Data Security and Data Masking in ClickHouse

Welcome to this comprehensive guide on implementing robust data security and effective data masking techniques in ClickHouse. Throughout this presentation, we'll explore the wide range of security features available in ClickHouse that help organizations protect sensitive information while maintaining database performance and functionality.

We'll cover everything from basic access controls to advanced encryption methods, providing practical implementation guidance for database administrators and data engineers. Let's dive into the world of data protection in ClickHouse!

by Shiv Iyer

Agenda: Data Security & Masking Overview

Access Control & Authentication

User management, RBAC, and authentication methods

Data Masking Techniques

SQL functions, custom UDFs, and view-based masking

Encryption Options

Disk-level, column-level, and in-transit encryption

Advanced Implementations

Complex solutions using materialized views and policies

Why Data Security Matters in ClickHouse

Regulatory Compliance

Meet GDPR, HIPAA, PCI DSS, and other regulatory requirements through proper data protection measures.

Breach Prevention

Protect against unauthorized access and potential data leaks that could damage reputation and finances.

Data Governance

Maintain control over who can access what data, ensuring proper data stewardship throughout your organization.

User Management Fundamentals

Creating Users with Specific Privileges

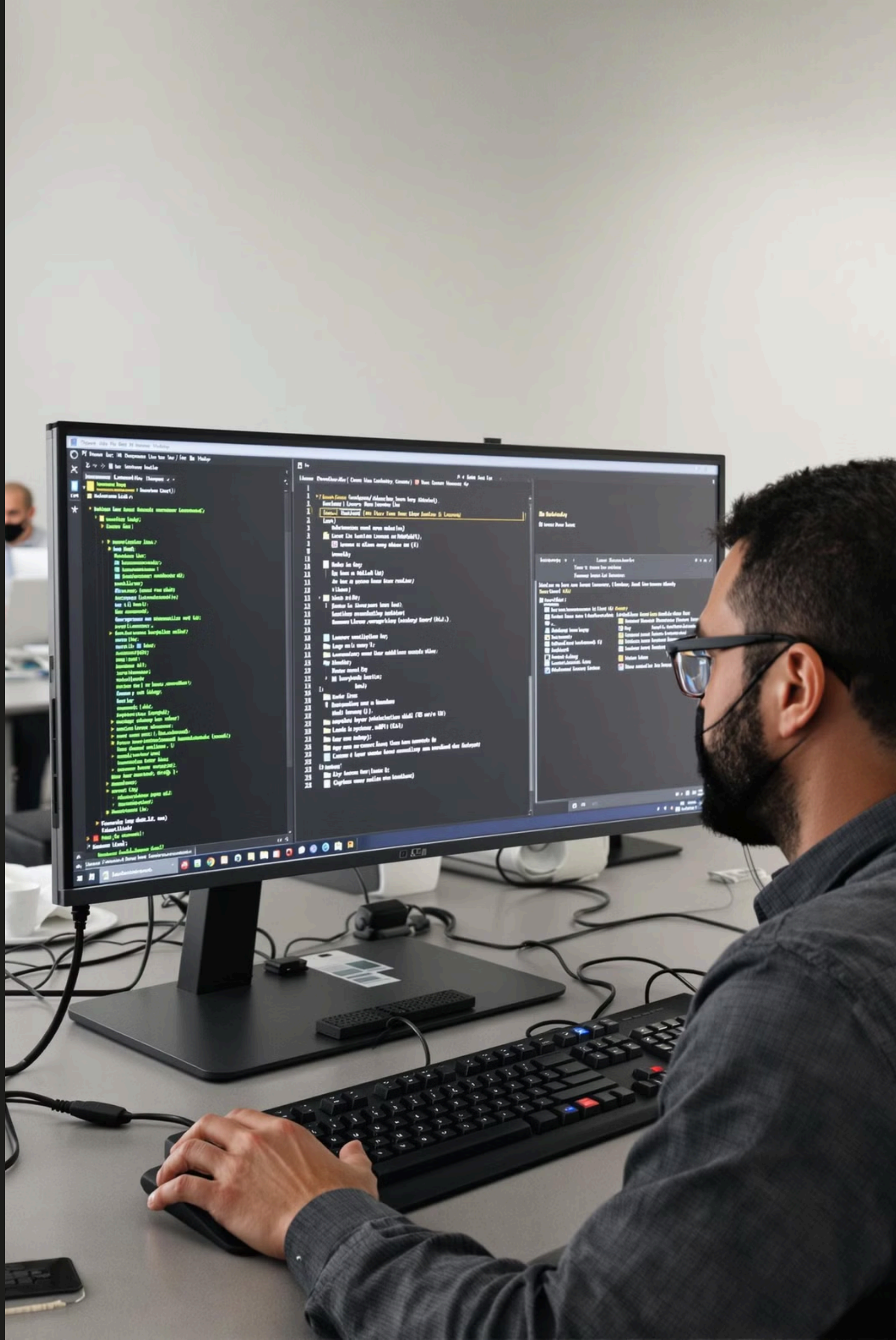
Define granular access permissions to limit data exposure based on job requirements.

Role-Based Access Control (RBAC)

Group common permissions into roles to simplify administration and ensure consistency.

Row-Level Security Policies

Implement data filtering at the row level to show only appropriate data to specific users.



Creating Users with Specific Privileges

SQL Commands

```
CREATE USER analyst
IDENTIFIED WITH sha256_password
BY 'strong_password'
SETTINGS max_memory_usage = 10000000000;

GRANT SELECT ON database.table TO analyst;
GRANT SELECT(id, name) ON database.sensitive_table TO analyst;
```

Best Practices

- Follow the principle of least privilege
- Regularly audit user privileges
- Implement password policies
- Remove unused accounts promptly
- Document access grants for compliance

Role-Based Access Control Implementation

Create Roles

Define roles that represent job functions or responsibilities within your organization.

Assign Privileges

Grant specific database permissions to each role based on access requirements.

Assign Users to Roles

Link users to appropriate roles instead of managing individual permissions.

Review & Update

Regularly audit role assignments and adjust as organizational needs change.

