Benchmarking Role-Based Access Control in Data Management Systems

Mads Cornelius Hansen*, Pınar Tözün, and Martin Hentschel

IT University of Copenhagen, Copenhagen, Denmark
{coha,pito,mhent}@itu.dk

Abstract. Role-Based Access Control (RBAC) is a widely used method for controlling access to data in data management systems, offering a scalable approach to enforcing security policies. Despite its broad adoption, RBAC implementations vary significantly across different systems, leading to inconsistencies in performance. Currently, there is no standardized benchmark for evaluating RBAC performance. In this study, we propose a benchmark that focuses on two core components: the creation of role hierarchies with varying structures and the access to securable objects and role metadata through SQL queries. We evaluate three systems—PostgreSQL, MariaDB, and Snowflake—across on-premise and cloud deployments. Our findings reveal notable differences in RBAC performance, driven by I/O demand, caching behavior, and system architecture. These results highlight the diversity of RBAC implementations and the need for a systematic approach to evaluate RBAC at scale.

1 Introduction

Role-Based Access Control (RBAC) is a widely used method for controlling access to data in data management systems. By assigning privileges to roles instead of individuals, RBAC simplifies access management and provides a scalable mechanism for enforcing security policies. For example, in a corporate system, roles such as 'HR' or 'Management' can be defined with specific access rights to employee records. This ensures users access only the information necessary for their roles, thereby enhancing security and regulatory compliance. RBAC is an effective strategy for managing data access at scale and has been adopted in most modern data management systems.

Despite its broad adoption, RBAC is not implemented uniformly across data management systems. Each system tends to interpret and execute RBAC principles in slightly different ways, resulting in inconsistencies in functionality and performance. From our experience in the industry, customers with large RBAC setups may experience slow system behavior due to caching effects, for example. While the security of RBAC implementations has been extensively studied, research into how RBAC mechanisms affect system performance, such as latency

* Student at IT University of Copenhagen.

and resource utilization, is still lacking. Notably, there is no standardized benchmark for evaluating RBAC performance, especially under varying workloads, deployment configurations, and scales.

This study addresses these gaps by investigating the performance of RBAC across different data management systems. We examine how RBAC is implemented in three systems—PostgreSQL, MariaDB, and Snowflake—propose a benchmark for evaluating RBAC performance, and use this benchmark to conduct a performance assessment. These systems were selected for their widespread use and because they represent a diverse range of architectural differences: open-source vs. commercial, transactional vs. analytical, and on-premise vs. cloud-native. One of the goals of this study is to understand how these architectural differences influence the implementation and performance of RBAC.

Our results reveal a range of behaviors: PostgreSQL performs well locally but suffers in the cloud, MariaDB degrades at scale, and Snowflake exhibits constant yet high latency. These findings indicate that (a) there is no standard for RBAC implementation, as each system shows different performance characteristics; (b) there is a need for an accepted performance benchmark for RBAC; and (c) although RBAC metadata handling is not a primary performance focus of data management systems, all tested systems show potential for performance optimization, for example, improving I/O behavior, caches, and network traffic.

Overall, this study highlights the importance of understanding RBAC performance across different data management systems, particularly at scale. By proposing a first step towards an accepted benchmark and evaluating multiple systems, we provide insights into how RBAC implementations can vary significantly, affecting both local and cloud-based deployments. Our findings show the need for standardized benchmarks in the industry to facilitate comparisons and optimizations of RBAC performance.

The paper is organized as follows: Section 2 provides background on RBAC and the data management systems evaluated in this study. Section 3 reviews related work. Section 4 describes our benchmark proposal for evaluating RBAC performance. Section 5 presents the experimental setup and results of the performance evaluation. Finally, Section 6 concludes the paper and provides an outlook on future work.

2 Background

In this section, we provide background on role-based access control and the data management systems evaluated in this study.

2.1 Role-Based Access Control

Role-Based Access Control (RBAC) was introduced as a model for access control in data management systems in the 1990s [9,16] and proposed as a NIST standard in 2001 [10]. RBAC is more general than the two traditional access control models, Mandatory Access Control (MAC) and Discretionary Access Control

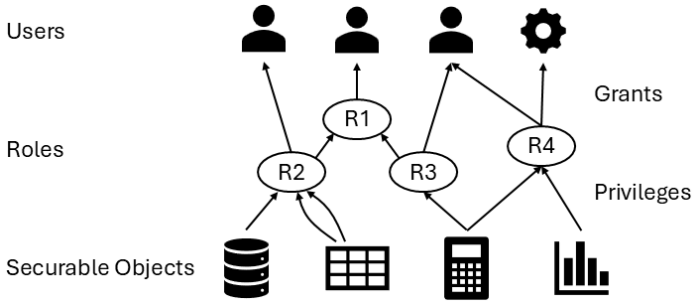


Fig. 1: The RBAC model consists of users, roles, privileges, and securable objects. Users can represent individuals or applications. Privileges are granted from securable objects to roles, and roles are granted to users or other roles.

(DAC). In MAC, a central authority controls access. Users and documents are assigned security labels, and access is granted based on these labels. For example, in a military setting, documents might be labeled "Top Secret", "Secret", or "Confidential", and users with clearance levels matching these labels can access the documents. MAC is highly secure but can be complex to manage. DAC, on the other hand, allows users to control access to their own data. For example, DAC is used in Microsoft OneDrive or Google Docs, where users can choose who can view or edit their documents. DAC is flexible and easy to use but can be less secure if owners are not careful with privileges. RBAC combines the strengths of both models by assigning privileges to roles instead of individual users. This shifts access control from many user-specific decisions to a smaller number of role definitions, simplifying administration. At the same time, users only receive privileges relevant to their roles, which reduces the risk of accidental overexposure of sensitive data and thereby enhances security.

