Virtuoso Database

Row Level Security

Abstract

This whitepaper introduces the Row Level Security feature in the Virtuoso Database. This functionality is also known as fine-grained access control or policy based security. This paper also covers why you might want to use row level security and the advantages of using row level security and technical details regarding policies and SQL complication.

Introduction

What is Virtuoso Row Level Security

Traditionally, database administrators assign access "privileges" or "roles" at the table and column level. Programmers writing applications are often required to write additional security layers of code on top of these tables in order to grant or restrict access to data stored therein.

Any other data access outside these applications could bypass this security and retrieve sensitive data.

Row level security in Virtuoso provides an additional level of security in the database as a way to protect data at the row level so that this mechanism cannot be circumvented.

Why Row Level Security in Virtuoso

Physically disconnected systems can be troublesome in an increasingly integrated corporate IS environment. Building access rules into applications can be complex and can run the risk of being circumvented by direct access to the database through other tools or utilities.

This is one of the primary reasons some database level security enforcement is required in applications today. SQL provides table and column level privileges that can be granted to users and roles. This type of security level does not address the situation where different users have differing rights to sections of one table, for example only to data about their own department.

This level of security in a table is typically done with views, which use hard-code selection criteria. The table itself will not be granted but views to specific ranges of rows will be granted to users of the applications or data access tools.

Virtuoso row level security removes the need for the cumbersome requirements of administration of views and requiring applications to use a different view for each type of application or end-user. Additionally the database also does not need to be split out into different departments and stored in separate tables to provide or restrict access thus cutting down on administrative work.

Row Level Security

Row level security or policy based security allows the SQL compiler to make choices according to which user is accessing any given table. Extra conditions will be introduced into a SQL statement at compile time in order to limit the user to a specific range of rows. This will apply equally to reading and modifying the table.

Row Level Security Advantages

There are several advantages to incorporating row level security in to the database. These include:

- Implementation of security policies that is difficult to maintain with views or queries.
- Isolating applications from implementing security polices
- Multilevel security and fine grained security with row-level access control
- Integrated security across the entire set of applications accessing the database deployed in the enterprise

Policy and SQL compilation

A policy is a SQL function that will be called by the SQL compiler each time a table having the policy is accessed in a user's query or stored procedure. The policy can

return extra conditions, which will be anded to the conditions in the query. After this is done, the new query is optimized and compiled.This mechanism makes it possible to transparently customize which information a user account sees without having to maintain a multiplicity of static views. This mechanism cannot be subverted by a user and will work no matter what application is used against the database.

As in Figure 1 detailed below, let us consider the example of a table of classified documents.To access a document, the user needs to have a record in a 'need-to-know' table, which forms a many-to-many relationship between classifications and users. If the user is a member of the security_auditor role, all documents will be accessible. Users themselves may neither read nor update the 'need-to-know' table.

Code Example: Creating a Virutoso Row Level Policy

Figure 1

```
create role staff;
create role security_auditor;
grant staff to security_auditor;

create table document (d_id varchar, d_changed datetime, d_author varchar

create table document_access (da_classification int, da_user  varchar,
primary key (da_d_id, da_user));

grant all privileges on document to staff;
grant all privileges on document_access to security_auditor;

create procedure d_policy (in tb varchar, in op varchar)
{
    if (user_has_role (user, 'security_auditor')) return '';
    if (user_has_role (user, 'staff'))
       return 'exists (select 1 from document_access where da_user = user
       d_classification)';
       return '1=2';

}

table_set_policy ('document', 'd_policy', 'IDUS');
```

Breaking down the Row Level Security Operation

These operations show above in Figure 1, define two roles:

- staff which is not granted access to document_access.
- security_auditor, which configures document_access.

The policy function will, for each query-accessing document, check if the user is a member of security_auditor.

If the user is a member, there will be no extra conditions and all rows will appear. If the user is a member of staff, an extra condition checking the existence of a need to know is added. This will check that there is an entry in the document_access where the accessing user is granted access to the document's classification. If the user is neither member of staff nor of security_auditor, no rows will be returned since the always false condition of 1=2 is added.

Besides SQL selects, the policy rule will apply to SQL inserts, updates and delete operations.This is caused by the 'IDUS' parameter to *table_set_policy*. Different policies may be defined for the select, insert, update and delete operations. Most often, these will be the same. Only one policy function is allowed per table and operation. In order to set a policy, one must be either the table's owner or dba. Policies do not apply to dba users.

We note that the 'need to know' check accesses a table that is not granted to staff, even for reading. This is OK, since the predicate coming from the policy function has access rights of the owner of the policy function. The predicate is treated like a view body that was granted to the user making the query without requiring that component tables of the view were granted.

Since policies are processed at the level of SQL compilation, these will equally apply to local and remote tables.

Queries inside stored procedures are subject to policy. The user whose policies are consulted is the owner of the stored procedure.The policy conditions are fixed at the time of compiling the procedure. Re-assigning a policy counts as a change to the table, causing recompilation of all concerned procedures, triggers and client statements.

Conclusion

Policies are an effective way of implementing a virtual private database, providing for isolation between classes of users of the same application. The maintenance overhead is small compared to the trouble of maintaining views for each class of users. The table concerned remains one schema object with one name.Policies can be set and maintained without modification to applications and existing applications can be made to use policy based access control at no additional cost.