RBAC SQL Examples

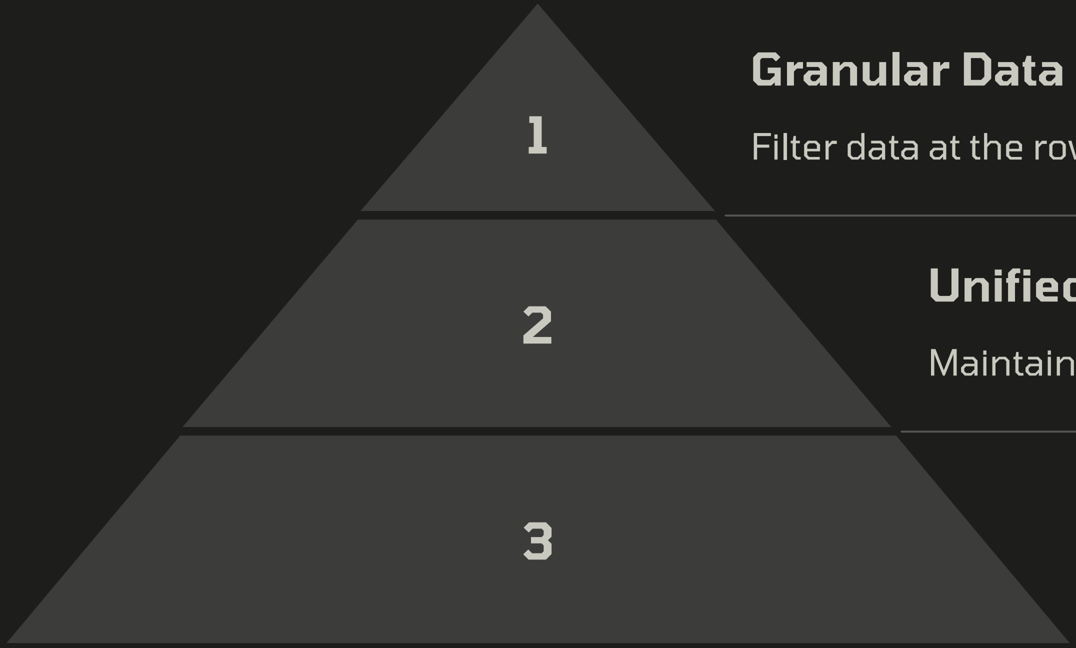
Creating and Assigning Roles

```
-- Create roles
CREATE ROLE analyst_role;
CREATE ROLE admin_role;

-- Grant permissions to roles
GRANT SELECT ON analytics.* TO analyst_role;
GRANT ALL ON *.* TO admin_role;

-- Assign roles to users
GRANT analyst_role TO user1;
GRANT admin_role TO admin_user;
```

Using Role Hierarchies

```
-- Create nested roles
CREATE ROLE junior_analyst;
CREATE ROLE senior_analyst;

-- Set up hierarchy
GRANT SELECT ON analytics.public_*
TO junior_analyst;

GRANT junior_analyst TO senior_analyst;
GRANT SELECT ON analytics.sensitive_*
TO senior_analyst;
```

Row-Level Security Policies



Granular Data Access

Filter data at the row level based on user context

Unified Data Model

Maintain a single table while controlling visibility

Transparent Security

Seamless to end users and applications

Implementing Row-Level Security

Example Policy Definition

```
-- Create row policy for regional access
CREATE ROW POLICY regional_access
ON sales.orders
FOR SELECT
USING region_id = currentRegion();

-- Policy for different roles
CREATE ROW POLICY manager_access
ON employees
FOR SELECT
USING (
  hasRole('manager') AND
  department_id = currentDepartmentId()
) OR hasRole('admin');
```

Key Considerations

- Define policies based on business rules
- Use user context variables or functions
- Combine multiple conditions for complex access patterns
- Test thoroughly to avoid unintended access restrictions
- Document policies for compliance audits

Authentication Methods in ClickHouse

Password Authentication

Basic method using SHA-256 password hashing for user verification

SSL Certificates

Certificate-based authentication for stronger security without password transmission

External Authentication

Support for Kerberos and custom authentication plugins

LDAP Integration

Centralized authentication using enterprise directory services

Setting Up Password Authentication

Server Configuration

```
sha256_password
*
```

Creating Users with Passwords

```
-- Plain password (less secure)
CREATE USER user1 IDENTIFIED WITH plaintext_password
BY 'password123';

-- SHA-256 hashed password (more secure)
CREATE USER user2 IDENTIFIED WITH sha256_password
BY 'securePassword!';

-- Double SHA-1 (legacy)
CREATE USER user3 IDENTIFIED WITH
double_sha1_password BY 'anotherPassword';
```

SSL Certificate Authentication

Generate Certificates

Create SSL certificates for your ClickHouse server and clients using a trusted Certificate Authority.

Configure Server

Set up the server to require and validate client certificates for authentication.

Distribute Client Certs

Securely provide certificates to authorized clients that need to connect to ClickHouse.

Configure Clients

Set up clients to present their certificates when connecting to the ClickHouse server.

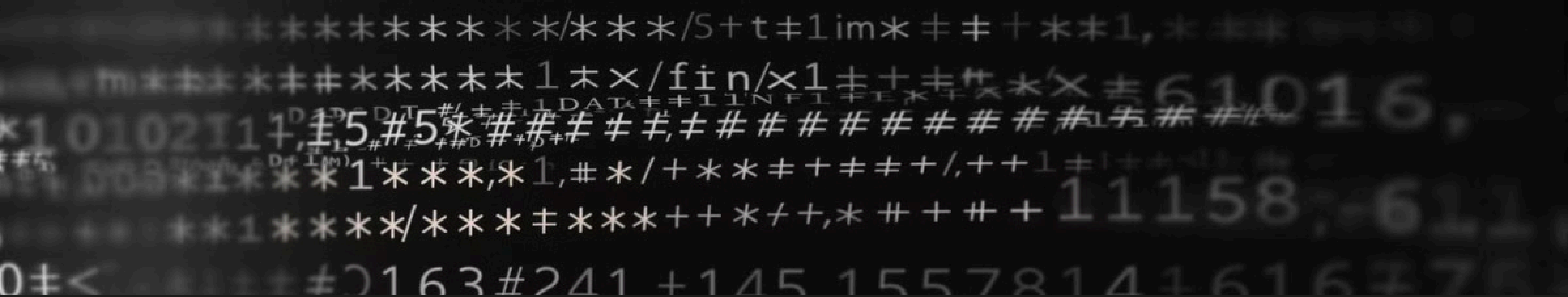
LDAP Integration

Benefits

- Centralized user management
- Simplified authentication
- Integration with enterprise systems
- Enforcement of password policies
- Reduced administrative overhead

Configuration

```
ldap.example.com
636
uid={user_name},ou=users,dc=example,dc=com
300
true
tls1.2
demand
```