RBAC is defined by four key components: users, roles, privileges, and securable objects. *Users* are individuals or applications that access the data management system. *Roles* are abstract entities that hold a collection of privileges. *Privileges* define what actions can be performed on securable objects. *Securable objects* are the resources or data that users can access, such as databases, tables, functions, and views. In RBAC, privileges are granted from securable objects to roles, and roles are then granted to users. This means that users obtain access to resources indirectly through roles, rather than receiving individual privileges as in MAC and DAC.

Figure 1 illustrates the RBAC model. Privileges are granted from securable objects to roles, meaning roles obtain the rights to perform specific actions on those objects. Roles are then granted to users, which can represent either individuals or applications. A user may thus receive privileges indirectly through one or more roles. For example, read and write privileges on a database table might be granted to a role, and that role might then be granted to several users, allowing them to read from and write to that table.

Roles can also be granted to other roles, forming a role hierarchy. In Figure 1, Role R1 is granted roles R2 and R3, which means R1 inherits all privileges granted to R2 and R3. Role hierarchies simplify privilege management by allowing roles to be structured in a way that reflects organizational needs. From our experience, role hierarchies with depths of up to 100 and overall hierarchy sizes of up to 100,000 roles are not uncommon.

Lastly, sessions are a key aspect of RBAC, representing the active instances of users accessing the system. A session is created when a user logs in and can include multiple roles assigned to that user. Typically, the user is assigned a primary role and can enable secondary roles. The combination of the primary role and secondary roles defines the privileges available to the user during that session.

2.2 Evaluated Data Management Systems

In this study, we focus on the implementation of RBAC in three popular data management systems: PostgreSQL [17], MariaDB [13], and Snowflake [7]. PostgreSQL and MariaDB are open-source database management systems that represent the class of online transactional processing (OLTP) systems, designed to support interactive applications requiring low-latency query responses and high-throughput transactions. While OLTP systems are typically deployed on-premises, they can also be hosted in the cloud, for example, on Amazon Web Services, as analyzed in this study.

Snowflake, on the other hand, is a commercial, cloud-native data management system that represents the class of online analytical processing (OLAP) systems optimized for analytical workloads. OLAP systems are designed to handle large datasets, up to petabytes of data, at the cost of increased latency and reduced transactional throughput. Analytical data management systems like Snowflake are typically hosted exclusively in the cloud, with no on-premise deployment options.

As mentioned, the choice of these systems allows us to evaluate RBAC performance across a range of architectures and deployment models from open-source to commercial, transactional to analytical, and on-premise to cloud-native. For MariaDB and PostgreSQL, we conduct the study on both on-premise and cloud deployments, while Snowflake is evaluated exclusively in the cloud. Details on the RBAC implementations of these systems appear in Section 5.2.

For the purposes of this study, we focus on RBAC features that are common across all three systems, such as role-to-role grants to create role hierarchies and `SHOW` queries to retrieve RBAC metadata. We exclude features that are not universally supported, such as privileges unique to a single system (e.g., PostgreSQL's `SUPERUSER` privilege or Snowflake's `OWNERSHIP` privilege) as well as features that are not relevant to our performance evaluation (e.g., row-level security policies or column-masking policies).

3 Related Work

Benchmarks around role-based access control in database systems have mostly focused on security aspects, such as the confidentiality and safety of data [21], as well as authentication, privileges, and encryption [22]. In a vision paper for benchmarking shared databases, the performance of verifiable transactions is considered [8]. YCSB++ [15] is an extension of the Yahoo! Cloud Serving Benchmark (YCSB) [6] that evaluates, among other features, the performance of authorization via access control lists.

Benchmarks for database and data management systems primarily focus on the performance of *data* management. Examples include TPC-C [18] for OLTP workloads; TPC-H [19], TPC-DS [20], and the Star Schema Benchmark [14] for OLAP workloads; and CH-benCHmark [5] for hybrid OLTP and OLAP workloads. In addition, there have been benchmarking standardization efforts for specialized data management scenarios, such as graph analytics [11], end-to-end data pipelines in AI [4], data correlations and skew [3], robustness [12], and dynamic or unexpected workloads [2]. However, these benchmarks do not address the performance of managing *metadata*, and specifically, RBAC metadata.

To the best of our knowledge, there is currently no benchmark that evaluates the performance of RBAC mechanisms, including role hierarchy creation and role hierarchy access, in data management systems. This study aims to take a first step toward filling this gap.

4 Benchmark Design

Based on our industry experience, operating on data protected by RBAC with large role hierarchies can result in high response times. To evaluate this, we designed a benchmark that measures the performance of creating role hierarchies with varying structures, as well as accessing securable objects and role metadata through SQL commands such as `SELECT` and `SHOW`. The primary goal is to assess how the structure and size of the role hierarchy impact query execution and metadata retrieval. The benchmark consists of two main components: Role Hierarchy Creation and Role Hierarchy Access, which are described in detail below. We have open sourced the benchmark on Github.[1]

4.1 Role Hierarchy Creation

In this component of the benchmark, we measure the performance of creating role hierarchies with varying structures. In general, role hierarchies can be of any graph structure, typically without including cycles. In this benchmark, we focus on tree structures because these are the most common based on our experience. The tree structures we consider are of the following types: deep linear hierarchies, wide linear hierarchies, and balanced hierarchies.

