



WHITE PAPER

Immuta + Unity Catalog

The Next Frontier
of Scalable Data Security

STEVE TOUW

Co-Founder and CTO,
Immuta

JONATHAN KELLER

Sr. Director, Product Management,
Databricks

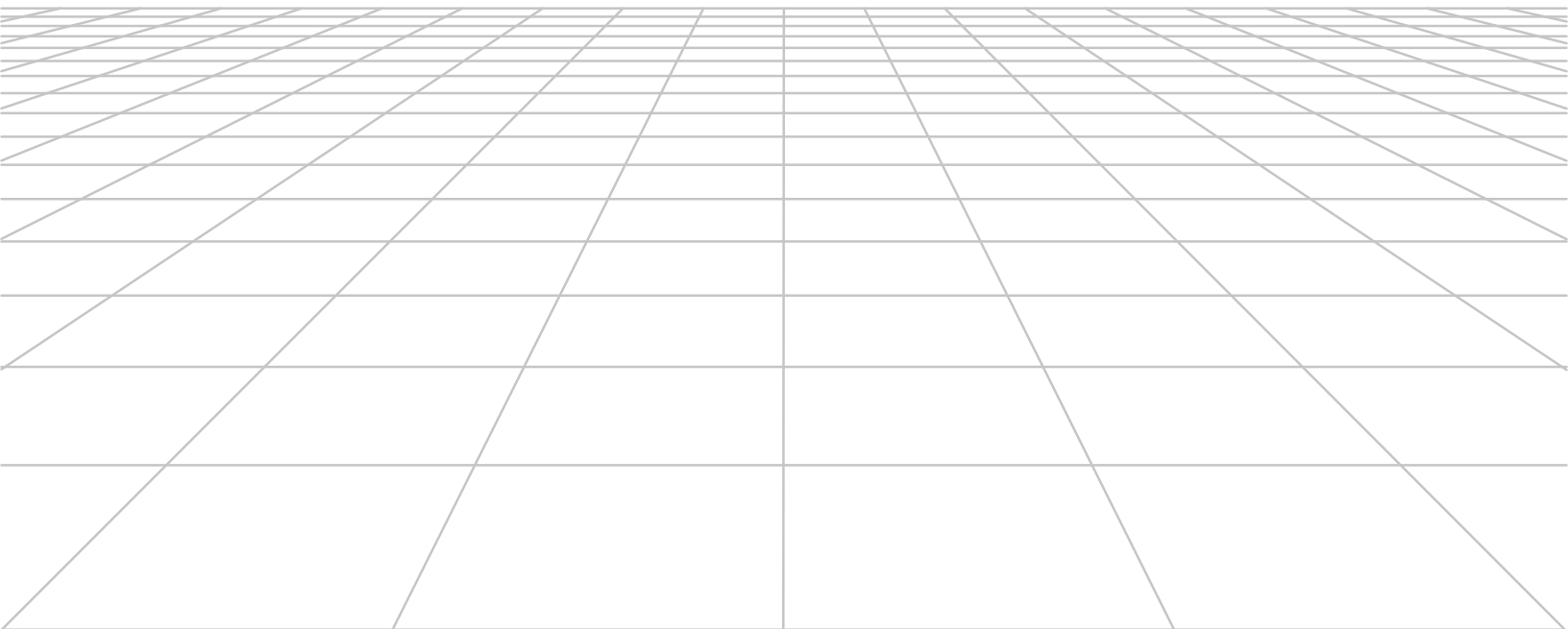


Table of Contents

Introduction	3
The Cloud Frontiers	4
The Unity Catalog Primitives	5
Why Do I Need Scalable Data Security?	8
The Scalable Data Security Frontier	10
Discover	11
Detect	12
Secure	13
Federated Governance	18
Conclusion	19

Introduction

We're excited to announce a new release of Immuta's integration with Databricks Unity Catalog that gives joint customers the ability to manage and scale data security on Databricks.

More specifically, the Immuta integration is powered by the release of Databricks Unity Catalog row filtering and column masking. By combining that new Unity feature with Immuta's Discover, Detect, and Secure platform we have reached a new frontier of cloud scalability: scalable data security. What is scalable data security? Why do I need it? How do Immuta and Databricks Unity Catalog get me there? In this blog, we will answer those questions and more.

The Cloud Frontiers

If you were to explain why cloud adoption has been so ubiquitous, a simplistic way to state it would be: *The cloud handles things you don't want to handle yourself so you can focus on more value-driving initiatives.*

There are many “things” it handles – frontiers, if you will – through abstraction and orchestration. As more frontiers are crossed, new and higher level abstractions emerge. The cloud continues to build up the value chain on the shoulders of earlier innovation.

