Enhancing and Simplifying Data Security and Privacy for Multitiered Applications

Walid Rjaibi[1]

IBM Canada Laboratory, 8200 Warden Avenue, Markham Ontario L6G 1C7 &
Department of Computing and Mathematics, Manchester Metropolitan University, Manchester M15 6BH

Mohammad Hammoudeh[2]

Department of Computing and Mathematics, Manchester Metropolitan University, Manchester M15 6BH

[1] Corresponding author's e-mail: wrjaibi@ca.ibm.com.
[2] Author's e-mail: M.Hammoudeh@mmu.ac.uk

Preprint submitted to Journal of Parallel and Distributed Computing

Abstract

While databases provide capabilities to enforce security and privacy policies, two major issues still prevent applications from safely delegating such policies to the database. The first one is the loss of user identity in multitiered environments which renders the database security features of little to no value. The second issue is the unsafe coexistence between the security capabilities and fundamental database tenets which creates data leakage vulnerabilities. This paper proposes extensions to database systems to allow applications, such as those used in managing the operations of energy clouds, to safely delegate the security and privacy policies to the database. This delegation reduces complexity for applications and improves overall data security and privacy. Our performance evaluation shows that almost all the TPC-H queries perform the same or better when the security policy is enforced by the database. For the set of queries that performed better, the improvement observed ranges from 8 to 68%.

Keywords: Applications, Security, Privacy, Databases, Energy Cloud

1. Introduction

Transitioning to an energy cloud is a significant challenge. This network of networks needs to be optimized to reduce cost and ensure overall security. This would also include reducing the complexity of building applications to manage the operations of energy cloud such as those related to customer relationship management. This type of applications has traditionally become quite complex partly due to the cost of implementing data security and privacy rules within the application logic itself. Fig. 1 shows the architecture of a classical 3-tier application, where the end user browsers, the application server and the database server represent the first, second and third tier respectively.

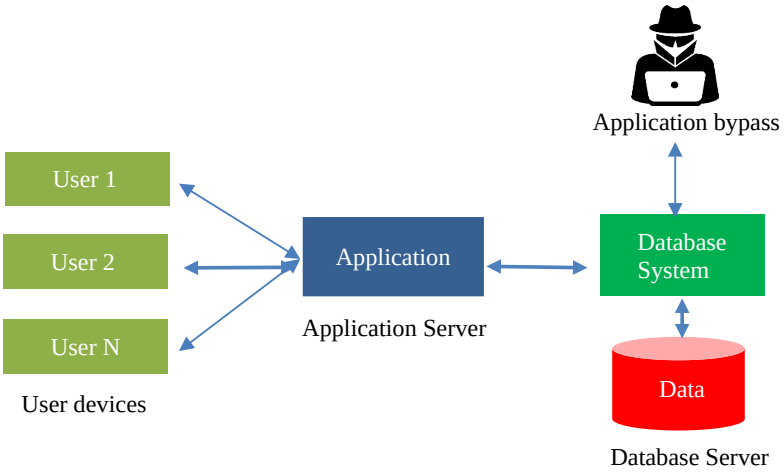

Fig. 1. Classical 3-tier application architecture and related security issues.

Under this model, end users access the application to perform tasks related to their job. The application authenticates such users to ensure they are authorized to use the application. To meet the needs of the end users, the application makes a connection to the database using a generic user ID identifying that application to the database. To ensure that the right content of the database is returned to the right users, the application logic typically includes a fine-grained authorization layer to do the appropriate level of

data filtering. This layer is usually implemented in one or a combination of these two options:

- The application builds the SQL queries in such a way that they include the appropriate predicates and functions to filter out and mask the table data as appropriate.
- The application builds a set of database views which perform the appropriate level of data filtering and routes the SQL queries to the appropriate views based on user identities.

Besides burdening the application with the task of implementing fine-grained authorization, this model also suffers from other security drawbacks including:

- The approach is not data centric. This means that the intended security policy is not enforced when the application is bypassed. An example of such bypass is when the application administrator chooses to abuse the application's database user ID to access the database directly. This is particularly important in today's world where internal threats are as concerning as external threats [1], [2].
- Over granting of database privileges. The application's database user ID is typically granted the privileges of a database administrator so that it can be used to do all things on behalf of all users. This means that when such user ID is abused, the consequences to the organization can be severe.
- Loss of end user identity at the database level. This is a consequence of the application doing all database accesses on behalf of all users using a single user ID. This makes it impossible to leverage database auditing to hold end users accountable for their actions. It also prevents the application from delegating the fine-grained authorization policy to the database as the user ID is lost at that level.
- Unnecessary exposure of the security policy to application developers.

We contend that applications complexity can be reduced by delegating the fine-grained authorization task to the database system. We also contend that this delegation will additionally address the security concerns raised above and enable applications to better adhere to compliance mandates such as the European General Data Protection

Regulation (GDPR) [3] and the Payment Card Industry Data Security Standard (PCI DSS) [4].

The crux of our contribution is the design of a holistic fine-grained database authorization approach which allows organizations to reduce the complexity of their applications and improve overall database security. We have also implemented the solution in a commercial database system (IBM DB2) [5]. Our approach improves over the state of the art as follows:

- Fine-grained authorization coexists in harmony with fundamental database tenets such as performance and integrity so that organizations are not forced to make compromises either on the security side or on the database side.
- Applications can safely delegate the security policy to the database system by leveraging the trusted context concept to propagate user identities to the database system, thus extending the value of fine-grained database authorization to multitiered applications.
- Organizations can leverage the trusted context concept to ensure that the application's database user ID cannot be abused by malicious entities who may want to leverage that user ID for accessing the database outside the scope of the application (i.e., application bypass).

The rest of this paper is organized as follows. Section 2 reviews the related work. Section 3 describes our fine-grained database authorization model. Section 4 introduces our trusted context concept which addresses the loss of user identity problem in multitiered environments. In section 5, we discuss how the new concepts introduced safely coexist with core database tenets. Section 6 describes the performance evaluation of our fine-grained database authorization model. In Section 7, we discuss a banking use case and show how our solution meets its requirements. Lastly, Section 8 summarizes our approach and outlines our future work.

2. Related work

Traditionally, fine-grained authorization in database systems has been implemented using the concept of database views [6]. Like database views, our approach is an

extension to SQL and is declarative in nature. Administrators are not expected to write any code to implement the fine-grained authorization rules. However, our solution improves over database views in two main ways. First, our approach defines the row and column controls directly on the database tables themselves. This means that the row and column authorization is always enforced regardless of whether the table is accessed directly or indirectly through a database view. In contrast, when implementing fine-grained authorization using views, the row and column authorization is enforced only when the access is made through those views. In other words, views do not provide any protection when the underlying tables are accessed directly. Additionally, our approach introduced the notion of trusted context to enable user identity propagation in multitiered environments so that applications can safely delegate fine-grained authorization to the database system.

Oracle Virtual Private Database (VPD) was, to the best of our knowledge, the first database system to introduce a fine-grained authorization model that improves over traditional database views [7] and is the closest to our work. There are however some important differences between Oracle VPD and our approach. First, the Oracle VPD approach is not declarative. It requires the administrator to code a PL/SQL program which computes a predicate string that is appended to any SQL statement accessing the table with which the PL/SQL program was associated. This also limits the benefits of SQL statements caching only to situations where the PL/SQL program is guaranteed to return the same results for all users. Our approach does not limit the benefits of SQL statements caching because it does not change the SQL statement text itself. Oracle VPD also includes the notion of an Application Context which can be used by applications to pass information to the database system such as a user ID in a multitiered environment. An Application Context is a set of name-value pairs the Oracle database systems stores in memory. Our trusted context concept provides a more robust framework for propagating user identities in multitiered environments as it first requires the establishment of a trusted relationship between the database system and the application before propagating a user ID is allowed. It also provides more control on which specific user IDs are allowed for propagation as well as the ability to associate the application's privileges with the trusted context only so they cannot be abused elsewhere.