[1] https://github.com/MaCoHa/Study_of_Role-Based_Access_Control_Mechanisms

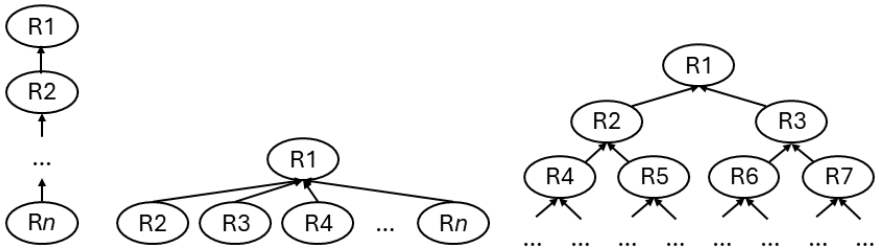


Fig. 2: Role hierarchy structures used in the benchmark: (a) deep linear, (b) wide linear, and (c) balanced.

Figure 2 illustrates the three types of hierarchies. *Deep linear hierarchies* (Figure 2a) consist of a single path of roles, where each role has exactly one parent and one child, resulting in a long chain of roles (e.g., R1 ← R2 ← R3 up to Rn, where n is the final number of roles). *Wide linear hierarchies* (Figure 2b) have a single root role with multiple child roles but no further descendants. *Balanced hierarchies* (Figure 2c) combine both deep and wide structures, where roles have a fixed number of children (e.g., always two children in Figure 2c) and can also be part of a deeper hierarchy.

We chose these structures for several reasons. Deep and wide linear hierarchies represent the two extremes of role hierarchy structures. With these structures, we aim to evaluate how well the systems handle both long chains of roles and wide role assignments. Balanced hierarchies represent a common structure in real-world applications. In one interesting example from the industry, a company has internal functionality where employees can "check out" projects consisting of databases and tables with sample data. Part of the check-out process is creating a subtree of roles that is added to the existing role hierarchy of the company. Over time, this leads to a role hierarchy that is large in size (up to 100,000 roles) and represents a wide hierarchy structure, however with subtrees of roles. It can be seen as a mix of a wide and a balanced hierarchy. The chosen structures in our benchmark therefore cover this scenario well.

The process for creating role hierarchies in this benchmark is as follows. Iteratively, the following commands are executed to create the role hierarchy:

1. Create a new role using the `CREATE ROLE` command.
2. Assign the new role to a parent role using the `GRANT` command, following the structure of the hierarchy (deep, wide, or balanced).
3. Repeat steps 1 and 2 until the desired number of roles is reached or until a time limit is reached.

For each command, we measure the execution time. After the time limit is reached, we also measure the total number of roles created.

4.2 Role Hierarchy Access

In the second component of the benchmark, we measure the performance of accessing securable objects and role metadata through SQL commands. The focus is on how the structure and size of the role hierarchy impact query execution and metadata retrieval. This component includes the following SQL operations: `SELECT * FROM table` and `SHOW ROLES`.

We selected these queries for several reasons. The `SELECT * FROM table` command is among the most common operations in data management systems. Measuring its performance allows us to assess how the size and structure of the role hierarchy affect authorization time when accessing a securable object. This query represents a broader class of "point access" operations, such as `ALTER`, `UPDATE`, `INSERT`, and `DELETE`, where a single securable object is accessed and authorization is performed.

The `SHOW ROLES` command is also widely used and is sometimes invoked automatically by third-party tools (e.g., graphical user interfaces or reporting systems). This command lists all roles within the role hierarchy, allowing us to evaluate how efficiently the metadata subsystem can retrieve role information.

The process for evaluating role hierarchy metadata access in this benchmark is as follows:

1. Create an empty table using the `CREATE TABLE` command.
2. Create a role hierarchy of size n using one of the three hierarchy structures, as defined in the Role Hierarchy Creation component.
3. Grant read access on the table to the last role (Role R_n) in the hierarchy using the `GRANT` command, and activate the root role (Role R_1) in the current session using either the `SET ROLE` or `USE ROLE` command.
4. Execute the `SELECT * FROM table` query to measure the performance of table access authorization.
5. Execute the `SHOW ROLES` command to measure the performance of role metadata retrieval.

For Steps 4 and 5, we record the execution time to evaluate system performance. These steps may be repeated multiple times after the role hierarchy has been created in Steps 1–3.

5 Experiments and Results

5.1 Experimental Setup

Hardware and Environment Configuration The experiments were conducted in an on-premise setup and in the cloud. For the on-premise setup, the experiments were conducted on a local machine, which was used both to execute the benchmark script and to host the MariaDB and PostgreSQL databases. The machine was a MSI GF65 Thin 10SER laptop running Microsoft Windows 11. It featured an Intel i7-10750H CPU at 2.60 GHz with 6 physical cores, supported

by 16 GB of DDR4 RAM and a 1 TB WDC PC SN730 NVMe SSD. GPU acceleration was not used in this study. The cache configuration included 64 KB of L1 cache and 256 KB of L2 cache per core, along with 12 MB of shared L3 cache.

