PostgreSQL Anomalous Query Detector*

Bilal Shebaro†, Asmaa Sallam†, Ashish Kamra‡, Elisa Bertino†
†Cyber Center and CERIAS, Purdue University, West Lafayette, IN 47907, USA
‡EMC Corporation, Bagmane World Technology Center, Bengaluru - 560038, KA, India
{bshebaro, asallam, bertino}@purdue.edu, ashish.kamra@emc.com

ABSTRACT

We propose to demonstrate the design, implementation, and the capabilities of an anomaly detection (AD) system integrated with a relational database management system (DBMS). Our AD system is trained by extracting relevant features from the parse-tree representation of the SQL commands, and then uses the DBMS roles as the classes for the bayesian classifier. In the detection phase, the maximum apriori probability role is chosen by the classifier which, if not matching the role associated with the SQL command, raises an alarm. We have implemented such system in the PostgreSQL DBMS, integrated with the statistics collection and the query processing mechanism of the DBMS. During the demonstration, our audience will be given the choice of training our system using either synthetic role-based SQL query traces based on probability sampling, or by entering their own set of training queries. In the subsequent detection mode, the audience can test the detection capabilities of the system by submitting arbitrary SQL commands. We will also allow the audience to generate arbitrary work loads to measure the overhead of the training phase and the detection phase of our AD mechanism on the performance of the DBMS.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems - Query processing

Keywords

Anomaly detection, Intrusion detection, Bayesian classifier, DBMS

1. INTRODUCTION

The goal of this work is to demonstrate the integration of a DBMS specific anomaly detection (AD) mechanism within the core of the DBMS functionality. AD mechanisms are essential to detect anomalies in data accesses by users. Such anomalies may be indicative of insider attacks or compromised database user accounts [1]. It is important to notice that even though AD mechanisms exist that work at the operating system (OS) level and at the network level, these AD mechanisms may not be effective against database related attacks. The reason is that user actions deemed malicious for a DBMS are not necessarily malicious for the underlying OS or the network. For example, consider the case of a database user/application that performs queries on a customer table containing name, address, phone numbers and e-mail of customers and thus has the select authorization on this table. Suppose that this user/application is in charge of shipping products to customers and each day typically performs 20 shipping operations. Thus the user/application typically accesses 20 records from the table and retrieves from these records only the customer names and addresses. Occasionally the user/application may access a phone number of some customer when there are problems with the product order. Now suppose that one day the user/application retrieves all the records from the customer table. Such access would be allowed by the access control system as the user/application has the select authorization on the table. However, the access is anomalous with respect to the usual access patterns by this user/application. Such anomalous SQL command may be the result of a SQL injection vulnerability or a data theft attempt by an authorized user and is not typically detected by an AD mechanism working at the OS level or at the network level as the OS and the network cannot understand the specific SQL command entered by users or applications. Our approach, that tightly integrates the AD mechanism with the DBMS, has three main advantages. First, AD is executed closer to the target (during query processing), thereby ruling out chances of a back-door entry to the DBMS that may bypass the AD mechanism. Second, as the AD mechanism is integrated as one of the features of the DBMS, the physical location of the DBMS is not a constraint with respect to using the AD service. Such requirement is important in the current era of cloud computing as organizations need the flexibility to move their databases to a cloud service provider. Third, the integration with the DBMS makes it possible for the AD mechanism to issue more versatile response actions to anomalies [5].

This demonstration paper is organized as follows: An overview of related work is presented in Section 2. In Section 3 we describe the technical details of our AD mechanism. Then we show the demonstration plan and how the audience can interact with the system in Section 4. Finally some conclusions are outlined in section 5.

2. RELATED WORK

Several AD approaches have been proposed for database systems. Spalka *et al.* [10] focus on detecting anomalies for relational databases based on state relations. They compare two approaches to deal with the database extension; one based on reference values and one based on Δ-relations. Mathew *et al.* [8] propose a data-centric approach for solving the AD problem in a DBMS. They model users' access patterns by profiling the used data points. Both these approaches complement our work as they focus on the semantic aspects of the SQL queries as represented by the data in the relations, while we focus on the syntactic aspects by detecting anomalous access patterns. Hu *et al.* [4] propose an approach for identifying malicious transactions

*This work is based on Ashish Kamra's Ph.D. thesis while at Purdue University.

