How Can Data Sources Specify Their Security Needs to a Data Warehouse?

Arnon Rosenthal
The MITRE Corporation
arnie@mitre.org

Edward Sciore
Boston College (and MITRE)
sciore@bc.edu

Abstract

In current warehouse systems, the warehouse administrator is given complete authority to grant user permissions. She is trusted to grant only permissions that respect the security needs of the underlying data sources, but has no tools to ascertain what those needs might be, or to detect changes. We believe that such trust is rarely a good idea, and often is not possible. In earlier papers, we proposed a different strategy, in which source permissions are used to automatically derive corresponding warehouse permissions. In this paper we pull the techniques together, identify concepts missing from standard SQL permissions, and show how to extend the syntax of SQL grant commands to include them.

1. Introduction

A data warehouse administers data collected from multiple sources. This transfer of control from the sources to the warehouse raises significant security problems. For example: How does the warehouse administrator know what permissions to enforce? How can a source administrator be assured that his source data will not be compromised by the warehouse? And how can multiple source administrators collaborate to make a combined view available to the warehouse? These problems are exacerbated by the possibility that the various administrators may not completely trust each other – a source administrator may want to limit what information the warehouse administrator sees and what permissions the warehouse administrator can grant.

This paper addresses these problems. Our approach is based on the principle of *derived permissions*: If a user has enough permission on the source databases to obtain some information, and that same information also exists in the warehouse, then that user can be authorized to access that information from the warehouse. When the warehouse is specified as a view over the source databases, then the system can use query processing theory to automatically calculate derived permissions on the warehouse [7].

The ability to automatically derive permissions is a big step forward, as it relieves the warehouse administrator of having to determine, specify, and maintain these permissions manually. However, the derived permissions obtainable using standard SQL only approximate what is needed. In particular, the following scenarios cannot be properly handled:

- a source administrator imposes limitations on when a derived permission can be used, or to whom it can be granted onward;
- the warehouse administrator overrides (or restrict) a user's inferred permission;
- multiple source administrators collaborates to create a combined view, with each administrator granting permissions on his contribution to it.

This paper solves such problems by extending the SQL Grant command along three orthogonal dimensions. As an example of these dimensions, consider the following standard SQL Grant command, which gives user Joe permission to read table T and to grant that permission to anyone else:

grant select on T **to** joe **with grant option**

Our extensions would allow the Grant command to be more accurately specified as follows:

grant info select on T **to** joe
within V
executeif ($DAY = 'Monday')
grantif ($GRANTEE in Accountant)

First, the *info* keyword specifies that this permission is an *information-factor* permission – that is, Joe is given permission on the information content of T (as opposed to permission to run against a particular physical instance of T). Factors are discussed in Section 2. Second, the *within* clause specifies that this permission is granted only so that Joe can execute view V. That is, this permission can be thought of as a partial

permission on V – when Joe obtains similar permissions on all tables mentioned in V, then Joe will be allowed to read from V. Partial permissions are discussed in Section 3. Third, the *executeif* clause contains a predicate that limits how the permission can be used. In this case, Joe can read from V only on Mondays. The *grantif* clause contains a predicate that limits how the permission can be further granted to others. In this case, Joe can grant the permission only to people who belong to the role *Accountant*. Limitation predicates are discussed in Section 4.

Our earlier papers ([8], [9], [10]) have discussed each of these dimensions in detail, but individually and without focusing on warehouse issues. The purpose of this paper is to show how they can be combined into a single framework not significantly different from standard SQL, and to apply these ideas directly to the data warehouse environment. Due to space limitations, we emphasize intuition and examples over formalism. For the latter, the reader is referred to the cited papers.

1.1. Why standard SQL is not sufficient

SQL (and also typical models used in operating systems and middleware) has several weaknesses in coordinating permissions for related data:

- It controls access to SQL tables (materialized, or derived from specific SQL tables) rather than to information (e.g., taxpayers' Income). As a result, applications must be written explicitly in terms of tables where the anticipated user has permission – excessively early binding. (An alternative, also only partly satisfying, is to give permissions to the application-writer and have her delegate).
- Even when content is identical or derivable, there is no formal consistency between tables' permissions (except for a view's owner), so one cannot save work by automatically maintaining that consistency.
- A table's administrator cannot grant permission to use it just for computing a view or procedure (except if they are the view/procedure definer and administer *all* tables needed for the computation).

There are also significant weaknesses in the model for delegating authority. Each privilege is delegated (Granted) completely or not at all, and (beyond our scope here) administrative authority (i.e., grant option) requires possession of the base privilege.

1.2. Previous research

Distributed database security involves all the issues that must be negotiated among systems, including schema integration, user/role sets, degree of autonomy and trust, authentication mechanisms, assigning enforcement responsibilities to servers, combining policies from various authorities, and delegation of authority. The range of tasks is set out in [2, 4, 6].

