

Flexible and Efficient Access Control in Oracle

Ravi Murthy Eric Sedlar

Oracle Corporation

500 Oracle Parkway Redwood Shores CA - USA

{first.lastname}@oracle.com

ABSTRACT

A single model for access control across the database and application server tiers is crucial to ensure consistent secure access to data in all the tiers. In this paper, we present the common model for access control within Oracle database and application tiers which is based on the standard WebDAV ACLs (Access Control Lists). Further, we discuss the flexible mechanisms for defining ACLs and associating them with data and various optimization techniques for efficiently evaluating ACLs in large scale enterprise applications.

Categories and Subject Descriptors

H.2.7 [Database Management]: Database Administration--- Security, integrity and protection.

General Terms

Security, Performance, Management.

Keywords

Access control, Oracle security model, enterprise applications, ACL.

1. INTRODUCTION

A common security model across the application and database tiers is extremely critical to ensure that consistent security policies are applied to data operations at all the tiers. Traditionally, database systems have provided simple access control mechanisms in the form of schema, table and in some cases, row level authorizations [1] tied to database sessions, users and roles. However, they are generally perceived to be heavy weight mechanisms and are also hard to enforce in other tiers e.g. access control to a mid tier data cache. Hence, applications have tended to define their own security policies and enforce them at the application layer. As a result, the database server is unaware of the security policies and does not enforce them.

This scenario poses several significant issues. Firstly, the same underlying data may have to be accessed via many different applications and it is hard to reconcile different application security mechanisms. Secondly, many data warehousing, data mining, etc tools require direct access to the database (via SQL) and it is

impossible to enforce application level security during direct SQL access. Thirdly the lack of a common security framework makes it very hard to administer the policies and increases the risk of security holes. Finally, there is a significant performance impact by always evaluating access checks at the application layer.

A new common security model and framework has been defined and developed for use across the Oracle application and database tiers. In this model, the security policy defined on an object can be enforced consistently in any tier. In the database, the security framework is composed of three main parts: Lightweight users and roles (LWUR), Lightweight sessions (LWS) and Access Control Lists (ACL).

LWUR provides mechanisms to efficiently create and manage a large number of logical users and roles and/or synchronize them from a common enterprise directory e.g. LDAP. LWS provides the ability to cheaply attach and detach specific user credentials to the same underlying (heavy weight) database connection. This enables a small number of database connections to be effectively used to service database requests on behalf of a large number of users – the main difference here is that the database is actually aware of the end user identity and performs access control checks within the database based on the end user credentials. ACLs provide flexible mechanisms to define access privileges and assign them to users and roles for one or more objects. The database enforces the ACLs transparently and efficiently during SQL operations. ACLs are also evaluated in other application tiers as needed.

This paper focuses on the ACLs component of the overall security framework. The rest of the paper is organized as follows. Section 2 describes the ACL model in Oracle. Section 3 discusses the mechanisms for creating ACLs and associating them with data in the Oracle RDBMS. Section 4 delves into the details of evaluating ACLs during SQL operations and various optimization techniques to improve performance. Section 5 highlights related work and possible future enhancements, followed by conclusions.

2. ORACLE ACL MODEL

The Oracle ACL model [5] is based on IETF standard WebDAV ACL [3] and NFSv4 security model [4]. An Access Control List (ACL) contains a set of access control entries (ACEs), where each ACE specifies a principal and a set of privileges that are either granted or denied to that principal. ACLs are represented in an XML format. We define an XML Schema for ACLs and validate all ACLs against this schema. In addition to the obvious benefit of using XML while transferring ACLs across database and application tiers, this representation also allows application specific extensions.

2.1 Example ACL

The XML document below represents an ACL that contains two ACEs. The first ACE grants the *is:Approve* user-defined privilege to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMOD'07, June 12-14, 2007, Beijing, China.

