Best practices in designing fine-grained privileges in multi-layered virtual models

Revision 20230308

Denodo Inc.: 525 University Avenue, Suite 31, Palo Alto, CA 94301. USA www.denodo.com

The Denodo Platform supports user and role-based authentication and authorization mechanisms with both schema-wide permissions (e.g., to access Denodo databases and views) and data-specific permissions. For the particular case of views, Denodo allows defining privileges at different levels of granularity:

- General or “coarse-grained” view privileges: To delimit whether a user/role can execute, edit, insert, update, delete on a certain view, as well as see its metadata as a whole.
- Fine-grained view execution privileges: It allows specifying what particular data of a view should be visible to a user/role when that view is executed as well as for INSERT/UPDATE/DELETE operations. There are different kinds of fine-grained privileges:
    - Column restrictions: Limit the columns a user/role can use on a query.
    - Row restrictions: Limits the rows that are visible to a user/role, allowing to mask values and defining custom policies for complex criteria.
    - Custom policies: Denodo offers an API to allow developers to provide their own access control rules.

This document is aimed at people that are already familiar with the Denodo security system and the different kinds of fine-grained view privileges described above.

The purpose of this document is to provide best practices on how to model these fine-grained privileges in complex environments where the virtual model in Denodo follows a multi-layer architecture with different development teams and users who can have multiple roles.

1 DOCUMENT STRUCTURE

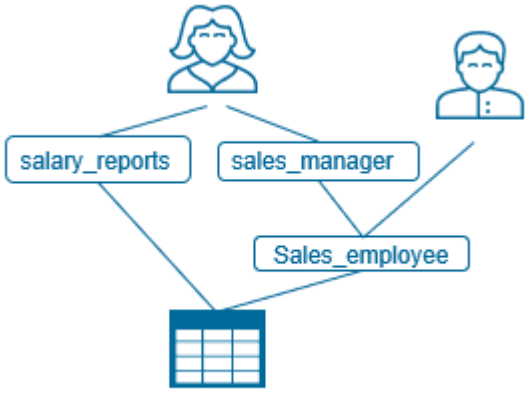
This document is structured on the following sections:

1. **Role Based Access Control model in Denodo**. Overview of the Role Based Access Control (RBAC) approach Denodo is based on.
2. **General best practices with fine grained privileges**. General considerations about where to define fine grained privileges in the view hierarchies defined with Denodo.
3. **Multi-layered virtual model with several development teams**. A virtual model in Denodo is usually structured in different layers that can involve different development teams with different people in charge of assigning the privileges of the views built by each team. This section describes the most common layers to understand the need of defining fine-grained privileges on certain layers.
4. **Setting Limits to the Views in Higher Layers**. In the multi-layered scenario described in the previous section, the stewards of the lower layers may want to set up limits to how their views are used in views from higher layers. This section describes how fine-grained privileges and global security policies can be used for this.
5. **Conclusions and takeaways**. This section summarizes the main ideas and best practices exposed in the document.

2 1 ROLE BASED ACCESS CONTROL MODEL IN DENODO

Denodo access control using roles is based on the NIST Role Based access control (RBAC) standard. In this regard, there are some considerations to take into account to understand how privileges are managed in Denodo:

- **Users acquire permission through roles**. It is important to understand when we talk about a role in RBAC we are referring to a collection of privileges and not a group of users. Sometimes a role can represent the privileges corresponding to a group of users (`sales_employee` in the image below) but not always (in the example, `salary_reports` represents the privilege to execute certain views accessing employee's salary to create reports).
- **Roles can be hierarchical**. For example, `sales_employee` can have execution privilege over certain views but not others. `sales_manager` can have the role `sales_employee` assigned and that way "inherits" its privileges. In addition, `sales_manager` can have extra privileges over the same or other elements.
- **The NIST RBAC model is based on positive permissions** as negative permissions can be very confusing and difficult to manage especially in the presence of hierarchies. This means that a role defines the actions a user can do and the data a user can see instead of specifying the actions that are forbidden.
- **A user can have several roles assigned and their permissions are additive**. The privileges of the user will be the privileges of all roles assigned simultaneously. For example, let's say `sales_manager` has the privilege to see the salary of all the employees in the sales department but not others. `salary_reports` has the privilege to see the salary of all the company employees. If a user has both roles, she will be able to see the salary of all employees.



