Fine-grained privileges and caching best practices

Revision 20220715

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

Denodo Virtual DataPort allows configuring a Cache Engine to store local copies of the data retrieved from the data sources. This can be useful for several purposes, such as enhancing performance, protecting data sources from costly queries, and/or reusing complex data combinations and transformations.

Since Denodo version 8, Denodo also includes smart query acceleration techniques using a new type of view called Summary. Summaries allow to store common intermediate results that the query optimizer can then use as a starting point to accelerate analytical queries.

At the same time, 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 fine-grained execution privileges to specify what particular rows and columns should be visible to a user/role when that view is executed as well as for INSERT/UPDATE/DELETE operations.

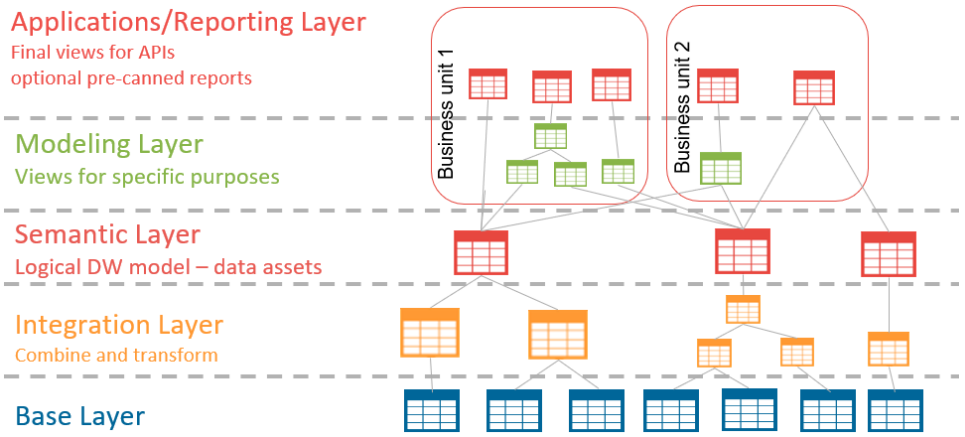
The purpose of this document is to provide best practices on how to design the caching and/or smart query acceleration strategies considering the security requirements on the views to make sure on the one hand, these requirements are preserved and, on the other hand, queries from restricted users can still benefit from these acceleration features.

This document is aimed at people who are already familiar with the caching system as well as with the Denodo security system and the different kinds of fine-grained view privileges available: column restrictions, row restrictions and custom policies.

1 MODELING IN DENODO FOLLOWING A LAYERED ARCHITECTURE

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 it is commonly used:

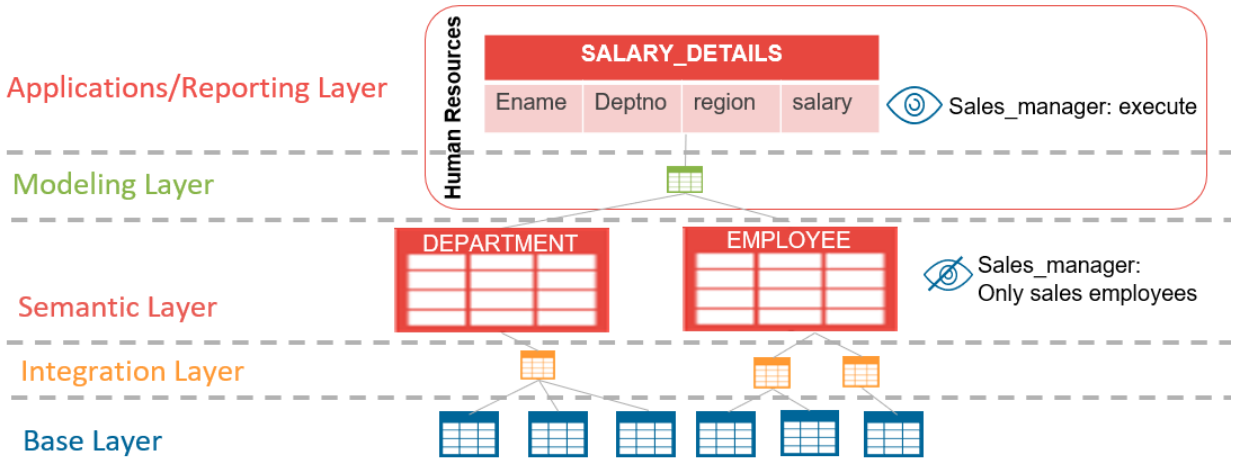
- A **base layer** or physical layer, containing the data sources' details and base views
- An **integration layer**, containing the intermediate transformations and combinations needed to create the views exposed to consumers.
- A **semantic layer** containing the canonical entities that act as data 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.



