [10] and Google [12], none of these systems support governance at a fine-grained level within the contents of a database. Another distinguishing factor of Purview is its unique integration with Office 365, a topic we discuss later in this section.

The Purview service consists of three parts. First, there is a *metadata catalog* that is maintained by automated scanning of all resources in the data estate to extract their schema and lineage; we call this the *Data Map*. The Data Map provides data administrators a central inventory of all of the organization's data. Second is a system to store and manage *data classification*. In order to classify data, Purview supplies an extensible taxonomy of *sensitivity labels* that a data security officer can customize for her organization. Purview also provides the ability to attach sensitivity labels to metadata elements such as columns. It does this by letting the data security officer supply a set of *classification rules* that are applied to the catalog in order to derive the classification. Purview also provides a class of in-built classifiers that the data security officer could apply. The classifiers tag at the metadata level, based on sampled subsets of the data as input. The results of classification are reflected in the Data Map.

Third, Purview provides a way for data security officers to state and enforce policies across the entire estate. Purview aims to support many kinds of policies ranging from access control to lifecycle management (e.g., data retention, deletion, sovereignty restrictions on data movement). In this paper, we focus on access control policies that can be applied across a data estate with heterogeneous subsystems. Some example policies include:

- *P1:* Only full-time employees can be granted access to data that is classified as PII (Personally Identifiable Information.)
- *P2:* Only full-time employees that are also part of the devops organization can be granted access to data that is classified as customer support.
- *P3:* Data scientist Alice can read any data that is classified as appropriate for reporting.

Access control policies, as in the examples above, use classification attributes to span across all subsystems in the data estate. The classification is a *attribute of the data*. The example policies also specify *attributes of the requestor*, and could additionally specify attributes about the environment (although we will not discuss these). Thus, Purview supports *Attribute Based Access Control*, or *ABAC* for short.

There are a few points worth calling out about Purview's ABAC policies. First, in contrast with the siloed approach in today's world, Purview policies offer a consistent and scalable way to govern the entire data estate. For instance, policy P1 above would be simultaneously interpreted in a relational database to apply to columns classified as PII, and in a storage system to apply to files classified as PII. Second, the succinctness of the policy matches the policy author's intent. The policy author does not need to express policies in source-specific languages, e.g., grant-revoke syntax for relational databases, or access control lists for file/cloud storage. Third, Purview policies co-exist with local policies in individual sources and are not intended to wholly replace them. Purview policies are interpreted as *mandatory*. For instance, policy P1 compels the entire data estate to deny access to data classified as PII to non-full-time employees. An individual administrator of a particular source can further restrict access to only a small subset of full-time employees, but cannot override the policy to grant access to a non-full-time employee. Fourth, our main focus with Purview is to enable governance to be carried out consistently and at scale. However, in terms of expressive power, it does not increase the scope of what can be done today. Increasing the expressive power of ABAC policies is a non-goal for our first step.

In this paper, we address various challenges in the design and implementation of Purview's ABAC system. By design, our system is distributed. Policies are authored and stored in the Purview service, but they are enforced by individual data sources like a relational database. Further, Purview needs to scan a source's metadata. A crucial question that arises in this setting is what is the source of truth for policies. One could imagine a system with dual sources of truth, where the local and Purview state is kept in sync using 2-phase commit. However, in our current release, we choose a simpler approach based on customers' requirements. We treat Purview as the single source of truth for policies and the local store as the source of truth for metadata. The relevant state, both locally and in Purview is updated *asynchronously*. Purview fetches metadata to update its catalog, and distributes policies to individual sources, which cache them locally and apply them in conjunction with their local policy.

The integration of Purview policies locally falls into two categories. For some engines like SQL Server and Azure Storage, we pursue *native* integration of Purview. In this approach, the permission logic of the engine is modified to also enforce Purview policies. Since the evaluation of Purview policies is identical across the various engines with native support, it is abstracted into a common component. For other engines, such as servers not owned by Microsoft (e.g., Oracle), we use a *proxy* based approach. Here, we develop a proxy that rewrites queries to enforce the Purview policy. While we briefly outline the proxy-based approach in this paper, our main focus is on describing native support. We use SQL Server as a detailed illustration of native Purview integration.

In the native integration approach, a data source locally *caches* policies fetched from Purview. The cache is used to enforce policies without a service call to Purview, and also to support the case where Purview is temporarily unreachable. Since permission checking is invoked during the execution of every query or request, caching

policies in the local engine greatly improves performance. A common access pattern with SQL is for a single user to connect to SQL and then issue queries in a loop, with queries often accessing the same tables and columns each time. As such, we cache the result of calls to the Purview component in order to evaluate repeated evaluation of the same policy, in the expectation that policy updates are not frequent. We empirically evaluate the impact of the above optimizations by running a benchmark based on TPC-C.

1.2 Integration with Office 365 Rights Management Service (RMS)

The Rights Management Service (henceforth, RMS) offered by Office 365 [17] classifies documents and offers central policy enforcement for unstructured data in Office 365, such as Excel spreadsheets and Word docs. RMS supports acess control policies through encryption. Briefly, an RMS protected document is encrypted with keys held by the RMS service. Access control is enforced through a key-release policy that determines the users who are authorized to receive keys that can decrypt the document. RMS policies are centrally enforced. At the time the document is accessed, a call is made to the RMS service, which checks whether the decryption keys can be released to the user attempting to access the document. The document may be stored in various file systems and storage accounts, and it may flow through various applications, e.g., as an email attachment or Office 365 Sharepoint. Throughout this lifecycle, the RMS policy is enforced.

Purview extends the notion of Rights Management to structured data. However, unlike Office 365's RMS, it extends across diverse and distributed data estates, and enforces access control over structured data through traditional access control mechanisms supported by the local data sources. Purview is integrated with Office 365 RMS through a hub-and-spoke model. A subset of the ABAC policies including some of the above examples are hosted in a policy hub shared by Purview and Office 365 RMS. The policies span structured data and documents. They are applied as RMS policies over documents and through traditional access control mechanisms backed by Purview over structured data engines. Furthermore, the policies are applied as the data flows. As a brief motivating scenario, consider a document authored in Microsoft Excel [15] that is classified as Confidential, protected through RMS and hence encrypted. A user who has access to the document could open it and load the data into a SQL database. When the data flows into SQL, it is re-classified by Purview and the same policies that were applied to Excel in the form of RMS are applied to the data in SQL in the form of ABAC policies, restricting access to columns classified Confidential. Now consider a business user authoring a report in Microsoft PowerBI [16] over the data in SQL. As noted, when queries are run over SQL, ABAC policies are applied. Furthermore, SQL propagates the labels on the base data into the columns returned as part of the query result and these labels *flow* into the PowerBI client. When the PowerBI report is exported in the form of an Excel document, the corresponding RMS policies are automatically applied and the document is encrypted if the policy requires it. We will illustrate the above scenario in greater detail later in this paper.

