ABSTRACT

ARJUN, VINOD. A Database Level Implementation To Enforce Fine Grained Access Control. (Under the direction of Dr. Ting Yu).

As privacy protection has gained significant importance, organizations have been forced to protect individual preferences and comply with many enacted privacy laws. This has been a strong driving force for access control in relational databases. Traditional relation level access control is insufficient to address the increasingly complex requirements of access control policies where each cell in the relation might be governed by a separate policy. In order to address this demand, we are in need of a more fine grained access control scheme, at the row-level or even the cell-level. A recent research paper proposed correctness criteria for query evaluation algorithms enforcing fine grained access control and showed that existing approaches did not satisfy the criteria. In addition, the paper proposed a query modification approach to implement a sound and secure query evaluation algorithm enforcing fine grained access control. To evaluate queries involving moderate table sizes of 50000 and 100000 records we experimentally find that the implementation takes approximately 8 and 32 seconds respectively. This is approximately 10 times, on an average, slower than query evaluation algorithms without access control. This performance gap increases significantly with increase in table size, thus rendering it impractical. In this thesis, we modify the query evaluation engine of POSTGRESQL to enforce fine grained access control at the database level. We address a few challenges and propose optimizations to counter inefficiencies that we encounter when moving the access control scheme to the database level. We analyze the performance of our implementation using data sets with various properties and find that it performs approximately 10 times better compared to the query modification approach on moderate table sizes of 50000 and 100000 records. Also, we find that our implementation scales well with table size. Experimental results show that our implementation is comparable to the performance of query evaluation algorithms without access control and hence is practical.

A Database Level Implementation To Enforce Fine Grained Access Control

by
Vinod Arjun

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fullfillment of the
requirements for the Degree of
Master of Science

Computer Science

Raleigh, North Carolina

2008

APPROVED BY:

Dr. Peng Ning	Dr. Rada Chirkova

Dr. Ting Yu
Chair of Advisory Committee

DEDICATION

To my parents

BIOGRAPHY

Vinod Arjun was born on 23^{rd} January, 1985. He received his Bachelor of Engineering degree, majoring in Computer Science and Engineering, from SSN College of Engineering, affiliated to Anna University, Chennai, India. He is currently a graduate student in the Department of Computer Science, North Carolina State University, Raleigh, USA.

ACKNOWLEDGMENTS

I would like to sincerely thank my advisor Dr. Ting Yu for his guidance. I thoroughly enjoyed working with him on this thesis topic. I cannot thank him enough for his constant motivation and support. His comments and suggestions were invaluable in improving the content of this thesis. I would also like to thank Dr. Rada Chirkova and Dr. Peng Ning for taking time off from their busy schedule and agreeing to serve on my thesis committee.

My appreciation also extends to Qihua Wang and Dr. Ninghui Li for sharing their work and providing helpful comments. I would like to thank all my friends in the Cyber Defense Lab for the wonderful times we shared together. Finally, I would like to specially thank my parents and sister for their love and moral support.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1

Introduction

As privacy protection has gained significant importance, organizations have been forced to protect individual preferences and comply with many enacted privacy laws. This has been a strong driving force for access control in databases. For example, in a typical bank setting, the bank database would contain the account information of many customers. When one customer tries to view his account details it is imperative that he view only his account information and not of any other customers'. The primary means of achieving this requirement is by enforcing appropriate access control.

Much work has been done on the specification of database access control policies [13, 15, 16]. Traditional policies treat tables or columns as the basic access control unit. Recently, security policy models have emerged to specify row-level or cell-level access control. Conceptually, with fine-grained access control, a policy defines for each cell whether it can be accessed by a user. In this thesis we focus on enforcing access control policies while evaluating a query issued by the user. We assume that the access control policies for each user are already in place. Hence, we do not focus on the specification of such policies.

1.1 Enforcing Fine Grained Access Control

There are two approaches to ensure the enforcement of access policies when a user queries a database. In the first approach, the access control is implemented as a middleware between the user and the DBMS. As shown in Figure 1.1, when the user issues a query Q, the middleware rewrites query Q to another query Q' , where Q' ensures only authorized

cells are accessed by the query, thus enforcing the access policies for that user. The rewritten query Q' is passed on to the DBMS for execution.

