Cyral | WHITE PAPER

# Limitations of Native Access Management Controls in Data Warehouses, and How Cyral Helps

## Contents

Database platforms’ native SSO integrations often leave gaps in security and oversight. Cyral closes these gaps.

## Overview

Cyral transparently integrates with your existing databases (MySQL, CloudSQL, RDS, etc.), data warehouses (Snowflake, BigQuery, etc.), pipelines (Kafka, Kinesis, etc.) and SSO providers (Okta, AD, etc.) to give better control and visibility into data accessed by various users and applications.

While many databases and data warehouses offer ways to integrate directly with Okta and other SSO platforms, these integrations leave gaps in security and oversight because they don’t extend the full power of the database’s native access controls to SSO-authenticated users, nor do they provide scalable access policy tools. In this white paper, we provide an overview of the gaps that exist when you combine SSO with a repository platform’s native access controls, and we show how Cyral can address these gaps. We’ll base our examples on Snowflake with Okta SSO, but these lessons apply all major data repositories and identity platforms, and Cyral’s solution supports all major repositories and platforms.

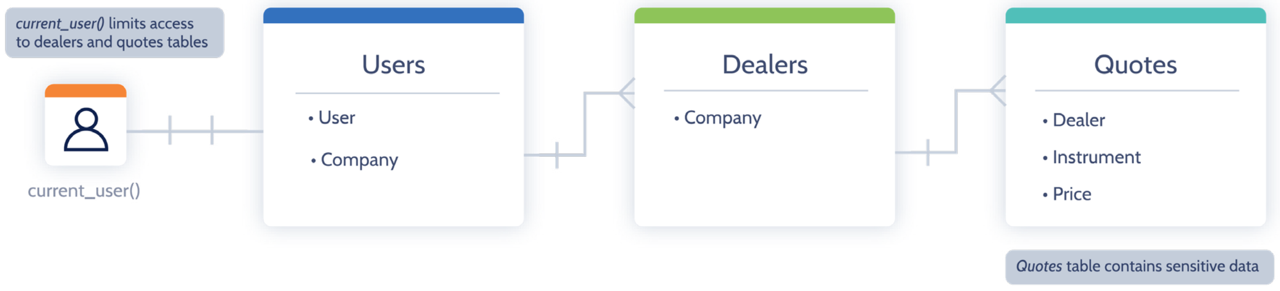
This document focuses on the benefits of Cyral policies and does not cover other aspects of Cyral, such as service credentials disambiguation, simplifying the approvals process, attribute-based authorization, automated threat detection and field-level data transformation. See cyral.com or ask your Cyral representative for more product details.

## Access Management: Snowflake + Okta vs Cyral

The key differences between the two approaches are highlighted in the table below:

| | **Native Controls** | **Cyral** |
|---|---|---|
| Policy Administration | Requires changing and managing data model | No change to data model; policies are clear and readable |
| Support for Data Modifications | Not possible to control access for DML commands | Fully supports access control to DML commands |
| Automation | Policy changes require manual, ad-hoc effort | Policies can be changed via APIs, integrated into CI/CD |
| Extensibility | Work cannot be leveraged for other data endpoints | Easily extensible to all data endpoints with no extra work |
| Visibility and Assurance | Validating enforcement of policies is highly complex | Easy-to-consume activity logs which enable auditing to track every policy change |
| Enforce Context in Access Policies | Not possible to regulate access using context | Verify the user is who they claim to be before they access the data |

### Example data model


*current_user()* limits access to dealers and quotes tables
current_user()
Users
• User
• Company
Dealers
• Company
Quotes
• Dealer
• Instrument
• Price
*Quotes* table contains sensitive data

In the data model shown above, the company wants to restrict users so that a user can see only the instruments and corresponding prices from dealers at the companies that the logged-in user is mapped to. In essence, they want the query to have a user field restriction, as shown here:

```
select instrument, price, dealer
from quotes, dealers, users
where users.user = current_user()
and users.company = dealers.company
and dealers.dealer = quotes.dealer
```

The challenge is that, by dropping the above highlighted restriction, a query will return all data in the 3 tables that meet the join criteria, regardless of who is executing the query.

```
select instrument, price, dealer
from quotes, dealers, users
where users.company = dealers.company
and dealers.dealer = quotes.dealer
```

## Access Management with Native Controls

Using native controls, the company would have to create a view with the additional join clause and have all users reference this view instead of the existing tables:

```
create view user_quotes as
select instrument, price, dealer
from quotes, dealers, users
where users.user = current_user()
and users.company = dealers.company
and dealers.dealer = quotes.dealer
```

While the above view will ensure that the user constraint is always enforced, the company now has to manage which user has access to which views, using Snowflake's role-based access control mechanism based on explicit grants. Any underlying changes to the data model will require manually refreshing and updating all the views.