Introduction to Data Masking

Definition

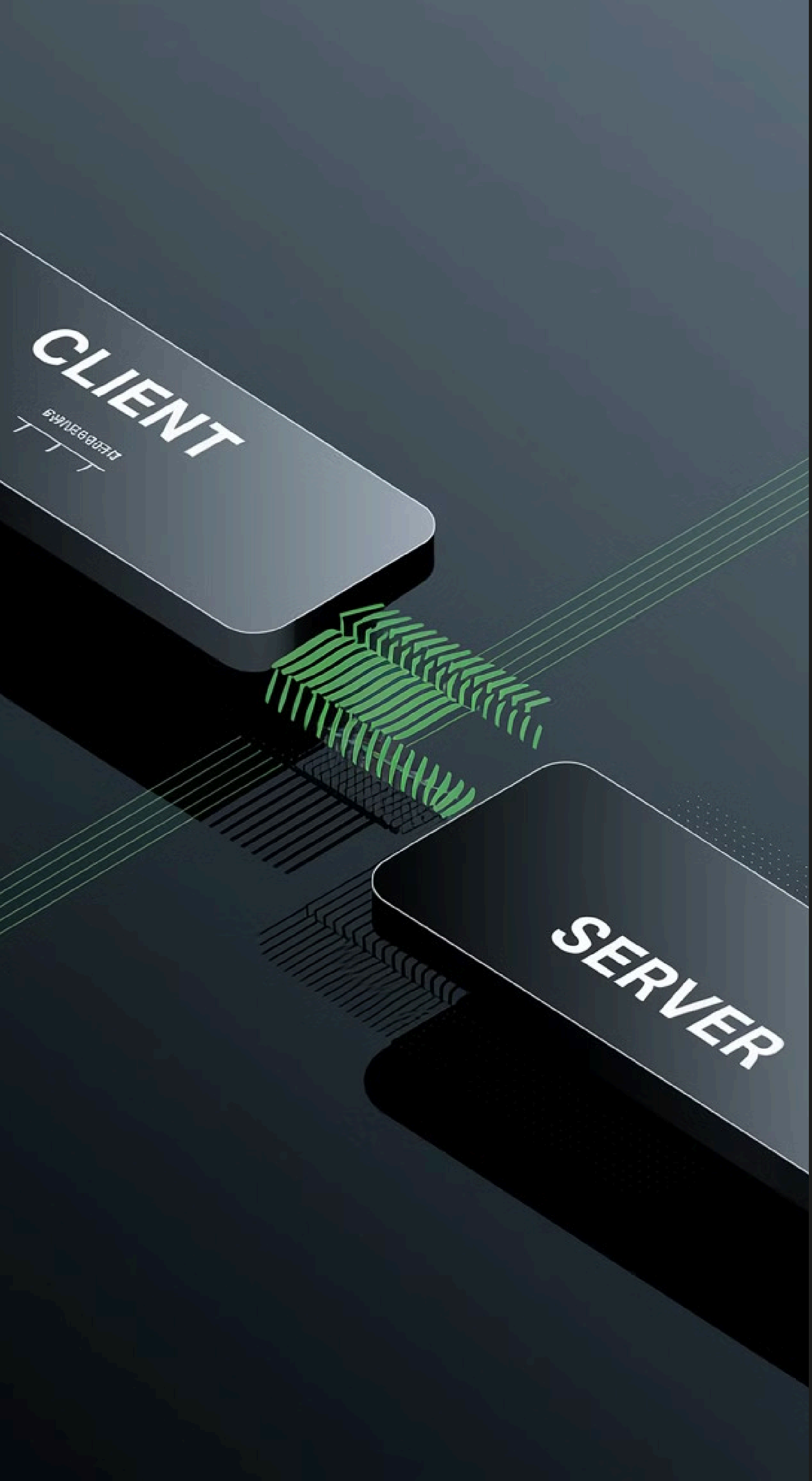
Data masking is the process of hiding original data with modified content while preserving the data format and usability for non-sensitive purposes.

Purpose

Protect sensitive information while allowing access to data structure for development, testing, or analysis without exposing protected information.

Common Use Cases

Customer data protection, PII compliance, test environment security, and limited data sharing with third parties or contractors.



Data Masking Approaches in ClickHouse

Built-in SQL Functions

Use ClickHouse's native functions like maskPhone() and maskEmail() for standard masking operations

Custom User-Defined Functions

Create specialized masking logic with UDFs for unique requirements

View-Based Masking

Implement column-level security using views with embedded masking logic

4

Advanced Solutions

Combine materialized views and access policies for comprehensive masking systems

Using Built-in SQL Functions for Masking

Available Functions

- **maskPhone()** - Masks phone numbers
- **maskEmail()** - Masks email addresses
- **maskCardNumber()** - Masks credit card numbers
- **maskData()** - General purpose masking

Implementation Example

```
SELECT
  id,
  name,
  maskPhone(phone_number) AS masked_phone,
  maskEmail(email) AS masked_email,
  maskCardNumber(credit_card) AS masked_cc
FROM customers;

-- Results:
-- 1, John Doe, +1-XXX-XXX-3456, j***@example.com,
XXXX-XXXX-XXXX-1234
```

Masking Function Behavior

Function	Input	Output	Description
maskPhone()	+1-123-456-7890	+1-XXX-XXX-7890	Keeps country code and last 4 digits
maskEmail()	john.doe@example.com	j***@example.com	Keeps first letter and domain
maskCardNumber()	1234-5678-9012-3456	XXXX-XXXX-XXXX-3456	Keeps only last 4 digits
maskData()	Secret123	XXXXXXX123	Configurable masking behavior

Creating Custom Masking UDFs

Custom Function Definition

```
CREATE FUNCTION maskCustomData AS
  (input, showChars) ->
    if(
      length(input) <= showChars,
      input,
      substring(input, 1, showChars) ||
      replaceRegexpAll(
        substring(input, showChars + 1),
        '.',
        '*'
      )
    );
```

Function Usage

```
-- Show first 2 characters
SELECT
  maskCustomData('12345678', 2) AS result;
-- Returns: 12******

-- Show first 4 characters
SELECT
  maskCustomData('ABCDEFGH', 4) AS result;
-- Returns: ABCD****

-- Varying amounts of visible data
SELECT
  maskCustomData(full_name, 1) AS name,
  maskCustomData(ssn, 0) AS ssn,
  maskCustomData(phone, 6) AS phone
FROM customer_data;
```

