

8-2016

Monitoring DBMS activity to detect insider threat using query selectivity

Prajwal B. Hegde
Purdue University

Follow this and additional works at: https://docs.lib.purdue.edu/open_access_theses



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

Recommended Citation

Hegde, Prajwal B., "Monitoring DBMS activity to detect insider threat using query selectivity" (2016). *Open Access Theses*. 952.
https://docs.lib.purdue.edu/open_access_theses/952

This document has been made available through Purdue e-Pubs, a service of the Purdue University Libraries. Please contact epubs@purdue.edu for additional information.

**PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance**

This is to certify that the thesis/dissertation prepared

By HEGDE, PRAJWAL BHASKARA

Entitled

MONITORING DBMS ACTIVITY FOR DETECTING INSIDER THREAT USING QUERY SELECTIVITY

For the degree of MASTER OF SCIENCE

Is approved by the final examining committee:

Dr. ELISA BERTINO

Co-chair

Dr. VICTOR RASKIN

Co-chair

Dr. BAIJIAN YANG

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy of Integrity in Research" and the use of copyright material.

Approved by Major Professor(s): VICTOR RASKIN

Approved by: EUGENE H SPAFFORD

Head of the Departmental Graduate Program

7/26/2016

Date

MONITORING DBMS ACTIVITY TO DETECT INSIDER THREAT USING
QUERY SELECTIVITY

A Thesis

Submitted to the Faculty

of

Purdue University

by

Prajwal B. Hegde

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

August 2016

Purdue University

West Lafayette, Indiana

I dedicate this to my parents without whom this would all still be a dream.

ACKNOWLEDGMENTS

I'm very grateful to Dr. Lorenzo Bossi who helped and guided me throughout the duration of my thesis. I would like to acknowledge the support of my committee members Dr. Elisa Bertino, Dr. Victor Raskin and Dr. Baijian Yang for having faith in me. Lastly, I would like to thank my friends and colleagues who were with me through good and bad times.

TABLE OF CONTENTS

	Page
LIST OF TABLES	vi
LIST OF FIGURES	vii
ABBREVIATIONS	viii
ABSTRACT	ix
1 Introduction	1
1.1 Scope	3
1.2 Assumptions	4
1.3 Limitations	4
2 REVIEW OF RELEVANT LITERATURE	6
2.1 Existing Anomaly Detection Mechanisms:	6
2.2 DetAnom	7
2.3 Advantages of DetAnom over other AD mechanisms	9
2.4 Limitations in DetAnom	10
3 FRAMEWORK AND METHODOLOGY	12
3.1 Query Selectivity	12
3.2 How Query Selectivity is evaluated	13
3.3 Applying Query selectivity to DetAnom	15
3.4 Implementation	16
3.4.1 Less than '<':	16
3.4.2 Equal to '=':	17
3.4.3 Greater than '>':	18
3.4.4 NOT IN:	18
3.4.5 AND:	19
3.4.6 OR:	19

	Page
4 RESULTS AND ANALYSIS	21
4.1 Setup	21
4.2 Analysis	22
5 Summary	24
LIST OF REFERENCES	25
A Demonstrate Figures	26
B Demonstrate Tables	28

LIST OF TABLES

Table	Page
B.1 Table Details	28
B.2 Normal-Anomalous Query Comparison	28
B.3 Results Comparison I	29
B.4 Results Comparison II	30

LIST OF FIGURES

Figure	Page
A.1 Populated Student Table	26
A.2 Student Table Schema	27
A.3 Customer Table Schema	27

ABBREVIATIONS

DBMS	Database management systems
RDBMS	Relational database management systems
ADS	Anomaly Detection System
SQL	Standard Query Language
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
NBC	Naive Bayes Classifier
MAP	Maximum A posteriori Probability
QI	Query Interceptor
DFS	Depth First Search

ABSTRACT

Hegde, Prajwal Bhaskara, M.S., Purdue University, August 2016. Monitoring DBMS Activity To Detect Insider Threat Using Query Selectivity . Major Professor: Elisa Bertino.

The objective of the research presented in this thesis is to evaluate the importance of query selectivity for monitoring DBMS activity and detect insider threat. We propose query selectivity as an additional component to an existing anomaly detection system (ADS). We first look at the advantages of working with this particular ADS. This is followed by a discussion about some existing limitations in the anomaly detection system (ADS) and how it affects its overall performance. We look at what query selectivity is and how it can help improve upon the existing limitations of the ADS. The system is then implemented using Java on top of the existing query parser used by the AD mechanism which in itself is written in Java. Towards the end, we look at how our version of the anomaly detection mechanism using query selectivity fares against a Relational database management system (RDBMS) query optimizer. With high accuracy results that closely match the results produced by the underlying query optimizer, we provide some proof of concept(PoC) for adding query selectivity to the existing AD mechanism .We conclude that a tool to analyze SQL and evaluate query selectivity is required to make the anomaly detection mechanism more maintainable and self-sustained.