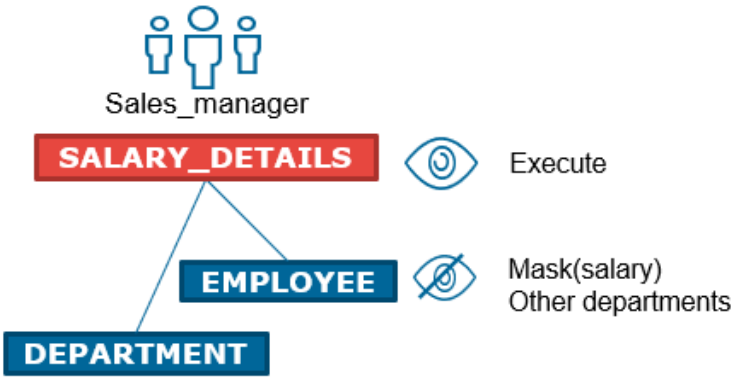
In general, data consumers can access subsets of the views in the applications and modeling layer (or even in the semantic layer) depending on their privileges.
Data stewards must decide which views will be available to what data consumers and what restrictions should apply to each one: different data services, business power users, data scientists, etc. For example, 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 views `EMPLOYEE` and `DEPARTMENT`. The data steward for this database has assigned the following privileges in the production environment:
  - Role `hr_manager` has `EXECUTE` privilege on `EMPLOYEE`.
  - Role `sales_manager` has a masking restriction on the `salary` column so that role can only see the salary for employees in the sales department.

- **HR_db**: This database contains the views of the Human Resources development team. The team has built a view `SALARY_DETAILS` combining `DEPARTMENT` and `EMPLOYEE`. The data steward for this database has assigned `EXECUTE` privilege to the role `sales_manager` on `SALARY_DETAILS`.



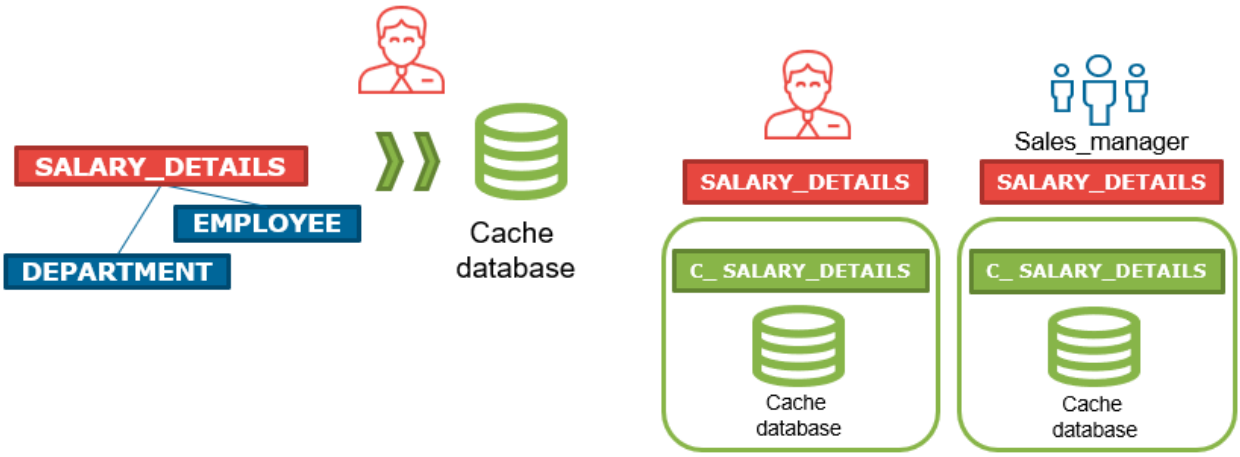
If you specify a fine-grained restriction on a view for a specific role, like masking the `salary` column on `EMPLOYEE`, that restriction is always applied on that view even if the user with that role executes a view built on top of it (unless the user has another role with more privileges). Therefore, if a user with role `sales_manager` executes the view `SALARY_DETAILS`, Denodo will still mask the salary information coming from `EMPLOYEE`:



See the document "Best practices in designing fine-grained privileges in multi-layered virtual models" of the Knowledge Base for more details on how to design the fine-grained privileges on a multi-layered architecture like this one.