Column-Level Security with Views

Create Base Tables

Store complete data in base tables with restricted access

2

Define Masked Views

Create views that apply masking functions to sensitive columns

Grant Access to Views

Allow appropriate roles to query masked views instead of base tables

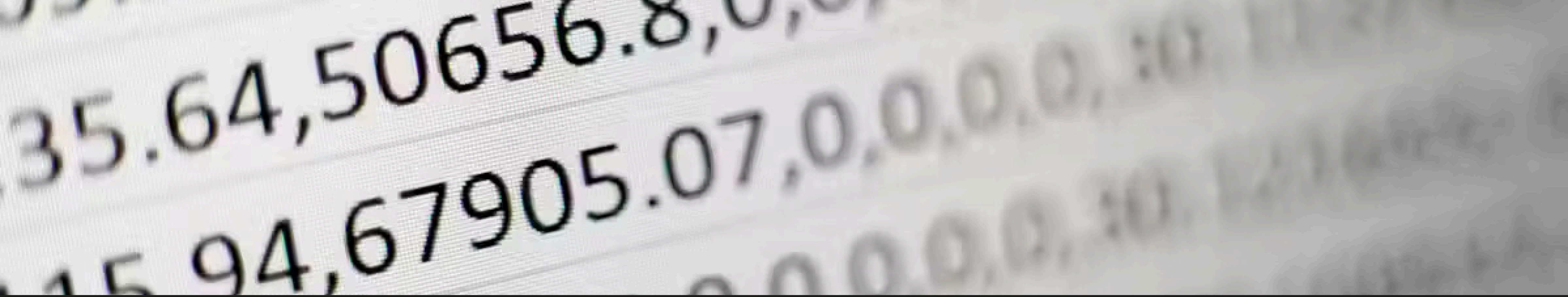
Implementing View-Based Masking

View Definition

```
-- Create masked view of customer data
CREATE VIEW masked_customers AS
SELECT
  id,
  name,
  maskCustomData(ssn, 0) AS ssn,
  maskCustomData(phone, 3) AS phone,
  city,
  state
FROM customers;

-- Grant access to analysts
GRANT SELECT ON masked_customers
TO analyst_role;

-- Revoke direct table access
REVOKE SELECT ON customers
FROM analyst_role;
```

Benefits

- Granular column-level protection
- Simplified access control
- Consistent application of masking rules
- Centralized management of masking logic
- Transparent to end users and applications



Encryption Options Overview

Disk-Level Encryption

Protect data at rest by encrypting the entire storage layer, securing all database files when the system is offline.

Column Encryption

Selectively encrypt sensitive columns within tables, maintaining granular protection even during database operation.

Data in Transit

Secure information as it travels between clients and servers or between cluster nodes using encryption protocols.

Disk-Level Encryption

Options

- **OS-level encryption**: Linux dm-crypt, LUKS
- **Filesystem encryption**: EncFS, eCryptfs
- **Hardware encryption**: Self-encrypting drives
- **ClickHouse native encryption**: Built-in encryption features

Configuration Example

```
/path/to/encrypted/storage/
/path/to/key.file




   encrypted
```

Column Encryption

1 **Generate and Secure Encryption Keys**

Create strong encryption keys and implement a secure key management system.

2 **Choose Columns to Encrypt**

Identify sensitive columns that require encryption while considering performance impact.

λ **Implement Encryption Functions**

Use built-in encrypt/decrypt functions or create custom encryption UDFs.

4 **Control Access to Decryption**

Limit decryption capabilities to authorized users with appropriate permissions.