1. INTRODUCTION

Insider threat is often perceived as an act of disloyalty carried out by disgruntled or old employees from an organization who benefit from causing harm to the organization. This insider perpetrator may or may not carry out a full-fledged cyber-attack on his/her own but can aid malicious activities or actors who possess sufficient resources to engage in such an attack. Cyberattacks are costing companies millions and millions of dollars despite them trying to safeguard their systems and networks against hackers. Globally, the insider threat has grown because the value of enterprise data has increased significantly in recent years [1]. Accordingly, the challenge of detecting insider threats has been emerging as a big research topic in the field of information security over the last few years. Most of the existing defense mechanisms like intrusion detection systems, firewalls, anti-virus software only protect us against external facing threats.

One of the main advantages of using Database Management Systems (DBMS) and RDBMS over file systems was the ability to provide access control. DBMS enables database administrators to grant users and applications specific roles and privileges based on their requirements. These access control mechanisms however are not enough to safeguard our databases because they don't defend against malicious insiders and compromised applications. Hackers for instance can use some unknown vulnerability in a privileged application and use it to fire malicious queries. OS and network level firewalls will not be able to detect such anomalies because they only monitor system logs and network logs respectively. Moreover, insider threat can be more than just disloyal and disgruntled employees. Social and political factors that shape an individual's ethics and morals play a big role in information security policy compliance.

Given all these reasons, the need for an anomaly detection that detects malicious activities from previously trusted applications and users is imperative. Some of the previous work done in this topic demonstrate the integration of a DBMS specific anomaly detection (AD) mechanism within the core of the DBMS functionality [2]. User actions that are deemed malicious for a DBMS are not identified malicious by existing OS and network level AD systems. To understand this better, consider an example from the healthcare industry. Medical records that are not maintained electronically, are worth more than credit card information because of the possibility of its abuse. Healthcare professionals like receptionists, nurses and doctors have regular access to such healthcare records. Consider a receptionist who needs to access the contact information of a hospitals patients to confirm their daily appointments. She uses some privileged application with her login credentials and fires a query to obtain the contact numbers of all the patients listed for a particular day. Occasionally, she might need to access the patients address or emergency contact information. This information is part of the hospitals patient database and the receptionist has been granted privilege to access it. Now, suppose that the residence and office address of a patient was queried using the receptionists login credentials on a particular day. How can this activity be flagged? This can be a premeditated attempt to steal the data or attackers getting access to the privileged application in some manner.

Most of the RDBMS systems which use SQL maintain their logs in forms of SQL queries which are not parsed by existing OS and network level AD systems. The approach described by Shebaro, B. et al. [2] in the above paper describes the implementation of an AD system that parses such SQL query logs and finds anomalies. In some proceeding work done in [3], we come across DetAnom which is an anomaly detection mechanism that does the AD work in 2 stages: Profile creation phase and anomaly detection phase. In the profile creation phase, DetAnom creates a profile of the application program which can succinctly represent the normal behavior in terms of its interaction with the database. After a sufficient profile creation, the

anomaly detection phase is used to detect anomalous queries before it even reaches the database. One of the main advantages of using this type of AD detection is that it tightly integrates the AD mechanism with the database. In other words, the anomaly detection is active and not passive.

However, we found some exceptional cases that can be termed as anomalies but are not caught by this AD engine. For each query in the profile creation phase, the corresponding signature and constraints are recorded. This signature is composed of a tuple representation of the query which is used to depict query features like table name, selected columns, and filter columns. When the system is in anomaly detection phase, these signatures and constraints are compared with the new query to decide if the query is anomalous or not. The scope of this research was to identify anomaly cases that were not handled by this existing AD mechanism. After identifying such cases, the end result of this research would be to add additional features to the AD mechanism to deal with such exception cases.

In this thesis, we have proposed and implemented an addition to the DetAnom mechanism by adding query selectivity to it. Using query selectivity, we were able to add more anomaly detection features to the mechanism making it more maintainable and self-sustained. We built this implementation on some ongoing research in query selectivity [4] to improve anomaly detection. By calculating the selectivity of queries fired by privileged applications against its database, we get an idea of the volume of data returned by these queries. Using this volume estimation, we can further fine tune the existing AD mechanism in DetAnom.