Hierarchical roles and functional roles

3 2 GENERAL BEST PRACTICES WITH FINE-GRAINED PRIVILEGES

As a general rule, fine-grained privileges should be defined on the final views that are exposed to data consumers, to decide what data is exposed to them. Defining fine-grained privileges in intermediate levels of the view hierarchies can lead to management complexities (some of them described later in this document) and, therefore, it is discouraged when not strictly needed. If the final view does not contain all the necessary columns required to evaluate the privilege, choose the highest level view with that information available (or create an intermediate auxiliary view with the required information if necessary).

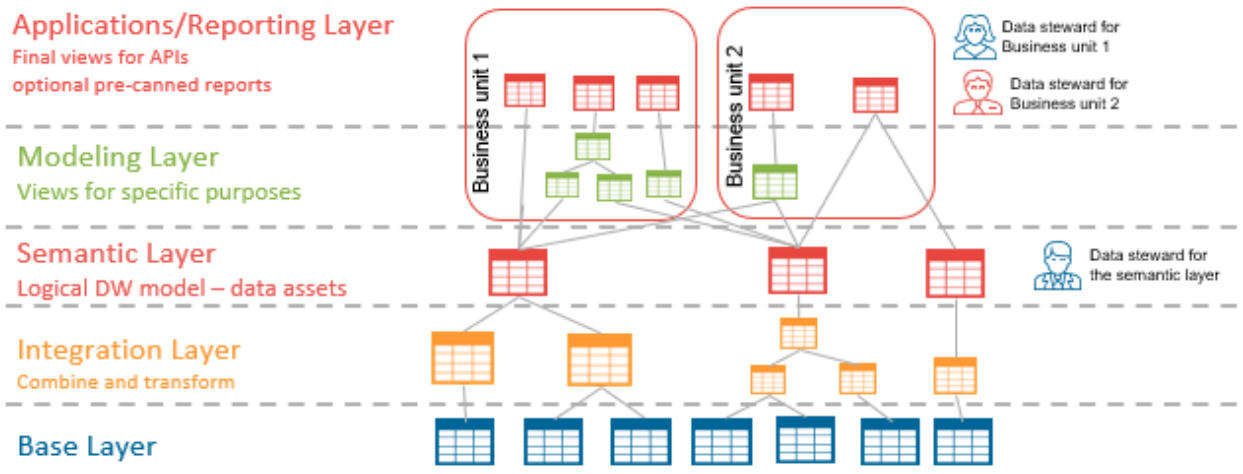
Nevertheless, in multi-layered virtual models with several development teams, defining restrictions at intermediate layers may be unavoidable: different layers may be managed by different teams with different security concerns and the owners of the views in a given layer may need to ensure that the data of their views is not inadequately exposed by views in the higher layers. The next sections of this document describe this situation in detail and explain the best practices that can be followed to avoid this while minimizing management complexity.

4 3 MULTI-LAYERED VIRTUAL MODEL WITH SEVERAL DEVELOPER TEAMS

Virtual models in Denodo are usually designed following a layered architecture in order to separate the different responsibilities, increase reusability and help the security management and maintenance. The chosen layers may vary but in order to illustrate the best practices we will use the following layered structure as an example:

- A **base layer** or physical layer, containing the data sources' details and base views
- An **integration layer**, containing the intermediate transformations and combinations needed to create the views exposed to consumers.
- A **semantic layer** containing the canonical entities that act as data assets for different purposes and can be transversal to different areas.
- A **modeling layer** containing views that different business units may need to create for specific purposes.
- An **applications and reporting layer** containing the final views which the different business units want to publish for specific applications, and optional pre-canned reports with calculated metrics. May also contain final views published to certain clients as an API.

Data stewards must decide which views will be available to the other data consumers and what restrictions should apply to each one: different data services, business power users, data scientists, etc.
In general, data consumers can access subsets of the views in the applications and modeling layer (or even in the semantic layer) depending on their privileges.



In big deployments, there may exist several development teams. In this example, we will separate the developers in two groups (there may be more in real cases, but this is enough for our discussion):

- Developers in charge of building the canonical views of the semantic layer. These developers build canonical views that can then be used for different purposes.

- Developers from specific business areas that will need access to the canonical views to build new views to fulfill the requirements of their particular data consumers.

The data steward responsible for defining and supervising the security rules for those view assets at the semantic level is commonly a different person from the data steward responsible for the data assets created inside a business unit like marketing, operations, etc. This scenario with different teams and different people in charge of assigning privileges on each one requires taking into account some security considerations that are described in the next section.

5 4 SETTING LIMITS TO THE VIEWS IN HIGHER LAYERS

One key point to bear in mind is that the privileges of a derived view are independent of the privileges assigned to its underlying views. This is not specific to Denodo, it is standard behavior in all commercial databases.

