KOANTEK
The Data and AI Partner

Secure Column-Level Masking in Databricks for Compliance-Driven Industries

Author:
Satya Sure,
Resident Solution Architect

November 21, 2025

Secure Column-Level Masking in Databricks for Compliance-Driven Industries

Summary

As industries face stricter data privacy mandates, protecting sensitive information without limiting analytics has become a top priority. This whitepaper explores a secure, scalable approach to column-level masking in Databricks, combining custom encryption logic, Unity Catalog governance, and secret-backed key management to meet compliance needs across finance, healthcare, and regulated sectors.

Enabling Secure Data Innovation

In finance, healthcare, credit card processing and other regulated sectors, ensuring sensitive data remains protected yet usable is a critical challenge. Modern lakehouse platforms like Databricks need to enable fine-grained encryption at the column level – so that only authorized users can see plaintext personal data (SSNs, credit card numbers, health records, etc.), while everyone else sees masked or encrypted values. This goes beyond basic disk encryption or table-level access controls; it requires dynamic data masking and encryption tied to user roles. Databricks provides native AES encryption functions, but companies often require alternative algorithms (PGP, SM4, etc.) for interoperability or regional compliance. In this post, we explore a universal approach to column-level data masking in Databricks that supports enterprise-grade encryption methods (from AES to PGP and SM4, even hardware security modules), all aligned with strict standards like HIPAA, PCI-DSS, and Basel III.

Why Now

Regulations are tightening, and platform architects must balance security with performance and usability. Frameworks like HIPAA and Basel III demand robust controls –end-to-end data lineage, auditability, encryption, and fine-grained access policies. A well-implemented column-level encryption strategy can fast-track compliance by enforcing policies natively in your data platform. The solution described here leverages Databricks Unity Catalog for governance, Databricks Secrets for key management, and Python UDFs for custom decryption logic, delivering a secure design that keeps sensitive data masked by default. Platform architects can implement this pattern to meet security requirements without impeding analytics – achieving both compliance and agility in their data pipelines.

Challenges with Native Approaches

Limited Algorithm Support:

Databricks SQL has native functions like aes_encrypt/aes_decrypt which are highly performant, but they only support AES encryption. If organizational policy mandates a non-AES cipher (PGP, SM4, etc.), native functions won't suffice. Many enterprises prefer PGP for its public/private key model or SM4 to meet specific national standards, requiring a more flexible solution.

Complexity of External Decryption:

Decoding data outside of Databricks (e.g. decrypting before ingestion or exporting encrypted data to an external tool) undermines the unified architecture. It adds extra steps, increases the risk of exposure, and breaks lineage and governance within Databricks. In short, moving sensitive data out of the lakehouse for decryption creates compliance blind spots and operational headaches.

Governance Gaps:

Approach above easily ties into fine-grained access controls. We need a way to conditionally mask or reveal data based on user roles within Databricks. Simply decrypting data for everyone or relying on application logic can lead to permission creep. A robust solution should leverage Databricks' access control mechanisms (like Unity Catalog policies) to ensure that only authorized users or groups ever see the decrypted values, with everyone else getting masked data – all on the same dataset.

Summary

While Databricks ensures encryption at rest and offers basic encryption functions, achieving column-level masking that satisfies strict enterprise and regulatory requirements calls for a more extensible pattern. We need to combine custom encryption/decryption logic with native governance features to get the best of both worlds.

TECHNICAL HIGHLIGHTS

Secret-backed encryption keys managed securely via Databricks Secrets and cloud Key Vaults.

Role-based masking conditional access enforced natively in Unity Catalog.

Universal compatibility supports AES, PGP, and SM4 for regional and enterprise compliance.

BUSINESS IMPACT

Stronger compliance meet HIPAA, PCI-DSS, and Basel III with ease

Reduced risk minimize data exposure through role-based masking.

Faster value up to 80% automation and 90% policy coverage with Koantek accelerators.