For the cloud setup, we used the Amazon Relational Database Service (RDS) to deploy PostgreSQL and MariaDB. Both systems ran on Free Tier instances (db.t4g.micro), each configured with 2 vCPUs, 1 GiB of RAM, 2.085 Mbps of network throughput, and 20 GiB (gp2) of SSD storage. All instances were deployed in the AWS Oregon (us-west-2) region, with encryption disabled. The benchmark script was executed on an Amazon EC2 instance (t2.micro) in the same region, configured with 1 vCPU, 1 GiB of RAM, and 10 GiB of SSD storage, running Ubuntu Server 24.04 LTS.

Snowflake was evaluated using a cloud-native setup, with the experiments conducted on a Snowflake account in the AWS Oregon (us-west-2) region. The Snowflake instance was set up with a single virtual warehouse at the X-Small size, with all other settings left at their default values. To assess the impact of geographic proximity on performance, we conducted two sets of experiments: in one, the benchmark driver was run from the local laptop located in Europe (Denmark); in the other, the benchmark script was executed on the Amazon EC2 instance in the AWS Oregon region, co-located with the Snowflake deployment.

Snowflake is a natively distributed service that replicates data and metadata across multiple servers, resulting in multiple additional network round trips. In contrast, the single-node setups of PostgreSQL and MariaDB exclude replication, avoiding this networking overhead. Benchmarking a replicated setup of PostgreSQL and MariaDB is left for future work.

System Configuration The configurations of PostgreSQL, MariaDB, and Snowflake were as follows. For PostgreSQL, we used version 17.4 in the on-premise setup and version 17.2-R2 in the cloud setup (both on the free tier and paid tier instances of RDS). For MariaDB, we used version 11.4.4 in both on-premise and cloud setups.

For Snowflake, at the time of the experiments, we used release version 9.9[2]. Snowflake follows a weekly release cycle, with all customers automatically upgraded to the latest version. Therefore, repeating the experiments at a later date may yield different results due to system changes.

5.2 Experimental Results

Role Hierarchy Creation Performance In this experiment, we executed the Role Hierarchy Creation component of the benchmark, measuring the performance of creating role hierarchies with varying structures (deep, wide, and balanced). The balanced hierarchy structure consisted of four children per node. We evaluated this component across the on-premise and cloud setups of PostgreSQL and MariaDB, as well as Snowflake accessed from both Europe and

[2] https://docs.snowflake.com/en/release-notes/2025/9_09

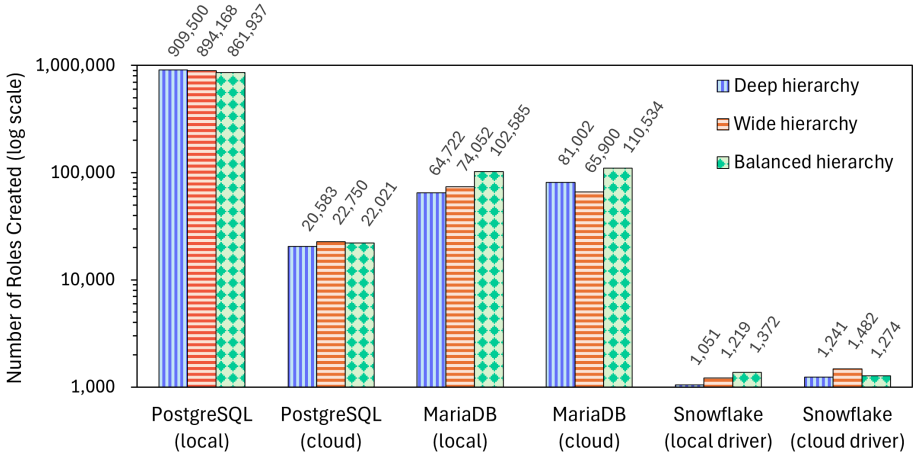


Fig. 3: Number of roles created within the 15-minute time limit.

an EC2 instance in the same AWS region. The benchmark was run with a 15-minute time limit, during which the benchmark script continuously executed `CREATE ROLE` and `GRANT` statements. We measured the execution time of each individual statement and the total number of roles created within the time limit. The experiment was repeated twice, and we report the average number of roles created across both runs.

Findings. Figure 3 shows the number of roles created within the 15-minute time limit for each system and hierarchy structure. Note the logarithmic scale of the y-axis. We observe the following findings: In the on-premise setup, PostgreSQL performed extremely well, creating hierarchies with around 900,000 roles within 15 minutes. However, in the cloud setup, PostgreSQL performed worse, creating hierarchies with only around 22,000 roles. MariaDB performed equally well in both the on-premise and cloud setups, creating around 65,000 to 110,000 roles within the time limit. Interestingly, MariaDB was able to create almost 50% more roles for the balanced hierarchy structure compared to the deep and wide structures. Snowflake was able to create around 1,000 to 1,500 roles within the time limit, regardless of the hierarchy structure or whether it was accessed from Europe or the EC2 instance in the AWS Oregon region.