These works aim at specifying ground rules for *pairwise* cooperation, e.g., between a component system and a federation interface. All issues are addressed at system granularity, in terms of a particular federated architecture. They provide limited integration, e.g., no theory to deduce end-to-end effects (even for cases where *some* deduction is possible). Also, they provide broad toolkits, rather than creating modular abstractions that can tackle each issue thoroughly. Our research addresses a narrower range of issues (just policies, delegation, and enforcement), but covers distributed systems beyond federations, and considers fine-grained permissions with a rigorous (rather than heuristic) inference theory.

Policy specification techniques beyond SQL are relevant to us. A series of papers culminating in [1] defines a calculus of negative permissions with two levels of priority and rules for overriding. We agree with their argument that some such facility is needed, but we seek one that scales up with less complexity. In [5], "provisions" were proposed, preconditions that the system state and history must satisfy before a request can be allowed. The action provisions are attractive, but some of the examples might violate a user's intentions as well as increase the size of the security system. One might move them to wizards and application code which are generated from the provisions.

2. Permission Factors

Each permission in standard SQL is actually the conjunction of two permissions:

- permission to see the specified information; and
- permission to access the place where the information is physically stored.

We call these smaller-scoped permissions *permission factors*. The first factor is called an *information permission*, and the second factor is called a *physical permission*. (In [10], each is subdivided further; physical permissions are extended to software processes, and renamed *run-here* permissions.)

We extend SQL so that a Grant command may contain the keyword *info* (denoting an information permission) or *physical* (denoting a physical permission); grants without either keyword are by default considered to be both. For example, the second Grant command of Section 1 used the *info* keyword. A user is allowed to

access a table only if the user has both corresponding permission factors.

The advantage of having permission factors is that they can be specified independently by different administrators. An administrator (likely at the source) decides what information ought to be released to what user (i.e., role). On the other hand, the administrator of each warehouse and data mart decides what users should access their system, i.e., have physical permission.

As a simple example, suppose a source has tables T1 and T2, which are replicated in the warehouse (as tables W1 and W2). The source administrator wants only Joe to be able to access T1, but T2 is public information. He thus issues the following Grant commands:

grant info select on T1 **to** joe
grant info select on T2 **to** Public

Note that corresponding information permissions will be derived at the warehouse for W1 and W2.

The warehouse administrator, because of resource constraints, wants only Management to access T1 and Accountants to access T2. He thus issues the following Grant commands:

grant physical select on W1 **to** Management
grant physical select on W2 **to** Accountant

The result for W1 is that only Joe will have access, and only if he is in the role *Management*. Only Accountants will be allowed to access W2. The source tables T1 and T2 are inaccessible as long as no physical permissions are declared for them.

The use of permission factors thus allows a source administrator to place bounds on who can see the corresponding warehouse data, and allows the warehouse administrator to independently limit access to the warehouse. A user will have access to warehouse data only if he has appropriate permission from both administrators.

3. Partial Permissions

It is common to allow users to access a view, while protecting confidential data. But what if the desensitizing view involves several, mutually suspicious sources? For example, a drug-effects warehouse might draw information from many hospitals. In order for an administrator at the warehouse to define the view, however, SQL requires that he be granted *select* permission on each underlying source table. If the underlying source data is so sensitive that no warehouse personnel are trusted to read or administer it, then how does the view get defined? And who is sufficiently authorized to grant others access to it?

Our approach is to allow sources to grant permission on a table, subject to the condition that its information be used only to compute a particular view. The constructs below allow the policy to be stated by source administrators, so the warehouse system knows what it *ought* to enforce. We can remove the threats of casual browsing or mistakes by warehouse personnel – to get access to the sources' exported data, an attacker must hack through the DBMS. (Encryption or immediately purging files of source data might further increase security). We rely also on the determination that the view is really secure, e.g., avoids problems of statistical and other inference [3].

We extend the SQL syntax to include a *within* clause, which was briefly mentioned in the example of Section 1. For a more concrete example, suppose the warehouse supports financial studies of hospitals, by providing summary cost data. Hospitals will only release their data if it is hidden within Statewide totals. Thus the warehouse defines a view COST_SUMMARY, and each hospital issues:

grant select on MyHospital.COST **to** Public
within COST_SUMMARY

That is, the sources assert that the information in view COST_SUMMARY is public knowledge, and their respective COST tables are available to the warehouse's query processor when it processes the view.

Thus our extension allows multiple sources to collaborate on a joint (less sensitive) aggregate view, without having to disclose any of their sensitive data. We can imagine several granting strategies. For example, each source could grant the warehouse administrator partial permission (with grant option) on their portion of the view; the warehouse administrator would then have grant-option permission on the view, and could grant permissions to others as desired. Alternatively, each source could grant partial permission to various users, and only those users who obtained partial permission for all underlying tables could access the view.