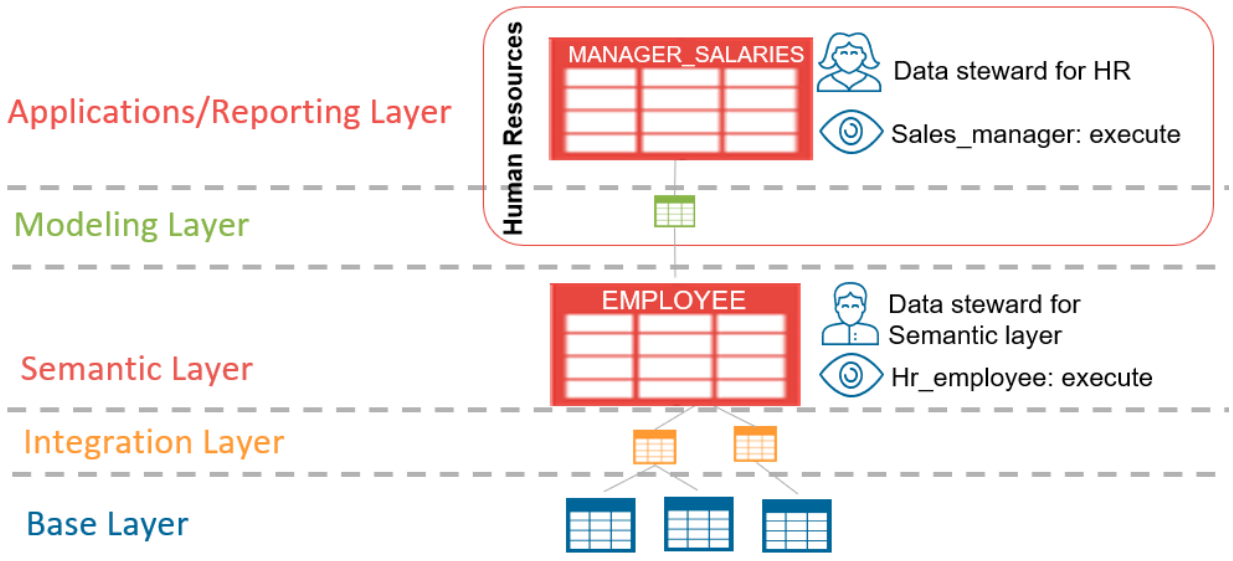
A sometimes underappreciated consequence of this is that when we allow a user to create new views and decide who can execute them, this user also gains the ability to decide who can indirectly see the data of the original views used to create the new one. Let's see an example to illustrate this point.

Example 1: let's say a company has the following two virtual databases in Denodo:

- **core_db**: This database contains the views from the semantic layer. Among others it contains the view EMPLOYEE.
- **hr_db**: This database contains the views of the Human Resources development team. The development team of the HR department is allowed to create their own derived views on top of the 'core_db' views, and they have built the view MANAGER_SALARIES, which is a derived view built on top of the EMPLOYEE view.

The steward of the 'core_db' virtual database (Ron) has decided to give the EXECUTE privilege to the 'hr_employee' role, but it has not given that privilege to the 'sales_employee' role to protect the employee salaries data from being disclosed.

The data steward for the HR database (Amanda) has the 'assign privileges' role in the HR database and, therefore, she can assign the EXECUTE privilege on MANAGER_SALARIES to any role she wishes, including the 'sales_employee' role. But the MANAGER_SALARIES view allows seeing salaries data, so now the users with 'sales_employee' role have *indirect visibility* over the data in the EMPLOYEE view. They cannot see the salaries by executing the EMPLOYEE view directly, but they can get whatever data from EMPLOYEE that is exposed through the MANAGER_SALARIES view.



This may not be the desired outcome. Ron may want to limit who can see the data coming from the 'core_db' views, even for indirect access through views in higher layers.

Key Takeaway: Giving execute access over a view to a user with privilege to create views and assign privileges, is also indirectly transferring the ability to grant privileges to third users over that data. In order to keep these indirect accesses to your views under control you can choose one or both of the following strategies: 1) Limit the indirect access to a list of trusted roles and 2) Define fine-grained privileges for the roles with indirect access.

The following subsections describe these strategies in detail and offer some extra considerations to take into account when different roles are combined together. In particular: Section (4.1) explains how to set limits defining an allowlist of trusted roles; section (4.2) shows how to set limits defining fine-grained privileges; and finally section (4.3) includes a detailed explanation on how the fine-grained privileges are computed when a user has several roles and how this should be taken into account in order to decide the best security strategy.