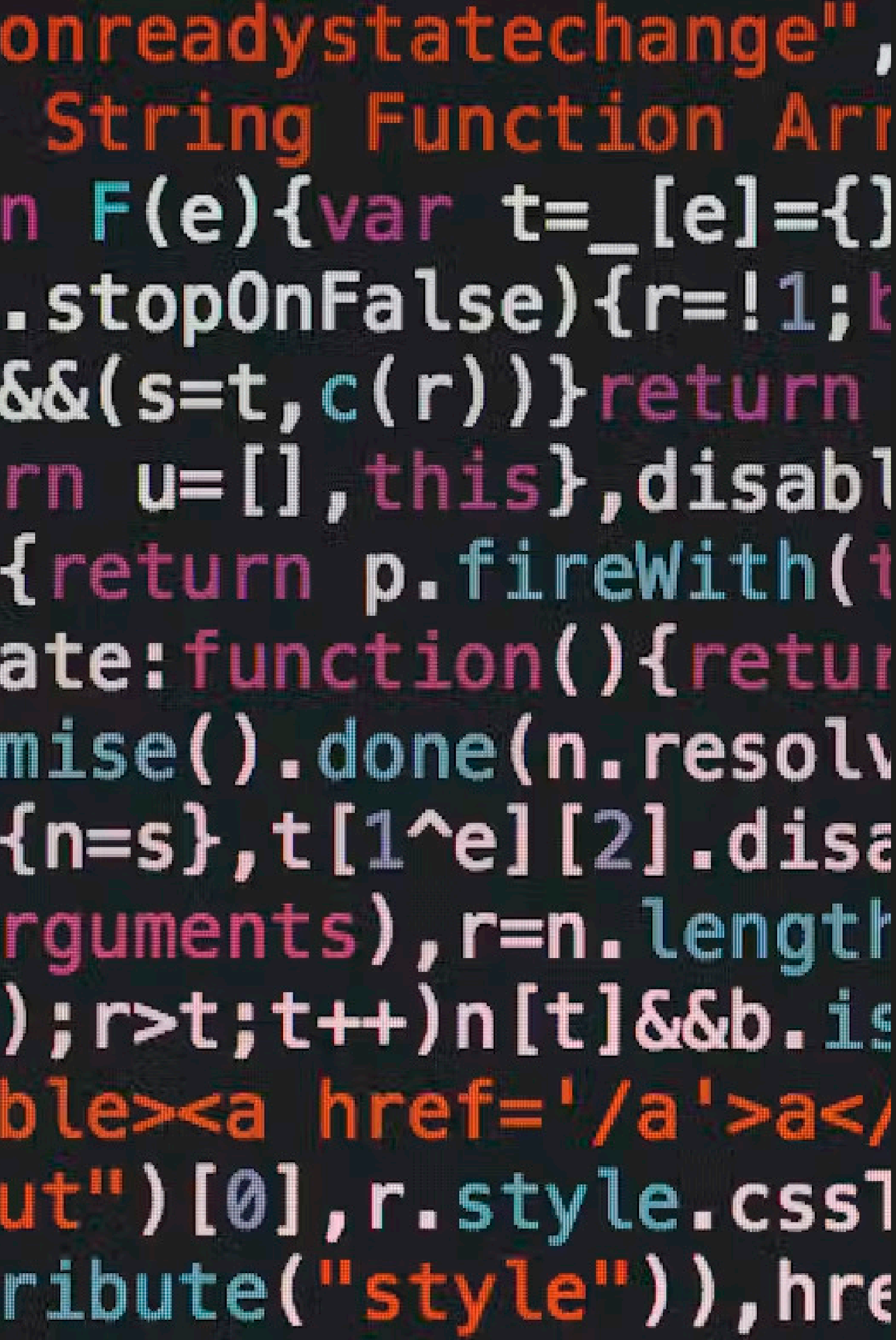
Column Encryption Implementation

Encryption Logic

```
-- Create function for encryption
CREATE FUNCTION encryptAES256 AS
  (data, key) -> encrypt('aes-256-cbc',
                 data,
                 key);

-- Create function for decryption
CREATE FUNCTION decryptAES256 AS
  (data, key) -> decrypt('aes-256-cbc',
                 data,
                 key);

-- Secure key retrieval function
CREATE FUNCTION getSecureKey AS
  () -> extractFromConfig('encryption_keys.user_data');
```

Usage in Table Operations

```
-- Insert with encryption
INSERT INTO sensitive_data
SELECT
  id,
  name,
  encryptAES256(social_security_number,
         getSecureKey()) AS encrypted_ssn
FROM source_data;

-- Query with decryption
SELECT
  id,
  name,
  decryptAES256(encrypted_ssn,
         getSecureKey()) AS ssn
FROM sensitive_data
WHERE id = 123;
```



Securing Data in Transit

443

Default SSL Port

Standard secure port for ClickHouse HTTPS connections

9440

Secure Native Protocol

Default port for encrypted native protocol

256

Bit Encryption

Recommended SSL encryption strength

SSL/TLS Configuration

Server Configuration

```
8443



    /path/to/server.crt
    /path/to/server.key
    /path/to/ca.crt
    strict
    true
    true
    sslv2,sslv3,tlsv1
    true
```

Client Configuration

```
clickhouse-client \
  --secure \
  --host=example.com \
  --port=8443 \
  --ssl_cert_file=/path/to/client.crt \
  --ssl_key_file=/path/to/client.key \
  --ssl_ca_file=/path/to/ca.crt


jdbc:clickhouse://example.com:8443/default?
ssl=true&sslmode=strict&sslrootcert=/path/to/ca.crt
```

Secure Internode Communication

Generate Node Certificates

Create individual certificates for each node in your ClickHouse cluster.

Configure Interserver Encryption

Set up encryption parameters for communication between cluster nodes.

Verify Certificate Trust

Ensure all nodes properly verify certificates from other nodes.

Test Secure Communication

Validate that nodes can securely communicate using encrypted channels.

Internode Encryption Configuration

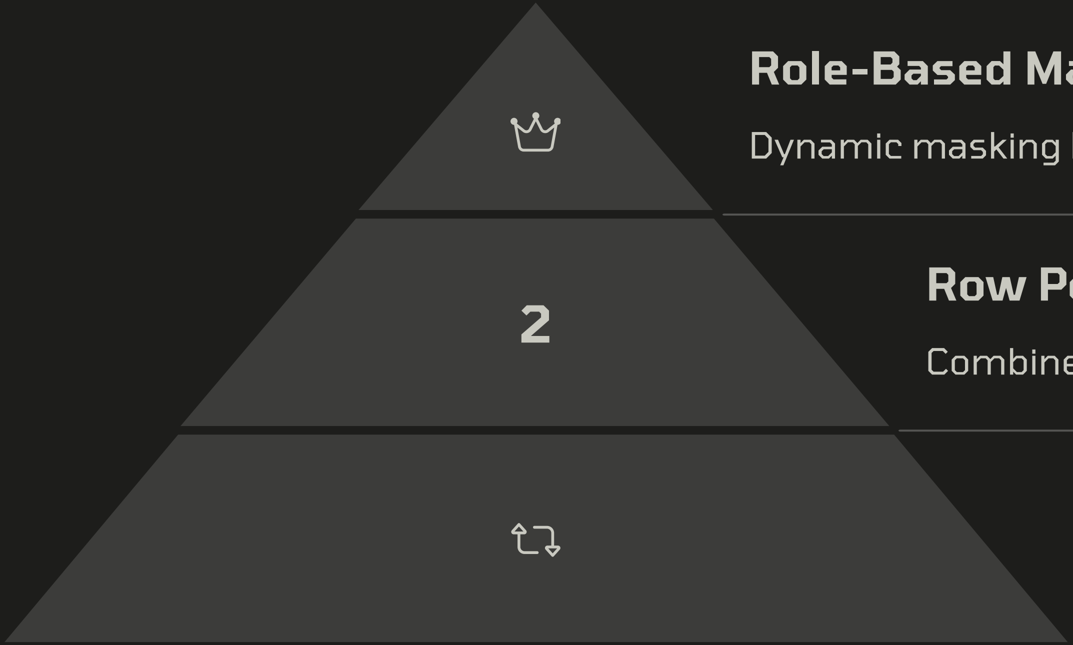
Configuration Settings

```
9010

  interserver
  password
```

Cluster Definition

```






        node1.example.com
        9010
        1



        node2.example.com
        9010
        1
```

Advanced Data Masking Implementation



Role-Based Masking

Dynamic masking based on user role

Row Policy Filters

Combine with row-level security

Materialized Views

Pre-computed masked data

Creating Row Policies for Different User Roles

Policy Definition

```
-- Create policy for different user roles
CREATE ROW POLICY sensitive_data
ON customers
FOR SELECT USING (
  -- Regular users see only their data
  hasRole('regular_user')
  AND (showCustomerData = 0)
  OR
  -- Admins see everything
  hasRole('admin')
);
```

Usage Considerations

- Combine multiple conditions for complex access patterns
- Use context variables to dynamically filter data
- Create separate policies for different operations (SELECT, INSERT, etc.)
- Test thoroughly to avoid unintended restrictions
- Document policies for auditing and compliance

Materialized Views with Conditional Masking

View Definition

```
-- Create materialized view with conditional masking
CREATE MATERIALIZED VIEW customer_data_secure
ENGINE = MergeTree()
ORDER BY id
AS SELECT
  id,
  -- Conditional name masking
  if(hasRole('admin') OR showCustomerData = 1,
    full_name,
    concat(substring(full_name, 1, 1), '***')
  ) AS name,
  -- Conditional email masking
  if(hasRole('admin') OR showCustomerData = 1,
    email,
    maskEmail(email)
  ) AS email,
  -- Conditional phone masking
  if(hasRole('admin') OR showCustomerData = 1,
    phone,
    maskPhone(phone)
  ) AS phone
FROM customers;
```