Figure 4 shows the individual execution times of the `CREATE ROLE` and `GRANT` statements for PostgreSQL across all hierarchy structures. We show the execution times of the second benchmark run only. The x-axis represents the experiment time in minutes, up to the 15-minute time limit. The y-axis shows the measured latency in milliseconds. We observe the following findings: In the on-premise setup, execution times for all statements remain steady at around 0.22 milliseconds, even as the role hierarchy grows over time. The structure and size of the role hierarchy do not affect the execution time of the `CREATE ROLE` and `GRANT`

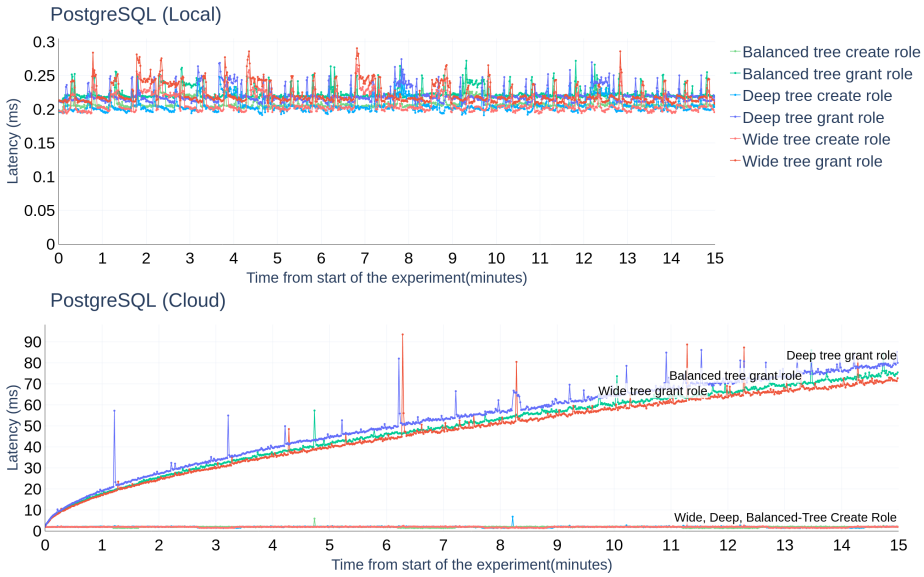


Fig. 4: Latency of `CREATE ROLE` and `GRANT` statements in PostgreSQL for all role hierarchies in the on-premise setup (top) and cloud setup (bottom).

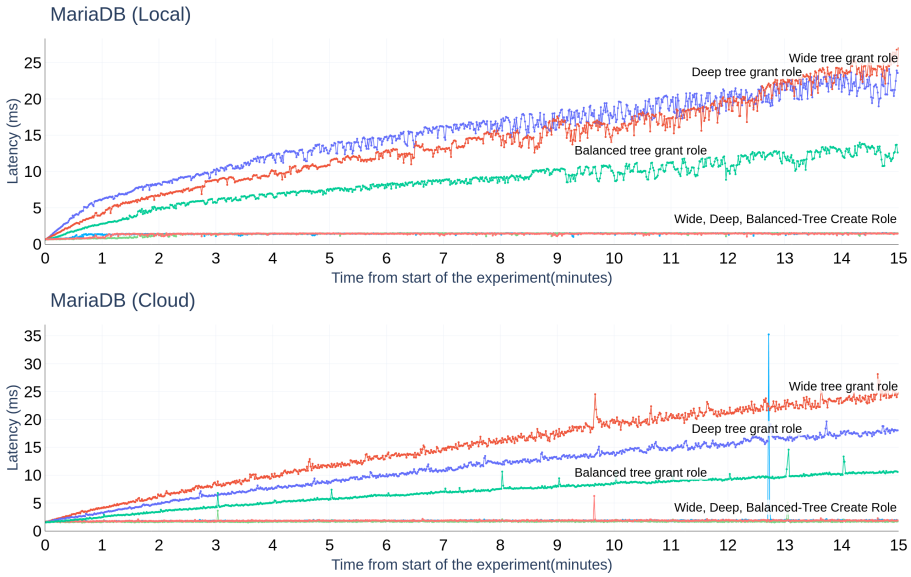


Fig. 5: Latency of `CREATE ROLE` and `GRANT` statements in MariaDB for all role hierarchies in the on-premise setup (top) and cloud setup (bottom).