Solution Overview: Universal Column-Level Masking in Databricks

Our approach builds a secure, dynamic masking pipeline using Databricks' latest capabilities: Extended Python UDFs, Unity Catalog's masking policies, and secret integration. This design can accommodate various encryption algorithms (PGP, AES, SM4, etc.) and enterprise key management (including cloud Key Vaults or HSMs), making it broadly applicable. The high-level flow is:

Encrypt Data Upstream

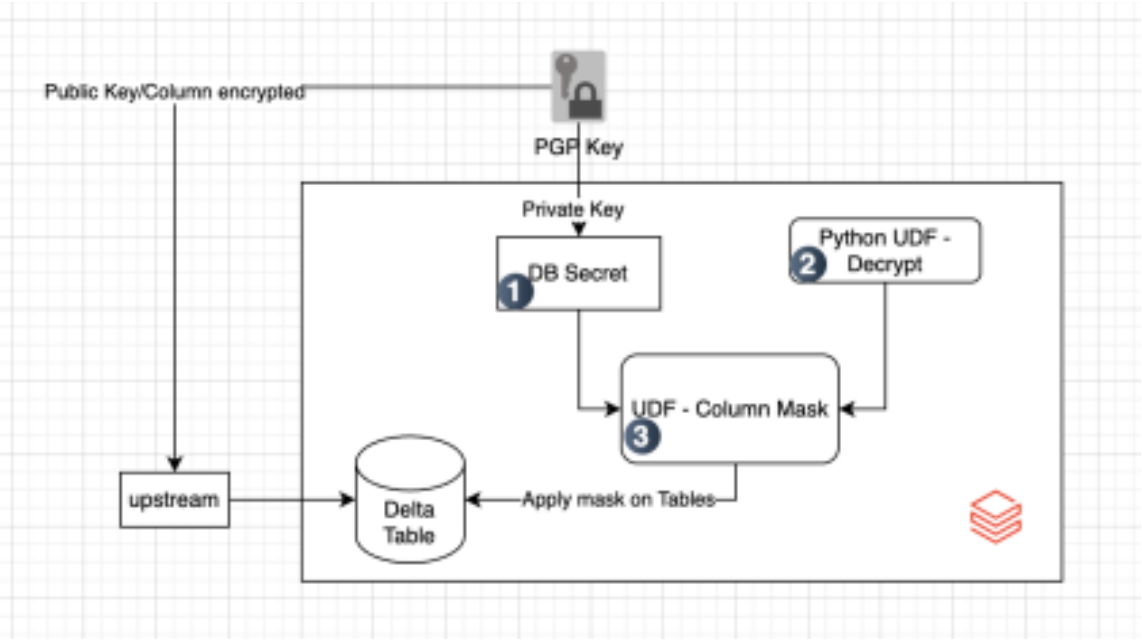
Ideally, sensitive columns are encrypted before or as they enter the Lakehouse (using a strong key/algorithm). For example, a source system might encrypt Social Security Numbers with a PGP public key. This ensures the raw data in Delta Lake is not human-readable to unauthorized eyes. (If data isn't encrypted upstream, you could also encrypt it on ingestion using PySpark with aes_encrypt or a custom function). The focus here is on how to decrypt and mask that data selectively for consumers

Manage Keys Securely

Store the decryption keys (e.g. the PGP private key, or AES symmetric key) in a secure Databricks secret scope or an external key management service. Databricks secrets are backed by cloud Key Vault/HSM services and ensure that the key

material never appears in plaintext in notebooks or logs. This is critical: all key handling must be abstracted away from end-users. Only the compute cluster will retrieve the key at runtime when needed, and only if the user/session has proper access.

With keys in place, we implement the core solution in Databricks as follows:



Architechture Diagram

This end-to-end solution integrates Databricks secrets, UDFs, and masking policies to enforce column-level encryption. As shown below, upstream systems provide data with a PGP-encrypted column. The corresponding private key is stored securely in a Databricks secret scope (1). A Python UDF (with the pgpy library dependency) is registered to decrypt the value using that key (2). A SQL masking function then calls this UDF, deciding whether to return the plaintext, a partially masked value, or the still-encrypted value based on

the querying user's group membership (3). Finally, this masking function is applied to the Delta Table column, so that any query against it automatically applies the correct decryption or mask. The result is that privileged users see the cleartext data, while others see only masked or cipher text – all from the same table, with no extra steps needed by the end user.