1.1 Scope

1. Identify queries that are exceptions and not handled by the existing DetAnom AD mechanism.

2. Identify additional functionality that should be added to the AD mechanism to handle these exceptional cases.
3. Evaluate query selectivity and how it will help make the DetAnom AD mechanism more maintainable and autonomous.
4. Using bulk data generation tools, compare the selectivity obtained by this extended DetAnom mechanism with the selectivity obtained by the underlying RDBMS query optimizer.

1.2 Assumptions

1. We assume that the user interacts with the database using SQL commands only.
2. We assume that the underlying database has role based access control (RBAC) where in the users are given privileges based on their roles.
3. The anomaly detection module will work in similar sync after adding additional query selectivity features to the query parser module.
4. Query selectivity for complex queries can be evaluated similarly and will remain consistent with the outcome of this thesis.
5. This additional module will improve DetAnom's anomaly detection accuracy but it will not make DetAnom foolproof.
6. Other DBMS systems (besides PostgresQL) will produce similar results.

1.3 Limitations

1. No connectivity with the back-end database. We are working only with the query parser module of the AD mechanism which intercepts every query before it reaches the database.

2. Within the scope of this thesis, we limit ourselves to handle mostly simple queries which use logical operators like AND, OR and NOT alongside conditional operators like $<$, $>$ and $=$.
3. Complex queries with subqueries and JOIN operators are not handled by the existing query selectivity addition.

2. REVIEW OF RELEVANT LITERATURE

In this section, we review all the existing work done on RDBMS anomaly detection. We look at the pros and cons of different AD mechanisms that exist and why this thesis is aimed on DetAnom? We look at the key advantages that separate this anomaly detection mechanism from other ADS. Lastly, we address the limitations that exist in DetAnom.

2.1 Existing Anomaly Detection Mechanisms:

As discussed in the introduction section, AD approaches for database systems came into play after the advent of relational database management systems over traditional file systems. RDBMS is the basis for the standard query language (SQL) which in turn is the basis for most database systems like MySQL, PostgreSQL, etc. These database systems use SQL queries to carry out its functionalities. SQL queries, unlike system and network level queries cannot be parsed by traditional AD systems that monitor operating systems or networks. While tools like intrusion detection systems (IDS), Intrusion prevention systems (IPS) and antivirus software among others defend our IT infrastructure from an external attacker trying to penetrate our systems, they are rendered useless against insider threats. The problem of securing any and all data from such insider threats is very complicated and requires the right combination of tools and policies to overcome it. Database based anomaly detection provide an anomaly detection tool that can parse through SQL queries – used by the user to interact with the database to flag anomalous behavior. Previous work suggests that this approach can be both syntactic and semantic.

Spalka et al. [5] proposed an approach of detecting anomalies using the state relations which depends upon the users behavior and the different data points the user hits during normal transactions. According to Spalka et al., their ADS is based on how a user executes syntactically related commands, which place a specific load on the DBMS. Mathew et al. [6] talks about another approach that models the users database access using a multidimensional vector and then uses a training phase to quantify these vectors. Chung et al. [7] proposed distance measure as a variable to understand the data structure and semantics associated with the given query. This distance measure is a quantity that explains the frequency of use of a particular dataset with respect to the currently working scope. Overall, we see that several syntactic and semantic attributes of SQL queries have been taken into consideration to establish a baseline that distinguishes between normal and anomalous queries.

2.2 DetAnom

In the PostgreSQL anomalous query detector demonstrated by Shebaro, B. et al. [2], the 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. Every SQL query is intercepted before it reaches the database. The mechanism is split into two stages: the training phase and the anomaly detection phase.

In the training phase, the mechanism relies on intrusion free transactions between authorized users and the database. With every transaction, the underlying access patterns are associated with user roles using a fine triplet representation. A fine triplet representation was first introduced by Bertino et. al [8]. Triplets are our basic unit for viewing the log files and are the basic components for forming user and role profiles, since subjects actions are characterized by sequences of such triplets. For sake of simplicity in the adopted notation, we represent a generic triplet with $T(c, R,$

A) 1, where c corresponds to the command, R to the relation information and A to the attribute information. These fine triplets of SQL queries are then fed to a Naive Bayes Classifier (NBC) which then outputs a corresponding identifier for every role of the role based access control.

In the anomaly detection phase, Maximum A posteriori Probability (MAP) is used to determine the role associated with a new query. 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.