1.3 Organization

While the integration with Office 365 discussed above is a crucial part of Purview, for ease of exposition, most of this paper focuses on structured data. The remainder of this paper is organized as follows. Section 2 gives an overview of all the Purview functionality and user experience, including a discussion of the hub-and-spoke model integrating Purview with Office 365. Section 3 discusses example scenarios illustrating the power of Purview, including data flows between Office 365 and structured data engines. Section 4 describes the overall architecture. Section 5 provides implementation details. Section 6 contains the results of performance experiments. Section 7 discusses related work. Section 8 summarizes the paper with avenues for future work.

2 OVERVIEW

In this section, we discuss how Purview is configured and the functionality it provides.

2.1 Initializing Purview

The first step in using Purview is to initialize it. A Purview account is initialized with all the classification taxonomy used in Office 365 [17]. As part of initialization, users can setup additional classification labels and classification rules, which will be defined precisely later in Section 2.2. Data sources are included in Purview governance through a process of *registration*.

2.2 Catalog and Classification

After registration, Purview scans the metadata in a given source and classifies it. As noted in Section 1, Purview governance can be applied to a variety of sources. Most of the discussion in this section will focus on SQL Server and Azure Storage.

A *classification label* is a free-form string. As noted above, there is a pre-defined taxonomy of labels that is drawn from Office 365. A Purview administrator can extend the taxonomy with additional labels.

In the case of Azure Storage, the unit of classification is a file. In the case of SQL Server, the unit of classification is a column. In future work, we will continue to extend the classification framework to include other object types. Every classified object can be associated with zero or more classification labels.

In Purview, there is also the concept of a *sensitivity label* which is also a free-form string drawn from a taxonomy separate from the one for classification labels. Unlike classification labels, every classified object is associated with at most a single sensitivity label. The sensitivity label associated with an object is supposed to aggregate its classification labels.

Example 2.1. Consider a customer of Purview named Contoso. As noted above, Purview provides pre-defined classification labels; e.g., MICROSOFT.GOVERNMENT.US.SOCIAL_SECURITY_NUMBER. Contoso could extend the taxonomy to include additional labels; e.g., CONTOSO.HR.EMPLOYEE_ID. Suppose Contoso uses employee social security numbers as a way to identify employees in a SQL database with a column SSN. Then, the column SSN could be associated with two classification labels: CONTOSO.HR.EMPLOYEE_ID, MICROSOFT.GOVERNMENT.US.SOCIAL_SECURITY_NUMBER.

However, it can only be associated with at most a single sensitivity label, e.g., Highly Confidential.

2.3 Classification Rules

Data is classified using an extensible framework of *classification rules*. For classification, both the metadata and a small subset of data is examined. Hence, the classification rules reference both. Purview supports a rich rule language that encompasses regular expressions and dictionaries. While Purview has an extensive set of in-built rules, users can extend them to include custom rules.

Example 2.2. An example of an in-built classification rule is a rule to identify US Social Security Numbers. A user could extend these rules with custom rules.

- If Contoso from Example 2.1 represents all employee identifiers with the prefix Employee followed by a five-digit numeral, then it could add a custom classification rule such as Employee[0-9][0-9][0-9][0-9][0-9] corresponding to the classification label CONTOSO.HR.EMPLOYEE_ID. The above rule would match actual data values.
- Rules can also reference dictionaries. For instance, Contoso could supply a dictionary of product categories corresponding to the label ProductCategory.
- In general, rules can also match metadata. For instance, if Contoso stores all customer names in columns that contain the string Name, then it could include a rule %Name% to match column names.

As noted above, classification rules are aggregated into sensitivity labels. The aggregation takes place using a set of *aggregation rules* that list the classification labels that derive a given sensitivity label. As with classification, while Purview has built-in rules, they can be extended by users.

Example 2.3. A Purview admin from Contoso could author a rule that stipulates that any object that is classified as MICROSOFT.GOVERNMENT.US.SOCIAL_SECURITY_NUMBER or CONTOSO.HR.EMPLOYEE_ID must be associated with the sensitivity label Highly Confidential.

Purview also uses rules based on the classifiers run by Office 365 to classify documents. These classifiers use sophisticated machine learning techniques that have evolved over time.

2.4 Attribute Based Access Control (ABAC)

We now describe the framework used to define Purview policies. As noted in Section 1, while our long-term goal is to support a rich class of data governance policies through Purview, in this paper, the focus is on access control.

2.4.1 Policy Language. An access control policy consists of four fundamental components listed below.

- A *principal* that refers to an identity, or a group of identities. For this paper, we assume a single identity provider such as Azure Active Directory [5]. For instance, an individual user, e.g., Alice is a principal, as is a group such as FTE denoting all full-time employees of an organization.
- A *resource* that refers to an object in a data source under Purview's governance. We associate every resource with a unique *name* drawn from a hierarchical namespace. The namespace is an extension of Azure's existing namespace for resources visible in Azure, e.g., a SQL database or an Azure Storage container. Azure resources are organized in a hierarchy where the *subscription* is at the root, followed by a *resource group* used to identify groups of resources, followed by an Azure resource. The name of a resource reflects the above hierarchy. For instance, the name /Subscriptions/D9E312A3-1672-44F6-AF82-72535AC74879/ResourceGroups/MyResourceGroup/Microsoft.SQLServers/MyServer/Databases/MyDb refers to the database MyDb contained within server MyServer, contained within the resource group MyResourceGroup, contained within subscription D9E312A3-1672-44F6-AF82-72535AC74879. We extend it to both address resources contained within, e.g., tables and columns for SQL and files for Azure Storage. For instance, the name /Subscriptions/D9E312A3-1672-44F6-AF82-72535AC74879/ResourceGroups/MyResourceGroup/ Microsoft.SQLServers/MyServer/Databases/MyDb/Schemas/MySchema/Tables/MyTable/Columns/MyColumn refers to the column MySchema.MyTable.MyColumn contained within the database MyDb. We also extend the namespace to address resources outside Azure.
- An *action* that refers to an action that a principal can perform on a resource. Example actions include read, insert, update, delete. While many actions such as the above are applicable to multiple data sources, some actions such as connect are specific to a data source such as a relational database.
- A *decision* that can take three values: grant, deny, inapplicable.

In order to succinctly describe policies, we introduce the concept of a *role* which is a set of actions. For instance, we could define a role reader consisting of actions connect, read. We also introduce the notion of a *scope* that defines a set of resources using a limited regular expression, as we illustrate below.