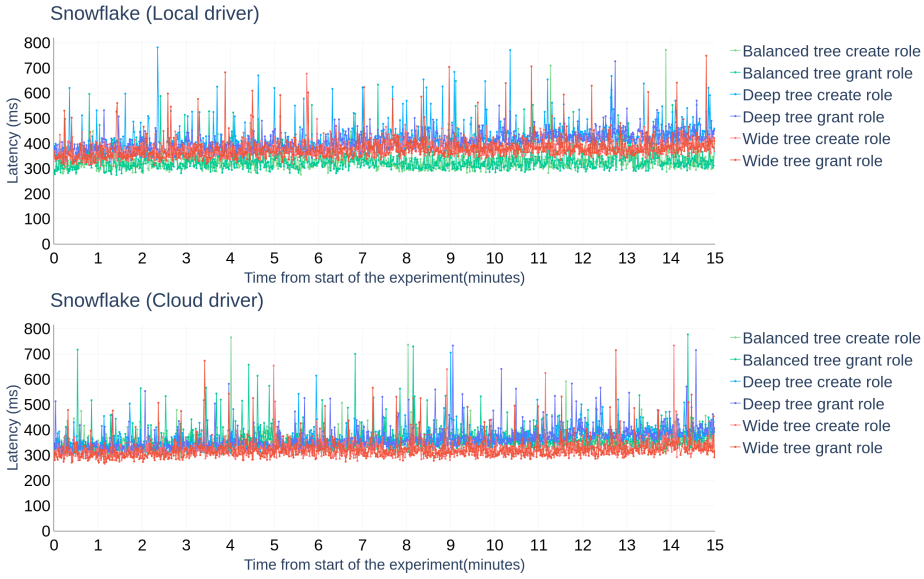


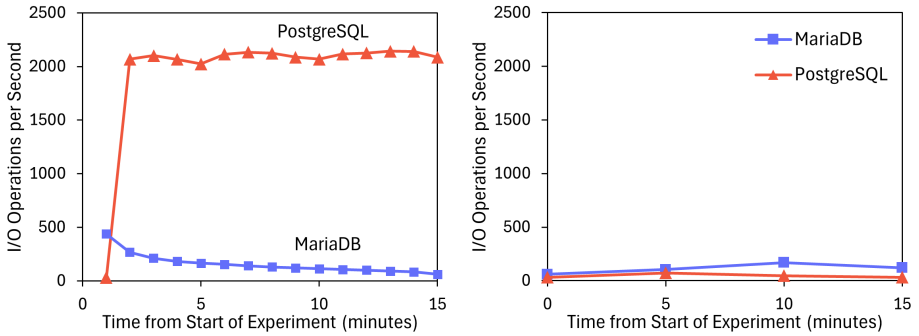
Fig. 6: Latency of `CREATE ROLE` and `GRANT` statements in Snowflake for all role hierarchies in the local-driver setup (top) and cloud-driver setup (bottom).

statements. In the cloud setup, however, the execution time of `GRANT` statements increases significantly over time (up to 80 milliseconds per statement), while the execution time of `CREATE ROLE` statements remains constant.

Similar to Figure 4, Figure 5 shows the individual execution times of the `CREATE ROLE` and `GRANT` statements for MariaDB across all hierarchy structures. We observe the following findings: The execution times of `GRANT` statements increase over time, while the execution times of `CREATE ROLE` statements remain fairly constant. Interestingly, the execution time of `GRANT` statements for the balanced hierarchy structure is significantly lower than for the deep and wide structures, capping at around 12 milliseconds instead of 25 milliseconds. The execution time of `CREATE ROLE` statements for the wide hierarchy structure is the highest in the cloud setup.

Figure 6 shows the individual execution times of the `CREATE ROLE` and `GRANT` statements for Snowflake across all hierarchy structures. We observe that the execution times fluctuate between 300 and 450 milliseconds, with frequent spikes up to 800 milliseconds. The execution times remain constant over time, with no significant differences between the hierarchy structures and statement types (`GRANT` vs. `CREATE ROLE`).

Explanation of Findings. The results of the Role Hierarchy Creation experiment show significant differences in performance across the evaluated systems and setups. In the following, we try to explain the observed findings.



(a) On-premise setup (b) Cloud setup

Fig. 7: I/O operations per second for PostgreSQL and MariaDB in the on-premise setup and the cloud setup.

PostgreSQL's drastic performance difference between the on-premise and cloud setups can be attributed to its high number of I/O operations per second (IOPS)—that is, the number of read and write operations performed on the storage device—when executing `GRANT` statements. This is illustrated in Figure 7, which shows the number of IOPS for PostgreSQL and MariaDB in the on-premise setup (Figure 7a) and the cloud setup (Figure 7b). IOPS were measured every minute using Windows Performance Monitor in the on-premise setup, and every five minutes using AWS CloudWatch in the cloud setup. In the on-premise setup, PostgreSQL uses approximately 2,000 IOPS, whereas in the cloud setup, it drops to a maximum of 70 IOPS. AWS imposes limits on IOPS: GP2 storage devices, as used in our experiment, are allowed to perform 3 IOPS per GiB with a minimum of 100 IOPS [1]. Actual performance may fall short of provisioned IOPS [1], which likely occurred in our experiment. We attribute the high IOPS demand to the `GRANT` statements rather than the `CREATE ROLE` statements, based on the elevated latencies observed in Figure 4.

MariaDB does not use the same I/O-intensive approach for grants as PostgreSQL and generally performs well in both on-premise and cloud environments. However, the performance of `GRANT` statements in MariaDB can degrade depending on the structure of the role hierarchy. This is due to the way MariaDB manages and caches privilege data at each role in the hierarchy. When a role is added or modified, MariaDB invalidates the privilege cache of that role and recursively invalidates all ancestor roles up to the root. Then, for each invalidated role, MariaDB traverses its subtree to rebuild the privilege cache, descending only into subtrees with invalidated caches.[3] In deep linear hierarchies, this results in high latency because every parent in the chain is invalidated and must be traversed again (complexity of $O(N)$). In wide hierarchies, any modification

[3] The source code for this process can be found in https://github.com/MariaDB/server/blob/main/sql/sql_acl.cc, beginning at line 6359 (accessed: 2025-05-28).

to a role causes the root's cache to be invalidated, requiring traversal of all its children during cache reconstruction (also $O(N)$). By contrast, balanced hierarchies are more efficient: only $O(\log N)$ ancestor roles are invalidated, and most subtrees retain valid caches, leading to lower latency when modifying a balanced role hierarchy.

Finally, Snowflake's high latency of around 300 to 450 milliseconds for `CREATE ROLE` and `GRANT` statements is due to its distributed architecture, which prioritizes large-scale data processing over low-latency operations. Snowflake's architecture consists of three layers: a service layer, a compute layer, and a storage layer [7]. Requests arrive at the service layer, which is a distributed mesh of nodes, and are forwarded to the compute and storage layers if necessary. Metadata in Snowflake is stored in a key-value store within the service layer. This key-value store is itself a distributed service in which metadata is replicated for durability. Modifying metadata, such as creating roles or granting privileges, therefore requires coordination across multiple nodes in the service layer. This coordination involves network communication, which results in higher latency compared to database systems like PostgreSQL and MariaDB. In our setup, these systems were installed as single-node instances without replication and additional network round trips.