The Row Level Security (RLS) and Dynamic Data Masking (DDM) capabilities in Microsoft SQL Server are conceptually similar to our row permission and column mask concepts [8]. But there are some important differences between the two approaches. First, the SQL Server DDM is static in the sense that the user either has access to the actual value in the column or a masked value thereof. The column mask concept in our approach is dynamic in the sense that the decision of whether the user sees the actual value, or a masked value is determined dynamically based on the conditions expressed in the column mask definition. Additionally, the SQL Server RLS requires the administrator to go through a two-step process: They first need to create a function which returns a filtering predicate, and then create a policy on the table to apply that predicate. In our approach, this is all done in a single step using the row permission concept. The user identity propagation in multitiered environments is supported through an application context concept similar to the Oracle VPD one discussed above.

The Vertica Row Access Policy (RAP) and Column Access Policy (CAP) concepts enable administrators to enforce access to table data at the row and column level respectively. The Vertica SQL syntax is very similar to ours. However, and to the best of our knowledge, the Vertica solution does not discuss how it enables user identity propagation in multitiered environments. Additionally, the Vertica solution does not show any performance evaluation to contrast implementing the fine-grained authorization rules within the database versus within the application.

The Sybase Row Level Access Control (RLAC) enables administrators to restrict access to data rows in a table by defining an access rule and binding it to a specific column of the table [9]. When a table is accessed, the access rules in place are automatically enforced by incorporating them into the query at compilation time. Our approach differs from the Sybase RLAC capability in several ways. First, RLAC is limited to row level access control only while our approach covers both the row and column level. Also, to the best our knowledge, the Sybase RLAC does not discuss how it enables user identity propagation in multitiered environments.

The fine-grained authorization model presented in [10] is also a declarative SQL model like ours. But there are some differences between the two approaches. The first difference is fairly minor. They have extended the GRANT SQL statement to give administrators the tools to define row and column authorization rules while our

approach introduced these constructs independently of the GRANT statement. However, the work presented in [10] did not cover user identity propagation in multitiered environments. It assumed it was taken care of through a method similar to the application context concept in Oracle VPD. Lastly, their work did not include any performance evaluation to contrast implementing the fine-grained authorization rules within the database versus within the application.

The fine-grained authorization approach discussed in [11] is also a declarative SQL model but there are some key differences with our approach. First, the focus of the work in [11] is on privacy policies. They introduced row and column restriction concepts for the purpose of being able to map privacy policies to them so the database system can automatically enforce privacy policies. It did not cover user identity propagation in multitiered environment. Also, the model described in [11] did not include any performance evaluation to contrast enforcing the privacy policy within the database versus within the application.

The model described in [12] can be regarded as a special form of fine-grained authorization. The focus of this work is more around introducing a flexible mandatory access control model which addresses some of the shortcoming of classical Multilevel Security [13]. It is a declarative SQL model and also ensures the security predicates are executed before any potentially unsafe predicates to prevent data leakage. However, it did not introduce the concept of secure functions as we did in this paper, so security predicates are always executed first even if that does not make sense from a performance perspective. Lastly, the approach discussed in [12] did not cover user identity propagation in multitiered environments.

Besides security built into database systems themselves, the importance of protecting databases has also led to the emergence of external database security tools. The leading tools in this context are Guardium and Imperva [14]. These tools can be thought of as complementary to our solution as they focus more on database auditing, compliance reporting and analytics on auditing data as opposed to fine-grained database authorization.

3. Fine-grained database authorization model

We extend the SQL table privileges model with two new concepts: *Row permissions* and *column masks*. Row permissions and column masks implement a second layer of security on top of table privileges. When a table is accessed, the privileges layer determines whether or not the table can be accessed. Next, row permissions are applied to decide what specific set of the table rows the user is authorized to access. Lastly, column masks are applied to figure out whether the user is allowed to see the actual value in a column or a masked value thereof. For example, row permissions ensure that when a doctor queries the patients table, they only see rows that represent patients under their care. On the other hand, a column mask on the phone number column ensures that the doctor sees only phone numbers for patients who consented to share their phone numbers with them. Fig. 2 shows our model as an extension to the SQL compiler.

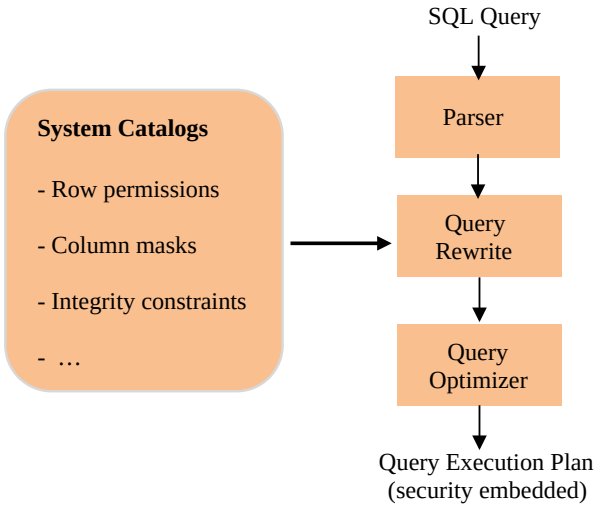

Fig. 2. Fine-grained authorization implemented in the SQL compiler.

An SQL statement first goes through the parser component where it is analyzed for syntactic correctness and a query graph is generated. Next, it goes into the query rewrite component where the graph is modified to inject additional objects such as integrity constraints and triggers. We have modified this component to inject the new row permission and a column mask concepts we have introduced. The modified graph then

goes into the query optimizer component where several execution options are examined, and the optimal plan is selected based on a cost function. We have also modified this component to protect against potential data leakage should an unsafe predicate be evaluated before the security rules expressed by the row permissions are evaluated.

Unlike database views [6] where the security policy is enforced only when the views themselves are accessed, row permissions and column masks are table centric. This ensures that the security policy is enforced consistently regardless of how the table is accessed. Row permissions and column masks are also applied uniformly across all users, including DBAs, which helps organizations better adhere to zero-trust security [15], [16], [17] and in particular ensuring that access control is based on "need-to-know". Additionally, row permissions and column masks are application transparent. Database applications can immediately benefit from these concepts without having to incur any code changes. The SQL syntax for row permissions and column masks is given below.

```
create permission permission-name on table-x
        for rows where predicate-clause
        enforced for all access [disable | enable]

create mask mask-name on table-x
        for column column-name
        return case-expression [disable | enable]
```

Example 1

The following row permission creates a rule that grants access to rows in the PAYROLL table only to users who are members of the HR role.

```
create permission rpayroll on payroll
        for rows where verify_role_for_user (USER, 'HR') = 1
        enforced for all access enable;
```

Example 2

The following column mask creates a rule that grants access to the salary column in the PAYROLL table only to users who are members of the SM role. Other users will see NULL when they query the salary column.

```
create mask msalary on payroll
        for column salary
        return case when verify_role_for_user (USER, 'SM') = 1
                        then salary
                        else null
                end
        enable;
```

Some applications may not desire receiving a NULL value. Instead, they may want to receive an alternate and format preserving data value [18]. Our model can easily support this use case. All that is needed is to register a User Defined Function (UDF) in the database and modify the CREATE MASK SQL statement above such that instead of returning NULL, call the UDF to return the desired output.