Let's explore some of those frontiers (there are many, but to name a few):

- **The Infrastructure Frontier**

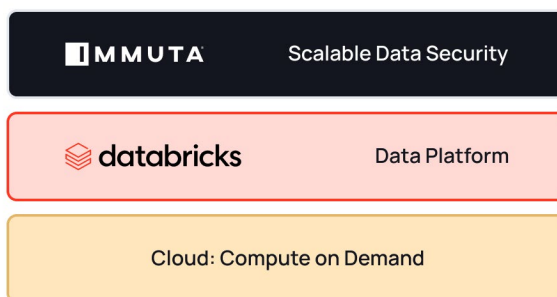
This is probably the most famous of the three. Before the cloud, you had to buy servers, find a place to keep them, power them, and keep them up to date with software, among other things. The cloud changed this – through abstraction and orchestration, you are able to spin up various different compute infrastructure on-demand. Cloud vendors handle the “primitives” (the Infrastructure), and provide you with an abstraction to orchestrate those primitives (APIs and user interfaces).

- **The Data Platform Frontier**

This is why you buy Databricks! Databricks builds on the Infrastructure frontier to offer a multi-persona experience that allows every practitioner to work on the same data with both SQL and Python, while abstracting and dynamically scaling the compute invisibly.

- **The Data Security Frontier**

This is what we are here to talk about in this blog. Without stealing too much thunder from the rest of the article, Databricks has delivered some new powerful primitives with its Databricks Unity Catalog, which provides a unified governance for data, analytics and AI. Immuta is the abstraction to orchestrate those Unity Catalog primitives to provide scalable data security.



The Unity Catalog Primitives

Before diving into why you need a scalable data security abstraction, we will provide some details on the new, powerful Databricks Unity Catalog primitives. Databricks has offered Unity Catalog for quite a while, however, its Unity Catalog fine-grained access controls are new. More specifically, column masking and row filtering are now available to customers (in private preview).

Column masking allows you to build rules about how to dynamically mask a column using a UDF, based on the querying user. Below is an extremely simplistic masking policy:

```
CREATE OR REPLACE FUNCTION `common`.`functions`.`mask_ssn`(ssn STRING) RETURN
(CASE WHEN upper('bob_loblaw') = upper(current_user()) OR is_account_group_
member('finance')
      THEN \ `SSN` \
      ELSE ('REDACTED')
      END) `;
```

This function takes in a social security number column, in string format, and converts it to REDACTED at query time, unless the user is Bob Loblaw or they are in the group finance, in which case they see it in the clear.

That function must then be added to the column in question:

```
ALTER TABLE `f_catalog`.`logs`.`user_records` ALTER COLUMN `SSN` SET MASK
`common`.`functions`.`mask_ssn`;
```

Row filtering allows you to build rules about how to dynamically filter out specific rows using a UDF, based on the querying user. Below is an extremely simplistic row filtering policy:

```
CREATE OR REPLACE FUNCTION `common`.`functions`.`sample_filter`(`ID` INT,
`SSN` varchar(100))
  RETURN (CASE
    WHEN upper('bob_loblaw') = upper(current_user()) OR is_account_
group_member('finance')
    THEN TRUE
    ELSE ((TRUE) AND (SSN LIKE '%5555') AND (ID LIKE '007%'))
  END);
```

This function takes in a social security number column in string format, and an ID column in int format, and only returns rows that have 5555 in the SSN AND 007 in the ID, unless you are Bob Loblaw or a member of the group finance, in which case you see all rows.

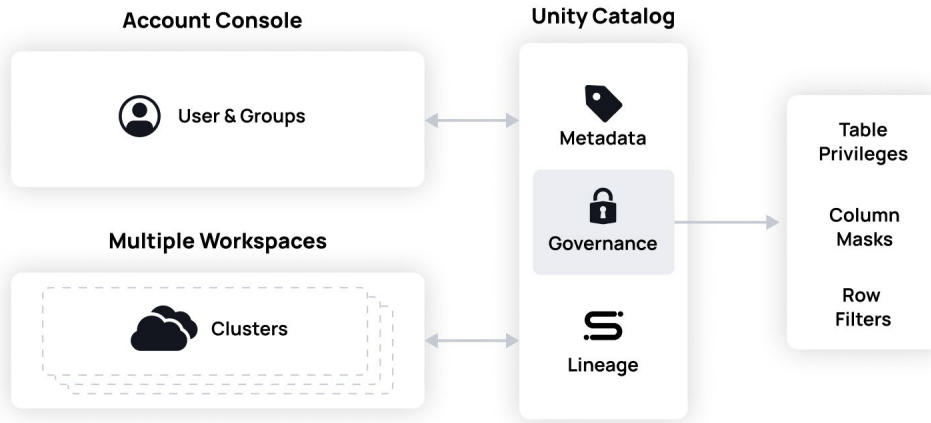
In the exact same manner we did with our column masking function, we must add the row filter function to the appropriate table:

```
ALTER TABLE `f_catalog`.`logs`.`user_records` SET ROW FILTER
`common`.`functions`.`sample_filter` ON (`ID`, `SSN`);
```