Additional challenges with this approach include:

- Any change in policy requires cumbersome view updates, which themselves could be error prone.
- There is no way to audit that all the changes were made.
- Multiple views might be needed to restrict/grant access to the proper columns. For example, if price should not be shown to a user then another view must be created excluding the price column.
- The views themselves are read only and updates and modifications are cumbersome.

For example, the following DML might be issued by a user, and there would be no way to monitor the statement or control it in terms of allowing or disallowing it.

```
update quotes set price = price/10.0
where dealer in (select dealer from dealers
where company = 'acme');
```

## Access Management with Cyral

With a Cyral policy, it's simple to ensure that the user restriction applies to every query. This can be accomplished creating a simple policy, expressed in the popular YAML format, as shown here:

```
data:
 - INSTRUMENT
 - PRICE

rules:
 reads:
   - data: any
     rows: any
    additionalChecks: |
      is_valid_request {
        filter.attr == "users.user"
        filter.op == "eq"
        filter.value == current_user
   }
```

The above policy applies to every query that attempts to read from the instrument or price columns in the quotes table.

But what if someone runs a query that does not include a user constraint in the WHERE clause, like the original query, shown here?

```
select instrument, price, dealer
from quotes, dealers, users
where users.company = dealers.company
and dealers.dealer = quotes.dealer
```

This query will be blocked by Cyral because it does not include the mandatory user column from the users table.

The same Cyral blocking behavior will also apply to queries on a single table:

```
select instrument, price
from quotes
```

Because instrument and price are sensitive (that is, your policies have told Cyral they're columns worth protecting), Cyral requires the user column from the user's table to be present. The only way this could be present here is through a three-way join of the quotes, dealers, and users tables.

Cyral's policy behavior can apply to current_user(), current_role(), and current_account() functions in Snowflake. These are the same functions that Snowflake supports for views to prevent accidental data sharing.

Additionally, it is possible to specify rich, granular context on how the policies may be enforced. For example, policy rules may be applied based on which Snowflake user is executing a request as well as the request type (a SELECT, or a data-altering DML statement such as an UPDATE or a DELETE).

Finally, updating Cyral policies is simple. If you decide that dealer is a sensitive column, then all you do is add dealer to your policy, and the dealer column is protected by Cyral. If you were to use views, you would have to drop and recreate all views that reference the dealer column.

Based on the updated policy shown to the right, any query that references instrument, price, or dealer will require the users table and be limited to return only rows whose user matches the current user who issued the query.

```
data:
- INSTRUMENT
- PRICE
- DEALER

rules:
 reads:
   - data: any
   rows: any
   additionalChecks: |
    is_valid_request {
      filter.attr == “users.user”
      filter.op == “eq”
      filter.value == current_user
 }
```

As we alluded to above, a policy in Cyral can contain a rich representation of the types of DML operations that are allowed, and who's permitted to perform each type. Below, we show an example policy that specifies the following rules:

- If a Snowflake user belongs to the “admin” role, both SELECTs and DMLs allowed with the additional restriction that DMLs are allowed to affect only one row at a time.
- For all other regular Snowflake users, only SELECT statements are allowed while DMLs such as UPDATEs and DELETEs are disallowed.

```
data:
 - INSTRUMENT
 - PRICE

rules:
identities:
     roles: [“admin”]
reads:
   data: any
   rows: any
     updates:
            data: any
            rows: 1
        deletes:
           data: any
           rows: 1

    reads:
       - data: any
           rows: any
           additionalChecks: |
              is_valid_request {
filter.attr == “users.user”
                filter.op == “eq”
                filter.value == current_user
 }
```