A table can have zero or more row permissions. When more than a single row permission is defined on a table, the predicates from each one of them are combined together by applying the logical OR operator. In other words, if a row permission R_1 gives user U_1 access to a set of rows S_1, and another row permission R_2 on the same table gives that same user access to another set of rows S_2, then both row permissions would give that user access to the union of S_1 and S_2. A column can have zero or one mask. We extended the SQL compiler so that during query compilation, row permissions and column masks are dynamically injected into the query graph. This ensures that the query execution plan generated automatically enforces the rules expressed by the row permissions and column masks.

3.1 Row permissions enforcement

Row permissions defined on a given table are automatically applied when that table is accessed through any table level SQL statements: SELECT, INSERT, UPDATE, DELETE, and MERGE.

For SELECT statements, the predicates from all the row permissions defined on the table are combined together through the logical OR operator to derive a master predicate. This master predicate acts as a filter to limit the set of rows returned. We extended the query optimizer component of the SQL compiler to ensure that this master predicate is evaluated before any other unsafe user predicates. This is important to guard against potential data leakage through such unsafe user predicates. For example, suppose there is a UDF which emails the table rows retrieved to some external party. If such UDF appears in a user predicate and that predicate is executed before the master predicate, then by the time the master predicate is applied it will already be too late as the row would have already been sent out

For INSERT statements, the rules specified in the row permissions defined on that table are used to determine whether or not the row can be inserted into the table. To qualify, the user attempting to insert the row must be able to retrieve it back through a SELECT statement. This semantic is analogous to how symmetric database views behave. More specifically, a user is not allowed to insert a row they cannot retrieve back.

For UPDATE statements, the rules specified in the row permissions defined on that table are used to determine whether or not the row can be updated. This is a two-step process. First, the row permissions are used to filter out the set of rows that can be updated. In other words, a user cannot update rows they are not allowed to see. Next, the updated rows (if any) must conform to the same semantic as for INSERT processing to ensure that the user does not inject rows they cannot retrieve back.

For DELETE statements, the rules specified in the row permissions defined on that table are used to filter the set of rows that can be deleted in order to ensure that the user can only delete rows they can see.

A MERGE statement can be thought of as a combination of an INSERT and an UPDATE statements. Therefore, a MERGE statement is processed as an INSERT when dealing with new rows and as an UPDATE when dealing with existing rows in the table.

3.2 Column masks enforcement

The goal of a column mask defined on a given column C1 is to ensure that when C1 appears in the final results set of a query, C1 values are masked out if the user is not authorized to see them. This has two important implications. First, the SQL compiler will enforce the column mask for SELECT statements only. INSERT, UPDATE, DELETE, and MERGE statements do not return a result set to the user, so the column mask does not apply in these cases. Secondly, the SQL compiler must ensure that the enforcement of a column mask does not break database applications as this can have severe business impact. For example, suppose that a column mask is applied when the column appears in a predicate. This may totally change the final results set and the database application may end up processing a different set of rows (e.g. giving a raise to the wrong employees). Consequently, we have extended the SQL compiler such that column masks do not interfere with the computation of the final results set and the order or grouping thereof. More specifically, column masks are not applied when the column appears in any of these situations: WHERE clauses, GROUP BY clauses, HAVING clauses, SELECT DISTINCT, and ORDER BY clauses. One consequence of this approach is that it may create opportunities for inferences. But as discussed in our threat model section, we focus on application access as opposed to free direct SQL access to the database. Furthermore, the trusted context concept introduced in this paper enables establishing a trusted relationship between the application and the database server as well as protecting against abuse of the application's database user ID.

4. User identity propagation in multitiered environments

In multitiered environments, the middle tier application serves the needs of several users over a pooled database connection. Under this model, the database server only

sees a generic user ID which identifies the middle tier application, not the actual users of that application. Despite being a very popular application model, the fact that the database server only sees a generic user ID for all accesses poses several challenges.

First, the middle tier application cannot benefit from fine-grained database authorization because the database server does not see the identity of the application user. Thus, instead of delegating the authorization burden to the database server where it can be enforced more effectively, the middle tier application is forced to implement that fine-grained authorization in the application itself. This renders the application more complex, exposes the security policy to application programmers, and forces unnecessary patching of the application each time the security policy needs to be updated.

Additionally, using a single user ID for all database accesses diminishes user accountability. For example, one of the very first tasks in a forensic investigation is to check the database audit logs for gaining insight into user activities. However, if all accesses by all users are made using a single user ID, the database audit log would unfortunately provide little to no value.

The naïve approach to address this issue is to have the middle tier application establish a separate database connection for each user. Unfortunately, this approach may not be always feasible as the middle tier application may not have access to the end user database credentials. Additionally, even if this were feasible, this approach would not be desirable as establishing a large set of database connections would introduce a database performance overhead. This is the overhead associated with user authentication and the setting of the actual connection structures on the database server side.

Clearly, a better approach is needed for relieving the middle tier application from the burden of enforcing fine-grained authorization, and for holding users accountable for their actions.

4.1 Trusted contexts

We extend database systems by introducing a new concept called *trusted context*. A trusted context is a database object which defines a trust relationship between the database server and an external entity such as a middle tier application server. The trust relationship allows the database security administrator (DBSECADM) to specify a set of conditions which, when satisfied by a database connection request, instructs the database server to internally mark that database connection as trusted. A trusted connection gives the entity that established such connection a set of privileges that are not available outside the scope of that trusted connection. One example of such privileges is the ability to reuse an existing database connection for a different user without having to re-authenticate that user at the database server. Reusing an existing database connection avoids incurring a performance overhead by eliminating the need to establish a new database connection. Therefore, a middle tier application server can take advantage of the trusted context concept to establish an initial trusted connection, and then reuse that trusted connection to propagate an end user identity to the database server before submitting database requests on behalf of that end user.

The DBSECADM can choose from a variety of attributes to set the conditions for a trusted relationship such as a user ID, an IP address, a domain name, a digital certificate, and the type of encryption used to protect the communication channel between the database server and the middle tier application (e.g., SSL). The SQL language syntax for our trusted context concept is given below.

```
create trusted context context-name
        based upon connection using system authid authorization-id
        attributes key-value-pair-list
        default role role-name
        with use for user | role | group name [without authentication |
        with authentication] [role role-name]
        [disable | enable]
```

Example 3

The following trusted context establishes a trusted relationship between the database server and a middle tier application. The attributes upon which this trusted relationship is based are the user ID identifying the middle tier application itself, the IP address of the server where that application is hosted, and the type of communication encryption used to protect the communication channel between the database server and the middle tier application.

```
create trusted context ctx1
        based upon connection using system authid midtierApp1
        attributes (address '174.94.142.56' encryption 'SSL')
        with use for role midtierApp1Users
        without authentication
        enable;
```

In our implementation of trusted contexts in IBM DB2, we have extended the database server connection processing as follows. When a database connection request is received, we go through the authentication process as usual, but we also compare the attributes of that request with the attributes of the trusted context objects defined at that database server. If there is a match, we mark that connection as trusted. We have also extended the DB2 Command Level Interface (CLI) with a new command to give applications the option to request switching the current user ID on a trusted database connection. On the database server side, when such request is received, we first verify this is within the scope of a trusted connection, and then ensure that the user ID to switch to is authorized as per the trusted context object definition. For example, the trusted context definition above states that it is only permitted to switch to users who are members of the role *midtierApp1Users*. Lastly, we also check whether the trusted context definition authorizes switching users without authentication or requires authentication. If authentication is not required as in example 3 above, then no further processing is required. Otherwise, the switch user request must provide a valid authentication credential. Once the checks above are completed and the switch user request is authorized, we reset the user environment over the current physical

connection to match the new user, and the application is now ready to start sending database commands under the scope of this new user.