Copyright 2007 ACM 978-1-59593-686-8/07/0006...\$5.00.

the *oa:Manager* principal. The second ACE grants the system privilege `<read-contents>` to a well known principal `<public>`. In XML, namespace URLs can be used to scope local names. The names of privileges and principals are XML namespace-qualified names. In this example, there are three namespaces – the default system namespace, the “is” (iStore) namespace that contains privileges pertaining to iStore data, and the “oa” (Oracle Applications) namespace that contains principals for the specific application. For clarity, the namespace definitions are omitted from the document below.

```
<acl>
  <ace>
    <grant>true</grant>
    <principal>oa:Manager</principal>
    <privilege>is:Approve</privilege>
  </ace>
  <ace>
    <grant>true</grant>
    <principal>public</principal>
    <privilege>read-contents</privilege>
  </ace>
</acl>
```

2.2 Principals

Every ACE contains a reference to a principal. Principals are used to represent end-users as well as groups of users (aka roles). These can be managed directly in the database or synchronized from external directories. A principal is represented by a XML namespace qualified name. Additionally, it can also be mapped to a URL resource.

An ACE refers to a principal to which the corresponding privileges are granted (or denied – see below for deny semantics). Alternatively, the ACE can refer to the entire set of users that are disjoint from the specified principal – by using an additional `<invert>` element wrapper as follows:

```
<ace>
  <grant>true</grant>
  <invert>
    <principal>oa:Manager</principal>
  </invert>
  <privilege>is:Approve</privilege>
</ace>
```

2.3 Privileges

A privilege controls access to a specific set of operations on a particular object. There are a set of system defined privileges that cover common operations such as select, insert, update, delete, etc. In addition, applications can define new privileges corresponding to application level operations e.g. `is:Approve` is a privilege required to execute the Approve operation. Application defined privileges are used to enforce function security i.e. they control the invocation of application functions. Note that these functional privileges are defined in an abstract manner and used within ACLs regardless of the specific language in which the function is implemented, the tier in which the function is invoked, etc.

New privileges can be created by aggregating a set of existing privileges. For example, the system *all* privilege is an aggregate of all the leaf system privileges. The definitions of privileges are managed as *security class* XML documents. A security class can define a set of leaf privileges and/or aggregate privileges. ACL documents reference the appropriate security class(es) to resolve and expand the privileges being used within the ACL.

2.4 Deny Semantics

An ACE can explicitly deny a privilege to the corresponding principal. The ACL evaluation processes each ACE in order, and terminates when either all the requested privileges have been granted to the current user (along with her enabled roles) or any of the requested privileges have been denied. In the former case, the ACL check returns a TRUE value, while the latter case returns FALSE. At the end of processing all the ACEs, if there are still some requested privileges that have not yet been granted, then again the return value is FALSE.

```
<ace>
  <grant>false</grant>
  <principal>oa:Salesperson</principal>
  <privilege>is:Approve</privilege>
</ace>
<ace>
  <grant>true</grant>
  <principal>oa:Manager</principal>
  <privilege>is:Approve</privilege>
</ace>
```

The set of enabled roles is the flattened list of roles derived from the explicit roles that were granted to the current user. This set is typically cached in the session context and updated whenever new roles are activated or existing roles are deactivated.

The ordering of ACEs affects the result of ACL evaluation. In the above example, the approve privilege is denied to salesperson and granted to manager. Consider a user who is both a salesperson and a manager. Given the order of ACEs in the above example, the approve privilege is denied to the user. However, if the order of ACEs is reversed, then the privilege is granted. This feature enables early termination of ACL evaluation as soon as all the requested privileges have been granted. Otherwise, the system would always have to evaluate all the ACEs completely. This *first-match* semantics is specified by WebDAV [3], NFSv4 [4] as well as Windows NT/XP.

2.5 ACL Inheritance

An ACL can be specified to inherit from another ACL. This allows security policies to be reused across objects and in different contexts. There are two forms of inheritance that are available – *extends-from* and *constrained-with*. These roughly correspond to the OR-combination and AND-combination of ACLs.

