A Cross-Database Examination of Next Generation Access Control Mechanisms: MySQL vs. PostgreSQL

Abstract

One important aspect of modern-day large-scale organizations is to provide users with controlled data. Researchers over the years have come up with different access control policies for this purpose. These policies limit users' capabilities to perform different operations. This is done using access control frameworks. Access control has been an ongoing challenge for decades. And different researchers have come up with solutions considering different aspects of challenges. Sometimes, newer models have been proposed to overcome the limitations of previous models. Other times, more complicated models have been introduced to face new challenges posed by the changes in organizational needs or in the need for different sets of capabilities from an administration perspective. It is of utmost importance to ensure security. But often, it comes at the cost of reduced performance. This paper focuses on a specific access control mechanism called Next Generation Access Control (NGAC). NGAC performs better than other access control mechanisms and offers much more flexibility but its implementation with MySQL has some performance issues when used at a very large scale. PostgreSQL, due to how it implements access control internally, performs faster and also offers more flexibility. This paper provides a comparative view of implementing NGAC with MySQL and with PostgreSQL.

1 Introduction

Among different access control techniques [1] Role Based Access Control (RBAC) and Attribute Based Access Control(ABAC) are the prominent ones. Whereas Role Based Access Control (RBAC) uses users' roles in an organization to restrict access, Attribute Based Access Control(ABAC) uses attributes, and environment conditions along with a set of policies for access control. RBAC can become less useful when the number of roles is extremely high. Big companies come with thousands of roles and there might be occasional changes in roles. These issues make RBAC less ideal option. On the other hand, ABAC struggles when it needs to process a huge number of policies. Next Generation Access Control (NGAC) is an attribute-based access control scheme developed by NIST(National Institute of Standards and Technology) [2] that tries to deal with this limitation. It draws decisions regarding access controls by modeling the access decision data as a graph. Some of its advantages are - its fine-grained access control with an increased level of granularity, its centralized policy and maintenance, central access control policy enforcement, and no access control requirements at other levels.

NGAC too suffers from some problems. SQL databases mostly do not have the level of granularity in access control as required in NGAC. Though some vendors developed average solutions, they still lack a global solution for this fine-grained access control. As part of the solution to this problem, authors have proposed that access decisions responsibility can be delegated to the NGAC policy functions. In this solution, NGAC stays on top of the local access control and takes all access decisions. All the query requests from database users are made on views instead of real databases. Only af-

ter ensuring that this query is compliant with NGAC policy, results are shown to the user. NGAC needs a strong database management system that has very good horizontal scalability, support for a wide range of data types, and very good concurrency control and partitioning features to achieve good performance when applied at a large scale. PostgreSQL has better support in all these aspects in comparison to MySQL.

2 Related Work

Access control in MySQL is achieved by using triggers, views, or through query augmentation. To accommodate complex policies, MySQL uses policy tables. Policy tables are not special tables. They are just like other tables storing policies in a structured way. Table 1 shows a sample policy table.

Access control on insert, update, or delete operations can be achieved by using triggers where the trigger body checks whether the action is allowed by the policies stored in the policy table. Or, we have to ensure that a query has been augmented properly before being executed on the system. For example, if a user tries to execute the following query,

SELECT * from patient;

It will be augmented to

SELECT * from patient WHERE
1 <= (SELECT check_access ('patients', 'write', 'patient'));

Here, check_access is a user_defined function that takes resource name, action type, and role as parameters and returns how many policies allow the action to occur for that particular role and resource. So, the query will be executed if there is at least one policy allowing the query. This ensures row-level security in MySQL. To ensure column-level security, MySQL uses views. For example, if a particular user group has access to only three out of five columns for a table, a view can be created consisting of only those three columns, and all the queries for that user will be made on the view instead of the original table.

PostgreSQL, on the other hand, has provisions for creating policies and applying restrictions on columns for different user groups. Access control in PostgreSQL will be discussed in detail in later sections.