A few points worth making on those functions that are relevant to the subsequent discussion in this article:

1. You are only allowed one row filter function per table, and one column mask function per column. This means that all logic required must exist in those single functions, even if it is different policies from different parts of your organization.
2. Column types matter. As you can see, the above column mask function expects a column type, so if you have SSN stored as a string in one table and an int in another column, you will need differing logic in those functions (an int type can't return the String 'REDACTED', for example).
3. To author a policy in Unity Catalog like the ones described above, you need to be the owner of a securable object above the object at hand, or the object's direct owner. For example, you could be the owner of the catalog in the metastore, and apply any policy to the tables in that catalog.

This brings us to the last point on primitives: Granting select privileges on tables cannot be ignored. Data access policy is not limited to row filtering and column masks only; who can access what table is obviously a key factor as well. Databricks Unity Catalog also has this primitive (and has for a while).



Why Do I Need Scalable Data Security?

Remember how we said the cloud becomes more powerful by building abstraction and orchestration on primitives that were built upon other abstractions and orchestrations? So, why do I need a scalable data security abstraction? Same answer: *it handles things you don't want to handle yourself so you can focus on more value-driving initiatives.*

But let's dive deeper. To do data security well, there are four critical goals you must handle:

1. Discover where your risks are
2. Detect where you have coverage gaps with those risks
3. Secure and remediate those gaps through data access policy
4. Empower data users to create their own data products and manage their own controls (federated governance)

Consider the following real world example:

“I want to ensure all PII is appropriately masked, unless used for a legitimate and temporary purpose”

1. To do this, you must Discover where your PII data exists
2. Then you must Detect where the PII data is not being used legitimately
3. Next, you must add data access controls proactively to Secure any gaps in PII access you detected
4. And finally, you must empower the data product owners to layer on additional data access exceptions or protections to their domain-specific data that contains PII

Doing this without an abstraction is extremely time-intensive, error prone, and unclear whether it was done correctly. Manually approaching this problem would involve the following steps:

- A survey would need to be done across all tables and columns to classify data at different sensitivity levels.
- Then, the data consuming side of the business would need to be audited to understand who is using those tables and why.
- Once all that information is gathered, security policies would need to be hand-crafted in SQL using the aforementioned Databricks Unity Catalog primitives to enforce access controls one-table-at-a-time and one-column-at-a-time.
- And finally, the creators of that data across the business need to work with the teams hand-crafting the Unity Catalog primitives to add or alter the policies with domain-specific policy logic (remember, you can only have a single function on a table/column).

But the job doesn't end there. Data is constantly changing, with new data products emerging nearly daily – this is not a one-and-done process, but rather a constant loop.

Also note that this problem must involve experts and stakeholders from across the business: legal and compliance teams need to define what is sensitive and the associated policies; the data consuming side of the business must understand how data is being used; the data platform team needs to constantly hand-craft the policies; and finally, the data product creators, must be consulted for domain-specific policy.



The Scalable Data Security Frontier is desperately needed, and Immuta and Databricks Unity Catalog provide it. How?

The Scalable Data Security Frontier

Let's walk through the above use case once again, but this time instead of manually solving the steps, we will instead use the Immuta abstraction to orchestrate solving them.

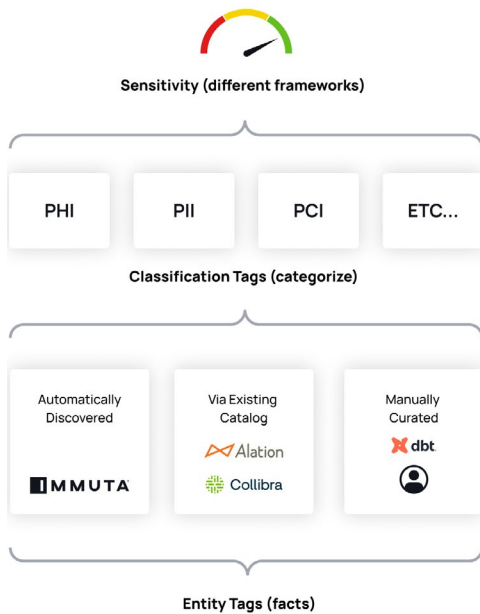
As a reminder, here's our use case:

"I want to ensure all PII is appropriately masked, unless used for a legitimate and temporary purpose"

1. To do this, you must Discover where your PII data exists
2. Then you must Detect where the PII data is not being used legitimately
3. Next, you must add data access controls proactively to Secure any gaps in PII access you detected
4. And finally, you must empower the data product owners to layer on additional data access exceptions or protections to their domain-specific data that contains PII

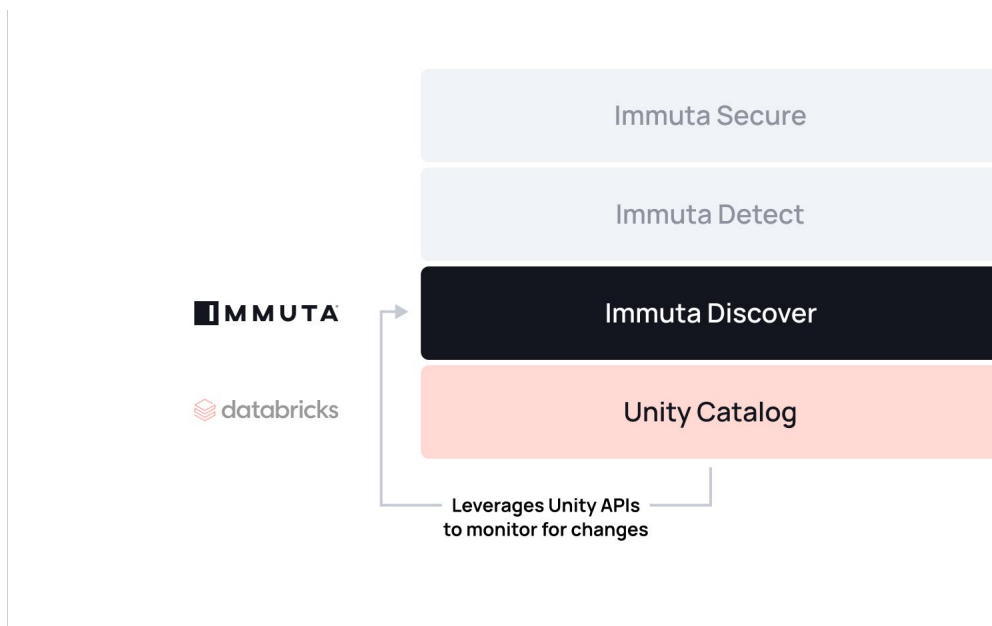
Discover

Think of Immuta as your central aggregation tier of metadata. Immuta can connect and monitor all Databricks objects as they change over time, inspect those objects to find sensitive data such as PII, and build higher level classifications and sensitivity scoring against what is discovered. Those detection algorithms can be customized to discover different entity types in your data ecosystem.



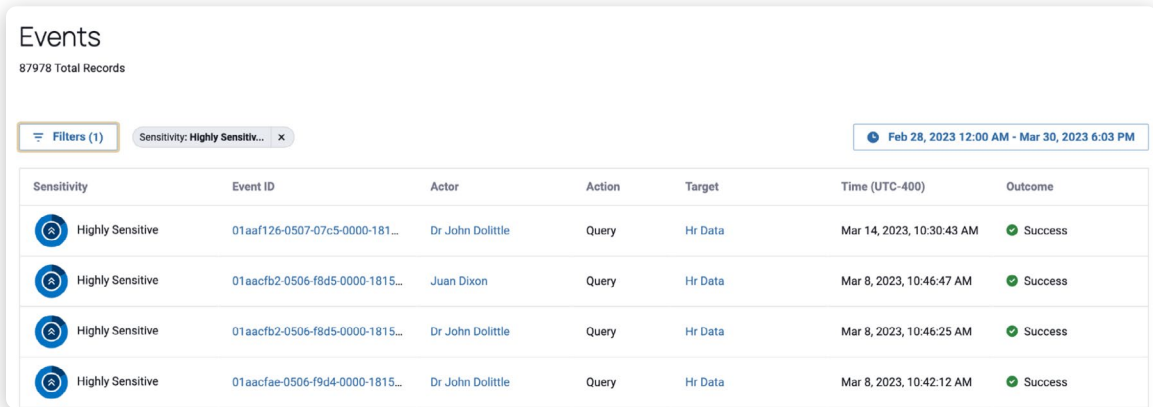
The diagram depicts the three levels of abstraction which can automatically be applied to data when discovered. You are able to define which discovered entities should receive higher level classifications, and each of those higher level classifications can map to different sensitivity levels. We call this logic "sensitivity frameworks". By way of example, sensitivity frameworks are important because a company in the healthcare industry may deem PHI more sensitive than a company in the advertising industry. Immuta Discover provides sensitivity frameworks that allow you to define and customize these levels, as well as ship with industry standards.