DetAnom [3] was built on top of this PostgreSQL anomalous query detector. This mechanism was designed to carry out concolic execution of a privileged application program that interacts with the database. Similar to the PostgreSQL anomalous query detector, DetAnom works in two stages: Profile creation and anomaly detection. In the profile creation phase, the application program is instrumented in a concolic execution environment where in different constraints corresponding to different input values are applied. The objective is to cover all paths possible. Thus, depending on the outcome, new inputs are provided that reverse branch conditions. All of these generated queries are passed on the profile builder module and then to the respective database. Once the query reaches the profile builder, corresponding query signatures are generated that are similar to the triplet representation used in the above mentioned paper. Queries' signatures and corresponding constraints are used to build the profile of the application. Thus, every query in the profile creation phase gets associated with a query record which is a combination of its constraints and associated query signature.

The anomaly detection phase of DetAnom involves each new query passing through the QI (Query interceptor) to the ADE (Anomaly detection engine). Here the first step is identifying the inputs of the executing application program. Based on the in-

put parameters, the ADE finds the corresponding constraints and simultaneously the query records associated with it. The signature generation sub-module then regenerates a signature of this received query and forwards it to the signature comparison sub-module which verifies if the newly generated signature matches with the one in the accompanying query record. If there is a mismatch, the AD engine raises a flag and response actions are taken as specified. Strict and Flexible denote two forms of response action that the ADE can take.

2.3 Advantages of DetAnom over other AD mechanisms

Some of the key advantages of using DetAnom in this thesis over other database AD mechanism are as follows:

1. The anomaly detection mechanism is active. Every query is intercepted by the DetAnom mechanism before it reaches the database. This eliminates the possibility of a back door entry that is observed in other passive AD mechanisms.
2. The anomaly detection mechanism is implemented as an additional feature of the database which makes the physical location of the underlying database unimportant. This feature is particularly important in today's day and age as enterprises are trying to move their data to the cloud or use third party database services.
3. DetAnom is tightly integrated with the database which makes it handy to do more than redirect anomalous queries. Several useful additions have been suggested to this mechanism over the years; this research being one of them. Database administrators for instance can handle different types of anomalous queries differently and implement response actions accordingly.
4. Since all SQL based database systems have the same nomenclature, DetAnom is not restricted to any one form of DBMS and can be used to detect anomalies in MS SQL, PostgreSQL and Oracle among others.

2.4 Limitations in DetAnom

The query signature generation mechanism used in DetAnom only keeps track of the tables and columns names mentioned in the query. Consider the following as example in [3]:

Query:

```
SELECT employee_id, work_experience FROM WorkInfo WHERE work_experience > 10;
```

Query signature:

$$\{1, \{\{200, 1\}, \{200, 2\}\}, \{200\}, \{\{200, 2\}\}, 1\}$$

This query signature is generated using the metrics in Table B.1 in Appendix.

Reading the query signature, we understand that the query is to SELECT (1) employee_id (200, 1) and work_experience (200,2) from WorkInfo (200). There where condition depends on work_experience (200, 2) and has 1 constraint (1).

This is a pretty accurate representation; more accurate than some other metrics used in other database ADS. However, as we can see, the query signature doesnt keep track of the constraints in the where condition except for the column name and count. This gives rise to some potential anomalous queries that cannot be detected by DetAnom.

To understand this limitation more clearly consider the following examples of queries from Table B.2 in Appendix that will return different results but will be treated the same by DetAnom.

DetAnom will generate the same query signatures for each of the query pairs shown in Table B.2 in Appendix. Hence, DetAnom will not raise any flags if a privileged

user is trying to fire queries that have the same table and column names but differ in some where condition constraints.

3. FRAMEWORK AND METHODOLOGY

In the following subsection, we look at:

1. What is query selectivity?
2. How can it be added to DetAnom?
3. How will it improve its overall performance?

3.1 Query Selectivity

Several relational database management systems maintain statistics in system catalogs that are used to estimate the cost of operation and result sizes. These statistics are maintained in system catalogs are used by query optimizers to estimate the cost of operations and result sizes. Consider the System R Optimizer [9], which is one of the most widely used query optimizers that works well for queries with less than 10 join conditions. When the number of joins in the query grows large however, this can take an enormous amount of time and space. The query optimizers consider a combination of CPU and I/O costs before deciding the most optimized path to take to carry out the query functionality. There are innumerable ways to go about this; the system R optimized only traverses the left-deep plans to find the most optimal path. Here left deep plans indicate the output of each sub-part of a query is pipelined to its adjacent left operation without storing it. Some other plans include single-relation plans and multiple-relation plans. Depending on the underlying query plan, the query optimizer estimates the CPU, I/O costs as well as the result sizes.

According to PostgreSQL documentation [10], The current PostgreSQL optimizer implementation performs a near-exhaustive search over the space of alternative strate-

gies. This algorithm, first introduced in the "System R" database, produces a near-optimal join order, but can take an enormous amount of time and memory space when the number of joins in the query grows large.