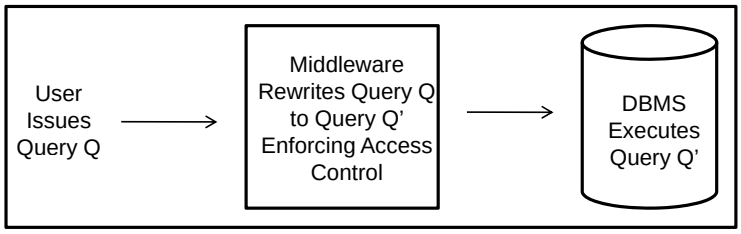

Figure 1.1: Application Level Access Control [1]

In this approach the underlying DBMS need not be changed. It is required only to write the middleware between the user and the DBMS in order to rewrite the query to enforce access control. It is easy to deploy in existing systems. On the other hand, the rewritten queries can be too expensive to execute. In addition, if the user has direct access to relations in the DBMS then he can bypass the middleware that rewrites the query, in which case the access policies do not get enforced [5].

In the second approach, access control is placed at the database level, as shown in Figure 1.2. In this approach, the query Q that the user issues does not get rewritten by a middleware as in the first approach. Instead, the DBMS receives and executes the query Q taking into account the access control policies formulated for the issuer of the query.

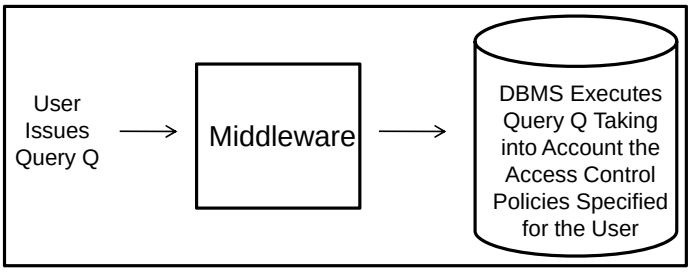

Figure 1.2: Database Level Access Control [1]

In this approach, the access control functionality is integrated into the DBMSs. As a result, it is possible to have optimizations targeted at access control. Another advantage with this approach is that it is harder to bypass the enforcement of access control policies

when they get integrated into the DBMS. The major drawback with this approach is that the underlying DBMS would have to be changed to support access control [5].

Some of the existing approaches to enforce fine grained access control include Oracle's Virtual Private Database [7], query modification in INGRES [8], access control in DB2 [9], and work in Hippocratic Databases [6]. How do we actually know that the results obtained by these query evaluation algorithms enforcing access control are indeed correct? Recently, Wang et al. [5] have proposed three correctness criteria that an ideal query evaluation algorithm enforcing fine grained access control should possess, namely, sound, secure and maximal. The algorithm is sound if the answer returned by it is consistent with the answer when there is no fine-grained access control. The algorithm is secure if the returned answer does not leak information not allowed by the policy. The algorithm is maximal if it returns as much information as possible, while satisfying the first two properties. In addition to the above stated three properties, the algorithm should be scalable, i.e. be able to handle databases of arbitrary size.

Unfortunately, none of the existing approaches satisfy all the correctness criteria proposed by Wang et al [5]. They argue that the existing approaches do not strive to achieve the maximal property. To appreciate this argument let us assume that there are two tables that are linked through an attribute A. If A is considered sensitive, then the existing approaches replace the cells in A with NULL value. As a result, it becomes impossible to maintain the association between the two tables. They also argue that the exisiting approaches violate the soundness property. They attribute this to the way in which existing approaches evaluate the SQL *set difference* operation. We encourage the readers to see the example in Section 3.1.1 to appreciate this argument.

Wang et al. have proposed a query evaluation algorithm to enforce fine grained access control, which satisfies the three correctness criteria of security, soundness and maximality. In their algorithm, they use named variables to mask unauthorized cells. If the attribute A involved in the association between two tables is considered sensitive, then they replace, consistently in both tables, each value of A with a different variable. This preserves the association between the two tables. In order to make their algorithm sound, they have introduced two different modes to evaluate queries and have proposed an evaluation technique

for queries involving SQL *set difference* operation, which have been discussed in Section 3.3. They have implemented their algorithm using a query modification approach.

Though their query evaluation algorithm was novel, Wang et al. were not able to show good performance through their query modification approach. For moderately sized tables having 50000 and 100000 records, their approach takes about 8 and 32 seconds to evaluate queries involving SQL *set difference* operation. For time critical applications this is not fast enough. Also, their approach does not scale well with tables of larger sizes. Thus, there is scope for further research in this area and this has precisely been the motivating factor for this thesis.