Solution Overview

Secure Key Storage:

The encryption key (e.g., a PGP private key or AES secret) is stored in a Databricks secret scope with tight access controls. This ensures keys are never hard-coded or exposed in code or Spark logs. For example, we might have a secret named private_pgp in a scope accessible only to privileged roles. This centralizes key management and allows easy rotation via the secret store without changing any application logic.

Python UDF for Decryption:

We create a Python UDF in Unity Catalog that handles the decryption logic. Databricks now supports external or extended UDFs with custom Python libraries, so we can package the required cryptography library (like pgpy for PGP, or a library for SM4/AES if needed) directly with the UDF. For example, a UDF decrypt_pgp(encrypted_value STRING, private_key STRING) RETURNS STRING can be defined in SQL, and under the hood it runs a Python function that loads the pgpy library to perform PGP decryption. The UDF reads the private key from the secret scope at runtime (passed in as an argument via the secret() function), decrypts the value, and returns plaintext. All of this happens within the Databricks runtime – no external services needed – and the logic is sandboxed by Unity Catalog's privileges. If the key is invalid or decryption fails, the UDF can return a sanitized error or null, ensuring robustness. Because this UDF is registered in Unity Catalog, it can be called from SQL, notebooks, or dashboards with proper permission, just like a built-in function. (For algorithms like AES which Databricks supports natively, you might not need a Python UDF at all – you could use aes_decrypt directly for better performance. The Python UDF approach is most useful when you require an algorithm not supported by Spark, or need custom logic.)

Secure Key Storage:

Next, we create a masking policy UDF – essentially a SQL function – that wraps the decryption UDF and adds conditional logic based on user group or role. This function will determine what a given user is allowed to see. For example, in SQL:

```
CREATE OR REPLACE FUNCTION can_read_ssn(masked_value
STRING) RETURNS STRING
RETURN CASE
WHEN is_account_group_member('data_privileged')
    THEN decrypt_pgp(masked_value,
secret('secrets_scope', 'private_pgp'))
WHEN is_account_group_member('data_partial')
THEN CONCAT('XXX-XX-',
RIGHT(regexp_replace(decrypt_pgp(masked_value,
secret('secrets_scope','private_pgp')), '[^0-9]',
''), 4)) ELSE '***-**-****'
END;
```

In this example, users in the data_privileged group get the fully decrypted SSN, users in a data_partial group get a partially masked result (only last 4 digits revealed), and everyone else gets a fully masked format. The use of is_account_group_member() is key – it checks the group membership of the querying user at runtime. You can define these groups in Databricks (often synced from your identity provider), allowing role-based data masking. This conditional UDF essentially encodes your policy: who can see what. It also pulls in the secret-stored key via secret() calls, so the key is fetched just-in-time and only if needed for a privileged user. Notably, the masked output for unauthorized users could also just be the original encrypted text or a constant mask – whatever makes sense (in our case we used asterisks). This approach is very flexible: you could add more branches for additional roles or implement other masking logic like generalization or hashing as needed.

Python UDF for Decryption:

Finally, we attach the masking function to the actual Delta table column that holds the encrypted data. Using Unity Catalog, we execute an ALTER statement to bind the policy: e.g., ALTER TABLE customer_data ALTER COLUMN ssn_enc SET MASK can_read_ssn. After this, any query on that table's SSN column will automatically invoke the masking UDF – no matter if the query comes through a SQL warehouse, a Python notebook, or a BI tool. The Databricks engine will check the user's group, call the can_read_ssn function, and return either decrypted or masked values accordingly. This means data scientists and analysts don't have to do anything special – they just query customer_data, and if they're not in the allowed group, they'll simply see masked SSNs. The policy is enforced consistently across all access paths. Crucially, because this is implemented at the table schema level in Unity Catalog, it's tamper-proof by users – only admins can alter or remove the mask, and it applies to all downstream consumers by default. This meets the compliance requirement of restricting sensitive data access even for internal users with raw table access. And since Unity Catalog logs these policy interactions, you also get an audit trail of who accessed what, which is gold for compliance audits.