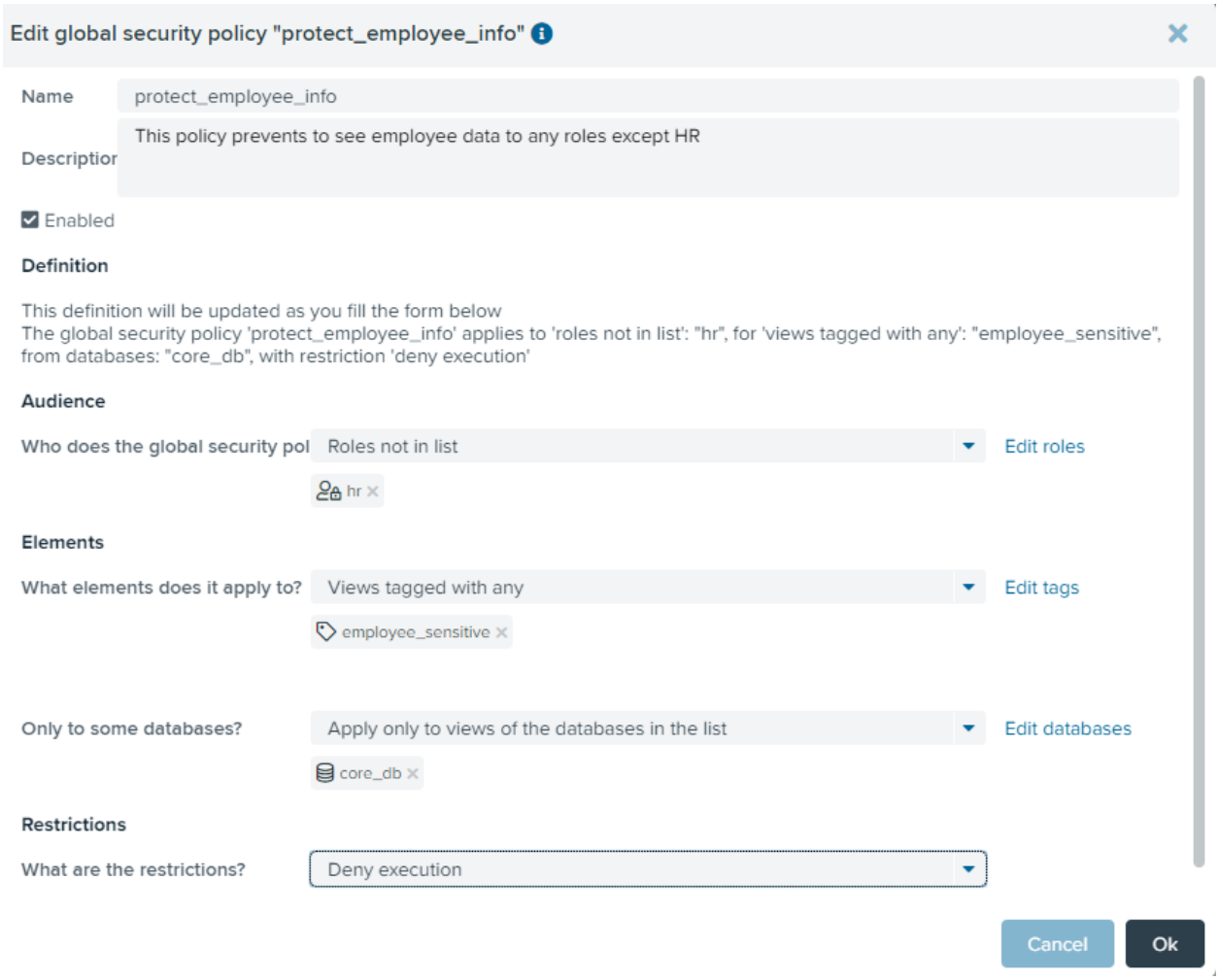
5.1 4.1 ALLOW INDIRECT ACCESS TO A RESTRICTED LIST OF ROLES

To avoid the previous situation, Ron can either make sure that he sets suitable restrictions for all the roles that can be used at higher layers, or he can also adopt a more conservative best practice: deny indirect visibility of the EMPLOYEE view to any role not included in an explicit list of exceptions.

The easiest way to achieve this is using Global security policies (only available with Denodo Enterprise Plus). Starting with Denodo 8.0u20230208, Denodo includes an alternative way to limit this access for any subscription bundle (See section Limit indirect access to views shared with other development teams of the user guide).

Let's see an example of how to do this using Global security policies.

Example 2: Ron could start by defining a global policy to deny the access to view EMPLOYEE to any role different from HR.