When an ACL A1 *extends from* another ACL A2, the ACEs in A2 are implicitly added to the end of the ACE list of A1. Thus the user is granted privilege if either A1 *or* A2 grants it. Note however that if A1 contains a deny ACE, then the privilege is denied (hence the ordering is important). This form of ACL inheritance is very useful for adding a few additional ACEs for one case without changing a widely used common ACL – and without duplicating the content of the common ACL.

Alternatively, an ACL A1 can be *constrained with* another ACL A2. In this case, the user is granted a privilege only if both A1 *and* A2 grant the privilege. For example, the ACL for “US-East” region orders may be constrained with the ACL for “US” orders which itself may be constrained with a common ACL for all orders.

The various advanced aspects of ACLs described above – including invert principals, user defined privileges, aggregate privileges, deny semantics and ACL inheritance – have all been proven to be necessary for modeling read-world security policies for enterprise applications. Hence, the Oracle security model provides native support for these features.

3. ACL USAGE

3.1 Creating ACLs

ACLs are XML documents conforming to the ACL XML schema. They can be inserted into the database using regular SQL, programmatic APIs in Java, C, PL/SQL or file-system protocols (such as FTP, HTTP/WebDAV, NFS, etc) provided by the XML DB repository [5,6]. When ACLs are inserted into the database, in addition to schema validation, the references to the principals and privileges are also validated.

ACLs can be identified by either their path in the repository (e.g. /apps/hr/acls/east-us_acl.xml – or a system assigned unique identifier.

There are multiple mechanisms for associating ACLs with data objects. Also, the same data object may be associated with more than one ACL. Finally, ACLs can be evaluated directly by the application without associating them with any data objects. These are described in the following sections. For simplicity, a row of a table is considered to be a data object. There are mechanisms for creating data objects from sets of rows from multiple tables but their discussion is outside the scope of this paper.

3.2 Dynamic ACL Association

A set of ACLs can be associated with one or more rows of a table dynamically by specifying an *instance set* – which is a list of ACL identifiers and corresponding table predicates. The ACL applies to all rows that satisfy the corresponding predicate. If a row satisfies more than multiple predicates, then all corresponding ACLs apply to the row – in the order in which they were specified in the instance set.

The example below shows an instance set for the Orders data table. Assume that the Orders table contains columns Region and Amount in addition to other columns. ACLs A1 and A2 are associated with all rows satisfying the predicates P1 and P2 respectively.

Table 1- Instance set for "Orders" table

ACL Identifier	Predicate
A1	(P1) Region = “West”
A2	(P2) Amount > 10000

Note that the predicates get evaluated only when ACLs checks are needed – hence the association between the ACLs and the data objects is completely dynamic. When the data is updated, the ACL

associations could implicitly change based on the instance set predicates.

The predicates specified in an instance set can be any valid WHERE clause condition – including single or multiple column conditions, exists sub-queries over one or more other tables, etc. It can also reference the current session context (SYS_CONTEXT) that contains current user settings such as language, locale, etc and also any application defined parameters.

Instance sets can also be **parameterized** i.e. the predicates can reference variables. The bindings of the variable to different values are provided within the ACLs. There are a couple of reasons for decoupling the predicate specification from the value bindings. One is that it reduces the number of explicit predicates in the instance set. Instead of listing multiple predicates that differ only in the constant values, they can be combined into a single parameterized predicate. A second and more important reason is that it allows for the set of users who can create predicates to be different from the users who can specify values for variables. This separation is very crucial because predicates need to be created carefully to ensure good performance e.g. using indexed columns, etc. Typically we expect that a few expert users who are familiar with the underlying schema are allowed to create predicates whereas a large set of business users can specify the variable values. For example, an expert user creates the predicate Region=\$r. A business user sets up the possible values (East, West, etc) and their corresponding ACLs.

3.3 Static ACL Association

ACLs can be statically associated with a data object by invoking system interfaces. In this case, the ACL identifiers are stored directly with the data row. This mechanism provides complete freedom to the application and/or administrators to set ACLs on data objects.