Also critical, but often ignored, is metadata about your users. This metadata can come from any system across your organization to enrich user data with contextual information that can be used for access control decisions.



Detect

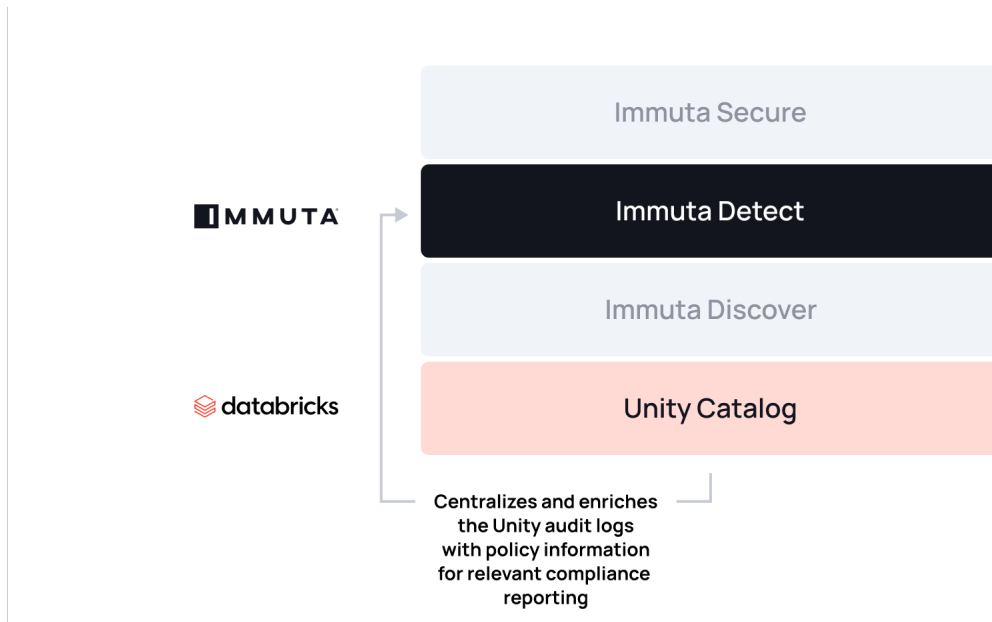
Now that you know where sensitive information exists, it's time to gain a deeper understanding of your current security posture. With Immuta Detect, you can see which users are accessing specific tables containing sensitive information, and how often. You can also understand your data security posture based on common recommended security configurations. Continuing our example, this will help understand what PII is being used and by whom, and identify appropriate data access control mitigation strategies.



The screenshot shows the 'Events' page in the Immuta Detect interface. It displays 87978 total records, filtered by 'Sensitivity: Highly Sensitive'. The table below shows four records of 'Query' actions on 'Hr Data' by various users, all resulting in 'Success'.

Sensitivity	Event ID	Actor	Action	Target	Time (UTC-400)	Outcome
Highly Sensitive	01aaf126-0507-07c5-0000-1815...	Dr. John Dolittle	Query	Hr Data	Mar 14, 2023, 10:30:43 AM	Success
Highly Sensitive	01aacfb2-0506-f8d5-0000-1815...	Juan Dixon	Query	Hr Data	Mar 8, 2023, 10:46:47 AM	Success
Highly Sensitive	01aacfb2-0506-f8d5-0000-1815...	Dr. John Dolittle	Query	Hr Data	Mar 8, 2023, 10:46:25 AM	Success
Highly Sensitive	01aacfae-0506-f9d4-0000-1815...	Dr. John Dolittle	Query	Hr Data	Mar 8, 2023, 10:42:12 AM	Success

Since your data platform is constantly evolving, even after you build your initial sets of access controls, PII (again, continuing the example, but could be other categories of data) can continue to evolve and arrive in it. As this occurs, Immuta's Discover and Detect work modules together to keep a watchful eye for compliance issues and help proactively address them.



Secure

At this point, we've discovered where PII data lives, which users are accessing it, and where we have gaps. It's time to close those gaps with Immuta Secure. As we showed above, it's certainly possible to hand-craft these policies using the Databricks Unity Catalog primitives. But with Immuta Secure, you are able to manage the authoring and orchestration of those policies in a simple and understandable manner – the scalable data policy frontier!

Let's imagine we have approximately 200,000 tables in Databricks Unity Catalog. How would you even begin to tackle this problem, remembering that you are only allowed one column masking and row filtering function per table?