2 FINE-GRAINED PRIVILEGE LIMITATIONS USING CACHE

If a view is configured to use cache, the restriction enforcement may not be possible. This is because when a user loads the cache, it's populated with the version of the data that particular user can see, and once that data is in the cache, Denodo will access the cached data regardless of the user. In the example, if a user without any fine-grained restrictions loads the cache for the view `SALARY_DETAILS`, a user with role `sales_manager` executing that view will see all data as Denodo will access the cache directly:



Best practice: When using cache in view hierarchies with fine-grained privileges, define the restrictions on those views with cache configured or on derived views on top of them, but avoid defining them on views under a cached view.

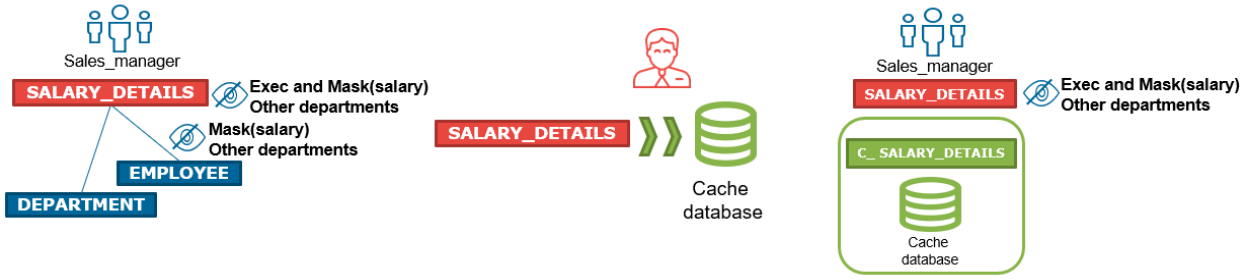
It's important to note that the best practices in designing fine-grained privileges establish that, in general, they should be defined on the views exposed to other users. Therefore, the best practice above will be already verified organically in most cases.

There are situations however, where following this best practice might not be possible. In the previous example, views `EMPLOYEE` and `SALARY_DETAILS` belong to different development teams with different data stewards. In addition, applying caching on `EMPLOYEE` would not be an option in this case as it would not obtain the same performance benefit as applying it on the aggregated view on top.

For those specific cases, there are different ways to design your caching strategy preserving the user restrictions but we will focus on 3 of them: 1) Defining the restrictions on the cached view, 2) creating views for each role, and caching them; and 3) using summaries for accelerating the execution without breaking the security rules set in the view's underneath.

2.1 DEFINE THE RESTRICTIONS ON THE CACHED VIEW

If the cached view projects the necessary fields to define the restriction, it would be a good idea to define the restriction on this view, either duplicating the one from the sub-view or adapting the restriction considering the specific audience of this particular derived view.
In the example, we could modify the permissions of `sales_manager` on SALARY_DETAILS to include the same masking policy. A better approach in terms of maintenance would be to use tag-based security policies as it would avoid duplicating the rule as long as the sensible fields have the right tags in place.