Role Hierarchy Access Performance In this experiment, we executed the Role Hierarchy Access component of the benchmark, measuring the performance of accessing securable objects and role metadata through SQL commands (`SELECT` and `SHOW` queries). We created role hierarchies of varying sizes (1,000, 10,000, and 100,000 roles) using three hierarchy structures: deep, wide, and balanced. The `SELECT * FROM table` and `SHOW ROLES` queries were each executed five times, and we report the median execution time. We measured the end-to-end round-trip time from the benchmark driver, which includes network latency from the driver to the database system.

Table 1 shows the execution times for the `SELECT * FROM table` and `SHOW ROLES` queries across the evaluated systems and hierarchy structures, with results reported in milliseconds. For role hierarchies that took more than 15 minutes to create, the experiment was aborted and the corresponding table entries are left blank.

For the `SELECT` query, we make the following observations: On-premise setups for PostgreSQL and MariaDB generally perform better than their cloud counterparts, which is expected due to the absence of network latency. Snowflake exhibits higher latency, which we speculate may be due to inefficient caching and the overhead of loading roles and privileges from its key-value store. A clear distinction can be seen between the local driver in Europe and the cloud driver in the same AWS region, due to the overhead of end-to-end network traffic. Overall, the performance of the `SELECT` query is not significantly affected by the structure of the role hierarchy, except in MariaDB's cloud setup, where the balanced structure outperforms the deep and wide variants. We currently have

	`SELECT * FROM table`			`SHOW ROLES`		
	Role Hierarchy Size			**Role Hierarchy Size**		
	1,000	**10,000**	**100,000**	**1,000**	**10,000**	**100,000**
PostgreSQL						
Local, Deep	0.28	0.22	0.24	1.41	7.53	69.57
Local, Wide	0.18	0.27	0.28	0.95	7.08	71.16
Local, Balanced	0.26	0.24	0.23	1.39	7.73	71.56
Cloud, Deep	0.46	0.50		1.74	9.28	
Cloud, Wide	0.68	0.46		1.75	9.16	
Cloud, Balanced	0.79	0.53		1.56	9.26	
MariaDB						
Local, Deep	0.13	0.16		1.47	7.22	
Local, Wide	0.22	0.16		1.35	6.87	
Local, Balanced	0.22	0.12	0.16	1.18	6.87	68.85
Cloud, Deep	0.44	0.53		2.08	11.19	
Cloud, Wide	0.46	0.40		1.85	9.81	
Cloud, Balanced	0.28	0.28	0.28	1.82	9.49	84.45
Snowflake						
Local driver, Deep	618			2354		
Local driver, Wide	456			2044		
Local driver, Balanced	571			2719		
Cloud driver, Deep	441			2580		
Cloud driver, Wide	391			559		
Cloud driver, Balanced	253			374		

Table 1: Median execution time (ms) for `SELECT * FROM table` and `SHOW ROLES` across systems. Blank cells indicate timeouts of hierarchy creation (> 15 min).

no clear explanation for this anomaly, and there are no noticeable differences or outliers between the five query repetitions in MariaDB's cloud setup.

For the `SHOW ROLES` query, we observe that PostgreSQL performs equally well in both the on-premise and cloud setups, with the `SHOW ROLES` command taking roughly 10 times longer as the role hierarchy increases in size, which is expected. MariaDB's performance is similar to PostgreSQL's. Snowflake's performance is considerably slower, in the range of 1–2 seconds. The outliers of 559 ms and 374 ms for the "cloud driver, wide" and "cloud driver, balanced" cases, may likely be explained by caching effects. Examining the individual execution times across the five runs, we recorded the following for "cloud driver, wide": 3359 ms, 344 ms, 2713 ms, 357 ms, and 559 ms (in that order). For "cloud driver, balanced": 3163 ms, 2724 ms, 365 ms, 351 ms, and 374 ms. Snowflake, being a distributed system, routes query requests to different nodes in the service layer, which can lead to varying latencies due to potentially different hardware characteristics and warm or stale caches.

6 Conclusion

In this study, we investigated the performance of Role-Based Access Control (RBAC) across three data management systems: PostgreSQL, MariaDB, and Snowflake. We designed a benchmark to evaluate the performance of RBAC mechanisms, focusing on the role hierarchy creation and access. The benchmark measures the performance of creating role hierarchies with varying structures (deep, wide, and balanced) and accessing securable objects and role metadata through SQL commands. The experiments were conducted in both on-premise and cloud setups.

Our results show significant differences in performance characteristics across the systems and deployment configurations. PostgreSQL performs well in the on-premise setup, creating large role hierarchies efficiently, but suffers in the cloud due to IOPS limitations. MariaDB shows consistent performance across both setups, with balanced hierarchies performing better than deep and wide hierarchies due to how MariaDB invalidates and rebuilds caches. Snowflake exhibits high latency for role hierarchy creation and access operations, due to its distributed architecture in which metadata is managed in a key-value store within one of its architectural layers. PostgreSQL and MariaDB were not set up in a distributed manner, so direct comparisons between these systems and Snowflake should be made with caution.