You would need to inventory every table, its PII, its associated policies per PII, and their column types, while also accounting for column types in the functions, and having a combined function for each combination of policy and type – and finally, roll that out to all appropriate tables across the 200,000.

With Immuta, you can accomplish our example scenario with a single policy that references the PII known through the Immuta Discover algorithms:

Global Data Policy Builder

[Add Certification](#) [SQL Support Matrix](#)

What is the name of this policy? ⓘ

Mask PII

How should this policy protect the data? ⓘ

Mask ▾ columns tagged ▾ Discovered . PII × add another tag (optional)

using hashing ▾ for ▾

everyone except ▾ when user is acting under purpose ▾ Legitimate Purpose

[+ Add Another Condition](#)

Enter Rationale for Policy (Optional)

[Add](#)

Where should this policy be applied? ⓘ

On data sources ▾ with columns tagged ▾ Discovered . PII

[+ Add Another Circumstance](#)

When activated, this policy will be propagated across all tables with PII by translating the policy logic into the Unity Catalog primitives. If there is a function that already exists on that table from a prior policy, Immuta is able to merge those policies together into a single function without requiring any human intervention.

You might be wondering how Immuta handles the column type problem? The quick answer is that Immuta has fallback algorithms. Should a masking policy hit a column that is unsupported for that masking type, Immuta will fall back to a masking type that works for that column type and has the same or more restrictive privacy guarantees than the original masking type – all automatically without you having to take action.

As you can see, this is extremely simple to manage and understand. The policy is in plain language, which makes it easy to prove compliance, unlike having to read and understand SQL (and its many different function variations across the PII combinations mentioned above).

The above example happens to be a column mask, but the same applies for row filtering. We could have instead made a more complex policy restricting anybody from seeing rows that contain PII unless the source country of the data matches the user's work location:

Global Data Policy Builder

[Add Certification](#) [SQL Support Matrix](#)

What is the name of this policy? ⓘ

Filter rows with PII that don't match work location

How should this policy protect the data? ⓘ

Only show rows ⌵ where user ⌵

possesses an attribute ⌵ in Work Location that matches the value in column tagged Source Country

+ [Add Another Condition](#)

for everyone ⌵

Enter Rationale for Policy (Optional)

[Add](#)

Where should this policy be applied? ⓘ

On data sources ⌵ with columns tagged ⌵ Discovered . PII and ⌵ ⓘ

with columns tagged ⌵ Source Country ⓘ

In this case, the policy will propagate to tables that were discovered to contain both PII and a Source Country, and enforce this row filtering policy by translating the policy logic to the Unity Catalog primitives. This will, again, automatically merge with existing row filtering functions on any given table.

Continuing other solutions to our example scenario, Immuta can also support complex table privilege grant scenarios, such as giving people access to entire tables containing PII, but only when they possess appropriate PII training:

Global Subscription Policy Builder

What is the name of this policy? ⓘ

Must have training to access tables with PII

How should this policy grant access? ⓘ

Allow users with specific groups/attributes ⌵

Who should be allowed to subscribe? ⓘ

Create using plain language Create using Advanced DSL ⓘ

Allow users to subscribe when user possesses attribute ⌵ in Training PII

[+ Add Another Condition](#)