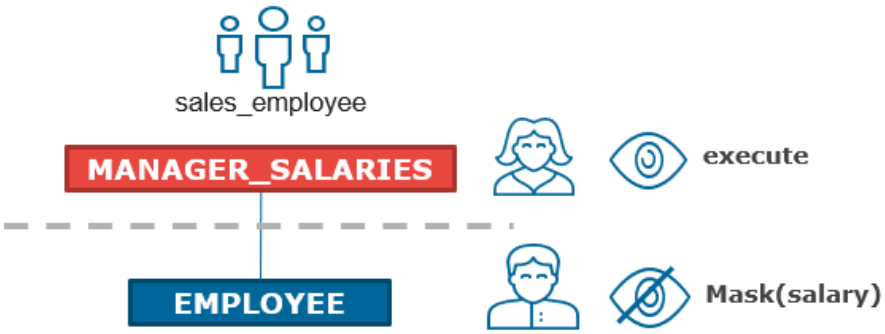
Now, even if Amanda grants execution privileges on `MANAGER_SALARIES` to the '`sales_employee`' and '`marketing_employee`' roles, the execution of these queries will fail because they are not allowed by the policy defined above for the `EMPLOYEE` view.

This best practice has the advantage of not requiring a thorough preventive analysis of the necessary limitations for each possible role. Instead, if some user or group of users need wider access, the data steward can study each case as it is requested and create the adequate policies for it. This is especially useful when the roles used at higher layers can change and/or are not known by the stewards of the lower layers.

5.2 4.2 DEFINE FINE-GRAINED PRIVILEGES TO LIMIT THE INDIRECT ACCESS

Following the same example, let's assume Ron has a global policy in place so only the role 'hr_employee' has indirect access to view EMPLOYEE. Then, upon Amanda's request, Ron can decide to add the roles 'sales_employee' and 'marketing_employee' to the allowlist of roles with indirect access, and use fine-grained privileges to restrict the data available to them.

Example 3: For the 'sales_employee' role, Ron can create a new row restriction on the EMPLOYEE view masking the salary column. This way, even if Amanda gives to the users with this role the privilege to execute queries on MANAGER_SALARIES, the salary data will be masked for them.



5.3 4.3 SPECIAL CONSIDERATIONS COMBINING ROLES WITH FINE-GRAINED PRIVILEGES ON DIFFERENT VIEWS OF THE SAME HIERARCHY

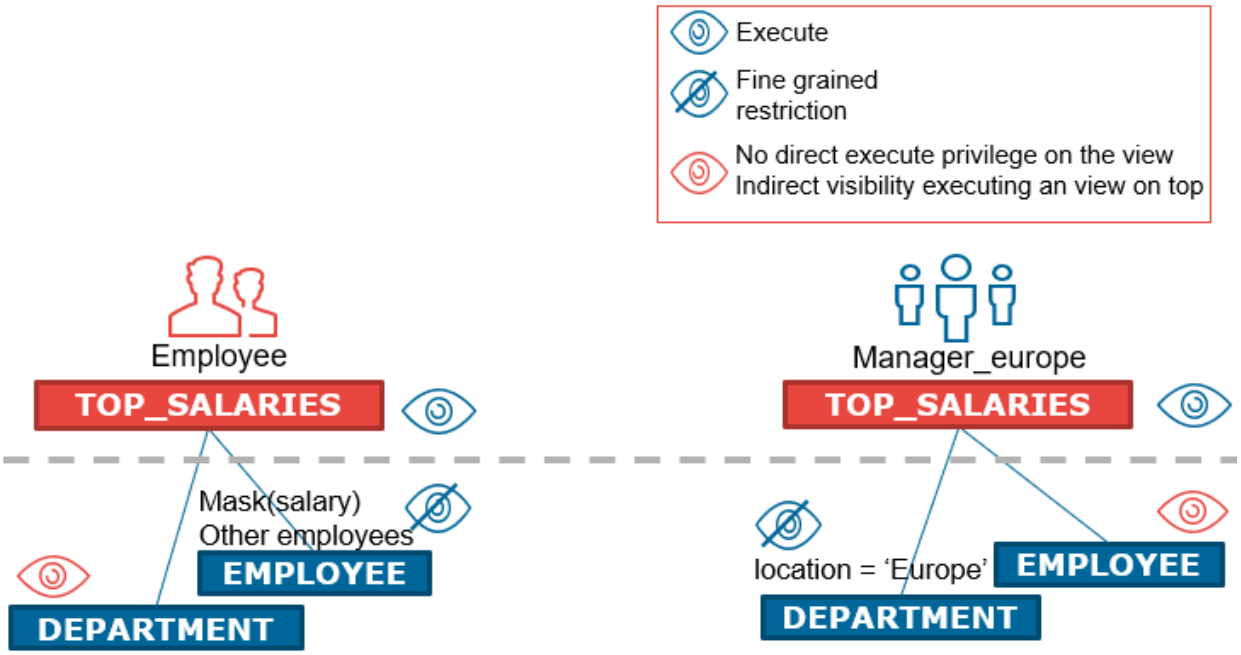
In the previous examples, the derived views created at the higher layers used only one view of the lower layer. When a higher view combines several views of the lower layer, the situation can get more complicated if there exist several roles with indirect access and each view has different restrictions for the different roles. Consider the following example to illustrate the point.