NGAC has the potential to become a state-of-the-art access control mechanism. To better understand its importance, comparisons between RBAC and NGAC and between XACML(another attribute-based access control) and NGAC have been provided below:

3 RBAC vs. NGAC

Here is a view of the limitations of RBAC that NGAC has successfully overcome:

- **Fined-Grained Access Control Based On Contextual Attributes:** RBAC depends on predefined roles and permissions. It might not always capture all the factors necessary for access control. NGAC uses attributes associated with the subject, object, and environment to make context-aware access decisions. For example, We can have a policy like below:

  create policy test_policy on
  patient_diagnosis to test_user
  using
  (diagnosis in ('flu','pox')
  and EXTRACT(HOUR FROM
  CURRENT_TIMESTAMP) > 18
  and current_user = 'test_user');

  Here, diagnosis is an object attribute, current_timestamp is an environment attribute and current_user is a subject attribute. Thus NGAC achieves fine-grained access control.

- **Dynamic Access Control:** RBAC policies are static and they need manual intervention for updates. It limits their ability to change in real time. NGAC's dynamic policies can evaluate attributes

Table 1: Policy Table

Role	Resource	Action	Condition
Patients	Patients_table	Read	Current_user = patient_name
Patients	Patients_diagnosis	Read	Current_user = patient_name
Doctors	Patients_table	Read	True
Doctors	Patients_diagnosis	Read	True

and adapt access permissions based on current conditions or events.

- **Integration With External Authorization Services And Policy Management Framework:** RBAC may require custom integrations with external systems which increases complexity. That is due to RBAC's feature of assigning pre-defined roles to users to handle authorization needs. NGAC, on the other hand, utilizes a decentralized approach. This means that authorization policies are not centrally defined but rather distributed across different systems and applications. This decentralized nature makes integrating with external authorization services and policy management frameworks significantly easier.

- **Policy Administration And Delegation:** RBAC's centralized policy management can limit flexibility in assigning access control responsibilities. NGAC's policy management framework allows for the delegation of policy administration and access control tasks. In NGAC, a superuser is capable of creating administrators and delegating administrative capabilities to them. [3]

4 XACML vs. NGAC

Extensible Access Control Markup Language or XACML and NGAC are both attribute-based access control techniques but they differ in their approaches. Key differences between these two approaches are given below:

- **Operational Efficiency:** NGAC is inherently more efficient.[3] This is due to NGAC's way of handling requests. NGAC combines multiple policies at once to decide on a particular privilege. XACML, on the other hand first finds out the trusted and untrusted policies and rules to decide by a search for targets. If these policies prove to be insufficient to resolve conflict, it searches for more attributes. The whole process requires at least two data stores.

- **Policy Management:** Data access control is crucial, particularly when multiple organizations share data. NGAC outperforms XACML in managing how access rules and attributes are created and modified. NGAC boasts a standardized approach for managing access control data, unlike XACML. NGAC also enables finer-grained control over who can create and modify these data elements. This methodical approach ensures policy preservation, starting with a single administrator and an empty data set, ultimately empowering users with data service, policy, and attribute management capabilities. NGAC grants administrative power down to the level of individual configuration elements, while also allowing for the denial of such power at the same level of granularity.[3]

- **Administrative Review and Discovery of resources:** Efficient review of user capabilities and object access control entries is a key feature of access control systems. NGAC excels in both per-user and per-object review, despite being slightly less efficient than pure ACL or RBAC mechanisms in their respective areas. This tradeoff stems from NGAC's

ability to handle multi-policy environments, a feature absent in RBAC and ACL(access control list), and inefficiently handled by rule-based mechanisms like XACML. Notably, XACML requires testing all possible decision outcomes to determine authorization status, a significant inefficiency compared to NGAC's approach.[3]

In summary, both NGAC and XACML offer advancements in attribute-based access control, providing more granular and context-aware access control decisions. However, NGAC excels in its decentralized approach and policy execution framework and its management compared to XACML makes it superior.

5 Solution

5.1 Objective

The objective of this project is to first explore the implementation of attribute-based access control in PostgreSQL and find out how PostgreSQL performs when applied at a very large scale.