- The scope /Subscriptions/D9E312A3-1672-44F6-AF82-72535AC74879/ResourceGroups/MyResourceGroup/Microsoft.SQLServers/MyServer/Databases/∗ refers to the set of all databases in the server MyServer.
- The scope /Subscriptions/D9E312A3-1672-44F6-AF82-72535AC74879/ResourceGroups/MyResourceGroup/Microsoft.SQLServers/∗∗ refers to all servers, databases and objects within, contained in the resource group MyResourceGroup.

Our limited regular expression language only allows the wild-cards ∗, ∗∗ to be used instead of a literal. The wild-card ∗ matches any single literal, whereas ∗∗ matches any path.

An access policy consists of a set of *rules*. We use the term *policy rule* to distinguish it from classification and sensitivity rules. A policy rule maps the combination of a principal, a role and a scope to a decision.

Example 2.4. An example policy rule maps principal Alice, a role reader and a scope /Subscriptions/D9E312A3-1672-44F6-AF82-72535AC74879/ResourceGroups/MyResourceGroup/Microsoft.SQLServers/MyServer/Databases/** to decision Grant.

Each policy rule is associated with an optional predicate on the sensitivity label.

Example 2.5. The policy rule in the above example could be associated with a predicate Sensitivity_Label = Public.

For ease of exposition, we focus on predicates in the class where the sensitivity label is constrained to be equal to a constant. It is straightforward to extend everything we describe in this paper to include (1) predicates on classification labels and (2) boolean combinations of predicates. An important property of the above class of predicates is that they are deterministic. When evaluated on a given catalog, classification and policy, the output of the predicate is fixed. We will discuss extensions to non-deterministic predicates later in the paper, e.g., a predicate TimeOfDay >= 9am and TimeOfDay < 5pm.

2.4.2 Policy Semantics. The semantics of the policy language described above is intended to produce a decision given the following triplet as input: an individual user, a fully specified resource, and an action.

A policy rule is said to *match* the above input if (1) the individual user in the input equals the principal or is a member of the user-group referred to in the rule, (2) the input resource is contained within the scope in the rule, (3) the input action equals the action in the rule. The overall decision is defined by aggregating the decision obtained from the matching rules as follows: (1) Grant if at least one matching rule results in a Grant and no rule results in a Deny, (2) Deny if at least one matching rule results in a Deny, (3) Inapplicable otherwise.

Example 2.6. Suppose that we have two policy rules.

- The first rule maps principal Alice, a role reader and a scope /Subscriptions/D9E312A3-1672-44F6-AF82-72535AC74879/ResourceGroups/MyResourceGroup/Microsoft.SQLServers/MyServer/Databases/** to decision Grant, hence allowing Alice to read any data in any database within the MyServer SQL Server.
- The second maps principal NonFTE, a user group denoting non-full-time employees, a role reader and a scope /Subscriptions/D9E312A3-1672-44F6-AF82-72535AC74879/** to decision Deny with the condition Sensitivity_Label = Confidential, hence denying access to data labeled as Confidential to non-full-time employees, anywhere within the entire subscription.

Suppose that principal Alice belongs to the group NonFTE, and that the server MyServer contains database MyDb that has a table MySchema.MyTable with two columns Column1, Column2 of which Column1 is labeled Confidential.

- For Alice to connect to the database MyDb, she requires a connect permission to the database. The above input matches the first rule above since the reader role contains the connect action. It fails to match the second rule since the database is not labeled Confidential. Therefore, under Purview governance, the connect permission is granted.
- Suppose Alice wishes to run the query select Column2 from MyTable. She requires a Read permission on the objects MyTable and Column2. The above input matches the first rule, but not the second because Column2 is not labeled Confidential. Hence, the output of the permission check is a grant.
- Suppose Alice wishes to run the query select Column1 from MyTable. She requires a Read permission on the objects MyTable and Column1. The above input matches both rules since Column1 is labeled Confidential. The first rule results in a grant but the second results in a deny. This is aggregated to return a deny.

2.4.3 Aggregation with local permissions. The final permission check during execution does not rely solely on Purview. It combines Purview checks with local checks. The combination semantics is identical to the single grant-no deny semantics described above. Any deny, whether local or Purview-backed, overrides a grant. In the absence of a deny, we require at least a single grant for the permission to be granted, whether local or Purview-backed. We illustrate with our running example.

Example 2.7. Suppose Alice wishes to run the query select Column2 from MyTable. The Purview check results in a grant but if the local SQL check results in a deny, then it overrides the grant and the query fails. Similarly, if Alice wishes to run the query select Column1 from MyTable then the permission check always fails regardless of the local decision because the Purview check results in a deny.

While Purview policies are aggregated with local policies in the manner described above, they cannot be over-ridden by local administrators. In that sense, once a data source is under Purview governance, Purview policies are mandatory and always applied.

2.4.4 Expressive Power. The goal of Purview policies is to express policies consistently and at scale across a variety of data sources. It is not a goal to increase the expressive power of policies that are expressible in several of the local sources, e.g., SQL Server with advanced security features. For instance, in examples 2.6 and 2.7, a Purview policy that denies Alice access to data labeled Confidential is equivalent to denying access to Column1 and other columns labeled Confidential in SQL Server's grant-revoke-deny permission model.

2.5 Integration with Office 365 Rights Management Service: Hub and Spoke Model

As discussed in Section 1, Microsoft Purview is integrated with the Rights Management Service (RMS) offered by Office 365 [17]. Similar to Purview, RMS classifies documents and offers central policy enforcement. RMS supports acess control policies realized through encryption. Briefly, an RMS protected document is encrypted with keys held by the RMS service. Access control is enforced through a key-release policy that determines the users that can decrypt the document. RMS policies are centrally enforced. At the time the document is accessed, a call is made to the RMS service, which

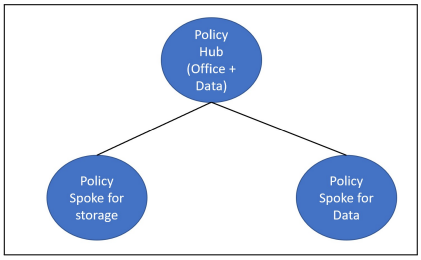

Figure 1: Hub and Spoke Model of Integration with Office 365 RMS

checks whether the decryption keys can be released to the user attempting to access the document.

Purview is integrated with the RMS system in the form of a hub-and-spoke model illustrated in Figure 1. At the level of the hub is a subset of the Purview policy language that can be applied across Office 365 and structured data engines. The subset only includes actions that are meaningful across documents and data, and a scope that is broad, e.g., at the level of a subscription or a tenant. We illustrate a policy stated in the hub through an example.

Example 2.8. An example policy rule stated in the hub maps principal group NonFTE, an action `read` and a scope `/Subscriptions/**` to decision Deny with a predicate Sensitivity_Label = Confidential. Stated in natural language, the policy denies access to all confidential data to principals that are not full-time employees of the organization.