Where should this policy be applied? ⓘ

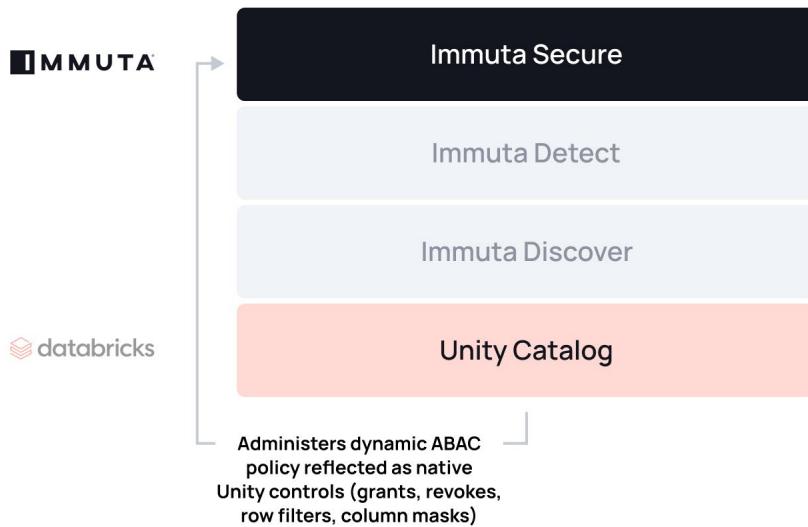
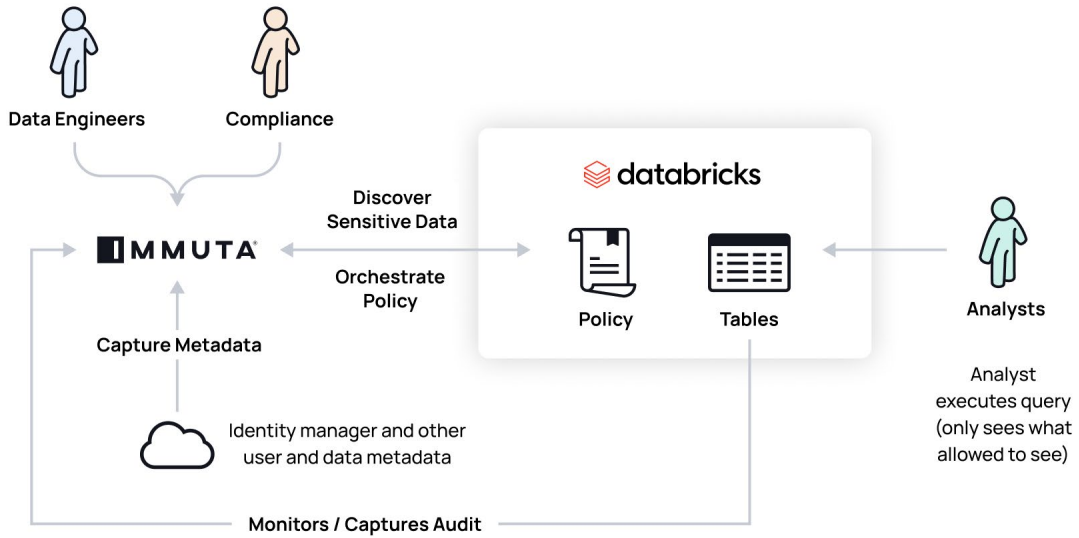
On data sources ⌵ with columns tagged ⌵ Discovered . PII

[+ Add Another Circumstance](#)

In this case, Immuta will leverage Unity Catalog table grants (in bulk) to provide access to tables that were discovered to contain PII for only people with PII training.

You might be wondering where the Training PII course came from? That is what we discussed in the Discover section earlier – Immuta is able to aggregate user metadata from across the business to use that logic in policy decisions. This is termed **Attribute-Based Access Control (ABAC)**, an approach that decouples policy logic from metadata about users (and data) to build scalable and future-proof policy. Immuta injects the ABAC model into Databricks Unity Catalog – this is the root of Immuta’s scalability. We could write paragraphs about the benefits of ABAC, but it’s beyond the scope of this article. If you are interested in learning more about the power of ABAC, please read [here](#).

As you can see from the below high-level architecture, Immuta acts as an abstraction and orchestration layer, built on top of the primitives provided by Databricks Unity Catalog.

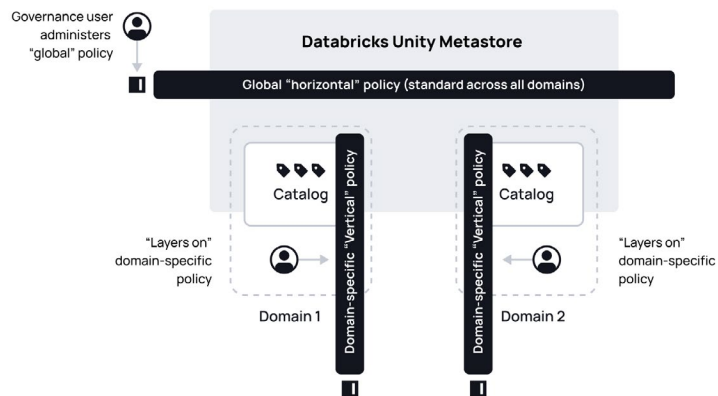


Federated Governance

The final piece of the puzzle is empowering different data owners and data product creators across the business to manage their own data access controls – without breaking policy. This is often termed federated governance, and can be a key component of data mesh architectures.

A good way to think about federated governance is like driving a car. You want people to be able to drive their own cars, but you need to provide a framework and rules for them to follow, like driving on the correct side, lines in the road, stop signs, stop lights, etc. Federated governance is no different – you need to provide a framework from which everyone can build policy that meets organizational standards and guidelines.

Consider the above diagram. Global or “horizontal” policies are those that must be enforced, no matter what – these are critical to the business and would horizontally apply to ALL tables. However, you also need to empower the domain-specific or “vertical” policies; these are very specific policies that are built by the data owners that ONLY apply to the data they own. Still, their rules must be built using the same framework as the horizontal policies and merge with them without conflict.