5.2 Methodology

Attribute-based access control in PostgreSQL involves three tasks - creating policies, ensuring row-level security, and ensuring column-level security. Row-level security is needed when users are restricted from accessing certain rows. Policies can be attached to row-level security. On the other hand, column-level security is needed when users are restricted from accessing certain fields/columns. For creating policy, we use the following command,

CREATE POLICY name ON table-name
[AS{ PERMISSIVE | RESTRICTIVE }]
[FOR {ALL | SELECT | INSERT |
UPDATE | DELETE }]
[TO {role name | PUBLIC | CURRENT ROLE | CURRENT USER |
SESSION USER } [, ...]]
[USING (using expression)]
[WITH CHECK (check expression)]

Policies can be made permissive or restrictive. Permissive policies applicable to a table are combined using the logical OR operation while restrictive policies applicable to a table are combined using logical AND operation. By default all policies are permissive. Policies can be attached to one or more commands like SELECT, UPDATE, or all commands. By default, policies apply to all commands. Moreover, we can attach policies to specific roles. By default, policies are public meaning they apply to everyone. In place of using expression and check expression, we can plug in SQL queries that evaluates to true or false. Using expression or check expression can not contain any aggregate or window functions and row-level security must be enabled on the table for applying policies.

To apply row-level policy on a table, the following command is used

ALTER TABLE table name ENABLE ROW LEVEL SECURITY

When enabling row-level security we must attach a policy to it. When row-level security is applied to a table all queries for this table must be allowed by the policies. Otherwise, the query is ignored. So, for example, if a system has three different categories of users and a policy has been created for only one category of users, all queries from other categories of users will simply be ignored. While creating enabling row-level security, we need to keep a few things in mind. First, it can be bypassed by a superuser or a role with a BYPASSRLS attribute or table owners who haven't opted for the RLS forcefully. Second, referential integrity checks, such as unique or primary key constraints and foreign key references, always bypass row security to ensure that data integrity is maintained. So, even if policies do restrict users from doing something on the database, users will be restricted by referential integrity if the action violates the referential integrity. Finally, row-level security should not be applied when taking backups because it might restrict some

records from being backed up.

Column level security on the other hand can be implemented in two ways. First, by enforcing access to selected columns of a table with direct access to the table. The following query can be used for this purpose,

GRANT command-list (column list) ON table-name TO user

Here 'command list' is a placeholder for select, update, and delete commands. We can specify more than one command. In place of the 'column list', we specify the columns that we will allow a specific user to access. Finally, the 'user' in this command will be replaced by a particular user group. Or, we can implement column-level security using views to limit access to selected columns of a table upon eliminating direct access to the table.

For demonstration purposes, let's try to implement the table 2 with the following policies:

1. An admin has full access to the PATIENT table. Admin can READ, UPDATE, INSERT, or DELETE records of the PATIENT table. But s/he can only read the ID and doctor-assigned field in the PATIENT DIAGNOSIS table.

2. A doctor has full access to the PAITENT DIAGNOSIS table. But s/he can only read the id and name field in the PATIENT table.

3. A patient can only read his/her records.

Table 2: Hospital Policy Table

User	Permissions	Table	Columns	Data
Mary (Admin)	Read, Update, Insert, Delete	PATIENT	id, name, birth_date, ssn	P1, P2, P3
	read	PATIENT DIAGNOSIS	id, doctor_assigned	P1, P2, P3
	No permissions	PATIENT DIAGNOSIS	diagnosis_date, diagnosis	P1, P2, P3
Bob (Doctor)	Read	PATIENT	id, name	P1, P2, P3
	Read, Update, Insert, Delete	PATIENT DIAGNOSIS	id, diagnosis_date, diagnosis, doctor_assigned	P1, P2, P3
	No permissions	PATIENT	birth_date, ssn	P1, P2, P3
P1 (Patient)	Read	PATIENT	id, name, birth_date, ssn	P1
	Read	PATIENT IAGNOSIS	id, diagnosis_date, diagnosis, doctor_assigned	P1
P2 (Patient)	Read	PATIENT	id, name, birth_date, ssn	P2
	Read	PATIENT DIAGNOSIS	id, diagnosis_date, diagnosis, doctor_assigned	P2
P3 (Patient)	Read	PATIENT	id, name, birth_date, ssn	P3
	Read	PATIENT DIAGNOSIS	id, diagnosis_date, diagnosis, doctor_assigned	P3