The policy is interpreted in Office 365 as an RMS policy enforced through encryption. Accordingly, all documents labeled Confidential would be encrypted and the encryption keys would be denied to principals that are not full-time employees. The same policy is interpreted in the structured data engines as an ABAC policy as described previously in this section.

In order to support the full range of policies defined above, Purview also supports spokes. Figure 1 shows one spoke each for policies specific to storage and structured data engines. In general, there could be multiple spokes and multiple levels of spokes. For instance, we could have a spoke specific to SQL underneath the one for structured data overall. For any given engine, policies defined at all scopes relevant to it are applied.

3 EXAMPLE SCENARIOS

Purview combines the data catalog together with classification and policies to offer a rich data discovery and governance experience. Users can browse and search for data sources both based on the nature of the source as well as classification. We illustrate with examples.

3.1 Self-service

Suppose that a data scientist wishes to analyze customer and sales data in her organization. With the data catalog, Purview allows her to *browse* various data sources to find relevant sources. She could also use the output of classification to *search* for data that is classified as customer or sales data. This would provide her an inventory of data that is relevant to her analysis, not just in terms of listing sources, but also the nature of the sources, e.g. SQL vs storage, that would inform her of the nature of the analysis. Purview allows her to then easily request access to these resources; it also allows an administrator to review the request and approve (or deny) it. Without Purview policies, she would need to consult with every relevant data owner to grant her access to their source. With Purview policies, the Purview administrator could single-handedly grant her the minimum set of required permissions, namely to connect to relevant sources and read data classified as customer or sales. In this way, the data scientist could use Purview to greatly improve the efficiency of discovery and governance, thereby enabling her to increase her focus on the main task, namely data analysis.

3.2 Governance through data flows

As discussed in Section 2.5, Purview is integrated with Office 365 RMS via a hub and spoke model. This not only enables governance across documents and data, but it also enables governance as data *flows* from one engine to another. We illustrate with an example.

Suppose that the policy hub has two policy rules.

(1) A rule that maps principal Alice, an action `read` and a scope `/Subscriptions/**` to decision Grant with a predicate Sensitivity_Label = Confidential. Stated in natural language, the policy grants access to all confidential data to principal Alice.
(2) A rule that maps principal group NonFTE, an action `read` and a scope `/Subscriptions/**` to decision Deny with a predicate Sensitivity_Label = Confidential. Stated in natural language, the policy denies access to all confidential data to principals that are not full-time employees of the organization. (Alice is a full-time employee.)

Suppose that Alice authors a document in Microsoft Excel [15] that is classified as Confidential. It would be protected through RMS per the above policy and hence would be encrypted. Alice has access to the encryption keys, opens the document in Excel and loads the data in the document into a SQL database. Once the data lands in SQL, it is re-classified by Purview. Suppose that Purview labels column Name in table Customer to be Confidential. When the data is in SQL, the policies stated above are applied as ABAC policies. Now, suppose Alice authors a report in Microsoft PowerBI [16] and fetches the data in SQL by issuing a query `select * from Customer`. As per the policy above, Alice has access to all columns in the Customer table and hence would be able to run the query successfully. Furthermore, SQL propagates the labels on the base data into the columns returned as part of the query result and the column Name returned in the query result would be labeled Confidential as it flows into PowerBI. When the PowerBI report is exported in the form of an Excel document, PowerBI applies the label Confidential to the entire document; the corresponding RMS policies are automatically applied and the document is encrypted.

4 ARCHITECTURE

In this section, we describe the overall system architecture shown in Figure 2. The rest of the paper focuses exclusively on Purview while keeping the details of the Office 365 RMS integration out of scope for ease of exposition.

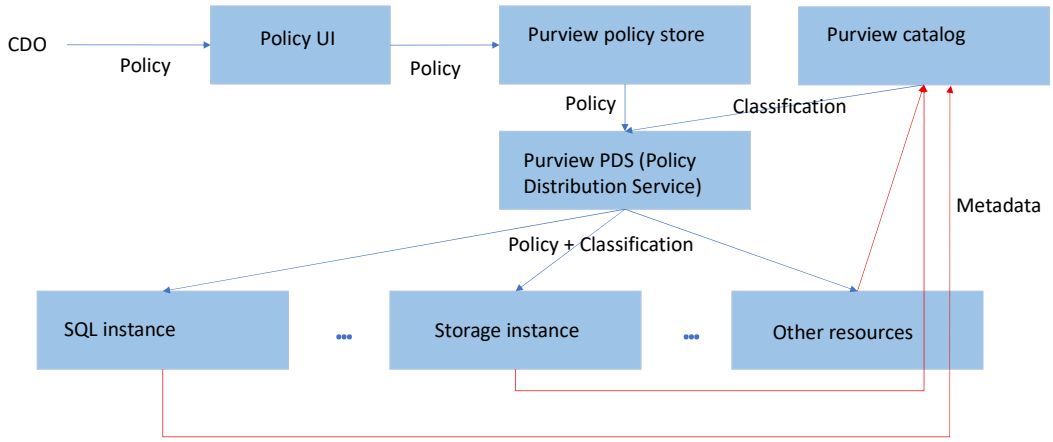

Figure 2: Purview system architecture.

4.1 Catalog and Classification

The catalog is populated by scanning every data source that is registered with Purview. Purview scans the source by connecting to it as a client. During registration of a given data source, a credential for the source is configured that lets Purview perform its scan. The scan involves examining metadata, but also (subsets of) data for the purposes of classification. Classification rules in general are applied on both metadata and data. In order to improve the efficiency of the scan, two optimizations are performed. First, the scan is scaled out by being performed close to a given source. In Azure, the scan is performed in the same region as the source. Second, the scan only examines a prefix of the data. Currently, it is a fixed prefix of 128 rows. In future, we plan to extend it to perform various kinds of random sampling. It is not required for all the scanned rows to match a given classification rule. The matching threshold is also input as part of configuring a rule.

We note that in our approach, only the final result of the scan, namely the metadata and the classification, is sent to the Purview service, thus minimizing the data transferred over the network and facilitating compliance with laws and organizational policies governing data movement. As with other data flows in Figure 2, the catalog is populated asynchronously. The default time period for scans is once every 8 hours, but this can be customized by users.

4.2 Policy Enforcement: Asynchrony

As shown in Figure 2, Purview policies are authored via a policy user interface and stored in a policy store. The figure hides the distinction between hub and spoke interfaces, which can be abstracted away into a single interface for the purposes of discussing the system architecture. Purview uses Azure CosmosDB [6] to store them centrally. Policies are enforced locally in the data source involved. We treat Purview as the source of truth for policies and the local store as the source of truth for metadata. Policies are enforced *asynchronously.* As noted above, the Purview catalog is also updated asynchronously. Purview policies are cached locally for enforcement. This asynchronous design gives us two crucial advantages. By serving permission checks off a local cache, we avoid a call to Purview in the inner loop, enabling (a) better performance, and (b) better fault tolerance when Purview is unavailable, since permission checks can then be served off an older policy.