An example might be, all PHI must be masked except for users in group Global Governance [global policy], and all PHI must be masked except for users in group Dharma Division [domain-specific policy]. When those two policies run into each other on the same table (remembering that the domain-specific policy will only be applied to tables they own), depending on which sides are deemed required or not, it may result in a merged policy such as: all PHI must be masked except for users in group Global Governance OR users in group Pharma Division. This is a rather simplistic example, but Immuta also supports other complex conflict management scenarios that are beyond the scope of this article.

Multiple Global Subscription Policies

Users with Specific Groups/Attributes

Global Moderately Restricted Locked

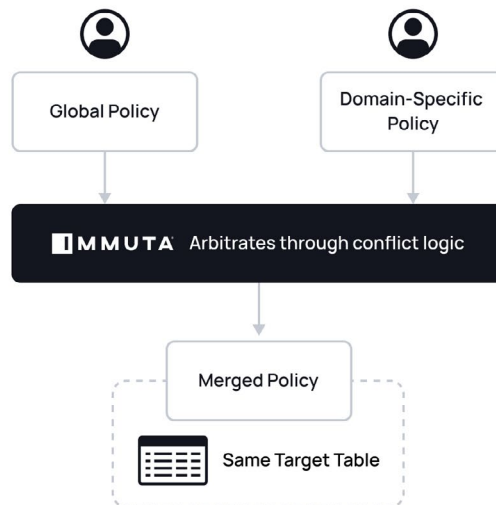
Users with the specified groups/attributes will be automatically subscribed to the data source.

The following Global Policies are being applied to this data source:

- Governance policy
- pharma

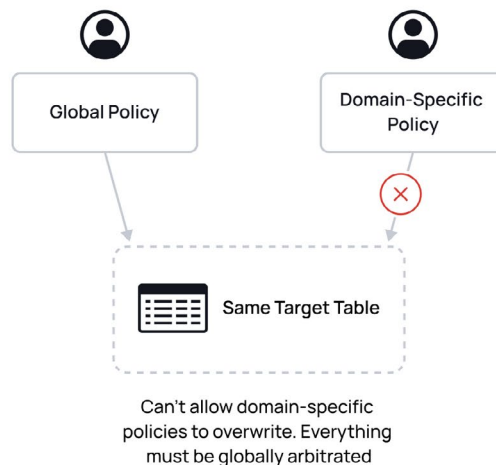
Allow users to subscribe to the data source when user (@isInGroups('Pharma Division')) OR (@isInGroups('Global Governance'))

Federated governance with Immuta is made possible through the framework that has been mentioned a few times already – it's the metadata about your data (the discovered information) and the metadata about your users. Since these are consistent across all data products, it is easy for everyone to build policies from the same sheet of music, so to speak. And in doing so, Immuta can handle conflicts and merge them successfully without any human involvement.



Also remember that Discover and Detect are constantly monitoring Databricks. So new tables or columns are scanned as they appear, and if the new data is discovered to include something relevant to an existing policy, that policy is attached *immediately*, with no human intervention required. If there is something newly sensitive that has yet to have a policy, Detect will identify that and may trigger you to create a new global policy to account for that now and going forward.

Without a framework like Immuta to manage policy authoring and conflict resolution, this becomes a large burden to the data platform team. Remember that with Unity Catalog, you must own the object or be the parent of the object for which you intend to build policy. That means that if the user building the global policy has the same level of control as the user building the domain-specific policy, they can easily blast each others' edits. To resolve this without Immuta, you need to have all policy changes go through the central authority, which results in bottlenecks and does not empower the data product owners.



Conclusion

In quick summary: Databricks has released new fine-grained access control primitives: column masks and row filters, in addition to their existing ANSI SQL DCL, for controlling table-level access. Immuta leverages these primitives to build a powerful abstraction and orchestration layer, which creates a new frontier for cloud scalability: the Data Security frontier.

This is similar to other well-known cloud frontiers that were built on deeper primitives like infrastructure and compute. Enabling yet another thing the cloud can handle that you don't want to handle yourself, so you can focus on more value-driving initiatives.

With Immuta's abstraction and orchestration, you are able to meet the four critical goals of cloud data security:

1. Discover where your risks are
2. Detect where you have gaps of coverage with those risks
3. Secure and remediate those gaps through data access policy
4. Empower data users to create their own data products and manage their own controls (federated governance)

About Immuta

Immuta enables organizations to unlock value from their cloud data by protecting it and providing secure access. The Immuta Data Security Platform provides sensitive data discovery, security and access control, data activity monitoring, and has deep integrations with the leading cloud data platforms. Immuta is now trusted by Fortune 500 companies and government agencies around the world to secure their data. Founded in 2015, Immuta is headquartered in Boston, MA.