Example 4: Now, the 'core_db' layer exposes to higher layers the views EMPLOYEE and DEPARTMENT:

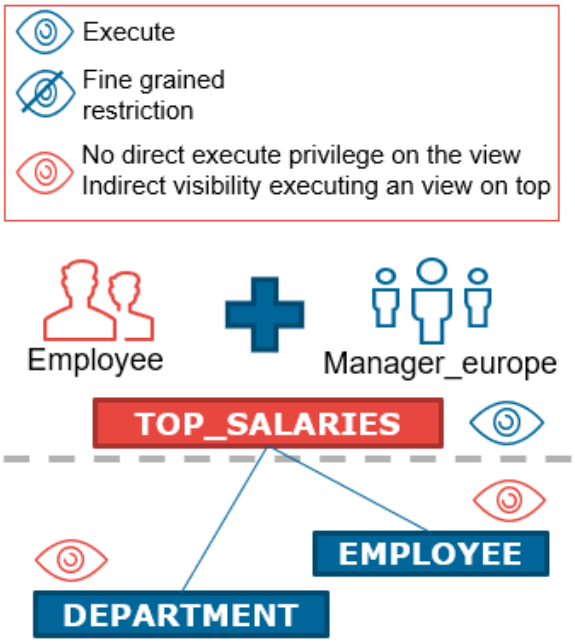
- EMPLOYEE includes among other things the employees' usernames, salaries and department ids;
- DEPARTMENT contains information about the company's departments including their id, name and geographical location.

In the 'hr_db' layer, there is a view TOP_SALARIES created by joining EMPLOYEE and DEPARTMENT:

- Roles ‘employee’ and ‘manager_europe’ have execute privileges on TOP_SALARIES.
- The steward of the ‘core_db’ team has not denied the indirect access to views EMPLOYEE and DEPARTMENT but has defined the following fine-grained privileges:
  - The EMPLOYEE view has a restriction on the ‘employee’ role which allows employees to see only their own salary (this could be implemented for instance by filtering the ‘username’ column by the value of the ‘current_user’ session variable).
  - The DEPARTMENT view has a restriction on the ‘manager_europe’ role which allows it to see only data about departments in Europe (value ‘Europe’ in the ‘location’ column).



Suppose now Sally has both the ‘employee’ and the ‘manager_europe’ roles, and she executes the TOP_SALARIES view. We will review step by step how Denodo evaluates the restrictions that are applied to this query. The key point is that Denodo will execute the query by going bottom-up in the view hierarchy and **computing the applicable restrictions at each individual view in the hierarchy**. More precisely:



- When evaluating the `EMPLOYEE` view, Denodo will detect that the '`manager_europe`' role does not have any restriction on the view and, therefore, has full *indirect visibility* of its data. Since roles are additive, this results in full indirect visibility for Sally on all the data of the view, even though the role '`employee`' does have restrictions.
- In a similar manner, when evaluating the `DEPARTMENT` view, Denodo will detect that the '`employee`' role does not have any restriction on this view and, therefore, has full indirect visibility on their data. So again, the role combination results in full indirect visibility.
- At the `TOP_SALARIES` level there are no further restrictions. Since no restrictions were applied on the lower views, Sally will be able to see all the data.

This is probably not the desired outcome in this scenario. As in Example 4 in the previous section, the root of the problem is that the steward of the '`core_db`' virtual database has not specified restrictions covering all the roles for the views. In this case, no restriction has been defined neither for the '`employee`' role in the `DEPARTMENT` view nor for the '`manager_europe`' role in the `EMPLOYEE` view.
But in this case, the problem cannot be solved by simply adding new row restrictions to `EMPLOYEE` and `DEPARTMENT`, because the desired policies require information that is not available in the views: for instance, we cannot specify a policy in the `EMPLOYEE` view stating that the '`manager_europe`' role can only see data of employees from departments located in Europe, because there is not a '`location`' column in the `EMPLOYEE` view.