While estimating result size, which indicates the maximum number of tuples returned in a query result, the query optimizer calculates a reduction factor (RF) based on the where condition. Reduction factor of a term in the WHERE clause is the ratio of (expected) result size to input size [11]. So consider a query as following:

Query:

```
SELECT * from students WHERE age > 20;
```

Here the number of tuples returned by the first part of the query `select * from students;` is limited by the conjunct where `age > 20;`. Thus the reduction factor will be calculated as the number of tuples returned by the above query divided by the number of tuples returned by same query without the where clause. Query selectivity is one of the parameters evaluated by the query optimizer to find the most optimized access path. Unlike RF, which evaluates each where clause conjunct, query selectivity is used to compare access paths. To better understand the interdependency between query selectivity and reduction factor, we say that query selectivity depends upon the several conjuncts used in the query and the fraction of tuples that satisfy a given conjunct is defined by the RF. In some cases, (not all), the query selectivity can be the product of the reduction factors for each of the conjuncts in the where clause.

3.2 How Query Selectivity is evaluated

We refer to the official PostgreSQL documentation [12] to understand how query selectivity is evaluated by its optimizer. Note that the reference to PostgreSQL is because the implementation of this thesis was performed on top of PostgreSQL v9.5. Similar query optimizers exist in almost all SQL based database systems.

Consider the same query listed above:

```
select * from students where age > 20;
```

As per the given documentation, "The planner examines the WHERE clause condition and looks up the selectivity function for the *operator < inpg_operator*. This is held in the column *oprrest*, and the entry in this case is *scalarltsel*. The *scalarltsel* function retrieves the histogram for *unique1* from *pg_statistics*."

Here, the histogram is one of the data structure in *pg_statistics* that divides the range of values in each column into buckets. For the above query, the planner (query optimizer) will retrieve the corresponding histogram by running the following query in the background:

Query:

```
SELECT histogram_bounds FROM pg_stats WHERE tablename = ' students'
AND attname = ' age';
```

Result:

```
          histogram_bounds
```

```
{0, 5, 25, 50, 60, 100, 160, 172, 220, 400, 995}
```

Using this *histogram.bound* values, the query selectivity is evaluated as follows:

$$\begin{aligned} \text{Selectivity} &= (1 + (20 - \text{bucket}[2].\text{min}) / (\text{bucket}[2].\text{max} - \text{bucket}[2].\text{min})) / \text{num_buckets} \\ &= (1 + (20 - 5) / (25 - 5)) / 10 \\ &= 0.175 \end{aligned}$$

This indicates one whole bucket (0.1) plus a linear fraction of the second bucket (0.075).

The number of rows returned can be calculated by:

$$\begin{aligned} \text{No. of rows} &= \text{selectivity} * \text{cardinalityof table}(\text{students}) \\ &= 0.175 * 100 \\ &= 17.5 \end{aligned}$$

This indicates that the query optimizer has now evaluated this access path obtaining a selectivity of 0.175 which eventually will return around 18 tuples. Notice that we have not actually parsed through the database to find out how many students aged 20 exist but are just using the histogram values. These statistics are then compared against the selectivity of other access paths and the optimal query execution path is selected.

3.3 Applying Query selectivity to DetAnom

As discussed in the Limitations of DetAnom section, the tuple representation used in DetAnom does not account for the constraints in the where condition except for its count and column names. Query selectivity is evaluated using the constraints applied to a query in the where clause. Thus, in this research, we suggest that we add an extra component to the query signature that indicates the query selectivity of the given query.

Lets consider the following two queries as given in Table 2.2:

Legitimate Query:

```
SELECT * from WorkInfo WHERE salary > 150000
```

Anomalous Query:

```
SELECT * from WorkInfo WHERE salary > 0
```

As discussed in the above section, the selectivity value for the first query will be calculated based on the logical operator used in the where clause ($>$) and comparison value (15000). Similarly, the selectivity for the second query will be based on the logical operator used in its where clause ($>$) and comparison value (0). Based on the formula used in the above section, we understand that the selectivity value for the anomalous query will be much higher than the legitimate query. In other words, we now understand the selectivity of queries issued against the database; which indicates the volume of data returned by that query can be used to differentiate anomalous queries from legitimate ones. Adding this query selectivity component to DetAnom

will enable the AD mechanism to detect anomalies in the volume of data return before even executing the query. Our research is supported by some other ongoing research [4] that explain how adding query selectivity improves the precision of the anomaly detection. In this following section, we look at how query selectivity can be calculated for different types of queries and how it can be integrated with the existing DetAnom mechanism.