Benefits

- Pre-computed masked data for performance
- Role-based conditional masking
- Consistent application of masking rules
- Reduced query complexity for end users
- Centralized definition of masking logic

Dynamic Data Masking with User Contexts

1

User Authentication

Establish user identity and role

Context Variables

Set session-specific masking controls

3

Conditional Masking

Apply masking based on context

Data Presentation

Show appropriately masked results

Implementing User Context Variables

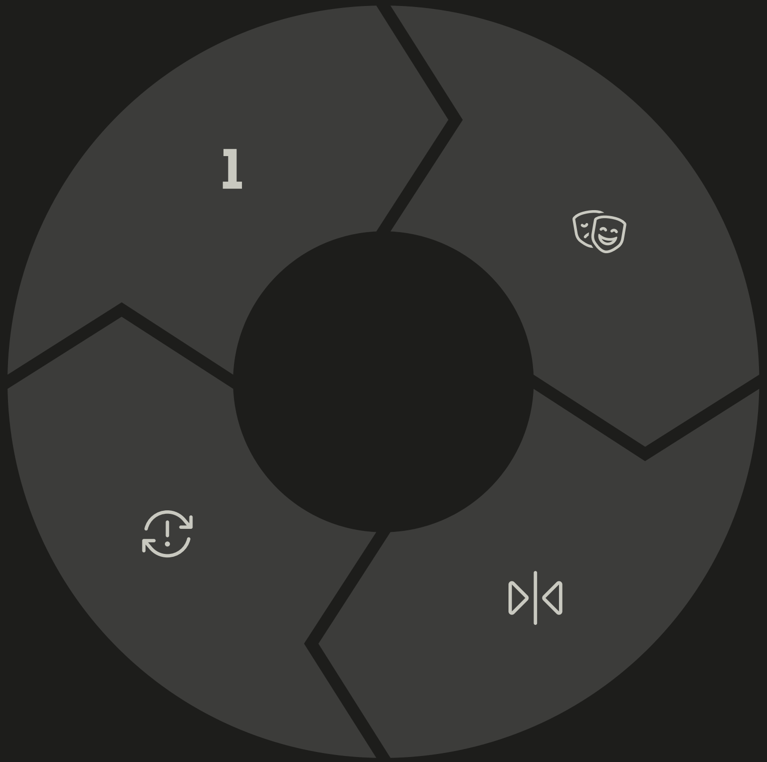
Setting Context Variables

```
-- At session start
SET allow_sensitive_data = 1;
SET current_department_id = 42;
SET current_customer_id = 12345;

-- In application code (example)
connection.execute(
  "SET allow_sensitive_data = ?",
  [user.hasPermission("view_sensitive") ? 1 : 0]
);
connection.execute(
  "SET current_department_id = ?",
  [user.departmentId]
);
```

Using Context in Queries

```
-- In view definition
CREATE VIEW employee_data AS
SELECT
  id,
  name,
  IF(allow_sensitive_data = 1,
     salary,
     NULL) AS salary,
  department_id,
  IF(allow_sensitive_data = 1 OR
     department_id = current_department_id,
     phone,
     maskPhone(phone)) AS phone
FROM employees
WHERE department_id = current_department_id
  OR allow_sensitive_data = 1;
```

Masking in Development and Testing Environments



Production Data

Original sensitive information in secure environment

Data Masking

Apply consistent masking before data migration

Regular Refresh

Update test data with newly masked production data

Test Environment

Use masked data for development and testing

Creating Test Data with INSERT SELECT

Data Masking During Migration

```
-- Copy and mask data to test environment
INSERT INTO test.customers
SELECT
  id,
  maskCustomData(name, 1) AS name,
  maskPhone(phone) AS phone,
  maskEmail(email) AS email,
  replaceAll(address, '.', '*') AS address,
  maskCardNumber(credit_card) AS credit_card,
  region,
  registration_date
FROM prod.customers;
```

Consistent Masking Approach

When creating test data from production, it's essential to:

- Apply consistent masking rules across all sensitive fields
- Preserve data relationships and referential integrity
- Maintain data format and validation rules
- Document the masking approach for developers
- Automate the refresh process with scheduled jobs

Testing Security Implementations

1

Verify Access Controls

Test that users can only access data appropriate for their roles and permissions.

Validate Masking Rules

Confirm that masking functions properly obscure sensitive data according to specifications.

Audit Encryption

Verify that encrypted data cannot be accessed without proper decryption keys.

Attempt Security Bypass

Try to circumvent security measures to identify potential vulnerabilities.

Security Testing SQL Examples

Testing Access Controls

```
-- Test as regular user
SET user_name = 'regular_user';

-- Attempt to access restricted table
SELECT * FROM sensitive_data;
-- Should fail or return filtered results

-- Test as admin
SET user_name = 'admin_user';

-- Attempt same query
SELECT * FROM sensitive_data;
-- Should return complete results

-- Test row-level policy
SET current_region_id = 5;
SELECT * FROM regional_data;
-- Should only show region 5 data
```

Testing Masking Rules

```
-- Test masking functions directly
SELECT
  maskPhone('+1-123-456-7890') AS masked_phone,
  '+1-123-456-7890' AS original_phone;
-- Should show masked version

-- Test conditional masking
SET show_sensitive = 0;
SELECT email FROM customer_view WHERE id = 1;
-- Should show masked email

SET show_sensitive = 1;
SELECT email FROM customer_view WHERE id = 1;
-- Should show original email if authorized
```

Auditing Security Measures

Enable Comprehensive Logging

Configure detailed logging of all data access, especially for sensitive information.

Regular Security Reviews

Schedule periodic audits of security configurations, permissions, and access patterns.

Monitor Suspicious Activity

Implement alerts for unusual data access patterns or potential security violations.

Maintain Compliance Documentation

Keep detailed records of security measures for regulatory compliance requirements.

