STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LI**, Number 1, 2006

A FRAMEWORK PROPOSAL FOR FINE GRAINED ACCESS CONTROL

COSTA CIPRIAN

Abstract. One of the main concerns in database security is confining a user to a specific set of the existing data. Current common DBMS implementations usually rely on specifying the columns a user might or might not see. The more intricate problem of restricting the user to several specified partitions of the data is usually left to the programmers and implemented through views (either simple or parameterized). This has long been acknowledged by the research community and the concept of query rewriting and "authorization transparent" queries has been introduced and discussed in great detail [1, 2, 4]. The scope of this paper is to introduce a framework for expressing the rules used in access control. The novelties introduced by this framework are: ability to choose between the type of access control (rewriting, checking), ability to use the structure of the query as input for the access control routines, ability to specify access control routines as a set of rules connecting specific users with specific constraints. The framework is part of the larger effort of defining AQL - a query language based on a code generation framework [5].

1. Introduction

The problem addressed by this paper is how we restrict the users of a database to specific sets of data. We have several answers to this problem and we will present some of them along with their advantages and disadvantages.

One of the solutions would be to create views that the users will use instead of the relations in the database. That means that there should be a view (simple or parameterized) for each user or group of users and that the client must be aware of which view to use for which user. Aside from that, the big draw-back here

Received by the editors: May 22, 2006.

1998 *CR Categories and Descriptors.* H.2.7 [**Information Systems**]: Database Management – *Database Administration - Security, integrity, and protection*; H.3.3 [**Information Systems**]: Information Storage and Retrieval – *Information Search and Retrieval - Information filtering* .

is maintainability: both changes of the underlying schema (database evolution) and changes in the security strategies (requirements evolution) result in all views needing to be rewritten.

A big advantage of using views is that they can introduce computed data as part of the schema. For example, a security restriction might be that the grade of a student should only be available as a logical typed column stating whether the student has passed or not.

A disadvantage of using views is that, in some cases, more relations need to be joined in order to compute the set of data the user is allowed to see. When a query uses several of these views, some of the tables might be joined several times needlessly, those consuming execution and optimizer time. The problem here is that when you define the view, you don't have information about the overall structure of the query - a problem that is addressed by our solution.

There are, of course, query rewriting algorithms that can do the job of replacing the relations with the corresponding views transparently for the user. This allows for the query writing and rights management issues to be orthogonal, and also might address some of the performance issues (since one of the primary usages of query rewriting is optimization). The disadvantages are that the views still have to be maintained and, more importantly, these features are not supported by commercial RDBMS's.

Another approach (that also takes the query rewriting path) is the Virtual Private Database feature of Oracle's 9iR2 RDBMS [2]. This feature allows for predicates to be added to the where clause of the query in which a relation appears. Although it allows for fine grained transparent access control and usage of data that is not in the original result set, it is still quite limited in terms of taking advantage of the query structure.

The approach introduced in [1] advocates that access control should not interfere with the expected behavior of a query since it is likely that it will produce unexpected results. The approach uses the notions of non-Truman models, authorization views and conditional validity in order to apply the declared restrictions. Although the inference rules proposed are capable of taking the query structure into consideration (to some extent), it still requires the creation and management of views and is not supported by commercial RDBMS's. Another difference of this approach is that it allows for transparent validation of queries but the query writing and access control are still tightly coupled and not at all orthogonal since no query rewriting takes place.

Our approach is based on representing the query as an XML document with a specified schema (defined in the AQL system). Although the system was first designed so that the user writes XML documents instead of queries, the XML document could just as well be generated by a translator based on the considered language's grammar.

Since our approach relies on modifying any aspect of the query, we are actually talking about a code generation framework, so we can also use here all the research on the topic of code generation as well [5, 6, 7].

2. AQL

For the purpose of self containment we will introduce here some issues regarding AQL, a declarative language that we developed for the broader topic of database evolution.

AQL is an instance of the code generation framework that we created and which is based on XML documents and transformations applied on those documents (either XSLT scripts or plug-ins written in any programming language).

From an architectural point of view AQL uses the compiler approach of multiple serial transformations. Each XML document contains tags that instruct the compiler which code generation strategy it should use. Once a strategy is identified, the steps from within that strategy are applied on the XML document and the result of the final step is the query statement.