3.4 Implementation

The profile creation phase of DetAnom uses a general SQL parser [13] that is used to dissect the SQL queries into components as desired by DetAnom. This SQL parser is written in JAVA and is capable of parsing through a query and separating out its tokens in a tree type representation. Since we have implemented the query selectivity functionality on top of this SQL parser, we work through the plan tree of the processed query from top to bottom in a Recursive-DFS (Depth first search) type algorithm. We calculate the individual selectivities for each where clause conjunct and then based on the operators used, calculate the overall query selectivity. The formulas for different comparison and expression type of SQL tokens and how their respective selectivities are calculated is given below:

3.4.1 Less than '<':

Query syntax:

```
SELECT column1 from table1 WHERE column2 < value;
```

As indicated in one of the examples above, the PostgreSQL documentation shows that the selectivity for '<' operator is calculated by the 'scalarltsel' function using its histogram_bounds. We query the underlying database to obtain the histogram of the respective column and use it in the following formula:

Optimizer query:

```
SELECT histogram_bounds FROM pg_stats WHERE tablename =
table1' AND attname = ' column2';
```

$$Selectivity = ((i-1)+(value-bucket[i].min)/(bucket[i].max-bucket[i].min))/num_buckets$$

Where,

- i : Index of the bucket in which the value lies
- bucket[i].min: lower bound of the ith bucket
- bucket[i].max: upper bound of the ith bucket
- num_buckets: total number of buckets

3.4.2 Equal to '=':**Query syntax:**

```
SELECT column1 from table1 WHERE column2 = value;
```

The selectivity of queries with = operator is determined using the function for =, which is eqsel. The eqsel function in turn looks up the most common values (MCVs), most common frequencies (MCFs) and number of distinct values (num_distinct) data structures to evaluate its selectivity.

Optimizer query:

```
SELECT n_distinct, most_common_vals, most_common_freqs FROM pg_stats WHERE
tablename='table1' AND attname='column2';
```

Now, if the value exists in the MCVs, its selectivity is the MCF value corresponding to the same index.

If the value does not exist in the MCVs, selectivity is calculated using:

$$Selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)$$

Where,

sum(mvf): sum of all values in the MCF list

num_distinct: number of distinct values in column2

num_mcv: count of MCVs

3.4.3 Greater than '>':

Query syntax:

```
SELECT column1 from table1 WHERE column2 > value;
```

The maximum selectivity value of a given query can be 1. We have already discussed the selectivity for less than (<) and equal to (=).

We evaluate the selectivity of queries with > operator by subtracting the selectivity of the same query with < and = operator from 1.

$$\text{Selectivity} = 1 - \text{selectivity}(\text{SELECT column1 from table1 WHERE column2} < \text{value;}) - \text{selectivity}(\text{SELECT column1 from table1 WHERE column2} = \text{value;})$$

3.4.4 NOT IN:

Query syntax:

```
select column1 from table1 where column2 NOT IN (value1, value2);
```

Again, similar to queries with > operator, we use mathematical logic to evaluate the selectivity of queries with NOT IN operator.

$$\text{Selectivity} = 1 - \text{selectivity}(\text{SELECT column1 from table1 WHERE column2} = \text{value1;}) - \text{selectivity}(\text{SELECT column1 from table1 WHERE column2} = \text{value2;})$$

3.4.5 AND:

Query syntax:

```
SELECT column1 from table1 WHERE column2 < value1 AND
column3 = value2;
```

Here, according to PostgreSQL documentation [12], The planner assumes that the two conditions are independent, so that the individual selectivities of the clauses can be multiplied together:

Hence,

$$\text{Selectivity} = \text{selectivity}(\text{SELECT column1 from table1 WHERE column2 < value1;}) * \text{selectivity}(\text{SELECT column1 from table1 WHERE column3 = value2;})$$

3.4.6 OR:

Query syntax:

```
SELECT column1 from table1 WHERE column2 < value1 OR column3 =
value2;
```

Using properties of set theory, we evaluate the selectivity of queries with OR condition as,

$$\text{Selectivity} = \text{selectivity}(\text{SELECT column1 from table1 WHERE } *column2* < *value1*;) + \text{selectivity}(\text{SELECT column1 from table1 WHERE } *column3* = *value2*;) - \text{selectivity}(\text{SELECT column1 from table1 WHERE } *column2* < *value1* AND *column3* = *value2*;)$$

Now, considering the research purpose of our thesis is to explain that a tool to analyze SQL and evaluate the query selectivity is required to make that project more maintainable and self-contained, we have not integrated more complex queries involv-

ing nested queries, JOIN conditions, sub-queries etc. Note that some of the complex queries can also be simplified into queries which involve the basic set of operators discussed above.