Also, in order to ensure database integrity is not compromised, we extended the database server processing such that switching users over a trusted connection is permitted only on transaction boundary. If such a request is made outside of a transaction boundary, the current transaction is rolled back, and the connection is put in an unconnected state, thus giving the middle tier application the opportunity to recover.

4.2 Trusted context-based authorization

Traditionally, database security models are such that the privileges granted to a user are universally applicable irrespective of any context. For example, if a user is granted SELECT privilege on the payroll database table, that user could exercise that privilege regardless of how they gain access to the database. The lack of control on when a privilege is available to a user can weaken overall security since the privilege may be abused. For example, an application administrator may choose to use the application's database credentials to connect to the database directly and make changes that are contrary to the application business logic.

To provide control over when privileges may be exercised, we extend the trusted context concept so that a DBSECADM can associate one or more roles with a trusted context. Roles that are associated with a trusted context are only exercisable when the user is acting within the scope of a trusted connection based upon that trusted context. This enables organizations to better adhere to zero-trust security, and in particular the "verify and never trust" tenet as the database system verifies more security attributes before granting a role to user [15], [16].

Example 4

The definition of the following trusted context is similar to example 3, but it specifies two database roles. The first role is *DBCONNECT* which the DBSECADM decided not to grant to the user ID *midtierApp1*. Instead, they assigned it to this trusted context. This

means that if the application administrator were to abuse this user ID by attempting to connect to the database from a server other than what is stated in the trusted context definition, that connection will be refused by the database server. The second role is HR, which is the role that grants access to the content of the payroll table as per the row authorization in example 1. This in turn means that members of the HR role will have access to the payroll table only within the scope of the trusted connection based upon this trusted context. In other words, they will only have access when they are using the application and not otherwise.

```
create trusted context ctx1
        based upon connection using system authid midtierApp1
        attributes (address 'srv.dep.org.com' encryption 'SSL')
        default role DBCONNECT
        with use for role midtierApp1Users
        without authentication HR
        enable;
```

In our implementation of trusted context-based authorization in IBM DB2, we have extended the database server authorization model as follows. When a database connection request is matched with a trusted context object, we check if there are any default roles assigned to that trusted context and add them to the user's roles list so they are used when deciding whether or not the user is authorized to connect to the database. Similarly, when a request to switch the current user on a trusted connection is received, we check if the trusted context definition grants any roles to the user to switch to and add any such roles to the new user's roles list accordingly.

5. Safe coexistence with fundamental database tenets

Database security needs to safely coexist with fundamental database tenets. Failure to do so may create database vulnerabilities and limit adoption of the solution.

5.1 User defined functions

A User Defined Function (UDF) is an important database concept which applications depend upon to delegate certain tasks to the database system. We extended the database system such that, by default, the row permission predicates are evaluated first to avoid potential data leakage through UDFs that may also appear in the set of predicates to apply on the table. The following experiment illustrates this extension and can be consistently repeated on any recent IBM DB2 system. The experiment creates a table T1 with 2 integer columns A and B. It inserts 3 rows into this table (1,1), (2,2) and (3,3). Then, we create a UDF which replaces any value in column A that is greater than 1 by 1. When we run the simple SQL query SELECT A, B FROM T1 WHERE F1(A) = 1, we expectedly obtain 3 rows because the values 2 and 3 in column A are changed to 1 by the UDF F1. Then we create a row permission with the predicate "A = 1". Now, when we run the SELECT query above any number of times, we consistently get back a single row. This is because our design ensures that the row permission predicates are executed before any unsafe UDF predicate. This is how data leakage is prevented because the UDF could have done anything with the data rows such as modifying them to alter the results set (as F1 does). But our design ensures that the UDF only sees the rows which are authorized for the user running the SELECT query. Below are the exact steps.

```
create table T1 (A int, B int);
insert into T1 values (1,1), (2,2), (3,3);
create function F1 (A int) returns int
        language SQL contains SQL no external action deterministic
        return (case when A > 1 then 1 else A end);
select A, B from T1 where F1(A) = 1;
create permission P1 on T1
       for rows where A = 1
       enforced for all access
      enable;
select A, B from T1 where F1(A) = 1;
```

While executing the UDF predicate last is good from a security perspective, it may not be necessarily good from a performance perspective, particularly if the UDF is a trusted function. Therefore, we extended the database system with the concept of secure UDF. By default, a UDF is not secure, but the administrator can alter the definition of a UDF to mark it secure. This means that the administrator confirms that the UDF is trusted. When a UDF is secure, the database system can order the evaluation of predicates based on such UDF anywhere the SQL compiler sees fit. Secure UDF enable performance and database security to coexist in harmony.

5.2 Materialized query tables

A Materialized Query Table (MQT) is a special type of database table which contains the results set of an SQL query. It is a critical database concept DBAs depend upon to maintain high performance for complex SQL queries. So, why does the design of database security need to pay attention to MQT? Suppose that the DBA creates an MQT M1 based on an SQL query affecting two tables T1 and T2. Further, suppose that table T1 is protected through a set of row permissions and column masks. If such row permissions and column masks are applied during the creation of MQT M1, the content of that MQT becomes dependent on what its creator can or cannot see in base table T1. This would negatively affect the accuracy of the database system's answers. For example, if the database system decides to use M1 to answer a query from a user U1, that user may get more data or less data than what they are authorized depending on whether they have access to more data or less data in base table T1 than the creator of MQT M1. A better approach is therefore to not enforce the row permissions and column masks on T1 during the creation of MQT M1 (or subsequent automatic refresh of its content). But we need to make sure that security is not compromised when doing so. In this context, we have extended the database system such that:

- Upon the creation of an MQT, the database system automatically generates and applies a default row permission with the false predicate "1 = 0". This ensures that direct SQL access to the MQT is blocked (i.e., "1 = 0" always evaluates to false). If certain users have a business need to access the MQT

directly, the administrator can create the appropriate row permissions on the MQT to give them access. Any such row permissions or column masks are enforced only during direct access to the MQT.

- When the database system decides to answer a user query from an MQT, it always ensures that any row permissions and column masks on any base table upon which the MQT is defined are automatically carried over and applied on the MQT itself. This ensures that users do not inadvertently get access to data in the base tables for which they are not authorized.

The following experiment illustrates how direct access to an MQT is automatically blocked when its underlying base table is protected by a row permission. This experiment can be consistently repeated on any recent IBM DB2 system. First, we create a table T1 with 2 integer columns A and B. We then insert 3 rows into this table, namely (1,1), (2,2) and (3,3). Next, we create an MQT M1 based on table T1. When we run the statement SELECT A FROM M1, we get the exact same data in base table T1. On the other hand, if we protect T1 with a row permission and retry that exact same statement, we now get zero rows returned. This is because our design automatically protects the MQT M1 to guard against data leakage. Below are the exact steps.

```
create table T1 (A int, B int);
insert into T1 values (1,1), (2,2), (3,3);
create table M1 (a, b) as (select A, avg(B) from T1 group by A)
        data initially deferred refresh deferred maintained by system;
refresh table M1;
select A from M1;
create permission P1 on T1
        for rows where A = 1
        enforced for all access
       enable;
select A from M1;
```

5.3 Database triggers

A database trigger is a critical database concept which applications depend upon to preserve data integrity. For example, a banking application may decide to use a trigger to ensure that each time a client's balance is updated in the clients table, a row is inserted into the statements table to record that particular withdrawal or deposit transaction. So, why does the design of database security need to pay attention to database triggers? Consider the banking application example above. Suppose that the clients table is protected with a set of row permissions and column masks. If such row permissions and column masks are blindly applied, then it may not be possible to update the statements table as the required input data could have been filtered out or masked. Clearly, this approach would negatively impact data integrity.