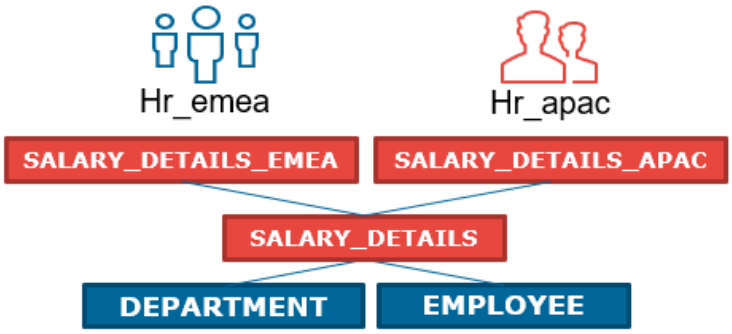


2.2 CREATE DIFFERENT VIEWS AIMED AT DIFFERENT ROLES AND CACHE EACH ONE WITH THE DATA FOR EACH ROLE

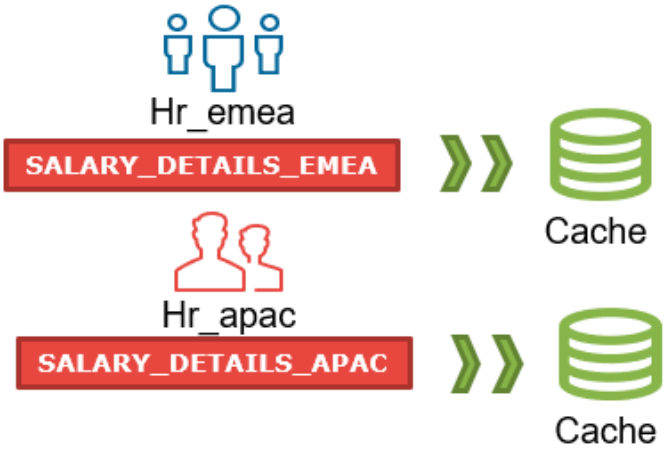
In order to provide different versions of the cached data to different users you can create different views in Virtual DataPort so each role has access to each copy of the view.
For example, consider in our example the company has two regional divisions: EMEA (Europe, the Middle East and Africa) and APAC (Asia-Pacific), and we have two different roles in HR for each region: `hr_emea` and `hr_apac`. The data steward for the semantic layer has assigned a row restriction to `hr_emea` and `hr_apac` on the view EMPLOYEE so these roles can only see the information from EMEA employees or APAC employees respectively.

Let's say it is not possible to define the same privilege at the top view level because the `region` column is not available. A different approach would be to create two

different identical views either by creating new views SALARY_DETAILS_APAC and SALARY_DETAILS_EMEA in the same virtual database or by creating different virtual databases hr_emea and hr_apac and create the same view SALARY_DETAILS on each one.

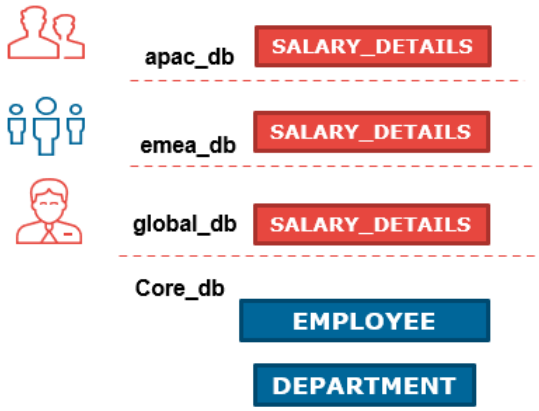


Then you can load the cache for each one using the right user for each load:



When hr_emea stores the data in the cache for SALARY_DETAILS_EMEA, as that role can only see employees from the EMEA region, the data stored in cache will contain the information from that region only. The same happens when the role hr_apac loads the cache for the view SALARY_DETAILS_APAC.

In the specific case where different roles have access to different virtual databases you just need to create the same view on each database and load the cache following the same procedure:



2.2.1

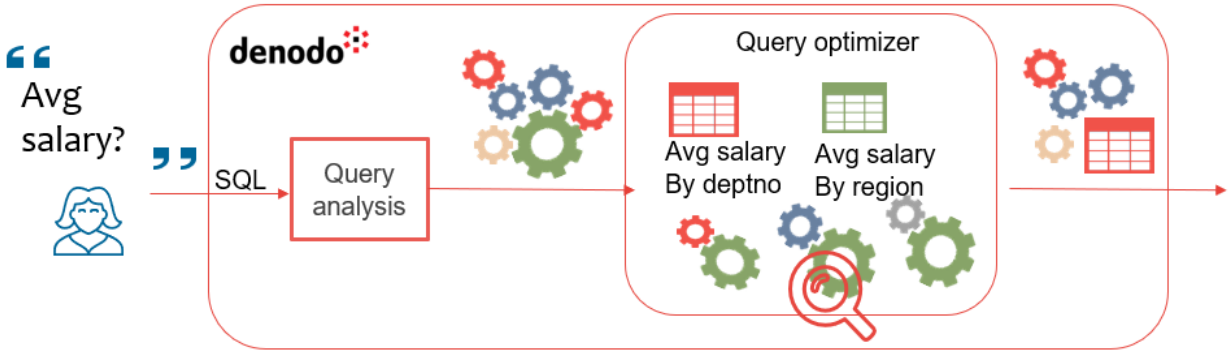
2.3 CREATE SUMMARIES USING DIFFERENT VERSIONS OF THE DATASET

An alternative approach available in Denodo 8.0 is to use summary views to accelerate queries instead of using traditional cache. Differences between caching and summaries make them more suitable for some of the modeling layers and not others (See section 'Summaries vs Cache' in the document "Best Practices to Maximize Performance III: Caching"). In the next section we describe how summaries can be a better strategy than caching when the security requirements include fine-grained privileges.

3 MODELING SUMMARIES CONSIDERING FINE-GRAINED PRIVILEGES

Denodo 8.0 includes a new feature called Smart Query Acceleration, which dynamically selects pre-stored data to avoid performing some of the same data combinations. It consists of: 1) pre-computing interesting data aggregates called "summaries," so then 2) the query optimizer can automatically decide to use these summaries in future queries.

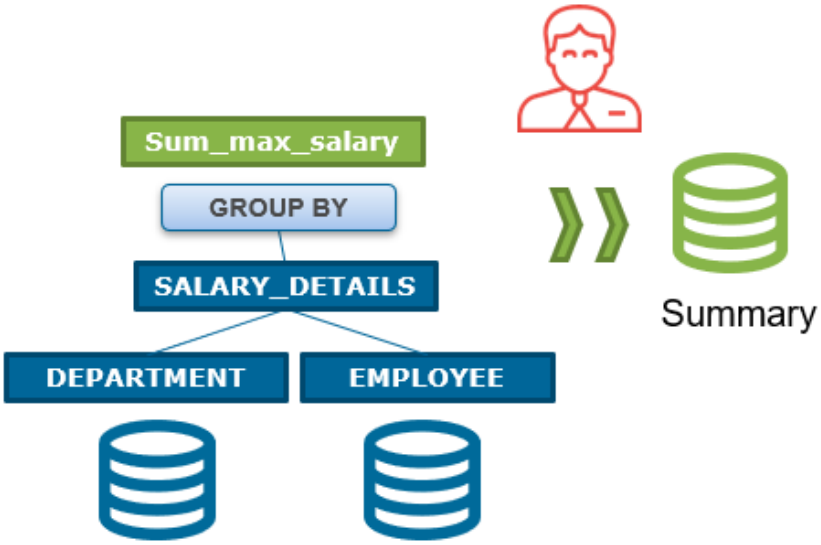
For example, let's say a user sends a query to obtain the average salary of the company employees. If there is a summary already created containing the average salary by department, the query optimizer will detect that it can use the data in that summary to get the final result by applying a final aggregation.



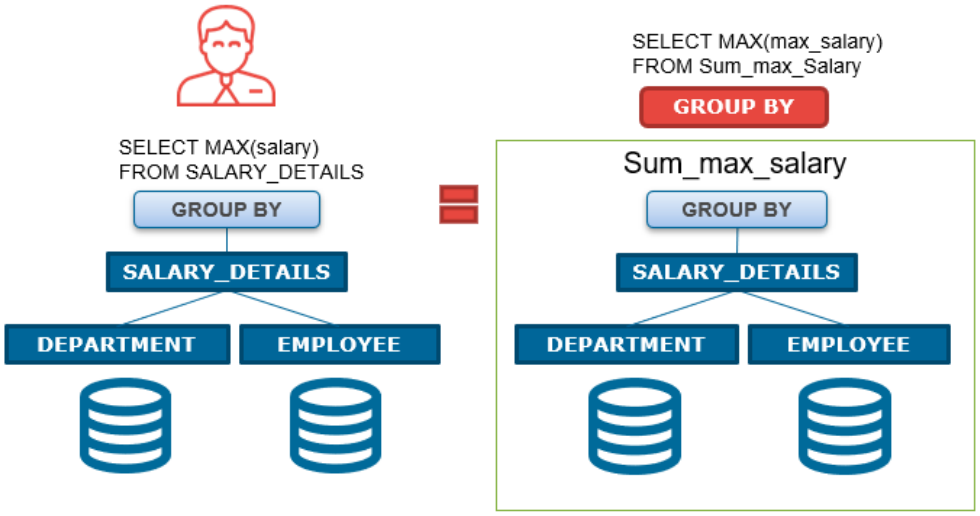
As summaries are transparent to the user and more flexible than traditional caching, they allow to keep the original view for all roles and use one summary or another depending on the user executing the query.