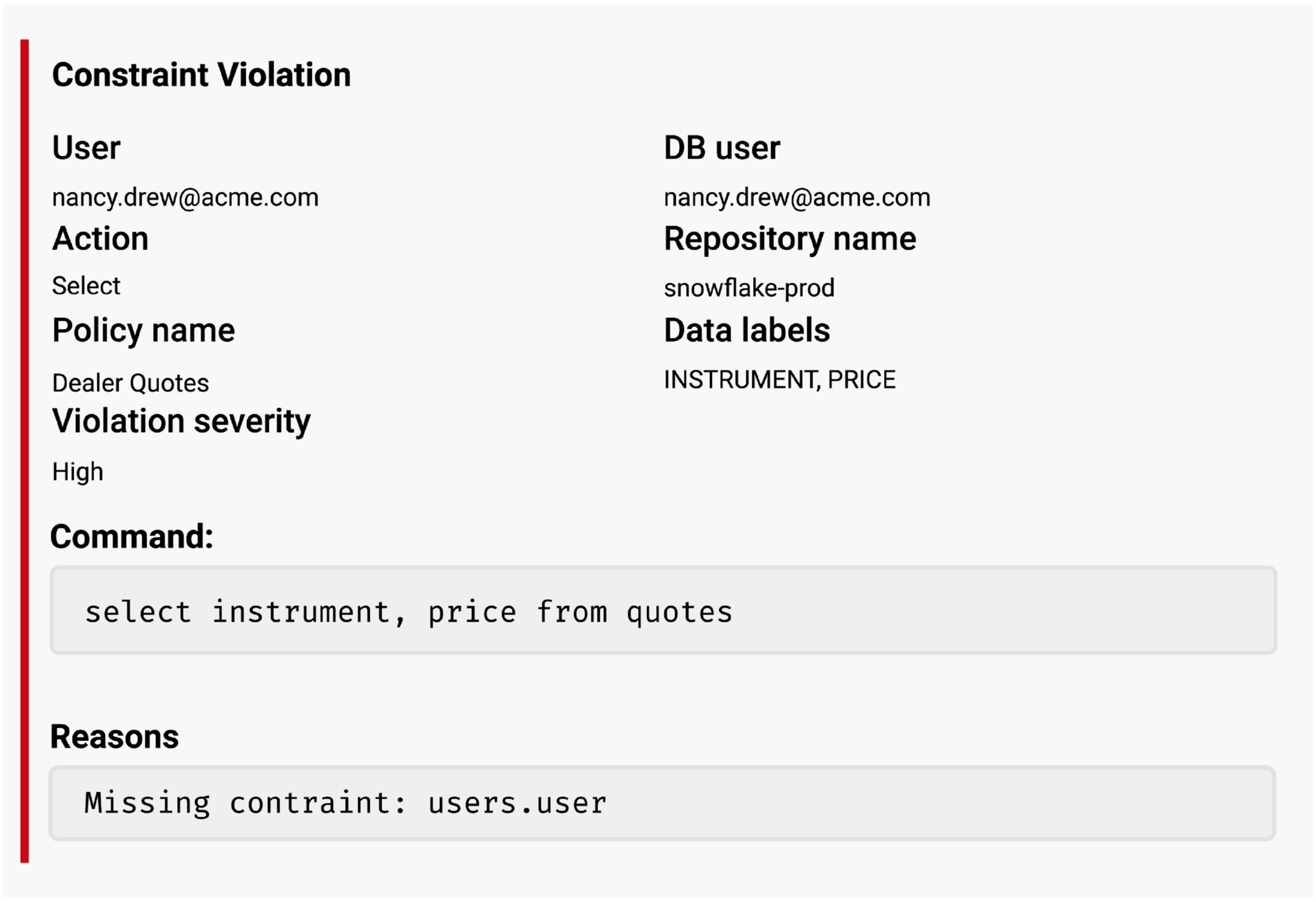
In addition, Cyral can help generate detailed activity logs and metrics that can be used for assurance and troubleshooting.

## Enriched Data for Auditing and Forensics

With Cyral, the logs contain the SQL statement that was attempted and the reason why the statement was blocked. In the following example, the query was blocked because the policy requires that any attempt to access the instrument or price columns must include a constraint on the user column in the users table.

```
{
 “rulesViolated”: [
  {
   “label”: [“INSTRUMENT”, “PRICE”],
   “policyName”: “Dealer Quotes”,
   “reason”: [“Missing constraint: users.user”],
   “severity”: “high”
 }
 ],
 “policyViolated”: true,
 “endUser”: “nancy.drew@acme.com”,
 “repo”: {
   “name”: “snowflake-prod”,
   “host”: “fgs8393.snowflakecomputing.com”
 },
 “client”: {
  “host”: “192.168.96.1”,
  “port”: 37526
 },
 “request”: {
   “timestamp”: “2020-10-28 16:19:10.5587894 +0000 UTC”,
   “statement”: “select instrument, price from quotes”,
   “statementType”: “SELECT”,
   “tablesReferenced”: [
   “public.quotes “
  ],
   “columnsReferenced”: {
   “public.quotes”: [
   “instrument”,
   “price”
  ]
},
   “labelsReferenced”: [
   “INSTRUMENT”,
   “PRICE”
  ],
 },
 “response”: {
   “status”: “blocked”
 }
}
```

In addition to blocking, you can receive an alert via Microsoft Teams that the inappropriate query occurred and was blocked by the Dealer Quotes policy.