A better approach is therefore to not enforce the row permissions or column masks on the clients table. However, not doing so may affect security as the data in the clients table now becomes visible to any triggers defined on such table and may be abused. In this context, we have extended the database system by introducing the notion of a *secure trigger*. By default, a database trigger is not secure, but the administrator can alter the trigger's definition to mark it secure. This means that the administrator vouches for the trigger as trusted and can be applied on a table protected with row permission or column mask constructs. Secure triggers enable database security and triggers to coexist in harmony.

6. Performance evaluation

We have conducted 4 different assessments during our performance evaluation. The assessments were conducted using IBM DB2, extended with our fine-grained authorization model, deployed on a dedicated AIX system with 8 processors @ 1452 GHz and 32GB of RAM. This is a fully dedicated system (CPU, memory, networking and storage) running only our experiment to ensure performance data stability. The time elapsed for a given query is measured from the time the query is submitted to the time the results are returned. Before a query is run, the database system is activated to ensure

a fresh database set up. The query is run several times. The first run is discarded from the statistics as the database bufferpool (i.e., database cache) is cold.

- **Assessment 1**: The goal of this assessment is to measure the impact to performance when an application chooses to delegate fine-grained authorization to the database. One of the key advantages of our fine-grained authorization model is that it relieves applications from the burden of enforcing fine-grained authorization by delegating such task to the database. But it is important that this reduction in application complexity does not result in any significant performance drawbacks for the application. This assessment confirmed that applications can safely delegate the enforcement of fine-grained database authorization to the database with no performance concerns.
- **Assessment 2**: The objective of this assessment is to measure the scalability of column masks. Linear scalability has been confirmed by this assessment.
- **Assessment 3**: The goal of this assessment is to verify the independence of column masks. This assessment has shown that the impact of all column masks defined on a table is never higher than the sum of the impact of each column mask defined individually.
- **Assessment 4**: The objective of this assessment is to measure the impact of row permissions. This test confirmed that the impact of row permissions is minimum.

6.1 Delegating fine-grained authorization enforcement to the database system

Methodology

We have selected TPC-H [19] as the application with which to conduct our assessment. TPC-H is an industry standard benchmark for measuring database performance. It consists of 22 queries representative of decision support systems that examine large volumes of data. The performance metric reported by TPC-H is called

the TPC-H Composite Query-per-Hour Performance Metric (QphH) and reflects multiple aspects of the capability of the database system to process queries.

We focused on two scenarios in our assessment. In the first scenario, we created a set of column masks and row permissions on the TPC-H database schema to specify a fine-grained authorization policy. Then, we ran the TPC-H benchmark and measured the QphH. In the second scenario, we created no column masks or row permissions in the database. Instead, we modified the SQL queries, so the same fine-grained authorization is enforced by the application.

Table 1 summarizes our findings. The ratio column represents the QphH of the fine-grained authorization policy delegated to the database divided by the QphH when that policy is enforced by the application itself and is plotted in Fig. 3. The numbers on the x-axis of this figure represent the 22 TPC-H queries referred to in Table 1. That is, 1 represents query Q1, 2 represents query Q2 and so on.

Discussion

Fig. 3 shows that almost all the TPC-H queries perform the same or better when the policy is enforced by the database than by the application. More specifically, 13 queries performed fairly the same in both scenarios. 8 queries performed better when the fine-grained authorization policy is enforced by the database system (i.e., the ones where the ratio column is coloured in green in table 1). The improvement observed ranges from 8 to 68%. Lastly, for query Q19, we observed a performance degradation of 15% when the fine-grained authorization policy is enforced by the database.

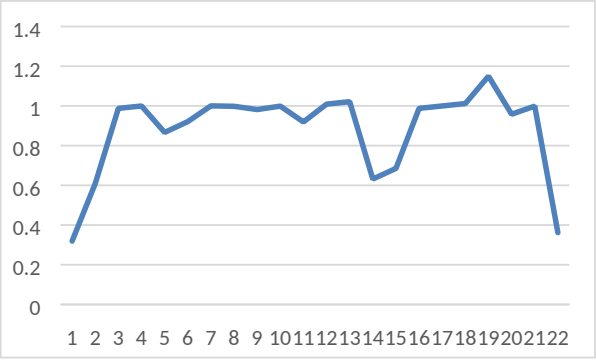

Fig. 3. Ratio of database vs application enforcement for TPC-H queries.

TABLE 1: Application vs Database Enforcement for TPC-H Queries.

TPC-H Query	QphH Application Enforcement (a)	QphH Database Enforcement (b)	Ratio (b/a)
Q1	1158.8	370	0.3193
Q2	19.7	12	0.6091
Q3	2350.6	2321.6	0.9877
Q4	6105.6	6103.4	0.9996
Q5	7352.6	6371.5	0.8666
Q6	27.8	25.6	0.9209
Q7	16654.1	16657.5	1.0002
Q8	884.2	882.5	0.9981
Q9	9653.8	9475.7	0.9816
Q10	8376.5	8367.3	0.9989
Q11	138.7	127.5	0.9193
Q12	112.6	113.6	1.0089
Q13	103.5	105.7	1.0213
Q14	22.8	14.4	0.6316
Q15	26.7	18.3	0.6854
Q16	24.3	24	0.9877
Q17	336.3	336.2	0.9997
Q18	288.5	291.9	1.0118
Q19	93.6	107.6	1.1496
Q20	73.9	70.8	0.9581
Q21	9655.1	9644.6	0.9989
Q22	90.9	32.9	0.3619

There are two main reasons for the results observed. First, the order in which predicates are evaluated is important, particularly for table joins. For example, consider the following query where tables T1 and T2 are joined on column C1: “SELECT * FROM T1 INNER JOIN T2 on T1.C1 = T2.C1”. When a row permission is enforced by an application, the application will modify the query above by adding the row

permission predicates to the SQL text directly as follows: “SELECT * FROM T1 INNER JOIN T2 on T1.C1 = T2.C1 AND <row permission predicate>”. Recall from section 3 that we extended the SQL compiler so that, by default, the row permissions predicates are evaluated first on the table to guard against potential data leakage by any unsafe predicates in the query. So, when the database enforces the fine-grained authorization policy, the query would actually look as follows within the SQL compiler “SELECT * FROM (SELECT * FROM T1 WHERE <row permission predicate>) INNER JOIN T1 on T1.C1 = T2.C1”. However, when there are no unsafe predicates in the query, we do not restrict the SQL compiler optimizer component from moving the row permission predicates higher or lower in the query graph if it leads to a better query execution plan. This was the case in our testing as we had no unsafe predicates. The only situation where the SQL compiler optimizer component did not move the predicate was for query Q19. This is because the row permission defined on the table did not refer to any data in the table itself as it was a simple rule to check whether or not the user issuing the query were a member of a given role. Consequently, the optimizer selected a merge-join instead of a hash-join [20] [21]. Normally, the merge-join would have performed better but because the row permission did not actually filter any rows, the merge-join ended up being more expensive, thus the observed degradation in query Q19.