4. Predicate-Limited Permissions

In standard SQL, a user who has a permission is able to use it in any circumstance. We can refine this "all or nothing" mechanism by attaching limitation predicates to a permission. For example, a limitation predicate may restrict a command to certain days or hours, to members or non-members of certain groups, to users above a certain rank, to tables above a threshold size, or to requests that are received along a trusted path.

We extend SQL so that two predicates are specified with each permission: The *execute predicate* limits how

the permission can be used; and the *grantonward predicate* limits how the permission can be granted to others. Limitation predicates can refer to environment variables, argument values, or the database state. These values are evaluated at the time a query is issued by a user.

For example, consider the second Grant command of Section 1. The execute predicate *($DAY = 'Monday')* limits when Joe can execute a query involving T. The grantonward predicate *($GRANTEE in Accountant)* limits when Joe is allowed to execute a Grant command for T. The argument value *$GRANTEE* denotes the user mentioned in the Grant command.

Our use of limitation predicates is a smooth extension of SQL. An SQL grant without grant option gives arbitrary execute privilege and no grant privilege; thus it is equivalent to

grant select **on** T **to** joe
executeif true
grantif false

An SQL grant with grant option gives arbitrary execute and grant privilege, and thus is equivalent to

grant select **on** T **to** joe
executeif true
grantif true

Predicate-limited permissions allow an administrator to more accurately specify his security needs, and thus are useful in any environment. In the warehouse environment, they also help source and warehouse administrators to work together without requiring complete trust between them.

For example, suppose a source administrator wants the warehouse administrator to help determine permissions on the warehouse. Currently, the only option would be to give the warehouse administrator full grant-option permission on the source tables, which requires great trust. In our model, a source who issues a grant to the warehouse administrator can impose a limitation predicate on using the privilege or on granting it to others, e.g., only to users in management, who have undergone strong authentication. Naturally, these limitations apply to any onward grants made by the recipient. This feature allows the source to set up security guidelines that are automatically enforced at the warehouse.

In the future, we hope to compare the power of limitation predicates, compared with "safety fence" permission factors, the negative permissions in [2, 4], and (especially) techniques that define view tables and grant permissions on them. We also hope to explore adequacy of the model for controlling data replication processors (which typically run with too few security controls).

5. Summary

In this paper, we have shown how data sources can more accurately specify their security needs to a warehouse. By extending the SQL Grant syntax in a few orthogonal ways, we obtain a simplicity, architecture and location independence, and compatibility with relational technology that is missing from other approaches. This paper also shows how our approach solves certain security problems that arise in a warehouse environment.

To date, our research has focused on solidifying the model. We are now beginning to look at implementation issues. In particular, we envisage that permission specification and derivation will be performed by a *permission manager* module, that would facilitate the communication between the warehouse and the sources.

6. References

[1] E. Bertino, S. Jajodia, P. Samarati, "A Flexible Authorization Mechanism for Relational Data Management Systems," *ACM Trans. Information Systems*, Vol. 17, No. 2, April 1999, pp. 101-140.

[2] S. De Capitani di Vimercati, P. Samarati, "Authorization Specification and Enforcement in Federated Database Systems," *Journal of Computer Security*, vol. 5, n. 2, 1997, pp. 155-188.

[3] S. Castano, M. Grazia Fugini, G. Martella, P. Samarati, *Database Secuirty*, Addison Wesley 1994.

[4] S. Castano, S. De Capitani di Vimercati, M.G. Fugini, "Automated Derivation of Global Authorizations for Database Federations," *Journal of Computer Security*, vol. 5, n. 4, 1997, pp. 271-301.

[5] S. Jajodia, M. Kudo, V. S. Subrahmanian, "Provisional authorizations," *Proceedings of the 1st Workshop on Security and Privacy in E-Commerce*, Athens, Greece, November 2000.

[6] D. Jonscher, K. Dittrich, "An Approach for Building Secure Database Federations", *VLDB 1994*.

[7] A. Rosenthal, E. Sciore, V. Doshi, "Security Administration for Federations, Warehouses, and other Derived Data", *IFIP 11.3 Working Conference on Database Security*, Seattle 1999. (Kluwer, 2000).

[8] A. Rosenthal, E. Sciore, "View Security as the Basis for Data Warehouse Security", *CAiSE Workshop on Design and Management of Data Warehouses*, Stockholm, 2000.

[9] A. Rosenthal, E. Sciore, "Extending SQL's Grant Operation to Limit Privileges", *IFIP Workshop on Database Security*, Amsterdam, August 2000.

[10] A. Rosenthal, E. Sciore, "Administering Permissions for Distributed Data: Factoring and Automated Inference", *IFIP Workshop on Database Security*, August 2001.