This approach is similar to the one adopted in [3] and the main advantages over other generation techniques (string based approaches, translators) are the ease of evolution and separation of concerns.

We will continue with a simple example (a select statement). Keep in mind that even if the XML document we start with does not appear to be user friendly, it is created by either a graphical user interface or by an SQL parser. Let's consider the select statement which returns all the quantities that have been shipped from an OrderDetails relation. The starting document would then be:

```
<StrategyParam Name="SimpleSelect">
  <Extraction>
   <HeaderColumns>
     <ColumnRef Name="Quantity"></ColumnRef>
   </HeaderColumns>
   <Source SourceID="0">
     <Columns>
       <Column Name="Quantity" Type="num"/>
```

```
        <Column Name="IsShipped" Type="logical"/>
      </Columns>
      <Table>OrderDetails</Table>
    </Source>
    <Filters>
      <CreationFilter>
        <Expression>
          <ColumnRef Name="IsShipped"></ColumnRef>
        </Expression>
      </CreationFilter>
    </Filters>
  </Extraction>
</StrategyParam>
```

Through a series of transformations that will transform the filters, collapse header columns, check for computed fields, etc., the final result will be:

```
select Quantity from OrderDetails where IsShipped = 1
```

The transformation that makes it possible to use in a filter a single column of type bit (which is not possible in TransactSQL - the targeted SQL dialect) has the following elements:

(1) An XPath expression identifying all the nodes that require the transformation - in our case all the ColumnRef tags that are not children of an operator
(2) An XSLT transformation that creates the expression Column = 1
(3) A hint on what to do with the result of the previous statement - in our case the action would be to replace the input tag

This is just one of the simplest applications of AQL. For a more in-depth analysis of AQL and it's features please refer to [5].

3. Fine Grained Access Control Framework

The access control framework that we propose is based on AQL code generation framework. As depicted in the above example, each step of an AQL code generation strategy has access to entire query and to any level of detail, so it is possible to take the best decision regarding how the query should be rewritten.

3.1. Access control scenarios.

In order to understand the complexity of the access control problem we will consider some of the scenarios that could become a reality in a large application.

The simplest and most common access control scenario is to restrict a user to only see some of the columns in a relation (this is the scenario that is taken into account by most of the commercial RDBMS's today). In our framework this is possible by removing all the top-level references to columns that are not visible to the user.

A derived scenario is when the user is not allowed to see the actual values in the columns but is allowed to use them for other operations (such as joins for example). If we want to implement this in a RDBMS we will need to create views or stored procedures that encapsulate the joins and then expose the result to the user. The big problem here is that the user will be confined to the set of joins that we create and this could be a serious drawback in open database schemas. In our framework this can be implemented by restricting the space in which the ColumnRef tags are searched to the Header and filter sections.

Another common access rule that involves columns is the restriction that certain columns cannot be used unless they are aggregated. For example a user might not be allowed to see the actual grade of a student, but might be allowed to see the average grade for the entire year.

The more complex access rules are the ones referring to the rows that a user is allowed to see. In most cases this translates in appending a predicate to the where clause and maybe join some other relations in order to get the required data. However, there are some hidden complexities for this scenario. For example, a user might want to find the difference between his grade and the average grade. The SQL statement from which they would start in AQL (this is not a correct SQL statement in most of the RDMS we are aware of) is:

```
select grade - avg(grade) from Grades
```

This sentence would be translated by AQL as

```
select grade - AVG_GRADE
from Grades, (select avg(Grade) as AVG_GRADE from Grades)
```

If the user is not allowed to see the grades of all the students (but, for example only for the student with the StudentID of 1), we must not apply this filter when computing the average, because, in that case, the difference would always be 0. So, the correct translation is:

```
select grade - AVG_GRADE
from Grades, (select avg(Grade) as AVG_GRADE from Grades)
where Grades.StudentID = 1
```

As you can see, it is not always enough to just add a predicate to the where statement. The rule in this case is: "if the query uses the Grades relation and uses a column outside of an aggregate function then append the StudentID = 1 predicate to the where statement".

3.2. **Framework description.** Recent work on rights management has identified the problem of access control checks as being a typical application of crosscutting concerns and there are many examples of how to use Aspect Oriented Programming to solve this. Although AQL is not a typical object oriented language, we still can apply some of the AOP techniques (at least the ones using the compile-time code generation approach).