Imagine we create a new summary `sum_max_salary` as:

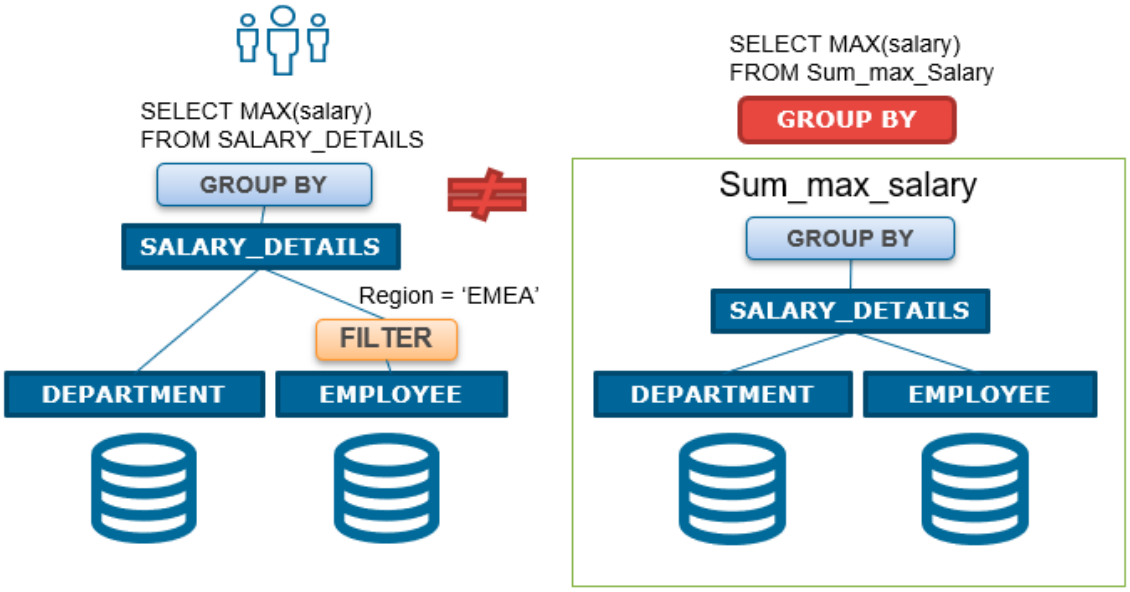
```
SELECT deptno, max(salary) FROM SALARY_DETAILS GROUP BY 1
```



In this case, if an admin user executes `SELECT max(salary) as max_salary FROM SALARY_DETAILS`, the query optimizer will detect it can use the data stored in the summary to answer that query.



On the other hand, if a user with role `hr_emea` executes the same queries, the query optimizer will detect that it cannot use the summary as the query for that user requires an extra condition that is not included in the pre-stored data (see image below).