Another mechanism for static ACL association is by marking an instance set as “static”. In this case, the predicates are evaluated when the instance set is created, and the resulting ACL identifiers are stored statically with the data rows. Also when additional data rows are inserted or existing rows are updated, the predicates are re-run and corresponding ACL identifiers are stored with the row. For example, if the instance set shown in Table 1 is marked as “static”, the following ACL associations get stored statically in the Orders table.

Table 2 - Static ACL Association for “Orders” table

OrderID	...	Region	Amount	ACLIDS (system column)
1	...	West	4000	A1
2	...	East	12000	A2
3	...	West	15000	A1,A2

Static ACL association often leads to improved performance of access checks but with additional overhead during row updates. The detailed analysis of ACL performance is presented in a later section. Note that static ACL association does not imply that it cannot change – it means that the association is evaluated and persisted in the table as opposed to dynamically evaluating the associations at query time.

3.4 SQL DML and Query Operations

Users can submit regular SQL DML and query operations. The system detects operations against *secured* tables i.e. tables with associated instance sets and/or ACLIDS system column, and implicitly transforms the SQL to perform the necessary access checks. The transformation involves attaching an additional predicate using a new CHECK_ACL SQL function.

3.4.1 CHECK_ACL Function

The new CHECK_ACL SQL function provides the ability to check access privileges within the context of a SQL statement. The function takes the following input parameters:

- Set of requested privileges – may include system defined as well as application defined privileges
- ACLIDS – column holding zero or more ACL identifiers
- Zero or more pairs of (ACLID, predicate). If the predicate evaluates to TRUE, then the corresponding ACL is applied, else it is ignored. The predicates are expressed as boolean scalar sub-queries (as shown in example below).

The function returns TRUE if all the requested privileges have been granted by the applicable list of ACLs, FALSE otherwise. When a query is submitted with the CHECK_ACL function in the WHERE clause, all rows that satisfy the security check are returned. The other rows that fail security check are not returned (note: no errors are raised in this case).

3.4.2 Implicit System Access Checks

When a user submits a SQL query against a secured table, the system implicitly attaches a CHECK_ACL predicate to perform the access checks. The requested privilege depends on the nature of the SQL statement. For example, SELECT query requires <read-contents> system privilege. If the table has dynamic ACL associations, the ACLID, predicate pairs are supplied as inputs to CHECK_ACL. If the table has static ACL associations, then the ACLIDS system column is supplied as an input. Note that the same table could have been set up with both static and dynamic ACL associations.

Let's consider the dynamic ACL association example in Table 1. The user query Q1 is transformed by the system into Q2. Since Oracle does not support boolean expressions as direct arguments to functions, we use integer value 1 to imply TRUE and NULL is interpreted as FALSE. The predicates are expressed as scalar sub-queries using the system table *dual* which is a special single row, single column table in Oracle. When the predicate is true, the scalar sub-query returns 1. If the predicate is false, the scalar sub-query returns NULL.

Q1: *select * from Orders*

Q2:

```
select * from Orders where  
CHECK_ACL('<read-contents>', NULL,  
    A1, (select 1 from dual where Region = "West"),  
    A2, (select 1 from dual where Amount > 10000)  
)
```

Now consider the static ACL association example from Table 2. The same user query Q1 would be transformed by the system into Q3.

Q3:

```
select * from Orders where  
CHECK_ACL('<read-contents>', ACLIDS)
```

In case of hybrid scenarios where the table has both static and dynamic ACL associations, the transformed query has both sets of ACL parameters.

3.4.3 Explicit Application Access Checks

Applications can check for additional access privileges during the SQL operation (also called *functional* security). For example, an application module may wish to only select "open" orders that the current user can approve. The CHECK_PRIVILEGE list of additional privileges can be attached to the table reference in the FROM clause. For example, the application would submit query Q4 which would then be transformed to Q5 (assuming static ACL association).

Q4:

```
select * from Orders CHECK PRIVILEGE ('<is:Approve>')  
where Status = "Open"
```

Q5:

```
select * from Orders where Status = "Open" AND  
CHECK_ACL('<read-contents>,<is:Approve>',ACLIDS)
```

The transformed query checks for the application requested privileges in addition to the default system privileges. Applications can also evaluate ACLs directly (i.e. outside SQL) by invoking programmatic interfaces in Java, C, PL/SQL, etc. It is important to note that the same ACLs can be evaluated in all these different environments and yield the same semantic behavior. This is the key benefit of a single unified model across all tiers.