1.2 Our Contribution

- We have implemented a query evaluation algorithm that enforces fine grained access control at the database level. We have chosen to modify the query evaluation engine of POSTGRESQL, which is an open source database management system, to incorporate the access control features. In this way, the overhead that is involved in query rewriting is overcome. Though the task of incorporating access control into the DBMS is difficult, the performance increase compared to the query rewriting approach is substantial, as can be observed from the graphs in chapter 5.

- We have come up with a few optimizations to improve the performance of the database level implementation. We have used an index structure, which has been discussed in chapter 4, to efficiently evaluate queries involving SQL *set difference* operation.

- Another challenging issue that we have addressed is that of scalability. When relations of arbitrarily large sizes are used in queries, then memory management becomes an important issue. The main memory might not be sufficient enough to hold the entire relation, in which case we might have to turn to the secondary disk as an alternative. Our implementation tries to make use of the main memory in an adaptive fashion, i.e. try to use as much main memory as available. We have also tried out a few optimizations in this regard, which are discussed in chapter 4.

- We have run extensive experiments to analyze the performance of our implementation and compare it with the performance of the query modification approach proposed by

Wang et al. [5]. The experimental results, which are presented in chapter 5, conclusively show that our database-level implementation to enforce access control presented in this thesis performs substantially better compared with the query modification approach. In fact, the results presented in chapter 5 show that the performance of the DBMS with access control implemented within it is comparable to the original implementation of the DBMS that has no access control features. This implies that it is not too expensive to incorporate access control within DBMSs and hence is practical.

The rest of the thesis is organized as follows. In Chapter 2, we discuss some of the work done by other researchers related to access control in databases. In Chapter 3, we discuss the correctness criteria, a sound and secure query evaluation algorithm to enforce fine grained access control and the query modification approach to implement the algorithm, proposed by Wang et al. [5]. In Chapter 4, we elaborate on our proposed implementation. We explain the index structure and other optimizations that play a significant role in achieving good efficiency. We also present some of the challenges that we encounter when access control features are implemented within the DBMS, and propose solutions. In chapter 5, we discuss the experiments that we performed, and analyze the results that we obtained. In Chapter 6, we conclude this thesis and outline some of the work that we wish to undertake at some point in the future.

Chapter 2

Related Work

The work presented in this thesis is related to access control, privacy policy specification and access control enforcement. We first discuss the basic concepts of access control. Towards this end, we first discuss the modeling of access policies and then elaborate on modeling the system state that is relevant to access control. We then discuss the existing work with regard to specification of access control policies in DBMS. Finally, we discuss some of the most relevant work in enforcing access control policies in DBMS.

2.1 Basic Concepts of Access Control

Two major ways to model access control policies include Discretionary Access Control (DAC) and Mandatory Access Control (MAC).

In the DAC scheme, the access control policies are formulated by the owner of the object. There is no notion of system-wide access control policies. DAC is supported by many of the operating systems, including UNIX. For example, if Alice owns a file called "security_concepts.doc", then she has all privileges on that file. In addition, she can decide as to which users get read, write and execute rights. Thus, one good aspect of DAC is that it is flexible and allows the owner to authorize those who can have access to the object she owns. On the contrary, this scheme does not impose any kind of restriction on information flow. Let's suppose that Alice is an instructor for the CSC574 class at NCSU. She gives read, and write access to a file named "security_concepts.doc" to all the students in the class. Bob, who is registered for the class, gets read and write access to the file. He in turn copies this

file into a new file called "chapter1.doc" and gives his friend Carol read and write accesses to the file. Thus, Carol, who isn't registered for the CSC574 class, gets access to the file that she isn't supposed to. For this reason, the DAC scheme is subject to Trojan Horse attacks. Figure 2.1 illustrates the typical way a Trojan Horse attack is carried out.

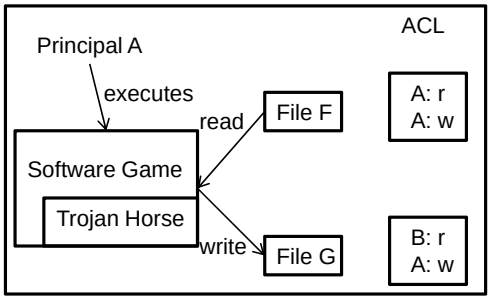

Figure 2.1: Trojan Horse Attack [3]

In the scenario depicted in Figure 2.1, user B wants to know the contents of file F for which she does not have read access. User A has read and write access to file F. In this situation, B embeds a Trojan Horse program into a program that A trusts, like a software game, and entices A into executing it. Also, B creates another file G, gives read access to herself and write access to A. When A plays the game thus executing it, the Trojan Horse program embedded within the game can access the contents of file F since it is running on behalf of A. Now, it copies the contents of file F into file G due to the permissions granted to A on file G. Hence, B can now read the contents of file F that has been copied into file G. Clearly, the DAC scheme has a weakness against Trojan Horse attacks. The MAC scheme, which is discussed subsequently, is designed to overcome this drawback.