**Constraint Violation**

| | |
|---|---|
| **User** | **DB user** |
| nancy.drew@acme.com | nancy.drew@acme.com |
| **Action** | **Repository name** |
| Select | snowflake-prod |
| **Policy name** | **Data labels** |
| Dealer Quotes | INSTRUMENT, PRICE |
| **Violation severity** | |
| High | |

**Command:**

```
select instrument, price from quotes
```

**Reasons**

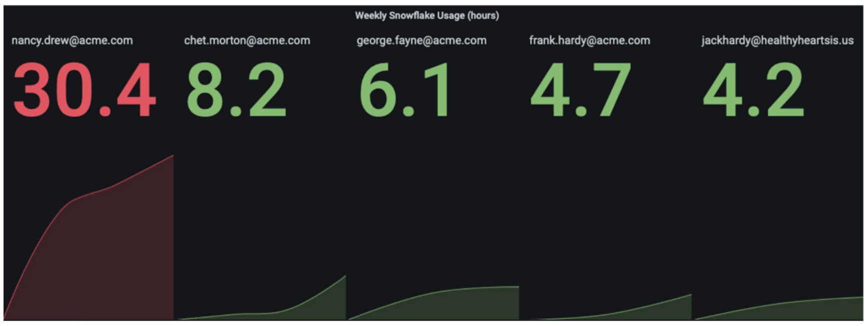
```
Missing contraint: users.user
```

# Metrics Reporting for Performance Debugging and Troubleshooting

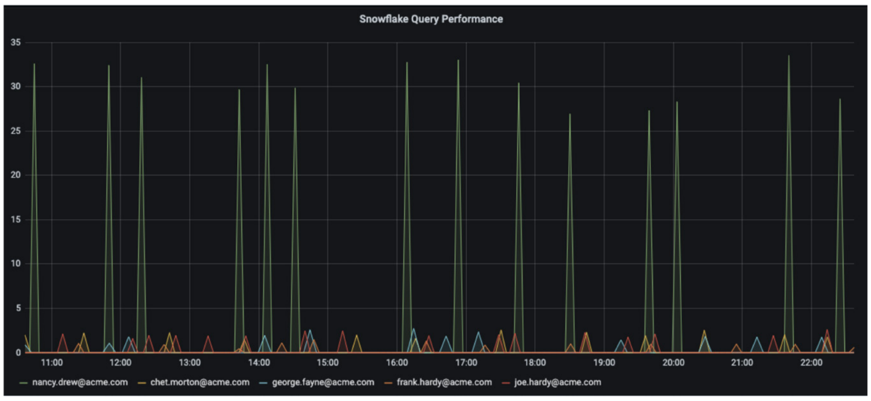
In addition to data activity monitoring for securing and controlling access to sensitive data, Cyral collects Snowflake query execution time metrics that are useful for monitoring service usage from a billing perspective. In any given billing period, Snowflake service usage tracks closely with the cumulative execution time of the queries.

Cyral can group the cumulative execution times by the individual Snowflake users and track their usage on a daily/weekly basis. This allows account and billing admins to monitor and optimize service costs for their Snowflake account.

For example, the chart below shows Snowflake usage over a time period for the five users contributing to the most usage, highlighting their total usage in the current cycle.



In this specific example, Nancy Drew (nancy.drew@acme.com) seems to have a disproportionate amount of Snowflake usage attributed to her, compared with the rest of the highly active users. Based on this, an admin can conclude either that Nancy is working substantially more than her colleagues, or that she's executing unusually longrunning queries. Digging deeper, the detailed metrics show the query execution times for the same five users.



The drill down shows that Nancy consistently runs queries which take an order of magnitude longer to execute, compared with those of her peers. This visibility allows an admin to check whether Nancy's queries are all in fact necessary, or whether they should be optimized in order to run faster to bring down the overall service costs.

# Learn More

As outlined at the beginning of this document, there are several other capabilities that Cyral provides in addition to better access management. This helps with scenarios such as the following:

- Ensuring that the same access policies, monitoring, and threat detection are enforced, even when a user connects to Snowflake via a BI tool like Looker or Tableau, and even when the BI tool relies on a shared service role to access Snowflake.
- Enforcing that users may read certain data only when they are logging from a specific country or putting limits on how may rows of data may be read.
- Implementing a simple break-glass procedure for giving users access to privileged roles to Snowflake (for example, admin roles with delete permissions, and so on).
- Automatically alerting the security team when threats are detected, such as a password spraying attack or a user running a full table scan.
- Implementing field-level encryption for certain sensitive data, using the customer's own keys.

## About Cyral

Cyral delivers enterprise data security and governance across all data services such as S3, Snowflake, Kafka, MongoDB, Oracle and more. The cloud-native service is built on a stateless interception technology that monitors all data endpoint activity in real-time and enables unified visibility, identity federation and granular access controls. Cyral automates workflows and enables collaboration between DevOps and Security teams to automate assurance and prevent data leakage. Cyral is venture-backed by Redpoint, A.Capital, Costanoa and SVCI. Follow the company on Twitter at @CyralInc.

cyral.com/tech-talk