The second reason for the results observed is how column masks are processed. When the database system enforces a column mask, it does so internally within the actual query graph built by the SQL compiler. So, when the same column appears multiple times within a query the SQL compiler does not need to duplicate the column masks. However, when the fine-grained authorization policy is enforced by the application, the rules representing the column mask end up being duplicated in the SQL query text as the application can only work with SQL. This explains the performance gain observed when the fine-grained authorization policy is enforced by the database.

Our tests have shown that enforcing the fine-grained database authorization policy by the database has not resulted in any significant performance drawbacks for the application. This means that the gains in security and the reduction in application complexity do not come at the expense of application SQL workload performance.

6.2 Scalability of column masks

Methodology

We have created a table T1 with 10 columns, all of the same type. We have populated the table with random data. No indices of any type were created on this table. We have run a "SELECT * FROM T1" as our baseline. Then, we created a column mask on the first column, ran the same query above and measured its performance. We have repeated this process for each of the remaining columns. The column mask created is exactly the same for each column. We have run the experiment twice: One where T1 contains one million rows and another one where it contains ten million rows. Table 2 summarizes our findings.

TABLE 2: Time elapsed (in seconds).

Test	1,000,000 rows	10,000,000 rows
Baseline (No Masks)	4.58	44.26
1 Mask	4.73	45.97
2 Masks	4.74	46.45
3 Masks	4.83	46.85
4 Masks	4.82	47.06
5 Masks	4.87	47.48
6 Masks	5	48.28
7 Masks	4.97	48.8
8 Masks	5.02	49.01
9 Masks	5.08	49.96
10 Masks	5.10	50

Discussion

Fig. 4 shows that for both the one million and ten million rows cases, the execution time of our query scales almost in a linear manner as the number of masks increases. This confirms our expectation as our design and implementation of column masks did not introduce any additional logic for coordinating the execution of multiple masks when they are present on a given table. Essentially, the overhead introduced is only the

one associated with the execution of the actual rule expressed in the column mask definition itself. In our experimentation, the rule was checking user membership in a role to decide whether they see the actual column value or a masked version thereof. It used the built-in SQL function VERIFY_ROLE_FOR_USER. This function is highly optimized. It keeps an in-memory list of users to roles mappings, making it very fast to decide whether or not a user is a member in a given role. We introduced this function to support the adoption of our row permissions and column masks as security best practices advocate for simplifying the management of authorization by assigning privileges to roles and assigning users to roles. Authorization then simply becomes checking user membership in roles.

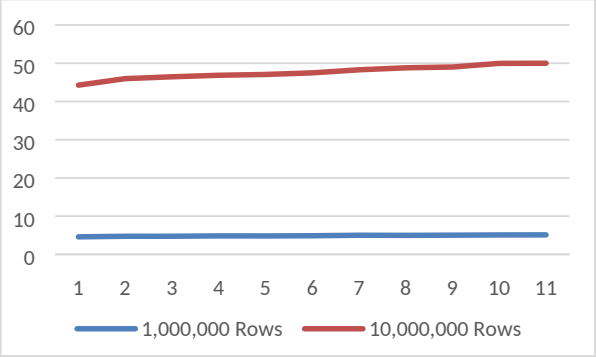

Fig. 4. Scalability of column masks.

6.3 *Independence of column masks*

Methodology

We have created three column masks on the CUSTOMER table in the TPC-H database schema: A simple column mask, an intermediate column mask, and complex column mask. The simple column mask is similar to the column mask shown in Example 2. It makes use of a single call to function VERIFY_ROLE_FOR_USER to check whether the user is a member of the given role. The intermediate column mask has four calls to the VERIFY_ROLE_FOR_USER function. Lastly, the complex

column mask is similar to the intermediate one but has a sub-select statement on top of that.

Our base line is a “SELECT * FROM CUSTOMER” query with no column masks defined on the CUSTOMER table. We ran this query, measured the elapsed time, and then performed the following tests:

- Run the same query with only the simple column mask enabled.
- Run the same query with only the intermediate column mask enabled.
- Run the same query with only the complex column mask enabled.
- Run the same query with all three column masks enabled.

Table 3 shows the time elapsed for each test when the CUSTOMER table contains one million rows, and ten million rows respectively. Table 4 shows the difference compared to the baseline for each of the tests conducted.

TABLE 3: Time elapsed (in seconds).

Test	1,000,000 rows	10,000,000 rows
Baseline (No Masks)	37.464	371.791
Simple Mask	38.812	387.457
Intermediate Mask	40.356	404.619
Complex Mask	58.592	556.439
All Masks	61.855	589.25

TABLE 4: Difference with the baseline.

Test	1,000,000 rows	10,000,000 rows
Simple Mask	1.348	15.666
Intermediate Mask	2.892	32.828
Complex Mask	21.128	184.648
Sum of all Masks	25.368	233.142
All Masks	24.391	217.459

Discussion

Fig. 5 contrasts the sum of the differences to the baseline for each of the simple, intermediate, and complex mask tests with the difference to the baseline for the test where all masks are enabled at the same time for both the one million rows and ten million rows cases. For both cases, we can observe that the difference with the baseline when all masks are enabled at the same time is never higher than the sum of the differences to the baseline for each individual mask. This confirms our expectation as our column masks design and implementation did not require introducing any coordination when multiple masks are enabled at the same time. The masks are in fact totally independent from each other.

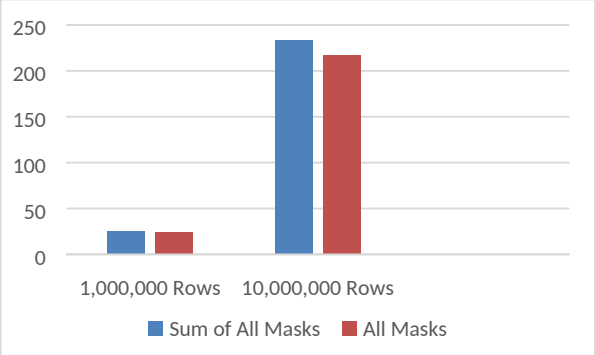

Fig. 5. Independence of column masks.

6.4 Row permissions impact

Methodology

We have created three row permissions on the CUSTOMER table in the TPC-H database schema: One row permission that returns zero rows, one permission that returns 50% of the rows, and another row permission that returns all rows. We have run "SELECT * FROM CUSTOMERS" as our baseline. Then, we run the same query with each of the row permissions above enabled individually (i.e. one row permission at a time). Table 5 shows the time elapsed for each test when the CUSTOMER table contains one million rows and ten million rows respectively.

TABLE 5: Time elapsed (in seconds).

Test	1,000,000 rows	10,000,000 rows
Baseline (No Permissions)	38.163	380.118
Permission (0 rows)	0.11	3.173
Permission (50% rows)	19.679	169.154
Permission (All rows)	38.679	383.93

Discussion

Fig. 6 and Fig. 7 contrast the performance for each of the 3 tests with our baseline for the one million rows and ten million rows respectively. The results are similar for each case and show that the overhead of row permissions is very minimal. For instance, when the row permission returns all rows, the performance is almost identical to the baseline. This is expected as the rule expressed in the row permission is internally implemented as a predicate. In our case, the predicate includes the built-in VERIFY_ROLE_FOR_USER SQL function. If a DBA decides to deploy their own UDF for use in a row permission definition, the performance implications may be different depending on several factors such as how optimized that UDF is and whether or not it is declared as trusted.