In figure 1 we showed the intended usage of the framework within a complex deployment. Besides enabling access control, we also get rid of problems like sending SQL code from the client (which, even if not recommended, is very often used in middle sized applications).

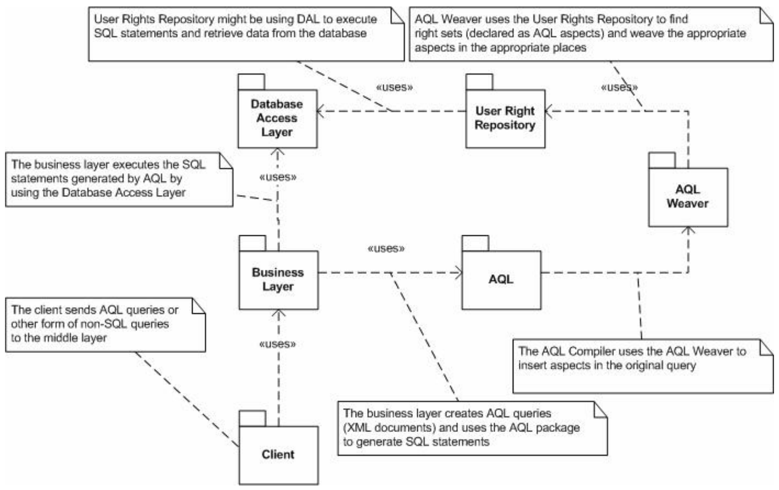
User Rights Repository might be using DAL to execute SQL statements and retrieve data from the database
AQL Weaver uses the User Rights Repository to find right sets (declared as AQL aspects) and weave the appropriate aspects in the appropriate places
«uses»
«uses»
Database Access Layer
User Right Repository
The business layer executes the SQL statements generated by AQL by using the Database Access Layer
«uses»
AQL Weaver
«uses»
Business Layer
AQL
The client sends AQL queries or other form of non-SQL queries to the middle layer
«uses»
The AQL Compiler uses the AQL Weaver to insert aspects in the original query
The business layer creates AQL queries (XML documents) and uses the AQL package to generate SQL statements
Client

FIGURE 1. Access Control Framework

One of the big advantages of using an XML based document to represent source code is the ability to execute complex queries on that source code. Many techniques used in refactoring are based on an intermediary, xml-based representation of the source code ([8]). The consequence in our case is that we can easily express elements of aspects by using an XML query language (in our case XPath).

An AQL Aspect is defined as AQLAspect = (Cond, JoinPoint, AspectCode) where:

- Cond is an XPath expression that evaluates to one or more XML nodes if the aspect is relevant for the AQL query. For example an horizontal expression on table A would have the condition ".//Table[@Name='A']" (which evaluates to one or more nodes if the table A is used in the query
- A join point is used to identify a set of points where the aspect code should be weaved. A JoinPoint is defined as JoinPoint = (BaseSelector, ExtensionSelector), where:
  - BaseSelector is an XPath expression that identifies the node which will be used to further identify the actual join point. In our previous example, the base selector should identify the select statement using table A: ".//Table[@Name='A']/parent::*"
  - ExtensionSelector is also an XPath expression that identifies the actual join point (relative to the node or nodes identified by the base selector). In our example the extension selector should identify the filters section of the select statement: "./Filters"
- AspectCode is an XML node that should be inserted at the join points previously identified. I our example it should specify a filter: "¡Filter¿Column = Value¡/Filter¿"

3.3. **Examples.** In order to better understand the usage scenarios of the framework and the extent of its capabilities, we created the following examples:

(1) Restrict access to the students of group 931. This could be done by adding a simple filter whenever the Students table is accessed.

```
A1 = (
".//Table[@Name='Students']",
(".//Table[@Name='Students']/parent::*", "./Filters"),
"<Filter> group='931' </Filter>")
```