Security and Performance Considerations

Any solution handling encryption must be validated for security and performance. Here's how our approach stacks up:

Keys and Secrets Management:

We never expose the encryption keys in any notebook, code, or output. By using Databricks Secrets and the secret() function in our UDF, the key is pulled securely at execution time. Databricks secret scopes can be backed by Azure Key Vault, AWS KMS, or GCP Secret Manager, and access to them is controlled via ACLs. This means only the Databricks compute (and only for authorized users) can retrieve the key – users cannot directly read the secret value. This setup supports separation of duties: security teams can manage the actual keys in the vault, while data engineers just reference a secret name. Additionally, Unity Catalog allows securing who can create or execute functions – ensuring only the right personas can deploy or use the decryption UDF. All usage of the secret() function can be logged for audit purposes. In short, the sensitive keys are locked down, meeting compliance requirements for strong key management (PCI-DSS, for example, mandates secure key storage with limited access).

Unity Catalog Enforcement:

Unity Catalog is the governance layer ensuring this masking policy cannot be bypassed. Even if a user knows there's a UDF, they cannot call decrypt_pgp directly on the table unless they have permission to use the secret and UDF. By attaching the mask at the table level, we rely on Catalog's enforcement: if a user is not in the allowed group, the policy will return masked data and they have no way around it (short of hacking privileges). This satisfies the principle of least privilege and the HIPAA/PHI rule that users should only access data necessary for their role. Moreover, Unity Catalog provides end-to-end data lineage and audit logs of accesses. For instance, you can trace which users accessed the SSN column and whether they saw masked or real values – a valuable report for compliance audits (Basel III operational risk management expects firms to monitor and audit data access rigorously). The solution maintains auditability and separation of duties: the people managing the keys and policies can be different from those consuming the data, adding an extra layer of internal control.

Fail-Safe Behavior:

It's important to handle errors and edge cases. Our Python UDF was written to catch exceptions (e.g., if decryption fails or input is malformed) and return a safe message or null. This prevents any unexpected crashes in production jobs – a must for pipeline reliability. We also advise adding monitoring: e.g., if the UDF returns an "Error:" string or null frequently, that could indicate an issue with keys or data. Unity Catalog doesn't natively alert on masking failures, so implement logging in the UDF or a periodic check on data quality.

Performance Profile:

One concern with using Python UDFs is performance. It's true that a Python-based decryption UDF will be slower than a native Spark SQL function. However, Databricks' implementation of external UDFs loads the library once per worker, not per row, which mitigates some overhead. Still, the decryption happens in Python for each value, which can be the gating factor in large scans. That said, in many scenarios the performance is acceptable given the trade-off of security – especially if the sensitive columns are not massive in size. Also note that the masking UDF itself is a lightweight SQL CASE expression; the heavy lift is the Python decrypt which only runs for privileged users (for non-privileged, it might return the pre-masked value without needing to decrypt at all). We tested this pattern on Databricks Runtime 16.4+ (which supports Python UDF with custom dependencies) and found it workable for moderate data volumes. Additionally, this feature is supported on both all-purpose clusters and serverless SQL endpoints – if using sql serverless, make sure to configure the cluster to allow downloading the pip package (pgpy, etc.) either via init script or environment, as internet access may be needed to fetch the library. In summary, design for scale: apply this masking selectively and monitor performance. If you have to decrypt billions of rows regularly, consider using native AES encryption (if allowable) or scaling out with more cluster resources. For most compliance use cases (e.g., decrypting a few columns for authorized analytics or reporting), the convenience and security of this approach far outweigh the performance cost.

Enterprise Ready

By addressing these aspects, the solution is enterprise-ready. You achieve the goal of column-level security without giving up the advantages of the Databricks Lakehouse. Data remains encrypted at-rest and masked in-use for most users, yet decrypts just-in-time for those with clearance – all within the platform's governance framework. This directly supports compliance with regulations like HIPAA (protecting PHI by default), PCI-DSS (card data is encrypted except when accessed by payment processing roles), and Basel III (strong internal data controls and audit trails for operational risk).