4. ACL EVALUATION

The performance of evaluating CHECK_ACL function is extremely critical to the overall efficiency of the application. There are several compile-time and run-time optimizations to improve the performance of ACL evaluation in the various scenarios. This section initially describes the baseline evaluation of the function followed by a discussion of the different optimizations. These optimizations are implemented within the core database kernel and are triggered by the presence of CHECK_ACL function.

4.1 Functional Evaluation

The baseline evaluation of the function initially processes the static ACLIDS (if any). For each such ACL, it processes its ACEs in order. If the specified principal matches the current user (given the current set of enabled roles) and the specified privilege matches any of the requested privileges, this privilege is subtracted from the requested set. However in case of a matching deny ACE, the processing immediately terminates (see case 3 below). After processing all the static ACLIDS, it then proceeds to the ACL, predicate pairs. For each such pair, the predicate is evaluated. If the predicate returns TRUE, then the corresponding ACL is processed as described above.

The processing terminates when any of the following conditions is satisfied:

1. the requested privilege set is empty. Result is TRUE.
2. all the ACLs have been processed and some requested privileges remain. Result is FALSE.
3. a requested privilege has been denied. Result is FALSE.

4.2 ACL and Result Caching

The performance of the functional evaluation is significantly improved by adding two levels of caching. A shared cache (across database sessions) stores the set of most recently accessed ACLs in an internal optimized format. A session private cache stores the set of privileges granted/denied to the current user (and current set of enabled roles) by each of the most recently accessed ACLs – in an internal compressed bitmap format.

In the best case, an ACL evaluation involves looking up the grant/deny privilege bitmaps from the private cache and performing AND-operations against the requested privilege bitmap. This operation is typically very fast - less than one thousand machine instructions.

In case of a miss in the private cache, the ACL is looked up from the shared cache (or loaded from the database if necessary), and evaluated to obtain the bitmap of the granted and denied privileges.

The session private cache needs to be updated (is typically flushed) when any role is enabled or disabled. The shared cache is updated whenever an ACL is updated. In practice, these operations are relatively infrequent compared to the number of ACL evaluations. Hence the benefits of caching easily outweigh the extra overhead of keeping the caches consistent.

4.3 Dynamic ACL Optimizations

4.3.1 Query Rewrite

A cost-based query rewrite optimization is available for queries with dynamic ACL association. Assume the original query contains CHECK_ACL function shown below:

Q6 :

*select * from T where*

CHECK_ACL(privs, NULL, A1, P1, A2, P2, ...An, Pn)

This query can be rewritten as follows. During query compilation, each of the ACLs A_i is evaluated. If the ACL grants any of the requested privileges, the corresponding predicate P_i is put on a OR-chain. At the end of evaluating all the ACLs in the list, the predicates on the OR-chain are linked via AND condition to the original CHECK_ACL predicate. Assuming that the ACLs $A_i, A_j, .. A_k$ granted some of the requested privileges, the above query Q6 can be in general rewritten as follows:

Q7:

*select * from T where*

(P_i OR P_j ... OR P_k) AND

CHECK_ACL(privs, NULL, A1, P1, A2, P2, ...An, Pn)

In general, the newly added OR-chain can be used as a pre-filter to eliminate many of the rows from the table and the original CHECK_ACL condition is applied to the remaining rows. There are

a few reasons why the original CHECK_ACL function needs to be retained (in the general case) :

- The set of requested privileges may be granted by different ACLs i.e. spread over multiple ACLs. In this case, the corresponding predicates have to be locally AND-ed to achieve the desired semantics.
- One of the ACLs may deny a requested privilege, but a subsequent ACL may actually grant it. In this case, the latter predicate would have to be AND-ed with the negation (NOT) of the former predicate. This is further complicated in the presence of multiple requested privileges.