4.3 Policy Distribution Service (PDS)

For the propagation of policies, Purview runs a *Policy Distribution Service (PDS).* Local stores fetch policies from PDS either by receiving notifications on updates, or by polling for updates. While policy updates are not immediately reflected in permission checks by design, our goal is to have them reflected without a significant delay. Hence, sources that poll do so frequently. SQL Server uses polling because we support Purview policies in both the on-prem and Azure products, and the on-prem products do not support any notification infrastructure. SQL Server polls PDS every five minutes.

A basic interface supported by PDS is a *full pull* where PDS returns the entire policy directly from the policy store. In addition, PDS supports an interface for incremental updates, both via polling i.e., *delta pull*, and notifications. PDS uses an off-the-shelf publish/subscribe system, namely Azure Event Hub [7] (henceforth referred to simply as Event Hub), in order to support deltas. This solution satisfies our scale and availability requirements. In terms of semantics, from the point of view of a source that is receiving events, Event Hub guarantees that every event is received exactly once but events could arrive out of order. For instance, if a policy is updated first by inserting a rule and then subsequently deleting it, the delete event could be received at a source before the insertion.

4.4 Policy Decision Point (PDP)

Policies are applied at local sources. We abstract the application of a policy into a component called the *Policy Decision Point (PDP).* The PDP assembles the policy fetched from PDS and evaluates a permission check against the policy. There are local sources owned by Microsoft, such as SQL Server and Azure Storage. Since Purview

supports governance of all sources, there are other sources, e.g., an Oracle database, that are not owned by Microsoft. Accordingly, the way in which PDP is integrated into a given data source falls into two categories.

For sources not owned by Microsoft: We enforce policies by running a *proxy* with PDP. We briefly describe the proxy using the example of a relational database such as Oracle. The proxy intercepts every query, parses it in order to identify the set of permissions required to run the query, and checks the permission against the policy using the PDP.

Example 4.1. We continue Example 2.6. Suppose Alice wishes to run the query `select Column2 from MyTable`. The proxy would parse the query to determine that she requires a `Read` permission on the objects `MyTable` and `Column2`. It would check the permissions against the policy using the PDP. As per the policy described in Example 2.6, the result is a grant. Hence, the query is forwarded to the underlying server for execution. If the result of the call to PDP is a `deny` then the query is rejected from the proxy itself.

For sources owned by Microsoft: PDP is integrated natively into the source code of the underlying engine, and the PDP checks are integrated into the core permission checking logic of the engine. The rest of the paper describes native enforcement in detail by using the example of SQL Server. Briefly, policies are fetched from PDS directly by individual SQL Server instances, and persisted in the local user database. The SQL Server process is modified to integrate the PDP. The PDP is prepared by giving as input the policy and attributes read from the user database. During query execution, at permission checking time, the PDP is invoked to obtain a decision from the Purview sub-system. The decision is aggregated with local permission checks to evaluate the permissions needed for the query. The performance advantages of native integration over the proxy based approach are discussed in Section 6.

5 IMPLEMENTATION IN SQL SERVER

We illustrate the native integration of Purview policy enforcement through the case study of SQL Server. Other systems such as Azure Storage and Cosmos Db also enforce policies through native integration. While some of the design details differ from SQL Server, the main points described below also apply to them. Also, for ease of exposition, most of this section focuses on policies without predicates, and hence without attributes. The extensions to include attributes are straightforward and discussed in Section 5.4.

5.1 PDP Algorithm

We begin by describing how the PDP component is implemented. Recall from Section 2.4 that a policy consists of a set of policy rules, each consisting of a principal, a scope, a role being mapped to a decision along with an optional predicate on sensitivity labels. The policy rules are provided to the PDP as a part of pre-processing and stored in an in-memory cache. In addition to the rules, the PDP also stores metadata per rule, specifically a rule identifier and a version number. The rule identifier and version number are both generated by the Purview policy store and distributed through PDS. The version number is a monotonically increasing counter that is incremented with every update.

5.1.1 Updating rules after receiving a delta. When the system performs a full pull, the PDP resets the policy state to reflect the new policy. When the system receives a delta (either through polling or notification), the policy state is incrementally maintained using the rule metadata. Specifically, every delta event contains a rule, including its metadata, and an indicator of the nature of update performed, i.e. whether it was an insert, delete or (in-place)update. We use the rule metadata to update the policy appropriately, also accounting for out of order delivery during delta pull, as follows:

- If the policy set has a rule with the same identifier as the incoming event, but with a higher version number, then we ignore the event.
- If the policy set has a rule with the same identifier as the incoming event, but a lower version number, then we update the rule in the policy set appropriately. If the incoming event is a delete, then the rule is marked with a boolean deleted flag in the policy set. We do not delete the rule (to account for subsequent events that might arrive for the same rule, but with a lower version number.)
- If the policy set does not have a rule with an identifier matching the incoming event, then we add a corresponding rule to the policy set. If the incoming event is a delete, then the rule is marked with a boolean deleted flag in the policy set.

We illustrate with examples. Suppose that a new rule is inserted into the policy set. The new rule would have a unique identifier. Since the delta API guarantees that all updates are delivered, the new rule is guaranteed to get inserted into the set stored locally by the PDP eventually, according to the method above. Similarly, if a rule gets updated, the update is guaranteed to eventually be applied in the local policy set.

We now illustrate the rationale for soft deletes. Suppose that a new rule is inserted in Purview but is immediately followed by a delete, which can happen if the policy author wishes to undo the insertion. It is possible that SQL receives the delete event before the insert event. When we receive the delete event and there is no existing rule with the same identifier, then according to the method above, we add the rule to the policy set but with a boolean indicating that it is deleted. When we receive the insert subsequently, the method would observe that there is already a rule with a matching identifier and a higher version number and hence ignore the insert. In contrast, if we do not add the rule during the processing of the delete event, then when we receive the insert out of order, we would be unable to determine if the rule being added was subsequently deleted.

Note that in the above method, the policy set keeps growing. We never actually delete a rule even if the incoming event is a delete. For garbage collection, we issue full pulls periodically. Since policy updates are rare, the frequency of full pulls can be low, e.g., once weekly.