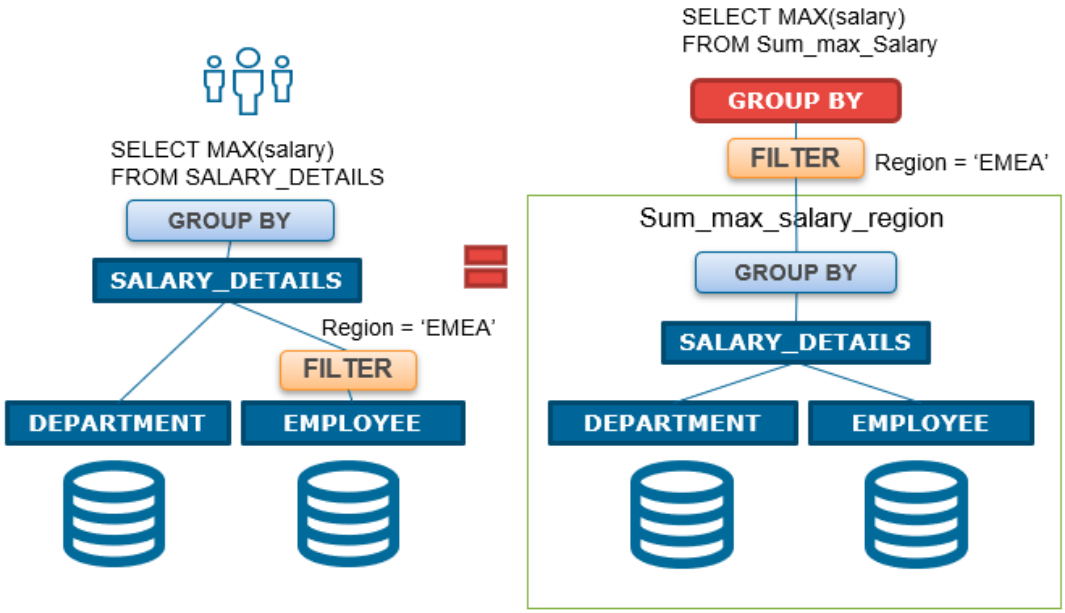
This means in this case the role `hr_emea` will not take advantage of the summary `sum_max_salary`.
In order to allow role `hr_emea` to take advantage of the summary acceleration there are two options:

- Create different summaries for different roles.
- Create a summary that includes the necessary fields to evaluate the restrictions.

For example, we could create a new summary `sum_max_salary_region` including the maximum salary by department and region:

```
SELECT region, deptno, max(salary) FROM SALARY_DETAILS GROUP BY 1,2
```

Once the summary is created and loaded, if a user with role `hr_emea` executes the previous query, the query optimizer will detect there is a summary available with a partial result that can use and apply the restriction and the final aggregation on top:



We could follow the same procedure for the role `hr_apac`.

This strategy has the advantage that it is completely transparent to the user as the view on your model is the same one in all cases. Denodo will take into account if the query used to load the summary includes the required user restrictions and therefore the user will never access data that is not supposed to see. In case there is no summary that matches the required restrictions it will access the original data instead of using the pre-stored one.

Using summaries has the advantage that the optimizer will not select a summary that does not fulfill the view's restrictions. At the same time, it is important to make the right design decisions to make sure the summaries you select are actually used for the restricted users.

3.1 BEST PRACTICES DESIGNING SUMMARIES ON RESTRICTED VIEWS

As mentioned before, 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. If that view does not have the necessary information choose the highest view possible containing all data required to apply the restriction.

The same way, summaries should be designed using the final views that are exposed to data consumers or the highest level containing the necessary information that you need to pre-calculate.

Therefore, it is a good idea to define summaries using the same views containing the fine-grained privilege, which will also allow to include in the summary those columns

that can be necessary to apply a restriction over the summary like in the previous example.

4 FINAL CONCLUSIONS

When your model requires fine-grained privileges in combination with either caching or smart query acceleration using summaries you need to take into account:

- **Summaries** offer more advantages in general, as more queries will potentially benefit from them. In addition, even if they are not configured taking the privileges into account the worst-case scenario will always preserve the security restrictions. In order to use summaries combined with fine-grained privileges there are two main best practices to follow:
  - Define the restrictions and the summaries on the same views, those top views that are accessible to other users.
  - Include in the summary definition those columns that are necessary to evaluate a restriction, or create different summaries aimed at different roles, including the specific restrictions on each one.
- **Caching** has additional configuration capabilities and it does not depend on the optimizer decisions, which means it will always access the cached data without any overhead from the optimization process. This makes caching especially attractive for final reports or base views requiring incremental mode. In order to use caching combined with fine-grained privileges:
  - Define the restrictions on those views with cache configured or on derived views on top of them, but avoid defining them on views under a cached view.