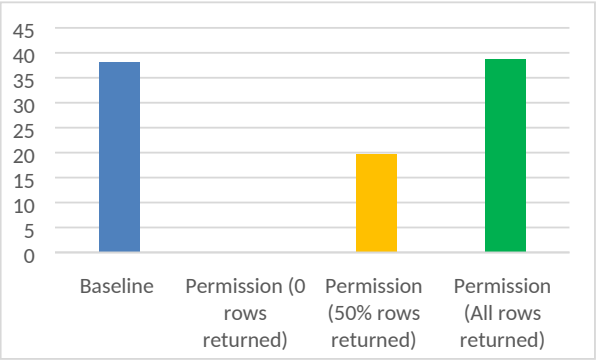

Fig. 6. Row permissions impact (1,000,000 rows).

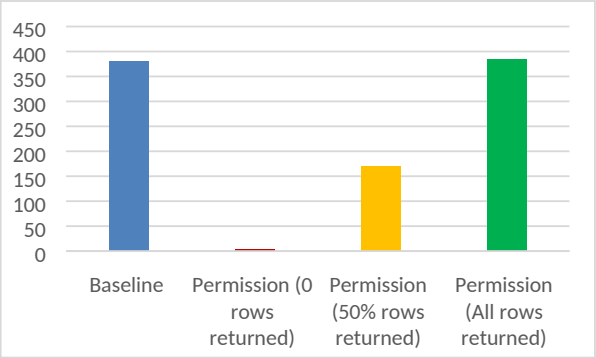

Fig. 7. Row permissions impact (10,000,000 rows).

7. Use case scenario

We describe how our row permissions and column masks can be applied to meet the needs of a banking application. All the SQL statements and outputs below have been fully verified with our implementation on IBM DB2. These requirements can be summarized as follows:

- Customer service representatives and telemarketers can see all data.
- Tellers can see only the data for their own branch customers.
- The customer account number is accessible only by customer service representatives. All other users can only see the last 4 digits.

Customer information is stored in a table called CUSTOMER and bank employee information is stored in a table called EMPLOYEE_INFO. The SQL statements for creating these two tables are given below.

```
create table customer (account varchar (9),
                       name varchar (20),
                       income int,
                       branch char (1) );
create table employee_info (branch char (1),
                            emp_id varchar (10) );
```

We assume that tables CUSTOMER and EMPLOYEE_INFO are already populated. Their content is given by tables 6 and 7 respectively.

TABLE 6: CUSTOMER table.

ACCOUNT	NAME	INCOME	BRANCH
1234-5678	Alice	22,000	A
2345-6754	Bob	71,000	B
3456-1298	Carl	123,000	B
4672-8901	David	172,000	C

TABLE 7: EMPLOYEE_INFO table.

EMP_ID	BRANCH
Amy	A
Pat	B
Haytham	C

Tellers, customer service representatives, and telemarketers are members of database roles TELLER, CSR, and TELEMARKETER respectively. SELECT privilege to the CUSTOMER table is granted to these three roles. Users Amy, Pat and Haytham are a teller, a customer service representative and a telemarketer respectively. The SQL statements for setting up these roles are given below.

```
create role teller;
grant select on customer to role teller;
grant role teller to user amy;
create role csr;
grant select on customer to role csr;
grant role csr to user pat;
create role telemarketer;
grant select on customer to role telemarketer;
grant role telemarketer to user haytham;
```

To implement the first rule which states that customer service representatives and telemarketers can see all customers, the following row permission must be created.

```
create permission csr_row_access on customer
        for rows where verify_role_for_user (USER, 'csr') = 1 or
                        verify_role_for_user (USER, 'telemarketer') = 1
        enforced for all access
        enable;
```

To implement the second rule which states that tellers can only see customers of their own branch, the following row permissions must be created. The sub-select in the permission definition ensures that the customer's branch and the teller's branch match.

```
create permission teller_row_access on customer
        for rows where verify_role_for_user (USER, 'teller') = 1 and
                        branch = (select branch from employee_info
                                where emp_id = USER)
        enforced for all access
        enable;
```

To implement the third rule, the following column mask is created. The mask ensures that when the user is not a member of the CSR role, they see only the last 4 digits of the account number. The rest of the digits are replaced by "X"s for them (masked out).

```
create mask csr_column_access on customer
        for column account
        return case when verify_role_for_user (USER, 'csr') = 1
                        then account
                        else 'XXXX-' || SUBSTR(ACCOUNT,5,4)
                end
        enable;
```

Now that the row permissions and column masks have been defined, any future access to the CUSTOMER table will see the database system automatically enforce the security policy. Table 8 contrasts the output when the application issues the query "SELECT * FROM CUSTOMER" for users Amy, Haytham and Pat respectively.

When the application issues that query on behalf of user Amy, the database only returns the rows for customers from branch A, which is where Amy works. Note that the account number is masked out because Amy is not a member of the CSR role.

On the other hand, when the application issues the exact same query on behalf of user Haytham, the database returns all the rows in the table which is in accordance with the first rule because Haytham is a telemarketer. Note that the account number is still masked out because Haytham is not a member of the CSR role.

Lastly, when the same query is issued on behalf of user Pat, all the rows in the table are returned and the account number is not masked out because Pat is a member of the CSR role.

TABLE 8: Output for users Amy, Haytham and Pat.

USER	ACCOUNT	NAME	INCOME	BRANCH
Amy	XXXX-5678	Alice	22,000	A
Haytham	XXXX-5678	Alice	22,000	A
	XXXX-6754	Bob	71,000	B
	XXXX-1298	Carl	123,000	B
	XXXX-8901	David	172,000	C
Pat	1234-5678	Alice	22,000	A
	2345-6754	Bob	71,000	B
	3456-1298	Carl	123,000	B
	4672-8901	David	172,000	C

This example has shown how the application logic can remain very simple. In all 3 user situations, the application simply issues the simple "SELECT * FROM CUSTOMERS" SQL query. The database system automatically applies the fine-

grained authorization rules, relieving the application from this burden, which in turn contributes to reducing the complexity of the application.

8. Conclusion

The rise of data breaches has driven many organizations nowadays to implement zero-trust security in order to reduce the risk of incurring a data breach. Like identity systems and networks, database systems also need to evolve to help organizations effectively adhere to zero-trust security. This is particularly important as database systems store an enterprise's most critical data and are often the primary target of attacks by both insiders and outsiders. This paper has introduced three new concepts to enable database systems for zero-trust security. Row permissions and column masks provide data-centric security so the security policy cannot be bypassed as with database views for example. They also coexist in harmony with the rest of the database core tenets so that enterprises are not forced to compromise neither security nor database functionality. Trusted contexts provide applications in multitiered environments with a secure and controlled manner to propagate end user identities to the database and therefore enable such applications to delegate the security policy to the database system where it is enforced more effectively. They also protect against application bypass so the application credentials cannot be abused to make database changes outside the scope of the application's business logic.

In our future work, we plan to focus on facilitating the adoption of our fine-grained database authorization model. For example, defining a column mask is a very easy task once you know which column to define it on. But in some situations, this knowledge may not be available (e.g., a database inherited through a merger or an acquisition). This is where data classification would be useful. The main challenges in this context would be to investigate how to do the data classification on the database efficiently and accurately. Additionally, we want to explore machine learning for automatically generating the appropriate row permissions and column masks. Machine learning has been explored for detecting threats [22], [23], [24], but here we would like to explore it for fine-grained authorization policy recommendation.