(2) Restrict a teacher to the groups he has classes with. We will assume that the relation between Teacher and Groups is kept in a separate relation TeacherGroups. In order to do that we will need to join the students table to the TeacherGroups and Teachers table and then add a filter on the UserID column of the Teachers table.

```
A1 = (
".//Table[@Name='Students']",
(".//Table[@Name='Students']/parent::*", "."),
"<Join>
<Table Name=" TeacherGroups"/>
<Condition Value="Students.Group = TeacherGroups.Group"/>
 <Join>")

A2 = (
".//Table[@Name='Students']",
(".//Table[@Name='Students']/parent::*", "."),
"<Join>
<Table Name="Teachers"/>
<Condition Value="Teacher.TeacherID = TeacherGroups.TeacherID"/>
 <Join>")

A3 = (
".//Table[@Name='Students']",
(".//Table[@Name='Students']/parent::*", "./Filters"),
"<Filter> UserID = @UserID </Filter>")
```

In the second example we used the following aspects:

- A1 in order to insert a join to the TeacherGroups table
- A2 in order to insert a join to the Teachers table
- A3 in order to restrict the selection to the groups assigned to the current user

3.4. **Advantages and disadvantages.** The most obvious disadvantages to this approach are the fact that the user must use AQL and that performance might be negatively affected if each query is serialized in an XML document, transformations are applied and, at the end, the actual query is obtained.

The usage of AQL might not be a disadvantage since we are creating a complete solution that addresses ease of querying, schema and requirement evolution, access control, etc. However, if the system is already implemented and has to be translated to AQL just for the sake of access control, this is probably not the best solution.

As far as the algorithms are not implemented within the native query compiler and the targeted level of generality is high, there will always be a certain amount of performance penalties. In order to mitigate this, smart caches could be implemented, but only if the queries that are sent to the database are not very diverse.

There are also several advantages, some given by the fact that we are using AQL (we mentioned them earlier). However, as far as access control is concerned the biggest advantages are the fact that the entire query can be analyzed in order to decide how to restrict access and the fact that there is no limit on the type of transformations that are applied to the query.

Another issue that is broadly discussed by the research community is the fact that query rewriting will change the expected behavior of queries and might introduce very subtle problems in complex scenarios. This is true, but we consider that in the case of AQL there is such a big amount of query rewriting that the user must acknowledge and understand the inner workings of the rewriting algorithm, and, as such, is less likely to overlook important details.

4. Conclusions and further work

The framework that we present here has a very specific set of characteristics that set it apart from other frameworks that are available ([2], [3]). Because of this, the framework is going to address a particular type of applications, in which the access control patterns are very diverse and there are many clients writing queries and we want to implement security transparently, without affecting the number and complexity of the relations in the database. Because of this, we can identify two main directions in which research should continue:

(1) Defining complete, reusable design patterns for access control and implementing them in AQL. For this purpose we must consider a broad range of commercial applications and identify requirements, define best practices and implement them in a reusable and extensible fashion
(2) Continuing the development of AQL and especially improving its performance and range of available mechanisms. This is a very important task if this language is to be adopted in real-life scenarios and used in large applications.

References

[1] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, Prasan Roy, Shariq Rizvi: Extending Query Rewriting Techniques for Fine Grained Access Control, SIGMOD 2004 June 13-18, 2004, Paris, France.

[2] The Virtual Private Database in Oracle9ir2: An Oracle Technical White Paper http://otn.oracle.com/deploy/security/oracle9ir2/pdf/vpd9ir2twp.pdf.

[3] Galen S. Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai: Clearwater: Extensible, Flexible, Modular Code Generation, ASE'05, November 7-11, 2005, Long Beach, California, USA.

[4] A. Halevy. Answering queries using views: A survey. The VLDB Journal, 10(4):270-294, 2001.

[5] C. Costa: An XML evolution based approach to code generation, about to be published

[6] Alfred V. Aho, Mahadevan Ganapathi, Steven W. K. Tjiang: Code Generation Using Tree Matching and Dynamic Programming, ACM Transactions on Programming Languages and Systems, Vol. 11, No. 4, October 1989, Pages 491-516.

[7] Mahadevan Ganapathi, Charles N. Fischer: Affix Grammar Driven Code Generation, ACM Transactions on Programming Languages and Systems, Vol. 7, No. 4, October 1985. Pages 560-599.

Babes-Bolyai University, Faculty of Mathematics and Computer Science, Department of Computer Science, Cluj-Napoca, Romania

E-mail address: `costa@cs.ubbcluj.ro`