5.1.2 Permission check. The input to a permission check is (1) a principal who is a user (as opposed to a group), (2) a resource that is specified through a path, and (3) an action. Given a set of policy rules, the goal of the permission check is to return an access decision, as defined by the semantics in Section 2.4. The current algorithm we implement in the PDP performs a linear scan of all

rules, identifies matching rules (as defined in Section 2.4) and aggregates their individual decisions into an overall decision. Checking whether an individual rule matches the input involves checking if (1) the input principal matches the corresponding principal in the rule, either directly or through a group membership, (2) the input action is contained in the role referred to in the rule, (3) the input resource matches the scope of the rule, and (4) the optional predicate on classification is satisfied. The details of the above checks are straightforward. In the case where the PDP is natively integrated, the host system provides the necessary inputs to the PDP to make its decision, e.g., the group memberships of a user principal.

Our algorithm based on a linear scan is optimized for the case of a concise policy. If policy sizes grow large, we could extend the algorithm to not have to scan all rules by building index structures, such as a trie on all scopes and a hash table on the principals that would help narrow down matches to rules with matching scopes and principals respectively. These extensions are part of future work.

5.2 Integration with SQL Query Processing and Security Caches

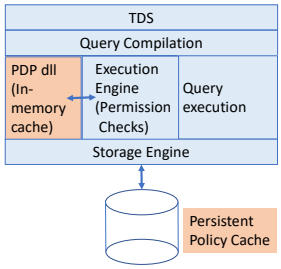

Figure 3: Purview integration with SQL query processing.

Figure 3 illustrates the integration of PDP into SQL query processing. Before we describe the integration, we first discuss how SQL permission checking works. Queries are compiled and the resulting execution plans are stored in a plan cache without any permission checking. When the query is to be executed, during a pre-execution step, permission checks are applied. If they succeed, then execution proceeds. But if the permission checks fail, then the query fails. Since permission checking is expensive, SQL has an array of in-memory security caches that reduce its cost. We describe two categories of caches that are relevant to this paper. SQL maintains an object level cache that stores for a given user, the result of a permission check for a table (and other objects like views), including all its columns. It also maintains a query-level cache that stores for a given user, the result of permission checks for all objects referenced in the query. If the user runs the same query but with different parameters, a common access pattern for SQL, all permission checks can be served off the query level cache. If the user changes the query but references the same underlying objects, then permission checks can be served off the object level cache. SQL's cache eviction policy is coarse-grained. If there is any change in the security metadata, then the cache for the entire database is cleared and re-populated on demand.

In order to fetch Purview policies from PDS, SQL Server runs a background task that polls PDS asynchronously. The task persists the fetched policy in the user database, so that queries can be served off the local cache even after a SQL restart, in the event that Purview is unavailable. SQL Server runs the PDP as a dll loaded into the SQL process. The PDP maintains an in-memory cache of the policy, which is input to it by the background task as the policy changes. SQL Server invokes PDP as part of its permission checking, and the result of the calls to PDP are aggregated with local checks *before* populating the security caches. In this way, we retain the use of security caches with Purview policies. In particular, for a given Purview policy, after the initial phase of populating security caches, in the "hot" path of iterative query execution for a given user, SQL Server returns all permission checks from its cache without invoking the PDP. We extend the cache eviction logic of SQL Server to clear the cache for the entire database when the Purview policy changes. In general, not all policy rules may apply to a given database. Before clearing the cache, we check if the scope of the updated rule includes the current database.

5.3 Discussion

We now expand on a few more implementation details.

First, in Azure, different clients do not share the same SQL Server process (i.e., we achieve multi-tenancy through process-level isolation.) Multiple databases in the same SQL process are guaranteed to belong to the same client. As clear from the example policies described in this paper, we expect a large degree of overlap between policies applicable to different databases. Therefore, running one PDP instance per database would be wasteful. Accordingly, we run a single PDP instance per SQL process. The single instance stores policies applicable to all databases contained in the SQL instance. Similarly, there is only one background task per SQL process fetching policies from PDS.

Second, since every SQL instance fetches policies directly from PDS, in order to minimize network traffic, we perform delta pulls through server restarts. As part of a delta pull, PDS returns a *sync token* that increments over time and needs to be presented to PDS as input to the subsequent delta pull. PDS guarantees that a sync token is unique within the context of a given resource path. As part of the persistent cache, we also store the sync token of the previous delta pull. When SQL restarts, we resume delta pulls with the persisted sync token. Notice that since a single SQL process can contain multiple databases, each with its own persisted sync token (and policy), we face the question of which sync token to use. While one could assume that in the common case, the persisted sync token and policy would be the same for all databases, in practice this is not in general true. For instance, if a new database is created, it may not yet have a persisted policy and sync token. A similar situation could arise if a database moves between different Azure resources, or if it is restored from an older backup. In order to address the above question, we also store a time-stamp indicating the local wall clock of the SQL process at the time we issue a delta pull in the persistent cache. When SQL restarts, we use the sync token of the database that has the most recent time-stamp. We also populate the

PDP with the corresponding policy that is used to answer queries in the duration before the first pull is initiated.

Example 5.1. Suppose a SQL process contains two databases, one of which is current and the other restored from an older backup. If the current database has a policy with a recent time-stamp, and the restored database has a policy with an older time-stamp, then when SQL restarts, it would use the policy and sync token associated with the current database.

We note that due to clock synchronization problems, it is possible that in the above example, the restored database has a more recent time-stamp than the current database. In such an event, there is no loss of correctness in the system. We would use the sync token associated with the restored database, which is also correct, because PDS guarantees that sync tokens are unique for a given resource. (When we pass a sync token that is outdated to PDS, it could result in an error if the relevant deltas have been deleted from Event Hub; in such an event PDS returns an appropriate error that triggers a full pull.)

With new and restored databases, there is a question of how their persisted policy is updated. For policies that are empty or completely out of sync, we cannot use deltas fetched by delta pulls since the deltas would be respective to a more recent policy state. Accordingly, we use the policy stored in the in-memory cache of the PDP to update the persistent cache for such databases. In general, if the sync token for a database is different from the sync token returned by the latest delta pull, then we over-write its persistent cache with the policy stored in the in-memory cache of the PDP.

Example 5.2. Suppose a SQL process has a single database with a PDP initialized by fetching a policy from PDS. Now suppose a new database is created within the same process. When the next delta pull is invoked, SQL checks that it has a database with no sync token, and updates the persistent cache in the database with the policy stored in the PDP. (We could optimize this further by only storing the subset of the policy applicable to a given database, but in our current implementation, we store a single server-wide policy that spans all databases, in all databases.)

Third, for read replicas, we use the policy state cached in the database obtained through log shipping and avoid a network call to PDS. Geo-replicas also receive policies through log shipping, but unlike local replicas, could belong to a different resource in Azure. The policies obtained through log shipping may not be applicable to geo-replicas. Hence, geo-replicas fetch relevant policies from PDS. However, they use log shipping as the source of information for classification.