Lastly, Fig. 8 gives a visual summary for how our row permissions, column masks and trust contexts contrast with the prior art. Multilevel Security (MLS) provides high security but is the least flexible because its authorization rules are rigid and cannot be changed. Database views and application-based fine-grained authorization provide high flexibility but the protection they offer can be bypassed by accessing the base tables directly. Oracle VPD, Microsoft RLS/DDM, Sybase RLAC and Vertica RAP/CAP improve over database views and application-based fine-grained authorization. Our approach provides an additional improvement by addressing the loss of user identity problem in multitiered environments and by coexisting safely with the rest of the database core tenets.

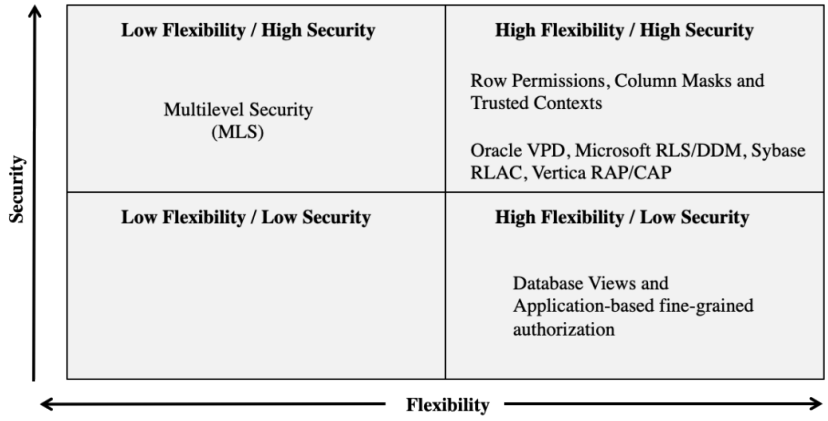

Fig. 8. Row permissions, column masks, trusted contexts and prior art.

References

[1] A. Zaytsev, A. Malyuk, N. Miloslavskaya, “Critical Analysis in the Research Area of Insider Threats”, *Proceedings of the IEEE 5th International Conference on Future Internet of Things and Cloud (FiCloud)*, 2017.

[2] I. Ghafir, J. Saleem, M. Hammoudeh, H. Faour, V. Prenosil, S. Jaf, S. Jabbar, T. Baker, “Security threats to critical infrastructure: the human factor”, *The Journal of Supercomputing*, Volume 74, Issue 10, Springer US, 2018.

[3] P. Voigt, A. von dem Bussche, *The EU General Data Protection Regulation (GDPR): A Practical Guide*, Springer, 2017.

[4] A. Chuvakin, B. Williams, *PCI Compliance: Understand and Implement Effective PCI Data Security Standard Compliance*. Elsevier, 2009.

[5] P. Zikopoulos, G. Baklarz, M. Huras, W. Rjaibi, D. McInnis, M. Nicola, L. Katsnelson, *DB2 10 for Linux, Unix and Windows New Features*, McGraw-Hill, 2012.

[6] R. Elmasri, S. Navathe, *Fundamentals of Database Systems 6th*. Addison-Wesley, 2010.

[7] S. Gaetjen, D. Knox, W. Maroulis, *Oracle Database 12c Security*, McGraw-Hill Education, 2015.

[8] P. Carter, *Securing SQL Server: DBAs defensing the database*, Apress, 2018.

[9] J. Garbus, *SAP ASE 16 / Sybase ASE Administration*, SAP Press, 2015.

[10] S. Chaudhuri, T. Dutta, S. Sudarshan, “Fine Grained Authorization Through Predicated Grants”, *Proceeding of the International Conference on Data Engineering*, 2007.

[11] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, W. Rjaibi, “Extending relational database systems to automatically enforce privacy policies”, *Proceedings of the International Conference on Data Engineering*, 2005.

[12] W. Rjaibi, P. Bird, “A Multi-Purpose Implementation of Mandatory Access Control in Relational Database Management Systems”, *Proceedings of the International Conference on Very Large Data Bases*, 2004.

[13] W. Rjaibi, “An introduction to multilevel secure relational database management systems”, *Proceedings of the conference of the Centre for Advanced Studies on Collaborative research (CASCON)*, 2004.

[14] W. Chen, B. Barkai, J. DiPietro, V. Langman, D. Perlov, R. Riah, Y. Rozenblit, A. Santos, *Deployment Guide for Infosphere Guardium*, IBM Redbooks, 2014.

[15] E. Gilman, D. Barth, *Zero Trust Networks: Building Secure Systems in Untrusted Networks*. O’Reilly Media, 2017.

[16] S. Walker-Roberts, M. Hammoudeh, A. Dehghantanha, “A Systematic Review of the Availability and Efficacy of Countermeasures to Internal Threats in Healthcare Critical Infrastructure”, *IEEE Access*, Volume 6, pp.25167-25177, 2018.

[17] S. Walker-Roberts, M. Hammoudeh, “Artificial Intelligent Agents as Mediators of Trustless Security Systems and Distributed Computing Application”. *In: Parkinson S., Crampton A., Hill R. (eds) Guide to Vulnerability Analysis for Computer Networks and Systems. Computer Communications and Networks*. Springer, Cham, 2018.

[18] A. Goldsteen, K. Kveler, T. Domany, I. Gokhman, B. Rozenberg, A. Farkash, “Application-Screen Masking: A Hybrid Approach”, *IEEE Software*, Volume 32, Issue 4, 2015.

[19] A. Thanopoulou, P. Carreira, H. Galhardas, “Benchmarking with TPC-H on Off-the-Shelf Hardware: An Experiments Report”, *Proceedings of the International Conference on Enterprise Information Systems*, 2012.

[20] N. Bruno, Y. Kwon, M. Wu, “Advanced Join Strategies for Large-Scale Distributed Computation”, *Proceedings of the VLDB Endowment*, Vol. 7, No. 13, 2014.

[21] C. Balkesen, G. Alonso, J. Teubner, M. Ozsu, “Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited”, *Proceedings of the VLDB Endowment*, Vol. 7, No. 1, 2013.

[22] M. Alloghani, D. Al-Jumeily, A. Hussain, J. Mustafina, T. Baker, A. Aljaaf, “Implementation of Machine Learning and Data Mining to Improve

Cybersecurity and Limit Vulnerability to Cyber Attacks". *In: Nature-Inspired Computation in Data Mining and Machine Learning*. Springer, Cham, 2020.

[23] S. Aljawarneh, M. Aldwairi, M. Bani Yassein, "Anomaly-based intrusion detection system through feature selection analysis and building hybrid efficient model", *Journal of Computational Science*, Volume 25, Elsevier, 2018.

[24] M. Aldwairi, R. Alsalman, "Malurls: a lightweight malicious website classification based on url features", *Journal of Emerging Technologies in Web Intelligence*, Volume, Issue 2, Academy Publisher, 2012.

 Walid Rjaibi is Distinguished Engineer and Chief Technology Officer (CTO) for Data Security with IBM in Toronto, Canada. Prior to his current role, Walid was Research Staff Member in network security and cryptography with IBM Research in Zurich, Switzerland. Walid's work on Data Security has resulted 26 granted patents and several publications in journals and conference proceedings such as the IDUG solutions journal, the international conference on security and cryptography (SECRYPT), the international conference on data engineering (ICDE), and the international conference on Very Large Databases (VLDB).

 Mohammad Hammoudeh is the Head of the CfACS IoT Laboratory and a Reader in Future Networks and Security with the Department of Computing and Mathematics, Manchester Metropolitan University. He has been a researcher and publisher in the field of big sensory data mining and visualization. He is a highly proficient, experienced, and professionally certified cybersecurity professional, specializing in threat analysis, and information and network security management. His research interests include highly decentralized algorithms, communication, and cross-layered solutions to Internet of Things, and wireless sensor networks.