In the MAC scheme, access control policies are formulated centrally by system administrators. The access of subjects to objects is restricted through the introduction of security labels. The data objects are assigned classification levels such as Top Secret, Secret, Unclassified etc., and each user is assigned a clearance level. The following two restrictions for access control are adopted in the MAC scheme [4].

- Simple Security: Subject S can read object O only if the clearance level of S dominates the security label of O. Intuitively, this restriction vouches for "no read up".

- Star Property: Subject S can write object O only if the security label of O dominates the clearance level of S. Intuitively, this restriction vouches for "no write down".

The Star Property stated above enables the MAC scheme to overcome the Trojan Horse attacks, as elaborated in Figure 2.2.

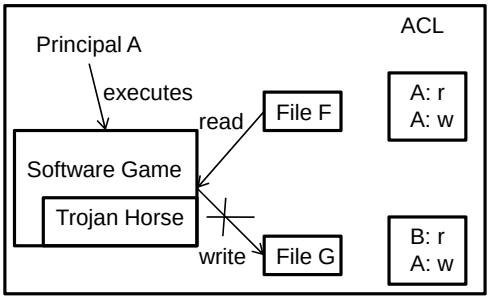

Figure 2.2: MAC preventing Trojan Horse Attack [3]

As shown in Figure 2.2, when principal A executes the program containing the Trojan Horse, the program can read the contents of file F since the program, executing on behalf of A, has the required permissions to do so. The program is prevented from writing the contents of file F into file G as this would otherwise violate the Star Property. Hence the Trojan Horse attack is thwarted. The MAC scheme is used for highly secure systems such as military applications, but turns out to be inappropriate when it comes to governmental and industrial environments where a lot if information is unclassified, but sensitive. Role-Based Access Control (RBAC), which is discussed subsequently, is used in such environments.

In the RBAC scheme [2], the system administrator assigns permissions to roles. When a user assumes a particular role, she would inherit the permissions that go with that role. Roles in an organization would fall into a neat hierarchy. Often, roles within the organizations remain fixed, but, the role that a user assumes varies from time to time depending on what job the user has to accomplish. Generally, the principle of least privilege has to be followed while assigning roles to the user. It is required that the user be given no higher privilege than what is needed to perform the job.

Now, having discussed ways to model access control policies, we briefly outline the modeling of system state pertaining to access control.

System state relevant to access control can be modeled as a matrix as shown in Figure 2.3. The rows of the matrix correspond to the various subjects. Subjects can be principals, or programs executing on behalf of the principals. The column of the matrix represents objects. Objects are anything on which subjects can perform operations, such as file, directory, memory segments etc., or they can be subjects too with operations like kill, suspend etc. The cells specify the access of the subject to the object.

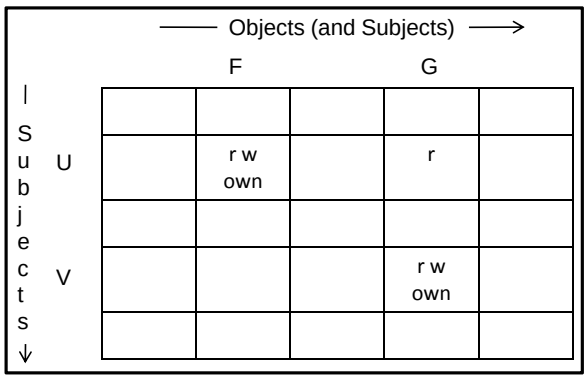

Figure 2.3: Access Control Matrix [3]

Access Control Matrices are not efficient to represent system state pertaining to access control. This is because, very often, they are sparsely populated, as a result of which too much space gets wasted. There are two alternatives to the matrix modeling, namely, Access Control List (ACL) and Capability List.