Configuring Security Logging

Log Configuration

```


trace
/var/log/clickhouse-server/clickhouse-server.log
/var/log/clickhouse-server/clickhouse-server.err.log
1000M
10



system
query_log
toYYYYMM(event_date)
7500


```

Querying Audit Logs

```
-- Find all queries accessing sensitive tables
SELECT
  query_id,
  user,
  query_start_time,
  query
FROM system.query_log
WHERE
  query LIKE '%sensitive_data%'
  AND event_date >= today() - 7
ORDER BY query_start_time DESC;

-- Check failed authentication attempts
SELECT
  event_time,
  user,
  auth_type,
  error_code,
  error_message
FROM system.text_log
WHERE
  event_type = 'AuthenticationFailed'
  AND event_date >= today() - 7;
```

Performance Considerations for Security Features

Data Masking

Minimal impact when using built-in functions; more complex UDFs may have higher CPU usage

Encryption

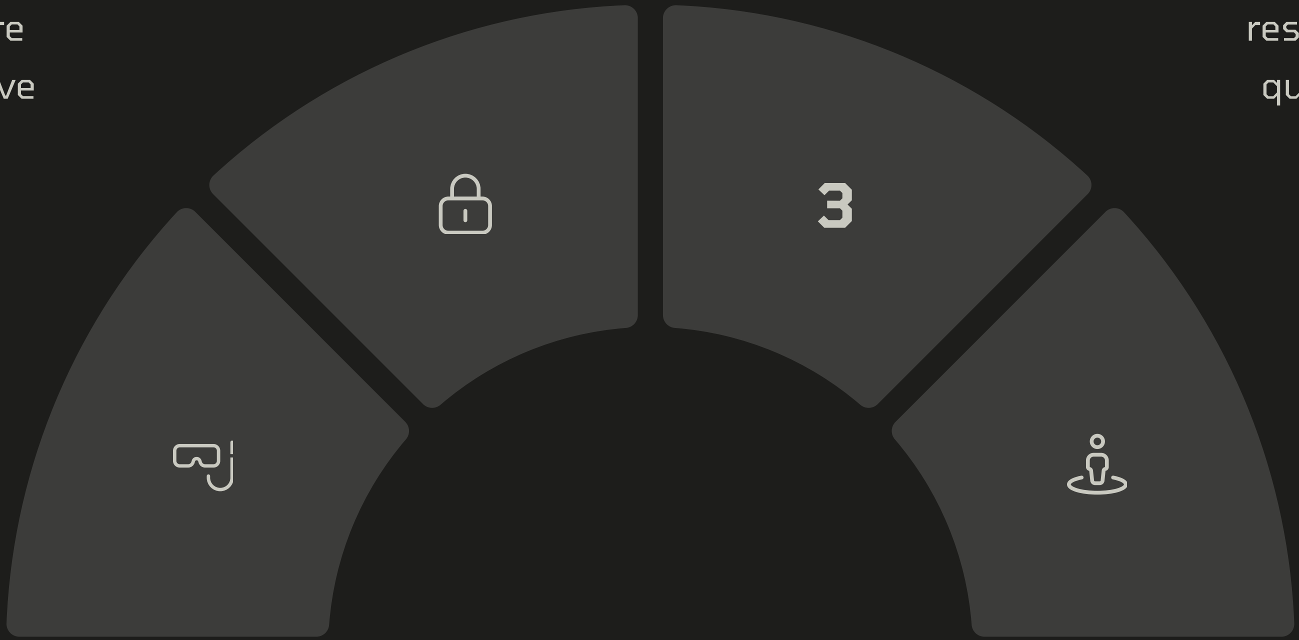
Column encryption adds significant CPU overhead; disk encryption primarily affects I/O operations

Row Policies

May reduce query performance when filtering large datasets, especially with complex conditions

Materialized Views

Initial creation requires resources, but subsequent queries benefit from pre-computed results



Optimizing Security Performance

Indexing Strategy

Ensure proper indexes on columns used in security filters to minimize scan operations.

Caching Mechanisms

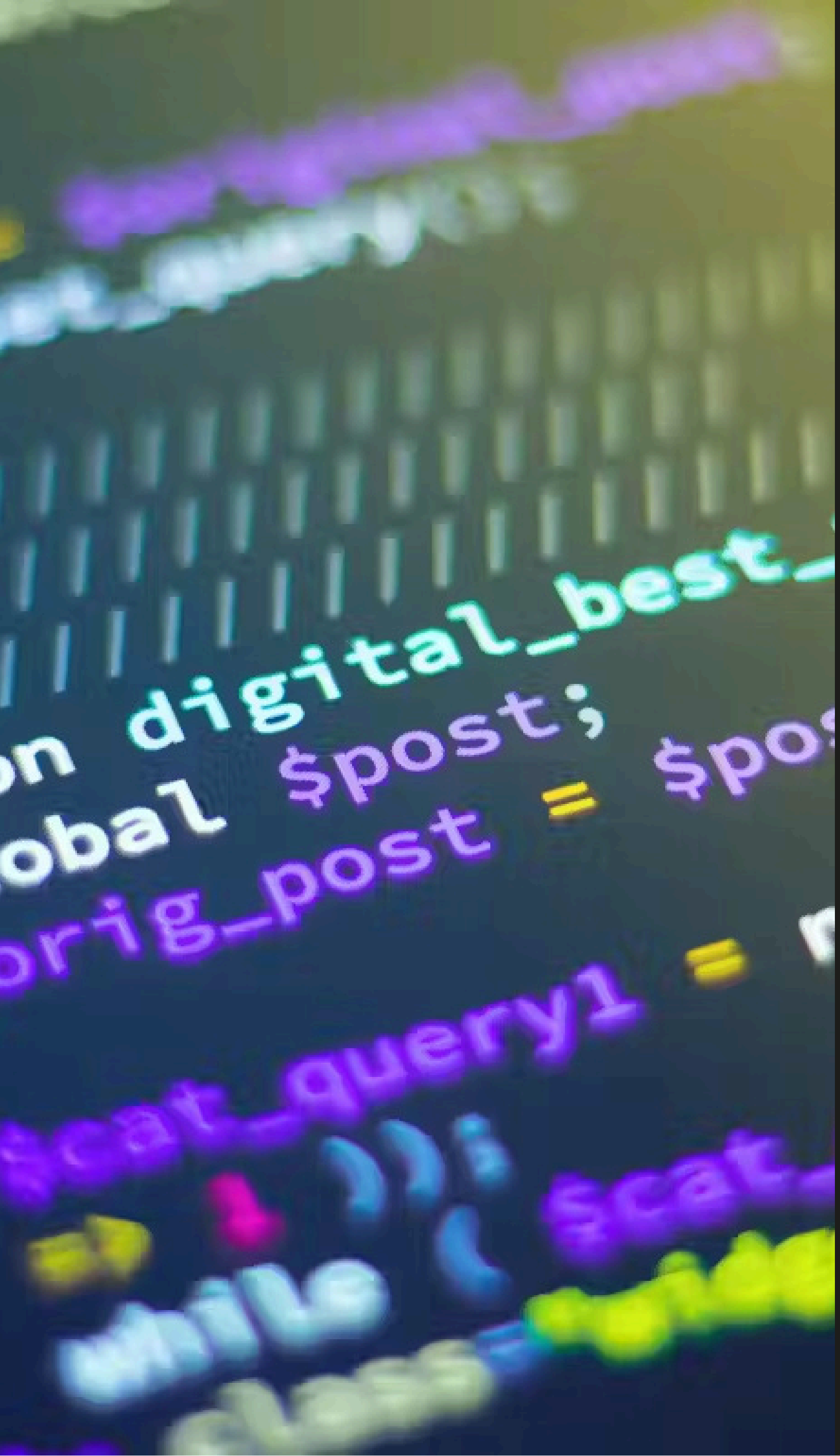
Utilize ClickHouse's caching features to reduce the overhead of repeated security checks.