4. RESULTS AND ANALYSIS

To evaluate the efficiency of our query selectivity module in DetAnom, we compare the selectivity and number of rows calculated by our project with the corresponding value obtained from PostgreSQL and its query optimizer. Note that we can do the same with other database engines like MySQL and Oracle since the underlying parser works for all SQL based DBMS systems.

4.1 Setup

Firstly, we need a tool to populate test data and transactions. With sufficient test data, we aim to expose any potential database performance issues. We looked at some benchmarking tools like TPC-C and OLTPBench that generate arbitrary tables and columns and provide a set of queries (in the form of transactions) that can be run against them. In the end, we decided on using named 'Datanamic Data Generator for PostgreSQL v6.1.0' [14]. We found this tool to be quite handy in creating different types of tables and populating them with meaningful attributes and columns depending upon our requirements. Several versions of Datanamic data generator exist for other database systems like Oracle, MS SQL, MS Access, Azure, etc.

After connecting to the PostgreSQL database server, we created a table named students and generated data according to specific requirements. In other words, it provides a well-defined data generation plan. The id column for instance was filled with sequential integers that start with 1 and increment by 1. The dept column was filled with a pre-existing list of department names. The departments will repeat 1 in 40 times; this was done so as to check the selectivity of queries that match a specific department name. Similarly, the name and age columns were also populated with

specific constraints to aid our calculations. Overall, 1000 tuples were populated as shown in Fig A.1 in Appendix. The syntax of the table is shown in Fig A.2 in Appendix.

Similarly, other tables named 'customer', 'owner', etc. are created. Some with only 50 tuples while others with around 5000 tuples. and its syntax is shown in Fig A.3 in Appendix.

4.2 Analysis

Now after populating the databases, we calculate the numbers of rows using our module and then using PostgreSQL try to gauge its accuracy. Before doing so, we run ANALYZE which populates the statistical columns like pg_stats within the PostgreSQL query generator. Some of the queries along with their selectivities are given in Table B.3 in Appendix.

Based on our evaluation of 50 queries, the accuracy of the query selectivity module implemented in this thesis is as provided in the Table B.4 in Appendix.

Based on these results, we come to the following observations:

1. The results produced by our query selectivity module (DetAnom using Query selectivity) is very accurate ($> 98\%$) for logical comparison operators like less than ($<$), greater than ($>$) and equal to ($=$).
2. These results are accurately replicated by the PostgreSQL query optimizer as well. Accuracy is approximately 100%

3. The accuracy of both 'DetAnom with query selectivity' and the PostgreSQL query optimizer for the NOT IN condition is almost perfect (100%) for our sample size.
4. The accuracy of 'DetAnom with query selectivity' for the queries with AND condition is very high (91.66%) and quite close to the accuracy of the PostgreSQL query optimizer (94.4%).
5. The accuracy of 'DetAnom with query selectivity' for the queries with OR condition is pretty high as well (88.8%) and quite close to the accuracy of the PostgreSQL query optimizer (92.2%).

5. SUMMARY

Overall, the implemented version of DetAnom with query selectivity was found to be very accurate (Overall accuracy $> 88\%$) in calculating query selectivities and very similar to the statistics obtained by the underlying query optimizer.

We started off with identifying anomalies in SQL based DBMS systems to detect insider threat in an organization. We saw that adding query selectivity as an additional constraint to the query signature during the training phase will improve its overall accuracy. After carrying out the research discussed in this thesis, we propose that adding query selectivity to the DetAnom ADS and other similar syntactic based anomaly detection systems will help us identify further anomalies and thereby defend against insider threat.