To accommodate these semantics without a post-filter, we would need a very complicated combination of predicates – and would defeat the intended optimization. Hence, the original CHECK_ACL function is retained as a post-filter – but it is omitted if each ACL in the list satisfies the conditions below :

- Either grants the entire set of requested privileges or does not grant any of them and
- does not deny any of the requested privileges.

In case the above conditions are true, the rewritten query is simplified to Q8.

Q8:

*select * from T where*

(P_i OR P_j ... OR P_k)

The decision to rewrite the query is made by estimating the costs of the original query and the rewritten query. Note that the rewritten query can potentially use multiple column indexes and combine them using bitmap OR operations. For example, consider query Q2. Assuming that ACL A1 does not grant (or deny) the requested privilege, but A2 does grant the requested privilege, the query Q2 can be rewritten to :

Q9:

*select * from Orders where Amount > 10000*

The above query can benefit from index on Amount column and will perform significantly better than the original query.

4.3.2 Cursor Sharing Implications

Queries are typically parsed, optimized and compiled into a shared cursor object which is cached for an extended period of time. Repeated executions of the query (perhaps with different values for bind variables) pick up the same compiled cursor and execute it – thereby avoiding the costly compilation phase.

If the optimizer decides to rewrite the CHECK_ACL query as described above, it has implications on the share-ability of the compiled cursor. Since the result of evaluating the ACLs has been exploited to perform the rewrite, the system needs to ensure the validity of sharing the compiled cursor by the following steps:

- Place dependencies from the compiled cursor on the ACLs specified within CHECK_ACL. This ensures that the compiled cursor will be invalidated if any of the pertinent ACLs are updated.
- Store the list of ACLIDs and the expected results from evaluating the ACLs for the requested privileges. During cursor lookup, the ACLs are reevaluated for the current user and current set of enabled roles. If the results match the

expected results, then the compiled cursor can be used. Else the system proceeds to check against other available compiler cursors. In the worst case, the query may have to be recompiled.

4.3.3 Predicate Re-ordering

There are two possible re-ordering optimizations to the functional evaluation algorithm described in Section 4.1. (Note that these optimizations are independent of the query rewrite and can be applied either to the CHECK_ACL functional evaluation in case of no-rewrite or the post-filter in case of rewrite) Consider the original query:

```
select * from T where
```

```
CHECK_ACL(privs, NULL, A1, P1, A2, P2, ...An, Pn)
```

If the cost of evaluating the predicate is lesser than that of ACL evaluation, then it is optimal to first evaluate the predicate, and based on the result, evaluate the ACL if necessary. On the other hand, if the cost of ACL evaluation is lesser than predicate evaluation, then we can re-order the steps – and evaluate the ACL first. Only if the ACL grants or denies any of the requested privileges, the corresponding predicate is evaluated.

There is another possible re-ordering optimization. Assuming that the ACLs don't deny any of the requested privileges (and that ACL evaluation is cheaper than predicate evaluation), all the ACLs can be evaluated first and then the appropriate predicates are evaluated – in the order of their increasing cost.

4.4 Static ACL Optimizations

A query on a table with static ACL association is optimized in two steps. Firstly, appropriate indexes are created on the ACLIDS column as well as the system ACL table. Secondly, based on estimated selectivity and cost, the query is rewritten to attach a primary filter predicate that can exploit the indexes. These steps are described in the following sections.

4.4.1 ACL Indexes

An index is created on the system ACL table to allow efficient lookup of all ACLs that reference a given principal. Since ACLs are modeled as XML documents, the XML Index technology [6] is used to create this index. A multi-valued index is created on the ACLIDS column of the user table to allow efficient lookup of all data rows that are protected by a given ACLID. With these two indexes in place, the query can be rewritten into an efficient index based evaluation.

4.4.2 Query Rewrite

The query against a table with static ACL association can be rewritten to use the indexes in the following manner.

1. Lookup the index on the system ACL table to retrieve all the ACL identifiers that reference the current user or one of the current enabled roles. This set is referred to as the set of all *relevant* ACLs.
2. For each of the *relevant* ACLs, the index on the ACLIDS column of the user table is accessed, and the corresponding data rows are identified.