5.3 Implementation

After creating the tables and inserting a few records in both tables, we need to create the three categories of users and assign appropriate roles to them. Then for creating policies and row-level access control, the following commands have been used.

alter table patient
enable row level security;

alter table patient_diagnosis
enable row level security;

create policy patients_policy on patient
for select to patients using
(current_user = name)

create policy doctors_policy on patient
for select to doctors using(true);

create policy admin_policy on patient
for all to admins using(true) with check (true);

create policy others_policy on patient_diagnosis to doctors, admins
using (true) with check(true);

create policy patients_policy on patient_diagnosis to patients using (EXISTS (SELECT 1 FROM patient WHERE patient.id = patient_diagnosis.id AND patient.name = current_user));

For column level security the following commands have been used.

grant all on patient_diagnosis
to doctors;

grant select (id, doctor_assigned)
on patient_diagnosis to admins;

grant select on patient_diagnosis
to patients;

grant select on patient to patients;

grant all on patient to admins;

grant select (id, name)on patient
to doctors;

5.4 Results

The commands mentioned in sections 5.2 and 5.3 successfully restrict patients from reading other patients' information. Patients can only view their records. Doctors on the other hand can view only the specified columns and have full access to the PATIENT_DIAGNOSIS table. Admins on the other hand have full access to the PATIENT table but can only access specified columns on the PATIENT_DIAGNOSIS table. Both the patient and patient_diagnosis had 1 million rows and there were 100 policies(including redundant policies). Read operations that resulted in 1 million rows took around 600 milliseconds in PostgreSQL and the same query(augmented properly) in MySQL took around 62.5 seconds. Insertion operations were faster both in PostgreSQL and MySQL. In PostgreSQL, insertion took around 60 microseconds whereas in MySQL it took almost 1700 microseconds. PostgreSQL has also tremendous support for other important concerns like error reporting and logging, summarizing a user's privileges on database objects, identifying redundant policies, etc. All these features reinforce PostgreSQL's eligibility to be used for NGAC.

6 Error reporting and logging

Error reporting and logging are important aspects of any access control system. It helps us identify any bugs or security vulnerabilities. It can work as a debugging tool. It also helps us track application performance and provides an audit trail for tracking and verifying system activities. Logs can be used not only as historical data but also to find security breaches. PostgreSQL has an open-source extension called

PGAudit that comes with auditing and monitoring capabilities. Its key features include:

- **SQL Statement Logging:** PGAudit can log SQL statements such as SELECT, INSERT, DELETE, and many more. This is important for monitoring purposes. It specifies which classes of statements will be logged by session audit logging. The possible values are: Read, Write, Function, Role, DDL, MISC, MISC_SET, and all. We can run the following command using psql.

  set pgaudit.log = 'write, ddl';
  to specify a desired value.

- **Audit Event Filtering:** It is possible to configure specific log events in PGAudit. It can reduce the amount of logs recorded while logging critical events. We can log connections, disconnections, checkpoints, application name, user name, session ID and many more. For example,

  log_checkpoints = on;
  log_connections = off;
  log_disconnections = off;
  log_duration = off;
  log_error_verbosity = default;
  log_hostname = off;
  log_line_prefix = '%t %u '

- **Output Formats:** PGAudit can return logs in CSV, JSON or syslog format. For example, we can use the command,

  log_destination = 'csvlog'

  Valid values are combinations of stderr, csvlog, jsonlog, syslog, and eventlog, depending on platform.

- **Granular Auditing:** We can specify which types of activities should be logged. Thus, it supports customized auditing. We can set the message security level. For example,

  pgaudit.log_level = notice

  It can be set to debug5, debug4, debug3, debug2, debug1, info, warning, error, log, fatal, and panic. Each security level has different implications.