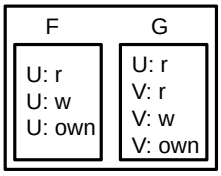

Figure 2.4: Access Control List [3]

In ACL, each column of the access matrix is stored along with the access permissions of the subjects appearing in that column. Figure 2.4 shows ACLs for two files F and G. The ACL for F indicates that the subject U has read, write, and own privileges on it. The ACL for G indicates that the subject U has only read permission, whereas the subject V has read, write and own privileges on it. In this method of modeling, it is easy to determine

all the subjects that have access rights on a given object, and is also easy to revoke all the subject's access rights on a given object, but, it is hard to revoke all the access rights of a given subject. This is because, to revoke all the access rights of a given subject, it is required to scan the ACLs of all the objects.

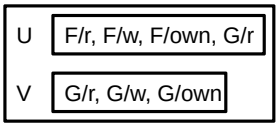

Figure 2.5: Capability List [3]

In Capability Lists, each row of the access matrix is stored along with the access permissions to objects appearing in that row. Figure 2.5 shows the Capability Lists of two subjects U and V. It indicates that the subject U has read permissions on both objects F and G. In addition, U has write and own privileges on object F. Subject V has read, write and own privileges on object G. Capabilities provide for superior access review on a per-subject basis, but, it is hard to revoke all the access permissions on a particular object [3].

2.2 Specification of Privacy Policies

The GRANT command in SQL [13] can be used to specify access restrictions at the table or relation level. When a user creates a new table, then he is entirely responsible for that table, and he can give privileges to whomever he wishes to. The privileges that the user can grant on a table include the ability to use that table in a query (READ), and ability to perform INSERTs, UPDATEs and DELETEs on the table. The syntax for the GRANT command is as follows.

GRANT {All Rights/<privileges>/All but <privileges>} ON <table> TO <user-list> [WITH GRANT-OPTION]

The table creator can specify the "All Rights" option to grant all privileges to the users, or he can specify the privileges explicitly, or he can use the "All but <privileges>" option if there arises a necessity to grant almost all privileges except a very few. The creator of the table can specify the GRANT-OPTION that would enable the grantees to further

grant their privileges on the table to other users. For example, let's suppose that user Alice creates a table named "ExamSchedule" to maintain the semester exam schedules. She, being helpful, decides to give her friend Bob the privilege to read from the table so that he too can benefit from it. In addition, she decides to give Bob the privilege to grant to other students hoping that other students too would make use of it. In order to accomplish this task, she issues the following GRANT statement.

Alice: GRANT READ ON EXAMSCHEDULE TO Bob WITH GRANT-OPTION

Griffiths et al. [14] extended this approach to handle conflicts that could possibly arise when multiple users grant conflicting privileges on a table to other users. They have done significant work in handling privilege revocations. One major observation to be made here is that the GRANT command of SQL allows a user to grant privileges at the table or relation level. However, this approach isn't sufficient enough to handle very complex privacy policies that are prevalent these days. There is a necessity for a scheme to specify access control policies at a more fine-grained level.

In their work on Hippocratic databases, Agarwal et al. [16] have specified access control in a "table-format". Their approach can be used to specify access control policies easily at the column level. For each purpose and for each attribute (column) accessed for that purpose, they identify the external recipients (whom the information can be given out to), the retention period (how long the information is stored), and the authorized users (those who can access this information). This is captured in two tables, namely, privacy-policies and privacy-authorizations. The schemas for the two tables are as follows.

Privacy-Policies (purpose, table, attribute, external recipients, retention)

Privacy Authorizations (purpose, table, attribute, authorized users)

The privacy-policies table specifies the external recipients and the retention period in addition to the purpose, and hence captures the privacy policy. The privacy-authorizations table specifies the specific users who are authorized to access the data, and hence captures the access control. Though this approach can be used to specify access control at the column level, it is quite difficult to extend it to specify access control at cell level.

Agarwal et al. [15] have provided access control constructs to specify access restrictions at the column level, row level and cell level. Figure 2.6 depicts the syntax, as defined by Agarwal et al., for specifying access restrictions. It states that those belonging to auth-name-1 list, but not belonging to auth-name-2 list, have restricted access on table-x. The restrictions can be specified at the column level by providing the list of columns to which the restriction applies, or at the row level by providing some search-condition in the WHERE clause or at the cell level by combining the restrictions at the row and column level. The syntax also allows the specification of a purpose, which determines the way the query result is going to be used, and recipient, which determines the users who would be using the query result. The restriction can be applied to any or all of select, insert, update and delete statements. Agarwal et al. have provided means to combine multiple restrictions that may be defined for a single user on a single table. In such a scenario, multiple restrictions can be combined using intersection or union operation. In the intersection approach, the user is allowed to access data that is defined by the intersection of the restrictions, thus reducing the user's access when more restrictions are added. In the union approach, the user is allowed to access data that is defined by the union of all the applicable restrictions, thus giving the user more access when restrictions are added.

```
create restriction restriction-name
on table-x
for auth-name-1 [except auth-name-2]
( (    (to columns column-name-list)
     | (to rows [where search-condition])
     | (to cells (column-name-list [where search-condition])+)
  )
  [for purpose purpose-list]
  [for recipient recipient-list]
)+
command restriction
```