EDBT/ICDT '13 March 18 - 22 2013, Genoa, Italy
ACM 978-1-4503-1597-5/13/03.

through a data dependency miner that correlates records from the database log. The transactions that do not comply with the data dependencies mined are identified as malicious transactions. Chung *et al.* introduce DEMIDS, a misuse-detection system, tailored for relational database systems [3]. DEMIDS uses audit log data to derive profiles describing typical patterns of accesses by database users. Since this approach is still theoretical and has not been implemented yet, there is no evidence of its capabilities in the AD domain. Lee *et al.* [7] propose an approach for detecting illegitimate database accesses by fingerprinting every transaction, mainly by summarizing SQL statements into compact regular expression fingerprints. To the best of our knowledge, no work demonstrating the capabilities of a DBMS AD mechanism exists.

3. TECHNICAL DETAILS

3.1 The Anomaly Detection Algorithm

In this section, we present our algorithm for detecting anomalous accesses to a database. We assume that the database has a Role Based Access Control (RBAC) model in place whereby authorizations are specified with respect to roles and not with respect to individual users. Our AD system builds a profile for each role that represents accurate and consistent behavior of users holding the role. We rely on the use of intrusion-free database traces where the record sequences of the database audit log represent normal user behavior. Thus the classifier is trained using these records and then used to detect anomalous behavior.

3.2 Data Representation

We assume that users interact with the database through SQL commands. For example, in the case of SELECT queries, such commands have the format:
SELECT [DISTINCT] {TARGET-LIST}
FROM {RELATION-LIST}

We use the fine triplet representation introduced in [2, 6] to capture all the information in the input query. The fine triplet consists of three fields in the form *(SQL-CMD, ACC-REL-BIN[], PROJ-ATTR-BIN[][])*. *SQL-CMD* represents the type of the query. *ACC-REL-BIN* is a binary vector, *ACC-REL-BIN[i]* is 1 if the i^{th} table is accessed in the query, that is, if it is present in the RELATION-LIST. *PROJ-ATTR-BIN* is a 2-dimentional vector; *PROJ-ATTR-BIN[i][j]* is 1 if the j^{th} attribute of the i^{th} table is projected, that is, if it is present in the TARGET-LIST. Consider as an example, a database that contains three tables R_1, R_2, and R_3, each with three columns C_1, C_2, and C_3. The corresponding representation of the SELECT, INSERT, and UPDATE queries is shown in Table 1. Since the triplet depends only on the tables and attributes accessed in the query, its representation is not affected by the attributes of the WHERE, VALUES, and UPDATE clauses printed after the SELECT, INSERT, and UPDATE statements respectively.

We address the AD problem as a classification problem. We apply the Naive Bayes Classifier (NBC), as discussed in [2]. NBC is a simple probabilistic classifier based on applying Bayes' theorem with strong independence assumptions. The NBC uses the attributes of the triplet representation of an input query and outputs the identifier of a role. To determine the role associated with a newly issued query, we use the Maximum Aposteriori Probability *(MAP)* decision rule. Using this decision rule, the NBC picks the correct classification that has the highest role probability. If the role predicted by the classifier is different from the original role associated with the query, an anomaly behavior is detected and the query execution stops.

SQL Command	Fine-triplet
Select $R_1.A_1$, $R_3.C_3$, $R_3.B_3$ From R_1, R_3 Where $R_1.A_1$=20 and $R_1.A_1$= $R_3.C_3$	select $<1, 0, 1>$ $<<1, 0, 0>$ $<0, 0, 0>$ $<0, 1, 1>>$
INSERT INTO R_2 VALUES (1, 1, 1)	insert $<0, 1, 0>$ $<<0, 0, 0>$ $<1, 1, 1>$ $<0, 0, 0>>$
UPDATE R_3 SET $R_3.C_3$ = 100	update $<0, 0, 1>$ $<<0, 0, 0>$ $<0, 0, 0>$ $<0, 0, 1>>$