5.3.1 Write Ahead Principle During Policy Persistence. Finally, we discuss an important principle we follow during policy pulls from PDS: we always persist the policy *before* we update the in-memory cache in the PDP that is used for query answering. The motivation for this order is that we want newer queries to use newer policies. In the event that SQL restarts, persisting first ensures that queries issued after restart before the first pull use the updated policy. Otherwise, if the in-memory cache in the PDP is updated first and SQL restarts before persistence, then we could have a scenario where queries before restart use the new policy, but queries after restart use an older policy.

5.4 Handling Rule Predicates

Thus far, the discussion focused on policies without predicates, for ease of exposition. We now discuss the extensions to include predicates, and hence attributes. Attributes are also distributed through PDS, just like policies. Attributes are also cached in a local persistent cache, like policies. Unlike policies, however, the ultimate source of truth for metadata is the local store. For instance, it is possible due to the asynchronous behavior of the system that PDS distributes the classification information for a table that no longer exists. In such a case, the classification information for the above table is ignored.

Attributes can take up more space than policies and are not pre-loaded into the PDP, hence reducing its in-memory footprint. The PDP fetches attributes on demand when the access check call is invoked using a callback mechanism. Also, since the classification for a database are a function of only its schema and data, replication including geo-replication would not change the attribute set. Hence, unlike policies which are fetched by geo-replicas from PDS, attributes are read from the transaction log that is shipped to maintain the replica.

5.5 Non-Deterministic Predicates

Many of the optimizations described above, in particular the integration with SQL Server's security caches, are dependent on the Purview access checks being *deterministic*, i.e., fixed for a given policy and classification. This assumption holds for the class of policies studied thus far. However, one particular source of non-determinism that is worth bringing up is predicates that are time-based. For instance, consider a policy that grants access to data classified a particular way, but only within a given time-period, say daily between the hours of 9am and 5pm. We could extend the class of predicates we support to also include such time-based predicates. In order to accommodate time-based predicates, we would need to clear the security cache at all "boundaries", i.e., times when the result of the predicate changes. In the above example, it would be 9am and 5pm. Extending our system to account for such predicates is future work.

6 PERFORMANCE EVALUATION

This section contains results of performance experiments with Purview and SQL. We report the results of an end-to-end evaluation of the impact of Purview based governance on the performance of SQL Server using a TPC-C [19] like benchmark.

6.1 Hardware configuration

Our experiments were run on a bare-metal machine running a 3.2GHz, 64-core Intel Family 6 processor, with 64GB of main memory. The machine was equipped with two separate drives to separate storage of data and log. We use a solid state drive for the log.

In order to run the TPC-C benchmark, we could not use standard benchmark drivers. This is because the standard benchmark driver inside Microsoft, *Benchcraft*, uses SQL authentication, whereas we needed support for a central identity service, namely Azure Active Directory (AAD) [5]. Accordingly, we used a modified driver for the benchmark that was integrated with AAD. Further, our goal

was to have all permissions be Purview-backed. Since we do not yet support all SQL permissions with Purview, specifically stored procedure execution, we further modified the benchmark driver to use prepared statements instead of stored procedures. Our benchmark driver was run on a machine with 4 cores and 14 GB of main memory.

6.2 Systems Compared

We compared the performance of three SQL Server configurations described below.

1. SQLWithoutPurview: This configuration runs SQL Server on TPC-C data with Azure Active Directory based authentication, but no Purview governance. This configuration serves as the baseline.

2. SQLWithPurview: This configuration runs SQL Server with Purview-backed governance. By comparing the performance of this configuration with the baseline above, we analyze the Purview related overheads. In order to study the effects of caching the results of permission checks in SQL Server, we also compare the results of two sub-configurations of the *SQLWithPurview* configuration, namely one with caching enabled and one with caching disabled. For the comparison with the baseline *SQLWithoutPurview*, we enable caching.

6.3 Benchmark and Purview Policy

We use the TPC-C benchmark [19] with the changes noted above to use AAD based authentication and prepared statements instead of stored procedures for our performance evaluation. The benchmark consists of nine tables and five types of transactions over these tables that simulate the business activities of a wholesale supplier. In the absence of stored procedures, the permissions needed to execute these statements include `read`, `insert`, delete and update permissions on the tables, as well as a `connect` permission on the database. Our Purview policy permits all of the above permissions on the AAD principals running the benchmark. Our policy remains fixed through the benchmark run. This is realistic in practice since permissions, once configured, do not change often.

One noteworthy aspect of our policy scenario is that there is *no* local policy. Every single permission needed to run the benchmark is derived solely from Purview. This configuration is intended to model the worst-case in terms of the performance challenges involved in running the benchmark, since every permission check would logically need a call to the PDP.

The benchmark includes a scaling factor `W` representing the number of warehouses. For our experiments, we used `W`= 200; consistent with the benchmark specification, this is the smallest scaling factor that maximized CPU usage for *SQL-NoPurview*, our baseline configuration.

6.4 Results

Due to SQL Server restrictions against publishing absolute throughput values in transactions per second (tps), we report normalized throughput values obtained by dividing all throughput values by the maximum observed value.

Figure 4 shows the relative (normalized) performance of the two configurations. We vary the number of TPC-C client driver threads

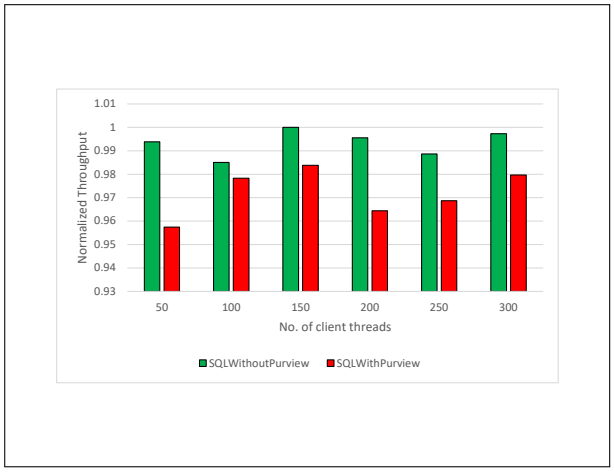

Figure 4: Normalized TPCC-like benchmark transaction processing rates for the systems compared for different number of TPCC client driver threads. The benchmark scaling factor was `W = 200`.

shown on the X-axis, and for each setting show the normalized throughput for the two configurations on the Y-axis. The maximum throughput occurs for 150 client driver threads. Under this load, the throughput of *SQLWithPurview* is 98.38% that of *SQLWithoutPurview*. In other words, the overhead of Purview governance is less than 2%. The overhead is higher when the workload is lighter but never exceeds 5%.

We also study the effect of caching the result of access checks. When caching is enabled, Purview is not in the "hot" path of data access. We find that without caching, the performance of the system drops by up to 31%. This shows the importance of our optimizations.