3. For each of the identified data rows, the CHECK_ACL function is applied as a post-filter (due to the reasons previously mentioned).

For example, the query Q3 can be rewritten to use the indexes as follows. For clarity, the rewritten query is expressed using the *relevant_acls* view (which represents the result of step 1 above).

Q10 :

```
select * from Orders where
```

```
CHECK_ACL('<read-contents>', ACLIDS) AND
```

```
exists(select 1 from TABLE(ACLIDS) a
```

```
where a.column_value IN
```

```
(SELECT ACLID FROM relevant_acls))
```

4.4.3 Selectivity and Cost Estimation

The optimizer decides to rewrite the original query based on its estimate of the overall cost of the original and rewritten queries. A good estimate of the CHECK_ACL predicate cost and selectivity is necessary to obtain a realistic query cost. However, it is tricky to estimate the selectivity of CHECK_ACL function because it depends on several factors including the current user and set of enabled roles.

The following statistics are gathered on the ACLIDS column of the user table.

1. For each user or role, the number of rows to which the principal has been granted any privilege. For efficiency reasons, it is sufficient to keep this information only for those principals for whom this number > X % of the total number of rows (we have chosen X to be 10 in our system). By exclusion, we can assume that all other principals can only access < 10% of the rows.
2. Average size of the ACLIDS column – This provides information about the average number of static ACLs that are evaluated for each row.

Based on the above statistics, the selectivity and cost are estimated as follows.

$$\text{Selectivity} = \frac{\text{Max}(\#rows-accessible(U), \#rows-accessible(R1), \dots)}{\text{Total number of rows in table}}$$

- U = current user
- R1, ... = set of enabled roles
- #rows-accessible(p) = Number of rows on which the principal p has been granted any privilege. If the principal occurs as part of ACLIDS column statistics, then use the value from stats. Else, assume default of (0.1 * Total-rows).

The above estimate is still rough because it does not yet account for the specific privilege requested by the query – but instead estimates an average across all privileges. There is a fundamental tradeoff between the size of maintained statistics vs. the accuracy of the estimates. It is possible to gather privilege level statistics and combine it with the user/role level statistics to obtain a better estimate. However, in practice, the above technique has proven to be adequate.

The cost of CHECK_ACL function is estimated based on the average number of ACLIDs per row and the expected hit/miss ratio in the session private result cache and the ACL shared cache.

5. RELATED WORK

The ACL functionality described in this paper is one component of the overall security framework in Oracle. This section briefly describes the other aspects of the framework as well as additional ACL features that are outside the scope of this paper.

LightWeight Users and Roles (LWUR) provides the ability to create and manage large user and group populations. The same user and role information is available consistently in both the database as well as the application server tiers. Hence, an application could choose to evaluate its ACLs in the database tier using the functionality described in this paper – or could evaluate the ACL in the application tier. The key aspect is that the ACL model and semantics are identical in both case and hence yield the same behavior. One common scenario is that data objects are cached in the middle tier – but access to these objects need to be controlled in the same manner as within the database. When a data object is fetched from the database, a new SQL function GET_ACLIDS can be used to also retrieve the corresponding ACL identifiers for the object. Both the data and its ACLs are cached in the middle tier. When the data object is accessed, the corresponding ACLs are checked – thereby ensuring consistent access control.

In this paper, we described techniques for associating ACLs with data objects. There are additional mechanisms for specifying ACLs on partial data objects i.e. column/attribute security. When the data object is accessed, different ACLs may be checked for different fields, and the returned object consists of only field values that have been granted. Column masks can be provided as default values for the data fields that are not accessible by the user. E.g. Show xxx-xx-xxxx as the social security number field if it is not accessible by the user. Also, the security mechanism described in this paper can be applied on the security metadata itself. For example, security can be enforced on the ACL documents by creating and associating ACLs on the ACL table.