Resource Allocation

Allocate sufficient CPU and memory resources to handle additional security processing.

Data Partitioning

Structure data partitions to align with security boundaries for more efficient filtering.



Balancing Security and Performance

Security Feature	Performance Impact	Optimization Strategy
Row-Level Security	Medium to High	Use materialized views for common filters
Column Masking	Low to Medium	Optimize UDFs, use built-in functions
Column Encryption	High	Encrypt only essential columns
Disk Encryption	Low (CPU), Medium (I/O)	Use hardware acceleration if available
SSL/TLS	Low to Medium	Session caching, connection pooling

Best Practices for Data Security in ClickHouse

Defense in Depth

Implement multiple security layers rather than relying on a single protection mechanism

Principle of Least Privilege

Grant minimum necessary access rights to users and applications

Monitor and Audit

Maintain comprehensive logging and regular security reviews

Security Implementation Checklist

User Management

- Create specific users for each purpose
- Implement role-based access control
- Enforce strong password policies
- Regularly audit user accounts
- Remove unused accounts promptly

Data Protection

- Identify and classify sensitive data
- Apply appropriate masking techniques
- Implement encryption for sensitive columns
- Configure secure data backups
- Test security measures regularly

System Security

- Enable secure authentication
- Configure SSL/TLS for all connections
- Secure internode communications
- Keep ClickHouse updated
- Monitor system logs for anomalies

Regulatory Compliance and ClickHouse Security

GDPR Compliance

Implement data minimization, masking, and right-to-be-forgotten capabilities to meet European privacy requirements.

HIPAA Requirements

Apply PHI protection through encryption, access controls, and comprehensive audit logs for healthcare data.

PCI DSS Standards

Secure payment card information using strong encryption, masking, and strict access limitations to meet payment industry requirements.

SOC 2 Auditing

Enable comprehensive logging and security controls to demonstrate proper system security during compliance audits.

GDPR-Specific Configuration

Data Protection Features

- **Right to be forgotten**: Implement deletion procedures
- **Data minimization**: Only store necessary fields
- **Purpose limitation**: Use row policies to restrict access
- **Storage limitation**: Configure TTL for data expiration
- **Processing security**: Apply masking and encryption

Implementation Example

```
-- Implement TTL for data expiration
CREATE TABLE gdpr_compliant_data
(
  user_id UInt64,
  name String,
  email String,
  preferences String,
  last_activity Date,
  created_at DateTime
)
ENGINE = MergeTree()
ORDER BY user_id
-- Auto-delete after 2 years of inactivity
TTL last_activity + INTERVAL 2 YEAR;

-- Data deletion procedure
CREATE PROCEDURE forget_user(user_id UInt64)
AS BEGIN
  ALTER TABLE user_data DELETE WHERE user_id = user_id;
  ALTER TABLE user_preferences DELETE WHERE user_id = user_id;
  ALTER TABLE user_activity DELETE WHERE user_id = user_id;
END;
```

Real-World Security Implementation Scenarios

Financial Services

Banks implementing column-level encryption for account data and transaction details, with role-based access for different staff positions.

Healthcare

Medical systems using comprehensive data masking for PHI in test environments while maintaining data utility for development.

E-Commerce

Online retailers applying dynamic masking for customer profiles based on service representative roles and access needs.

Troubleshooting Security Issues

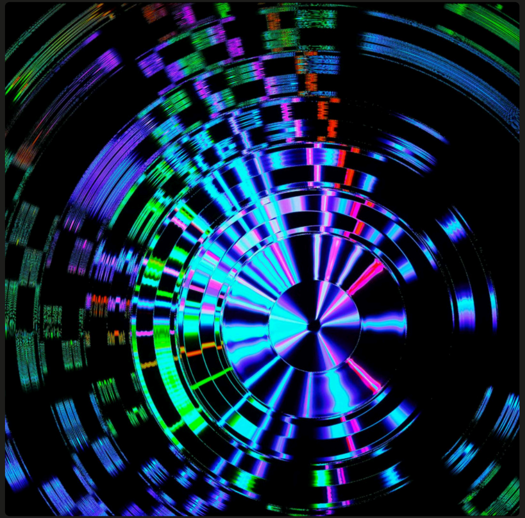
Common Problems

1. **Access denied errors**: Users unable to access needed data due to overly restrictive policies
2. **Performance degradation**: Queries running slowly after implementing security measures
3. **Inconsistent masking**: Data appearing masked in some queries but not in others
4. **Certificate errors**: SSL connection failures due to certificate misconfigurations
5. **Key management issues**: Problems with encryption key access or rotation

Diagnostic Approaches

- Check system.query_log for errors and execution details
- Verify user grants with SHOW GRANTS FOR user
- Inspect role assignments with SHOW CREATE USER
- Test security functions directly with simple queries
- Review server logs for authentication errors
- Use EXPLAIN to analyze query execution with security predicates

Future Security Enhancements in ClickHouse



The ClickHouse security landscape continues to evolve with upcoming features like enhanced anomaly detection for identifying suspicious access patterns, improved integration with zero-trust security frameworks, and preparation for post-quantum cryptography. Watch for advancements in unified security management that will simplify configuration and monitoring while strengthening protection.

Key Takeaways: Data Security & Masking in ClickHouse

1

Defense in Depth

Combine multiple security techniques including access control, masking, and encryption

2

Performance Balance

Carefully implement security features with performance considerations in mind

3

Continuous Monitoring

Maintain comprehensive logging and regular security reviews

4

Regulatory Alignment

Configure security measures to meet relevant compliance requirements