Figure 2.6: Fine Grained Restriction Syntax [15]

There are certain drawbacks of this construct. In many situations, the time at which the user tries to access the database is critical. The location of access might be very critical too. For example, some organizations do not allow the user to access tables from outside the organization. This construct does not allow the specification of restrictions in terms of time

or location. In spite of these limitations, this method can be very effective in restricting access to data at column, row or cell level.

2.3 Enforcement of Privacy Policies

Traditionally privacy policies were enforced in relational databases through views. Let's suppose there exists a relation named "Student" with attributes Student Id, Name, Date of Birth, and CGPA, and the privacy policy states that every student is allowed to view his details alone and should not be able to view records corresponding to any other students'. In this scenario, the system administrator would have to create views for each of the students. For example, if we assume that the "Student" relation contains a record corresponding to Student-Id 1001, then the view would be manually created as follows.

CREATE VIEW Student1001 as SELECT Student Id, Name, Date of Birth, CGPA from Student WHERE Student Id = 1001;

The same thing would have to be repeated for each student, replacing the view name and the student id present in the WHERE clause. Once the views have been created, the system administrator would have to grant permissions for each user to access his view. One major drawback is that this approach isn't scalable with the increase in number of users. Also, minor modifications in privacy policies would lead to the manual creation of a number of additional views. These limitations make the trivial view based approach unsuitable for enforcing access control policies.

Stonebraker et al. [8] proposed an algorithm to enforce access control in INGRES that is similar to the view based approach. In their approach, they collect and store the access permissions for each user in access control interactions, which is quite similar to views. When the user issues a query, the algorithm would search for the access control interactions associated with that user. Among those, it finds the interactions that contain the attributes used in the query. It would then combine the qualifications or conditions used in the query with that present in the user interactions using AND operator. If the user interactions turn out to be complex as a result of the policies being complex, then the query modification approach might turn out to be an expensive one. As pointed out by Wang et al. [5], this

algorithm fails to specify clearly the manner in which operators such as set difference need to be handled, which apparently requires special treatment. If while handling the set difference operator, the algorithm treats the "incomplete" or "hidden" information as "NULL", then it fails to obey the soundness property. This has been explained with example in chapter 3.

Oracle's Virtual Private Database (VPD) approach [10], which is yet another query modification approach, overcomes the drawbacks of the trivial view based approach by dynamically modifying the queries issued by the users in order to enforce access control as shown in Figure 2.7. In the scenario depicted in Figure 2.7, two users, one with an id of '10' and another with an id of '20', try to access the "ORDERS" relation. The privacy policies formulated by the database administrator allows the user with id '10' to access his record alone, but does not allow the user with id '20' to access the relation. Now, when the user with id '10' issues a select query over the "ORDERS" relation, the Oracle server dynamically generates a predicate, that is a 'where' clause and appends it to the user query, thus modifying the query transparently to the user. When the modified query is issued against the relation, the result contains the record pertaining to the issuer of the query. In the second case, when the user with id '20' issues the select query, the server, as before, appends the predicate generated to the query. Since the issuer does not have access to the relation, as reflected by the predicate generated, the result returned is NULL.

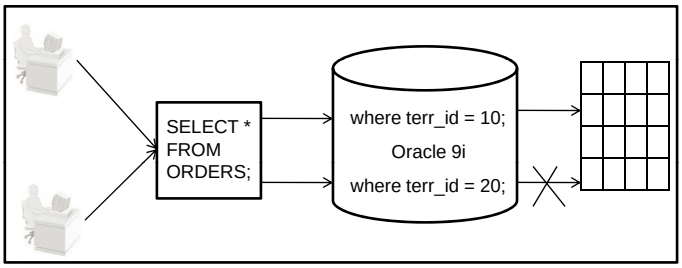

Figure 2.7: Oracle's Virtual Private Database Approach [10]

Generally, the predicates are generated by policy functions that can include callouts to other functions. It is possible to embed C or Java callouts within the policy functions, thus giving a greater flexibility. The policy functions would generate different predicates for different users based on the context of the queries. For each user, a secure application context is created in the database, which would contain user-specific information. This

information is used by the policy functions to generate predicates for different users. One major drawback with this dynamic query modification approach is that the predicates that are appended to the user issued query can be complex, for example, they can be sub-queries by themselves. In that case, the execution time of the modified query can greatly increase.