There are two possible ways of solving this:

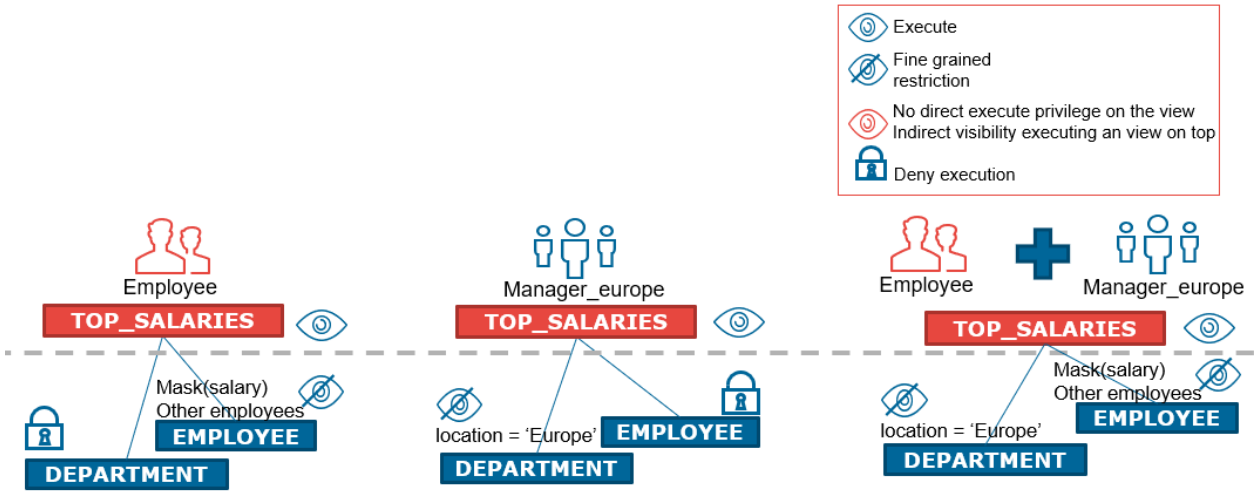
1. Deny indirect invisibility using global policies as described in the previous section.

2. Creating a new view having all the columns required to define the desired policies and expose only that view to the higher layers.

These alternatives result in different effective privileges for Sally in our example, so let's analyze each of them in detail.

Example 5: To implement the first approach we could specify the following policies:

- A policy on `EMPLOYEE` denying indirect visibility to all roles except a list of explicit exceptions (e.g., the '`employee`' role and the '`hr_employee`' role), and an additional policy for specifying the restrictions to apply to each exception (e.g., the '`employee`' role only allows users to see their own salary).
- In a similar way, a policy on `DEPARTMENT` denying indirect visibility to all roles except a list of exceptions (e.g., the '`manager_europe`' role, the '`manager_us`' role and the '`hr_manager`' role), and additional policies to specify the restrictions for the exceptions (e.g., allow the '`manager_europe`' and '`manager_us`' roles to see only the departments in their regions).



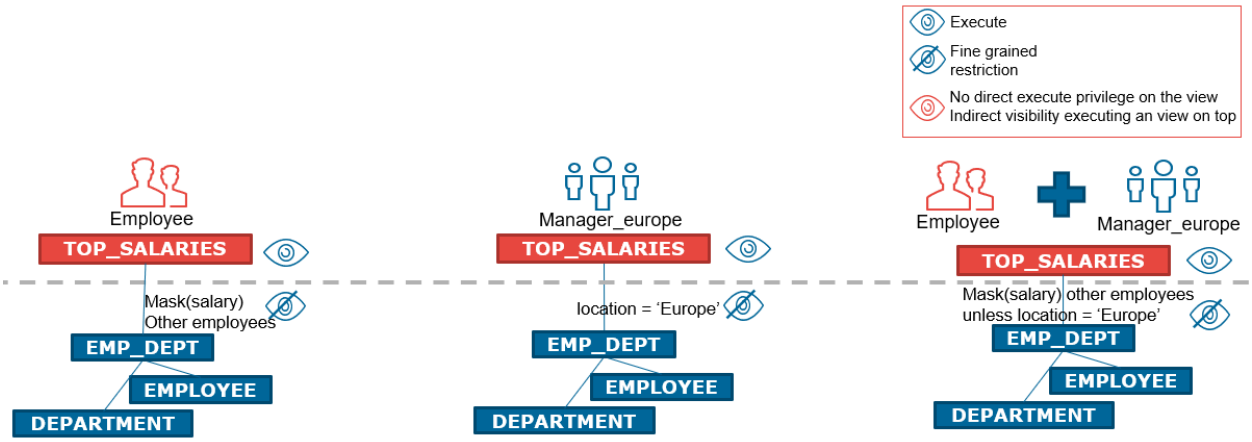
Let's see what happens when Sally executes the `TOP_SALARIES` view in the new situation:

- When evaluating the `EMPLOYEE` view, Denodo will detect that only the '`employee`' role grants Sally indirect visibility on `EMPLOYEE`, restricted to the employee's own salary. As a result, in our example all rows will go up in the execution tree and the salary column will be masked except for the one representing Sally.
- In a similar manner, when evaluating the `DEPARTMENT` view, Denodo will detect that only the '`manager_europe`' role grants Sally some indirect visibility, but restricted to the departments located in Europe. So only rows of departments located in Europe will go up the execution tree.
- At the `TOP_SALARIES` level, the rows from both underlying views are joined, resulting in those rows corresponding to employees in departments located in Europe, with the salary masked for all of them except for Sally's own salary.

This adequately protects sensitive data, but it may seem too restrictive given Sally's roles. So, let's consider now the second alternative of exposing to the higher layers a combined view containing all columns required to implement the desired policies.

Example 6: Exposing to the higher layers a single view combining EMPLOYEE and DEPARTMENT and define the restrictions in it:

- The new view is called EMP_DEPT and joins the EMPLOYEE and DEPARTMENT views by the department id. It contains, among other data, columns for the employee salaries and the location of their departments. We define in EMP_DEPT restrictions analogous to the ones in our initial example:
  - The 'employee' role allows employees to see their own salary.
  - The 'manager_europe' role allows seeing the rows of EMP_DEPT with the value 'Europe' in the 'department_location' column.
- Now TOP_SALARIES is defined on top of EMP_DEPT, since EMPLOYEE and DEPARTMENT are no longer exposed to the developers of the higher layers.



Let's see what happens now when Sally executes TOP_SALARIES:

- When evaluating the EMP_DEPT view, Denodo detects that both Sally's roles have assigned restrictions. Since the restrictions are defined on the same view, Denodo can compute the logical union of the privileges granted by the two restrictions (recall again that roles are additive), so the rows sent to the higher layer will include Sally's row and all the rows corresponding to employees located in Europe.
- At the TOP_SALARIES level there are no further restrictions, so no additional filters are executed.

Key Takeaway: When several views of a lower layer can be combined to create views at a higher layer, make sure to define relevant restrictions for each role in all the views of the lower layer. When this is not possible (e.g. because some restrictions depend on columns that are not available in all views), consider either creating new views with all the needed columns, or using global policies to deny indirect visibility and further restrict access.

As a side note, when creating this type of combined views, make sure to define the relevant referential constraints between them so Denodo can apply the 'join pruning' optimization when needed. See document "Best Practices to Maximize Performance II: Configuring the Query Optimizer" of our Knowledge Base for details.

6 5 SUMMARY AND KEY TAKEAWAYS

1. As a general rule, you should define fine-grained privileges on the final views that are exposed to data consumers, in order to minimize the management complexity introduced by defining fine-grained privileges in intermediate views. Nevertheless, in multi-layered virtual models with several development teams this may not be enough.
2. When we allow a user to create new views and decide who can execute them, this user also gains the ability to decide who can indirectly see the data of the original views used to create the new one.
3. Row restrictions allow to set limits on this privilege, ensuring that certain security rules are always applied over the data of the lower-level views. Since the privileges granted by roles are additive, it's important to ensure that appropriate row restrictions are defined for *all* the roles that can be used to execute the higher-level views.
4. Alternatively, to have tighter control and simplify management, stewards may also consider denying all indirect visibility of the data on their views, and then defining specific policies for the desired exceptions. Indirect visibility can be denied either using global policies or implicit access privileges. This can be especially useful when the roles used at higher layers can change and/or are not known by the stewards of the lower layers.
5. Denodo applies fine-grained privileges separately at each view. This may have subtle implications when a higher view combines several views of the lower layer, each one with different restrictions for different roles. As explained in takeaway 4, starting by denying indirect visibility, and then granting access for specific exceptions simplifies management and ensures tighter control. Nevertheless, in some cases, it may be advisable to create new views for typical combinations of the views of the lower layer, in order to be able to define the most appropriate restrictions for all roles.
6. For specific recommendations on scenarios using caching or smart query acceleration strategies in combination with fine-grained privileges see the document "Fine-grained privileges and caching best practices" of our Knowledge Base.