Table 1: Triplet construction example

3.3 Implementation in PostgreSQL

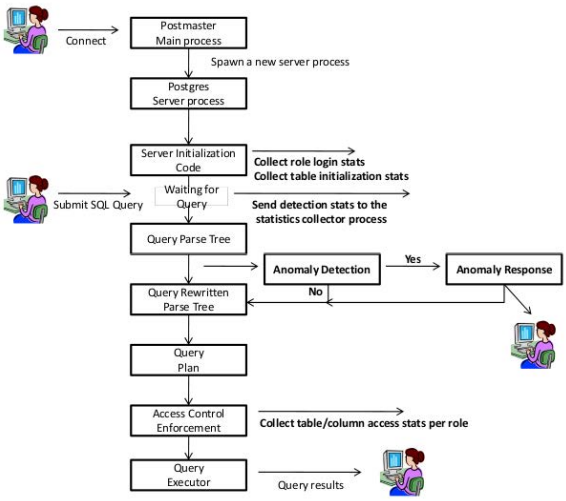

Figure 1: Anomaly Detection and Data Collection Hooks in PostgreSQL.

We implemented our AD algorithm in the open-source DBMS PostgreSQL. Figure 1 shows the query processing architecture and the application of our algorithm in the execution pipeline of a query. For every new connection to the database, the main server process spawns a new server process called Postgres. Every SQL query sent on that connection is handled by this new Postgres process. When a new connection to the database is established by a user, we report the login statistics to the statistics collector process (server-side process) that includes the roles activated by the user, and the list of tables under AD.

When the user submits a query, the query string passes through the query parser which creates a parse tree of the query structure. The parse tree is then modified by composing into it any *views* or *rules* that may apply to the query. This is performed by the query rewrite system. After the query has been rewritten, the query optimizer takes the parse tree and generates an optimal query plan that contains the operations to be executed for processing the query. The plan is then passed to the query executor that is responsible for executing of the query and passing the results back to the client. Before the executor begins executing the query, it checks whether the user has the privileges (directly or indirectly through role membership) to execute the query under consideration during

False Negatives	3.4%
False Positives	16%
Recall	84%
Precision	96.1%

Table 2: Classifier Accuracy

the access control enforcement procedure.

The major portion of the information (statistics) required to carry out the AD task are collected during the access control enforcement procedure. The collected statistics include the command count, the table access count, and the column access count on a per role basis. We assume that a strict RBAC model is under operation and thus all privileges required to access any portion of the database table are inherited by a user's role membership. Thus, the table and column statistics are aggregated on a per role basis before being sent to the statistics collector. The statistics collector, upon receiving the statistics, updates the memory resident role profiles that are then periodically recorded in our system.

The AD algorithm is executed on the query parse tree so there is no need to parse the query every time to get the features required for the detection task. The algorithm uses pre-defined functions to access the statistics required for the NBC, and the result of the AD algorithm is whether an anomaly has been detected or not. We mark a query as anomalous if the role associated with the database user (submitting the query) does not match the role predicted by the NBC. Our current implementation, thus, only supports single role activation by a single user per session.

The performance of our classifier is tested by using a real dataset to train the system, containing 6602 SQL traces from 4 different applications submitting queries to an MS SQL database server [2]. The database itself consists of 119 tables with 1142 attributes in all. To generate anomalous queries, we change the role information in normal queries. Table 2 shows the precision/recall and false-positive and false-negative statistics, representing the accuracy of our classifier. With fine triplets, we were able to achieve high accuracy of detection.

4. DEMONSTRATION

For demonstrating our work, we have extended the interface for pgAdmin, that is, the GUI interface for PostgreSQL [9]. Figure 2 shows the extended pgAdmin query interface that allows users to submit SQL queries which generate an execution report after every transaction. Specifically, we allow the audience to toggle between the training and the detection phase and specify the roles under AD, and test the detection mechanism implemented. We also demonstrate the different behaviors of our system depending on the various input parameters (such as role probabilities, *etc.*.), and we display the corresponding statistics of the DBMS and performance measurements of our detection algorithm when varying the input workload. Details are discussed in what follows.