The CHECK_ACL function returns a boolean TRUE/FALSE result. When the function returns FALSE, in many cases it is desirable to find out what privileges have not been granted and further, obtain some indication about the underlying reasons. For example, the if the role internetUser is enabled (indicating that the user is outside the firewall), then certain operations that are usually available may be prohibited. In such cases, it is useful for the application to return an informational message to the user e.g. “You cannot access this data or operation because you are outside the firewall” – instead of a cryptic “Operation prohibited” message. However, the algorithm for identifying the underlying reason for access check failure (e.g. absence or presence of a specific role) is inherently ambiguous and works on a best-effort basis. Also, many applications may turn this feature off due to security concerns.

There has been ongoing research into security mechanisms within databases for many years [9]. Some of them [8,10,14] focus on the role based access control model. Our ACL model uses roles as principals but also augments it with system and application defined privileges, deny semantics, principal inversion and ACL inheritance. Further it has flexible mechanisms for associating ACLs with data objects. The dynamic (instance set) ACL association is similar to the authorization view mechanisms [7] in

that they both use arbitrary SQL predicates as filtering criteria for access control. However, our model decouples the association mechanism (could be dynamic or static) from the policy itself viz. ACLs. Some of those query rewriting techniques [7] could be used within our system for dynamic ACL query rewrites. The problem of returning informational messages during access check failures has been studied in [11, 12].

Recently, standards such as XACML [13] have arisen that allow business users to express security policies via business rules. These rules specify who can perform what actions on which resources. It is primarily based on an XML data model and uses XML Schema datatypes and XPath like functions to express the rules. In our ACL model, rules are expressed using SQL expressions (via instance sets) against a mostly relational data model. Also, our model had many additional features such as static and dynamic ACL assignments, parameterized instance sets, etc. We are investigating the possibility of mapping between the ACL and XACML models.

6. CONCLUSIONS

This paper presented the common security model for Oracle Applications across the database and application server tiers. It discussed the details of Access Control Lists, one of the key components of the framework, flexible mechanisms for associating ACLs with data and several database optimizations to improve the performance of ACL evaluation.

Real world enterprise class applications require the broad set of security features as well as the high degree of performance and scalability made possible by the optimization techniques discussed here. The benefits of a common security model across all the tiers include consistent access control, simplified administration, reduced risk of security holes, direct SQL access possible by all tools and improved performance. The technology described in this paper will be at the core of the next generation of Oracle enterprise applications.

7. ACKNOWLEDGMENTS

Our thanks to all the members of the Fusion Security and XML DB development teams for their help and support.

8. REFERENCES

- [1] Oracle Database Security Guide, Oracle RDBMS 10gR2.
- [2] HTTP Extensions for Distributed Authoring - WebDAV. RFC 2518. <http://webdav.org>
- [3] WebDAV Access Control Protocol. RFC 3744. <http://webdav.org/acl>
- [4] Network File System (NFS) version 4 Protocol. RFC 3530. <http://www.ietf.org/rfc/rfc3744>
- [5] Oracle XML DB Developer's Guide, Oracle RDBMS 10gR2.
- [6] R. Murthy, et al. Towards an enterprise XML architecture, SIGMOD 2005.
- [7] S. Rizvi, A. Mendelzon, S. Sudarshan, P. Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. SIGMOD 2004.
- [8] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. IEEE Computer, 29(2), 1996.

- [9] S. Castano, M. Fugini, G. Martella, and P. Samarati. Database Security. Addison-Wesley, 1995.
- [10] V. Gligor. Characteristics of role-based access control. In ACM Symp. on Access Control Models and Technologies, 1996.
- [11] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In ICDE, pages 339–347, 1989.
- [12] A. Motro. Panorama: A database system that annotates its answers to queries with their properties. Journal of Intelligent Information Systems, 7(1):51–73, Sept. 1996.
- [13] OASIS eXtensible Access Control Markup Language (XACML) TC. <http://www.oasis-open.org/committees/xacml>
- [14] D.F. Ferraiolo, D.R. Kuhn, R. Chandramouli, Role Based Access Control, Artech House, 2003.