Business Impact and Next Steps

Regulatory Compliance, Simplified:

The approach meets or exceeds data security mandates in healthcare, finance, and beyond. Fine-grained policies ensure that only authorized personnel can view PII/PHI, helping avoid costly breaches or compliance violations. Controls can be mapped directly to standards – for example, HIPAA's minimum necessary access principle is enforced by group-based masks, and PCI-DSS requirements for encryption of cardholder data are met by keeping PAN encrypted in Delta tables. Unity Catalog's centralized policies let you "define once, enforce everywhere," which is exactly what auditors love to see

Increased Trust and Data Utility:

By protecting sensitive data at the column level, you enable broader data analytics and AI on the lakehouse without exposing raw sensitive values. Teams can work with masked data for development and testing, and only unlock real data in production with proper approvals. This balanced approach fosters an environment of trust – both internally (between data engineering and compliance teams) and externally (with customers whose data is protected). It maintains data usability (no need to completely omit or heavily obfuscate critical fields) while still safeguarding privacy.

Performance and Scalability for the Long Term:

While Python UDFs add some overhead, this approach is scalable with your Databricks infrastructure. As your data grows, you can allocate more resources or optimize your usage of the UDF (e.g., caching results, using materialized views for frequent decrypts, etc.). The solution is cloud-native and works across Azure, AWS, and GCP Databricks deployments – important for multi-cloud compliance needs. Moreover, because it uses Databricks' built-in features (as opposed to an external tool), it benefits from continuous improvements in the platform. We anticipate that Databricks will keep optimizing UDF performance and adding more native encryption options; by building on their framework, you're future-proofing your data security architecture.

Integration with Koantek Accelerators

Koantek, as a Databricks partner, has developed accelerators that embed these best practices so you don't have to start from scratch. For instance, our Unity Catalog "Governance by Design" solution comes with pre-built masking policies and compliance rule sets mapped to regulations like HIPAA and Basel III. This means we have templates for common sensitive fields (SSN, DOB, Credit Card, etc.) and default policies (masking, hashing, role-based filters) ready to deploy – accelerating your implementation while following proven blueprints. Similarly, Koantek's AI Factory (Ascend AI Brickbuilder) includes governance guardrails as a core tenet, so any AI/ML pipelines you build have data masking, lineage, and security controls from day one. By leveraging these accelerators, platform teams can achieve up to 80% automation and 90% policy coverage out-of-the-box when securing sensitive data in Databricks. In practical terms, that's faster time-to-value (weeks instead of months to reach a compliant state) and confidence that your data platform is meeting industry benchmarks for security.

What are Brickbuilders

Based on a foundation of proven customer deployments, Databricks Brickbuilder Solutions and Accelerators package together the experience and knowledge of partners into pre-built code, modular frameworks, and custom services to help you unlock the full potential of the Databricks Data Intelligence Platform to boost productivity and extract value from data.

- **Brickbuilder Accelerators**: Pre-built code and frameworks for quickly implementing a specific methodology or Databricks capability.
- **Brickbuilder Solutions:** End-to-end lakehouse solutions and services for cloud migrations and common industry use cases.

Conclusion

Column-level encryption and dynamic masking in Databricks empower enterprises to unlock analytics without compromising security or compliance. By extending beyond AES to support PGP, SM4, and other encryption standards, platform architects can meet diverse regulatory and interoperability requirements within the lakehouse. Using secret-backed Python UDFs and Unity Catalog masking policies, this native, scalable solution ensures the right users see the right data while others view only masked or encrypted values, minimizing exposure risk.

The results are clear: stronger compliance, reduced manual governance effort, and greater agility in using sensitive data for insights. Organizations can automate up to 80% of data handling and cut policy management by over half, freeing teams to focus on innovation instead of access concerns.

Next Steps

If your organization is ready to strengthen data compliance and implement column-level security on Databricks, explore Koantek's accelerators from Unity Catalog Unlocked to AI Factory framework is designed to fast-track secure key management and masking policies. Developed in collaboration with Databricks engineering, our solutions embed governance, security, and automation into every deployment.
Don't let data security be an afterthought make it a catalyst for trust and innovation. Secure your data at the column level, meet compliance with confidence, and turn your lakehouse into both an analytics powerhouse and a fortress for sensitive data.

Ready to strengthen your data security posture?

Partner with Koantek to implement governance-first architectures on Databricks that protect sensitive data, simplify compliance, and unlock analytics with confidence.

Contact us at **sales@koantek.com** to start your secure data transformation journey.