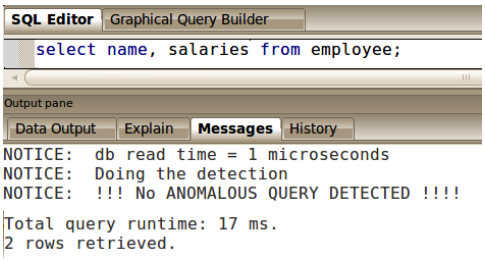

Figure 2: pgadmin query interface and transaction details.

The anomaly detection:

For demonstration purposes, we prepared a set of different synthetic datasets that can train the AD mechanism in many different scenarios. These datasets are carefully prepared to demonstrate different schemes for each training session based on different roles. After these files are given in input to the AD mechanism for training, the system will report the time spent in training the classifier.

Our audience will also have the option to train the AD mechanism through customized input queries and will be able to perform training without using our preset synthetic datasets. After the training phase is over, the audience can submit arbitrary queries to the system to test its AD capabilities. While choosing the detection option and upon the execution of each query, the execution time is displayed and the detection time is reported. Since our implementation of the AD mechanism is learning-based, any query that is not anomalous is used to train the classifier even while the system is operating in the detection mode. At the end of each query execution, the time spent in this feedback operation is reported as well.

Statistics:

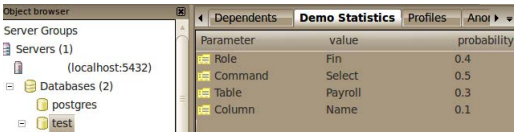

Figure 3: Anomaly Detection Statistics interface.

In Figure 3, we show the extended pgAdmin interface for displaying the various AD statistics (profile for a role). For a specified role, we display the probability of specific command types, tables, and columns. For example, if we choose the 'Fin' role and the 'Select' command, the command probability shown in Figure 3 is computed using $P(command = Select|role = Fin)$. Similarly, if we choose the 'Fin' role, the 'Payroll' table, the 'Select' command, and the 'Name' column, the column probability is computed using $P(command = Select, column = Name, Table = Payroll|role = Fin)$. These statistics are indicative based on the type of query that can be accepted (assumed not to be anomalous) from the user holding this specific role.

Performance Analysis:

We allow the audience to assess the performance of the AD mechanism when varying a number of input parameters.

1. **Changing the Database size:**

   In this set of experiments, the audience can run tests on an increasing size of the database to measure the anomaly detection time and detection overhead when executing queries. The database size is measured in terms of the number of tables and columns per table. To test the performance of our system, we initialized a table of 10 columns with 100 rows of random data. The database is configured with 3 roles and 3 users. Every user is assigned to exactly one role in the database. For training the classifier, we divide the tables into 3 groups; each user submits a query on each table in one of the groups. We measure the performance of our system by executing 5 runs; in each run the database size is doubled, starting with database size of 5 tables. Figure 4 reflects the results of executing 5 SELECT, 5 INSERT, and 5 UPDATE commands using the fine triplet representation. The detection time for the fine triplets, however, does increase and becomes noticeable when the database size is large. This is primarily because the

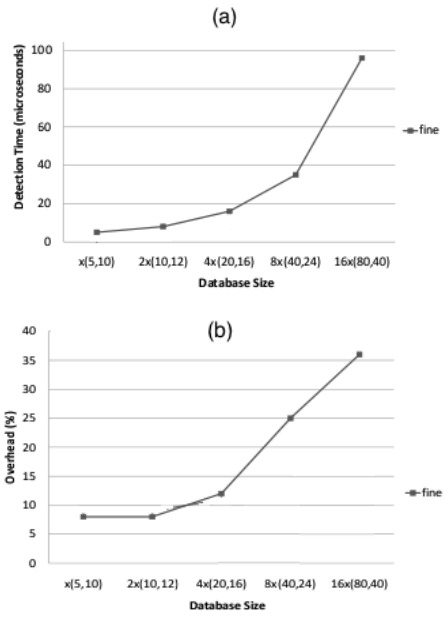

Figure 4: Changing Database size affecting: (a) Detection time (b) Detection overhead.