While we have not empirically evaluated the performance of a proxy based approach, it is worth noting that at least for systems where native integration is possible, e.g., SQL Server, a proxy based approach would be less efficient than native integration. First, a proxy adds an extra hop and hence an extra delay in the system. Further, native integration enables us to closely integrate the Purview PDP's decisions with SQL Server's security caches, which would not be possible with a proxy based approach; the impact, as discussed above, is significant.

7 RELATED WORK

There are several industry offerings that support a central data catalog, e.g., Alation [1], Collibra [9], Google Data Catalog [11] and Informatica [14]. They provide different levels of scanning, classification and lineage tracing, much like Purview [8]. (While this paper focused only on the catalog and classification, Purview also supports lineage tracing.) Purview is notable in several respects. First, it scales to multi-petabyte data sources and exabyte scale data estates, as can be seen from the fact that it has been operational over Microsoft's own large internal data estate (with hundreds of instances each of several database systems including SQL Server and Kusto, as well as an internal Hadoop-like big data system called Cosmos with over 10 exabytes). Purview also supports a very large and

diverse set of sources through its scanning algorithm. In particular, it is unique in its native integration with the Microsoft ecosystem, including data sources in Azure such as the various Data services and Azure Storage, as well as the Office ecosystem. For instance, its classification taxonomy fully integrates with the Office 365 [17] taxonomy, it understands sensitivity labels and Office'scalability Information Protection and Data Lifecycle Protection frameworks, and it supports a single sign-on experience integrated with Azure Active Directory [5].

The main open-source system in the catalog space is Apache Atlas [3]. Purview is fully integrated with Atlas and the entire catalog can be accessed using Atlas APIs. In this sense, we can think of Purview as a significant extension to Atlas. Further, this integration with Atlas differentiates Purview from industry competitors including those listed above.

The most important Purview differentiator, however, is its support for central policy management. Among the major industrial systems, the closest alternatives are offerings from Amazon Web Services [2], Databricks [10] and Google [12]. However, all of them only support coarse-grained policies that are limited to the level of a database. None of them supports fine-grained policies on the contents of the database. In the open-source world, we have Apache Ranger [4] and Open Policy Agent (OPA) [18]. Ranger also supports tag based policies similar to the classification based policies this paper focuses on. However, Ranger's policies focus exclusively on the Hadoop ecosystem, which is then integrated by systems like IBM DB2 as a part of their Hadoop extension [13]. OPA provides a policy language with similar expressive power to Purview's access control policies. It provides a toolkit that engines can use to integrate OPA. Thus, in principle OPA is extensible and not restricted in any way. But it doesn't come integrated with data sources by default. In contrast, Purview is an integrated governance offering out-of-the-box, allowing the instantiated catalog, aka the Data Map, to be used to specify policies uniformly over all data sources. Policy enrollment covers a diverse set of sources, many with native integration and others through a proxy framework. The diversity of sources Purview supports imposes unique scalability challenges, e.g., in the design of the Policy Distribution Service (PDS) as discussed in this paper. The above scalability problems are addressed neither by Ranger nor OPA. Further, the native integration with SQL Server described in this paper is the first of its kind, to our knowledge, of any commercial relational database system.

8 CONCLUSIONS AND FUTURE DIRECTIONS

This paper described Microsoft Purview, a service for central governance of data, including fine-grained mandatory ABAC. It is deeply integrated with Rights Management in Office 365, and essentially extends that across all structured sources of data as well. Purview enables users to govern their entire data estate, split across heterogeneous sources, structured and unstructured, relational and non-relational, on-prem and cloud, OLTP and analytics. It does so by a combination of three components: (1) a Data Map that catalogs the meta-data from various sources, (2) a classification framework that enables identification of sensitive data, and (3) a framework to express and enforce policies across the entire data estate. Purview is the first system that can offer the above combination of features, and is further distinguished by the scale and diversity of sources that it supports.

We described the challenges associated with designing and implementing the Purview system, focusing in depth on the policy evaluation component. We focused on attribute based access control policies in this paper. We described both native and proxy based policy evaluation. For data sources owned by Microsoft, for which we support native integration with Purview, we described how we combine asynchronous semantics and an aggressive use of caching to achieve Purview based governance with minimal performance overhead. Our experiments based on the TPC-C benchmark indicated a drop in throughput of less than 2% due to Purview backed governance.

Centralized governance of distributed and diverse data estates is a central problem in modern enterprises. Purview is the first in what will likely become a growing number of systems that seek to address this problem. What we've accomplished is a good beginning, but the scope of policies that can be centrally governed is vast. We have only taken a first step with access control based policies. Even with access control, while we support the main permissions of engines like SQL Server, we hope to be comprehensive in our coverage of the permission set in future work. The central Data Map opens the door to expressing a rich set of policies, data masking, retention, deletion, data movement, and more. We aim to expand our support for additional classes of policies in future.

REFERENCES

[1] Alation Data Catalog and Data Governance 2021. https://www.alation.com/.
[2] Amazon Web Services Identity and Access Management 2023. https://aws.amazon.com/iam/.
[3] Apache Atlas: Data Governance and Metadata Framework 2021. https://atlas.apache.org/.
[4] Apache Ranger 2023. https://ranger.apache.org/.
[5] Azure Active Directory 2023. https://azure.microsoft.com/en-us/services/active-directory/.
[6] Azure Cosmos DB: NoSQL Database 2021. https://azure.microsoft.com/en-us/services/cosmos-db/.
[7] Azure Event Hub 2021. https://azure.microsoft.com/en-us/services/event-hubs/.
[8] Azure Purview: A unified data governance solution that maximizes the business value of your data 2021. https://azure.microsoft.com/en-us/services/purview/.
[9] Collibra: The Data Intelligence Cloud 2021. https://www.collibra.com/us/en.
[10] Databricks Unity 2023. https://www.databricks.com/product/unity-catalog.
[11] Google Data Catalog 2021. https://cloud.google.com/data-catalog.
[12] Google Identity and Access Management 2023. https://cloud.google.com/iam.
[13] IBM Db2 Big SQL 2021. https://www.ibm.com/docs/en/db2-big-sql/7.1?topic=authorization-ranger.
[14] Informatica Enterprise Data Catalog 2021. https://www.informatica.com/products/data-catalog/enterprise-data-catalog.html.
[15] Microsoft Excel 2023. https://www.microsoft.com/en-us/microsoft-365/excel.
[16] Microsoft PowerBI 2023. https://powerbi.microsoft.com/.
[17] Office 365 2023. https://www.office.com.
[18] Open Policy Agent 2021. https://www.openpolicyagent.org/.
[19] TPC-C Benchmark 2019. http://www.tpc.org/tpcc/.