The column level VPD [7] allows one to enforce row-level security when a column that is considered sensitive is used in a query. It has two modes of operation, one is a default behavior and the other is a column masking behavior. In the default behavior, the number of rows returned as result is restricted. In the masking behavior, all rows are displayed, but the sensitive columns are replaced with NULL values. As pointed out by Wang et al. [5], the default behavior violates the "maximality" criteria, and the column masking behavior violates the "soundness" property, both of which have been briefly discussed about in chapter 1. It is explained in detail with an example in chapter 3 as to why the soundness property is violated when incomplete data is treated as "NULL" value. A more formal definition of the correctness criteria can be found in chapter 3.

In addition to the above listed drawbacks of Oracles' VPD approach, as pointed out by Rizvi et al. [11], quite often the results returned by this approach can be misleading to the user. For example, if a student issues a query to a "Student" relation to obtain the average grade for a particular course, and the privacy policy allows the student to access only his own record, then the average grade returned would be the student's grade itself. Thus, the student is misled into believing that his grade is the average grade of the class in that particular course.

Rizvi et al. [11] came up with yet another query rewriting approach in order to avoid misleading the users while enforcing access control. They have introduced the concept of parameterized authorization views, which are similar to the SQL views, but make use of parameters like user-id, time, user-location etc. Contrary to the trivial view based approach that required creation of views for each user, parameterized authorization views can be used to define a single view applicable for multiple users. For example, let's suppose we have a "Student" relation with three attributes, namely, StudentId, Name and CGPA. Let's suppose that the privacy policy states that each student can view only his record. Using parameterized authorization views, we can achieve this using the following query.

CREATE AUTHORIZATION VIEW MyRecord as SELECT StudentId, Name, CGPA from Student WHERE StudentId = $userid;

$userid appears as a parameter in this view definition. Thus, this single view definition would suffice for all the students. Based on the privacy policies established, multiple authorization views could possibly be defined for each user. Rizvi et al. try to validate the query before actually executing it. In order to do so, they determine if the issued query can be answered based on the authorization views associated with the issuer. If it's possible to do so, then they execute the query unmodified over the appropriate authorization view and return the result, else they reject the query. By rejecting the query, they violate the "maximality" criteria [5].

Bond et al. [9] used a label based approach to enforce access control in DB2. The concept of security label is central to the label based access control scheme. As defined in IBM's document [12], a security label component is defined as "a database object that represents a criterion you want to use to determine if a user should access a piece of data". For example, in order to view the grades of all students in a course, the criterion could be that the user is an instructor or a teaching assistant for that course. A security policy describes the criteria that will be used to decide who has access to what data. When the security policy is in place, security labels, which are a collection of security label components, can be associated with cells in a relation to protect it. If the user needs to access data in a particular cell, then he must obtain the appropriate security label from the database administrator, which is also called a credential. When a user tries to access protected data, that user's security label is compared to the security label protecting the data. If the user has no access to that column, then an error message is flagged. If the user has no access to the row in which the data is present, then those rows are hidden from the user, violating "soundness" and "maximality" principle.

LeFevre et al. [6] came up with an algorithm to enforce access control that is based on a cell level disclosure policy. This policy would specify the cells that could be used in order to answer a given query Q. Those cells that have been prohibited from accessing for a given query Q are then replaced with NULL values. Once this is done, the query Q is run on the new version of the relation obtained by applying the cell level disclosure policy on

the original relation. While executing the query on the new relation, it is considered that $NULL \neq NULL$ and $NULL \neq c$, for any constant c. As pointed out by Wang et al. [5] and illustrated by an example in chapter 3, replacing incomplete or sensitive information with NULL leads to the violation of the soundness property.

Chapter 3

A Sound and Secure Query Evaluation Algorithm to Enforce Fine Grained Access Control

In this chapter we discuss the correctness criteria for query evaluation algorithms enforcing fine grained access control proposed by Wang et al. [5]. We then introduce some important concepts that lead to a sound and secure query evaluation algorithm to enforce fine grained access control, also proposed by Wang et al. These play a significant role in enabling the readers understand our contribution.