These findings reveal that each system exhibits different performance characteristics, and all tested systems show potential for optimization. PostgreSQL could be improved with respect to I/O operations when executing `GRANT` statements. MariaDB could benefit from optimizing the invalidation mechanism of its role hierarchy cache. Snowflake's RBAC performance will improve if Snowflake optimizes network patterns within its architecture.

Our study also highlights the need for a standardized benchmark for RBAC performance in data management systems. This is important as organizations increasingly centralize their data management and provide access to a growing number of users and applications, regulated via RBAC. This leads to increasingly large role hierarchies, which can impact performance. The benchmark we designed provides a first step toward evaluating RBAC performance, specifically focusing on the role hierarchy component at scale.

Future work includes extending the benchmark to cover additional RBAC features, such as granting different privileges, scaling the number of securable objects accessed via the role hierarchy, and evaluating more complex hierarchy structures such as DAGs and disjoint graphs. Furthermore, various other queries to access RBAC metadata can be considered, for example, `INFORMATION_SCHEMA` queries. Finally, we plan to evaluate RBAC performance in additional data management systems as well as in replicated setups of PostgreSQL and MariaDB, and to investigate data governance performance in open-source metadata catalogs, such as Apache Polaris and Unity Catalog.

References

1. Amazon Web Services: Amazon RDS DB instance storage. https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/CHAP_Storage.html, accessed: 2025-05-28
2. Benson, L., Binnig, C., Bodensohn, J.M., Lorenzi, F., Luo, J., Porobic, D., Rabl, T., Sanghi, A., Sears, R., Tözün, P., Ziegler, T.: Surprise benchmarking: The why, what, and how. In: DBTest. pp. 1–8 (2024)
3. Boncz, P.A., Anadiotis, A.G., Kläbe, S.: JCC-H: Adding join crossing correlations with skew to TPC-H. In: TPCTC. Lecture Notes in Computer Science, vol. 10661, pp. 103–119 (2017)
4. Brücke, C., Härtling, P., Palacios, R.E., Patel, H., Rabl, T.: TPCx-AI – an industry standard benchmark for artificial intelligence and machine learning systems. Proc. VLDB Endow. **16**(12), 3649–3661 (2023)
5. Cole, R., Funke, F., Giakoumakis, L., Guy, W., Kemper, A., Krompass, S., Kuno, H., Nambiar, R., Neumann, T., Poess, M., et al.: The mixed workload CH-benCHmark. In: DBTest. pp. 1–6 (2011)
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM symposium on Cloud computing. pp. 143–154 (2010)
7. Dageville, B., Cruanes, T., Zukowski, M., Antonov, V., Avanes, A., Bock, J., Claybaugh, J., Engovatov, D., Hentschel, M., Huang, J., et al.: The Snowflake elastic data warehouse. In: Proceedings of the 2016 International Conference on Management of Data. pp. 215–226 (2016)
8. El-Hindi, M., Arora, A., Karrer, S., Binnig, C.: Towards a benchmark for shared databases [vision paper]. Datenbank-Spektrum **22**(3), 227–239 (2022)
9. Ferraiolo, D., Kuhn, R.: Role-based access controls. In: Proceedings of the 15th National Computer Security Conference. pp. 554–563 (1992)
10. Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Transactions on Information and System Security (TISSEC) **4**(3), 224–274 (2001)
11. Iosup, A., Hegeman, T., Ngai, W.L., Heldens, S., Prat-Pérez, A., Manhardto, T., Chafio, H., Capotă, M., Sundaram, N., Anderson, M., Tănase, I.G., Xia, Y., Nai, L., Boncz, P.: LDBC graphalytics: A benchmark for large-scale graph analysis on parallel and distributed platforms. Proc. VLDB Endow. **9**(13), 1317–1328 (2016)
12. Kersten, M.L., Kemper, A., Markl, V., Nica, A., Poess, M., Sattler, K.U.: Tractor pulling on data warehouses. In: DBTest. pp. 1–6 (2011)
13. MariaDB Foundation: MariaDB server. https://mariadb.com, accessed: 2025-05-28
14. O'Neil, P.E., O'Neil, E.J., Chen, X.: The star schema benchmark (SSB). https://www.cs.umb.edu/~poneil/StarSchemaB.pdf, accessed: 2025-05-28
15. Patil, S., Polte, M., Ren, K., Tantisiriroj, W., Xiao, L., López, J., Gibson, G., Fuchs, A., Rinaldi, B.: YCSB++ benchmarking and performance debugging advanced features in scalable table stores. In: Proceedings of the 2nd ACM Symposium on Cloud Computing. pp. 1–14 (2011)
16. Sanhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Computer **29**(2), 38–47 (1996)
17. Stonebraker, M., Rowe, L.A.: The design of Postgres. ACM Sigmod Record **15**(2), 340–355 (1986)
18. Transaction Processing Performance Council (TPC): TPC Benchmark™ C (2010), http://www.tpc.org/tpcc/

19. Transaction Processing Performance Council (TPC): TPC Benchmark™ H (2022), http://www.tpc.org/tpch/
20. Transaction Processing Performance Council (TPC): TPC Benchmark™ DS (2024), http://www.tpc.org/tpcds/
21. Vieira, M., Madeira, H.: A dependability benchmark for OLTP application environments. In: Proceedings 2003 VLDB Conference. pp. 742–753. Elsevier (2003)
22. Vieira, M., Madeira, H.: Towards a security benchmark for database management systems. In: International Conference on Dependable Systems and Networks. pp. 592–601. IEEE (2005)