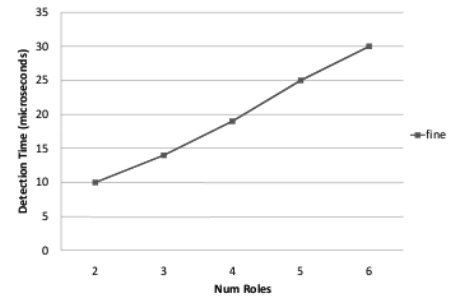

Figure 5: Detection time affected by number of roles.

number of features considered by the detection algorithm for fine triplets becomes very large in case of a large database size. For example, considering the database size of $16x$, corresponding to a database of 80 tables and 40 columns per table, gives rise to $80 * 40 = 3200$ features to be considered by the detection algorithm (for calculating likelihood of every role) for the fine triplets. Thus, if fine triplets are being used for the AD procedure, the number of tables in the database that need to be considered for the AD procedure must be carefully configured so as not to adversely impact the performance of the database. While the detection time remains constant for the case of simple and join queries, the overhead for simple queries is large, while it is less in the join queries case as the time spent in query processing itself is large.

2. **Anomaly Detection time *vs.* Number of roles:**

   In this experiment, the audience can measure the effect of increasing the number of roles in the database on the AD time. The audience can define the base number of roles and the maximum number allowed to use in the DBMS, where each user belongs to one role. For our experiment we set the base number of roles to 2 and the maximum number to 6. The result of executing 5 SELECT, 5 INSERT, and 5 UPDATE commands is plotted in Figure 5. The detection time increases with increasing the number of roles because in the AD algorithm we have to calculate the role likelihood for every role in the database. However, the performance impact is small since computing the role likelihood only requires a summation over the relevant feature probabilities.

5. CONCLUSION AND FUTURE WORK

In this work, we have demonstrated the design and implementation of an anomaly detection mechanism implemented in the PostgreSQL DBMS. We have developed interfaces in the pgAdmin tool for the audience to interact with our system. Our demonstration allows the audience to test the capabilities of the anomaly detection mechanism by first training it manually or using synthetic query traces (generated using probability sampling), then by testing whether an arbitrary query under anomaly detection is detected as anomalous or not.

We plan to extend this work by adding support for *group-by* type of queries that can help train our system in further anomalous situations. We also plan to extend our approach by also considering the query semantics in terms of data values used for the query computation. Another complementary area for future research is the response of the system when an intrusion is detected.

6. REFERENCES

[1] E. Bertino. *Data Protection from Insider Threats*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2012.

[2] E. Bertino, E. Terzi, A. Kamra, and A. Vakali. Intrusion detection in rbac-administered databases. In *Computer Security Applications Conference, 21st Annual*, pages 10 pp. –182, dec. 2005.

[3] C. Y. Chung, M. Gertz, and K. Levitt. Demids: A misuse detection system for database systems. In *Proceedings of the Integrity and Internal Control in Information System*, pages 159–178, 1999.

[4] Y. Hu and B. Panda. A data mining approach for database intrusion detection. In *Proceedings of the 2004 ACM symposium on Applied computing*, SAC '04, pages 711–716, New York, NY, USA, 2004. ACM.

[5] A. Kamra and E. Bertino. Design and implementation of an intrusion response system for relational databases. *Knowledge and Data Engineering, IEEE Transactions*, 23(6):875 –888, June 2011.

[6] A. Kamra, E. Terzi, and E. Bertino. Detecting anomalous access patterns in relational databases. *The VLDB Journal*, 17(5):1063–1077, Aug. 2008.

[7] S. Y. Lee, W. L. Low, and P. Y. Wong. Learning fingerprints for a database intrusion detection system. In *Proceedings of the 7th European Symposium on Research in Computer Security*, ESORICS '02, pages 264–280, London, UK, UK, 2002. Springer-Verlag.

[8] S. Mathew, M. Petropoulos, H. Q. Ngo, and S. Upadhyaya. A data-centric approach to insider attack detection in database systems. In *Proceedings of the 13th international conference on Recent advances in intrusion detection*, RAID'10, pages 382–401, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] pgadmin development team. pgadmin postgre sql tools.

[10] A. Spalka, J. Lehnhardt, S. Jajodia, and D. Wijesekera. *A Comprehensive Approach to Anomaly Detection in Relational Databases*, volume 3654, pages 924–924. Springer Berlin / Heidelberg, 2005.