7 Summarizing User's privileges on database Objects

Analyzing user's access to different database objects can help identify potential security vulnerabilities. It is important for ensuring compliance requirements and hiding sensitive information. It can assist not only in resource optimization but also in performance management decisions. Overall, access control and data ownership can be ensured by properly analyzing user's access to different databases.

7.1 The information_schema

The information_schema is a schema itself. It exists in all databases by default. Initial database owner owns this and can drop this. As, it takes very less space, it is usually kept. It consists of a set of views. These views contain information about current database objects. This schema contains privilege information on different database objects. To know the names of this table we use the following query,

SELECT table_name FROM information_schema.tables WHERE table_schema = 'information_schema' AND table_name LIKE '%_privileges';

It returns all the table names containing privilege information for a database. By querying on these table we can find out the tables along with privilege type for a particular user. For example, to know doctors privileges in table 2, we can use the following query,

select table_name as object_name, grantee, privilege_type from information_schema.table_privileges where grantee = 'doctors'

union

```
select table_name, grantee,
privilege_type from
information_schema.column_privileges
where grantee = 'doctors'
union

select routine_name, grantee, privilege_type
from information_schema.routine_privileges
where grantee = 'doctors'
union

select udt_name, grantee, privilege_type
from information_schema.udt_privileges
where grantee = 'doctors'
union

select object_name, grantee, privilege_type
from information_schema.usage_privileges
where grantee = 'doctors';
```

This will return the database objects, user, and privilege types. But it does not return details on the column advantages. So, to know which particular columns of which objects a user has access to, we can use the following command,

```
select grantee, table_name,
column_name, privilege_type from
information_schema.column_privileges
where grantee = 'doctors';
```

This will return the user and table names with specific column names and privilege types for that user.

7.2 Redundant Policy Identification

Redundant policies refer to policies that do not contribute to any changes to user access to database objects. It is important to identify redundant policies for the following reasons,

- **Improved query performance:** When a query is processed it is checked against each existing policy. So, removing redundant policies will help the number of policy checks that can significantly improve query performance.
- **Simplified database administration:** It becomes easier to track which policies are giving access to which database objects and overall management of the database becomes simpler.

We can use views along with some queries mentioned above to determine whether a policy is redundant or not. We have created **view 1** that contains the result of the query mentioned above that returns a list of tables and associated privilege types and **view 2** that contains the result of the query that returns a list of tables with associated column names and privileges on these columns. After creating a new policy (say P) we will create **view 3** that will use the same query as view 1. Finally, we will create **view 4** that will use the same query as view 2.

Now, we will check the number of records in all the views. If view 3 has a different number of records than view 1 or view 4 has a different number of records than view 2, then policy P is changing some user access. So, P is not redundant.

But if view 3 has the same number of records as view 1 and view 4 has the same number of records as view 2, then We will apply union on view 3 and view 1. We will also apply union on view 4 and view 2. If the first union returns the same number of records as view 3 and the second union returns the same number of records as view 4, then we can say policy P is redundant. Otherwise, policy P is not redundant.

8 Conclusion

PostgreSQL is more suitable to address the scalability issues of NGAC and it also has other important features that make it ideal for NGAC.

References

[1] M. May, C. Gunter, and I. Lee, "Privacy apis: access control techniques to analyze and verify legal privacy policies," in

19th IEEE Computer Security Foundations Workshop (CSFW'06), pp. 13 pp.–97, 2006.

[2] D. Ferraiolo, S. Gavrila, G. Katwala, and J. Roberts, "Imposing fine-grain next generation access control over database queries," in *Proceedings of the 2nd ACM Workshop on Attribute-Based Access Control*, pp. 9–15, 2017.

[3] D. Ferraiolo, R. Chandramouli, R. Kuhn, and V. Hu, "Extensible access control markup language (xacml) and next generation access control (ngac)," in *Proceedings of the 2016 ACM International Workshop on Attribute Based Access Control*, ABAC '16, (New York, NY, USA), p. 13–24, Association for Computing Machinery, 2016.