LIST OF REFERENCES

LIST OF REFERENCES

- [1] G. Fyffe, “Addressing the insider threat,” *Network security*, vol. 2008, no. 3, pp. 11–14, 2008.
- [2] B. Shebaro, A. Sallam, A. Kamra, and E. Bertino, “Postgresql anomalous query detector,” in *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT ’13, (New York, NY, USA), pp. 741–744, ACM, 2013.
- [3] S. R. Hussain, A. M. Sallam, and E. Bertino, “Detanom: Detecting anomalous database transactions by insiders,” in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY ’15, (New York, NY, USA), pp. 25–35, ACM, 2015.
- [4] A. Sallam, E. Bertino, S. R. Hussain, D. Landers, R. M. Lefler, and D. Steiner, “Dbsafe: An anomaly detection system to protect databases from exfiltration attempts,” *IEEE Systems Journal*, vol. PP, no. 99, pp. 1–11, 2015.
- [5] A. Spalka and J. Lehnhardt, *A Comprehensive Approach to Anomaly Detection in Relational Databases*, pp. 207–221. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005.
- [6] S. Mathew, M. Petropoulos, H. Q. Ngo, and S. Upadhyaya, *A Data-Centric Approach to Insider Attack Detection in Database Systems*, pp. 382–401. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
- [7] C. Y. Chung, M. Gertz, and K. Levitt, “Demids: A misuse detection system for database systems,” in *Integrity and Internal Control in Information Systems*, pp. 159–178, Springer, 2000.
- [8] E. Bertino, E. Terzi, A. Kamra, and A. Vakali, “Intrusion detection in rbac-administered databases,” in *21st Annual Computer Security Applications Conference (ACSAC’05)*, pp. 10 pp.–182, Dec 2005.
- [9] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, *et al.*, “A history and evaluation of system r,” *Communications of the ACM*, vol. 24.
- [10] M. Utesch, *Genetic query optimization in database systems. Postgresql 6.3 Documentation*, 1997.
- [11] M. Christian, *A Survey of Database Query Optimization and Genetic Algorithms*, 2002.
- [12] PostgreSQL, *PostgreSQL 9.5.3 Documentation*, 1996-2016.
- [13] G. S. Parse, *Professional SQL engine for various databases*, 2002-2016.

APPENDICES

A. DEMONSTRATE FIGURES

Edit Data - PostgreSQL 9.5 (localhost:5432) - employee - public.student

File Edit View Tools Help

100 rows

	id [PK] integer	dept character(50)	name character(50)	age integer
5	5	Facilities	Jules	65
6	6	Facilities	Jules	65
7	7	Facilities	Jules	65
8	8	Facilities	Ron	65
9	9	Facilities	Ron	65
10	10	Facilities	Al	65
11	11	Facilities	Al	65
12	12	Facilities	Al	65
13	13	Facilities	Al	65
14	14	Facilities	Harold	65
15	15	Facilities	Harold	65
16	16	Facilities	Nick	65
17	17	Facilities	Nick	65
18	18	Facilities	Nick	65
19	19	Facilities	Alba	65
20	20	Research and Development	Alba	56
21	21	Research and Development	Oliver	56
22	22	Research and Development	Oliver	56
23	23	Facilities	Oliver	56

Fig. A.1. Populated Student Table





	Name	Data type	Fill type
<input checked="" type="checkbox"/>	public.student		1000 rows
	id	INTEGER	Sequential integer
	dept	CHARACTER(50)	Business - Department name
	name	CHARACTER(50)	Personal - First name
	age	INTEGER	Random integer

Fig. A.2. Student Table Schema





	Name	Data type	Fill type
<input checked="" type="checkbox"/>	public.customer		200 rows
	cid	INTEGER	Sequential integer
	first_name	CHARACTER(50)	Personal - First na...
	last_name	CHARACTER(50)	Personal - Last na...
	credit_card_type	CHARACTER(20)	Business - Credit C...

Fig. A.3. Customer Table Schema

B. DEMONSTRATE TABLES

Table B.1
Table Details

Table ID	Table Name	Attribute Name	Attribute ID	Type
100	PersonalInfo	employee_id	1	varchar(10)
		employee_name	2	varhar(50)
200	WorkInfo	employee_id	1	vachr(10)
		work_experience	2	number
		salary	3	number
		performance	4	varchar(20)

Table B.2
Normal-Anomalous Query Comparison

NORMAL	ANOMALOUS
Select * from WorkInfo where salary >150000	select * from WorkInfo where salary >0
select * from WorkInfo where employee_id = 'x' AND salary >10000	select * from WorkInfo where employee_id = 'x' OR salary >10000
select * from WorkInfor where employee_id='x'	select * from WorkInfo where employee_id <>'x'

Table B.3
Results Comparison I

Query	No. of rows returned		
	DetAnom using Query selectivity	PostgreSQL Query Optimizer	Actual count
Select * from student where id <200;	200	199	199
Select * from student where dept = Facilities;	183	183	183
Select name from student where age >40;	546	546	546
Select * from customer where credit_card_type NOT IN (VISA,MASTERCARD);	184	184	184
Select * from customer where cid >100 AND first_name NOT IN ('Elene', 'Klara');	127	139	131
Select * from student where id <10 OR name = Duncan;	8	13	11
Select * from owner where age <40 and vehicle = Maserati;	6	9	7

Table B.4
Results Comparison II

Query	Percentage Accuracy	
	DetAnom using Query selectiv- ity	PostgreSQL Query Opti- mizer
Less than operator (<)	98.3	100
Greater than operator (>)	100	100
Equal to (=)	100	100
NOT IN conditions	100	100
AND condition	91.66	94.4
OR condition	88.8	92.2