3.1 Correctness Criteria

We use the same notations and terminology used by Wang et al. [5]. Let A be the query evaluation algorithm enforcing access control in database D. Let P be the access control policy that decides which portion of D could be used in evaluating a query Q over D, Q being issued by the user U. In most cases, P depends on U, the purpose of Q, the place and time Q was issued, and many other factors. For simplicity, Wang et al. assume that there is a policy P for each query Q based on the context of Q. Thus the input to the algorithm A would be the query Q, the policy P, and the database D. The output or result of the algorithm is defined as $R = A(D, P, Q)$. How do we decide if the result R that was obtained from the algorithm A is indeed correct? Wang et al. [5] provide an answer to this

question by formulating three correctness criteria, which seem to be very intuitive. Their three correctness criteria are as follows.

3.1.1 Sound

Let S be the algorithm that evaluates a query Q over a database D when there is no access control. The result returned in this scenario is $S(D, Q)$. Now, the soundness criterion requires that $A(D, P, Q) \subseteq S(D, Q)$. The intuition behind this is that, the policy P actually limits information from D that can be used to answer Q. The information that could be used to answer Q, when there is fine grained access control, would be a portion of the entire relation that would be used to answer Q, when there is no access control. Thus, the result obtained while enforcing access control is a portion of the result that is obtained without access control. Wang et al. formalize this criterion as follows.

$$\forall_P \forall_Q \forall_D A(D, P, Q) \sqsubseteq S(D, Q)$$

Now, let us look into an example that illustrates why the approach that treats hidden information as simply NULLs is unsound. The example is similar to the one that Wang et al. [5] have used to illustrate the idea.

Let us consider a relation titled "Student" with schema Student(Student ID, Name, DEPT, CGPA), which is shown in Figure 3.1(a). Cells marked as "(Y)" are allowed by the policy and cells marked as "(N)" are not allowed by the policy, when answering a query Q.

Consider the query Q = "SELECT Student ID, Name FROM Student EXCEPT SELECT Student ID, Name FROM Student WHERE CGPA $\geq$ 3.00". The query is of the form $Q = Q_1$ - Q_2, where Q_1 = "SELECT Student ID, Name FROM Student" and Q_2 = "SELECT Student ID, Name FROM Student WHERE CGPA $\geq$ 3.00". When the there is no access control, the result is as shown in Figure 3.1(e). When the algorithm proposed by LeFevre et al. [6] is used, the result is as shown in Figure 3.1(d). The result would be the same for all algorithms that treat unauthorized cells as simply NULLs. We can clearly observe that the algorithm proposed by LeFevre et al. leaks more information than what is allowed when there is no access control policy. Thus the class of query evaluation algorithms that

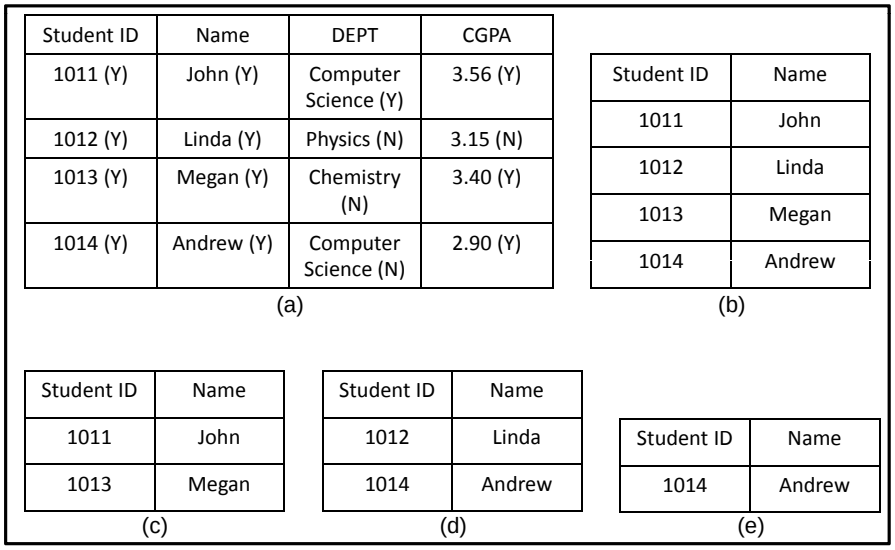

Figure 3.1: Example to illustrate why existing approaches are not sound

treats unauthorized cells as simply NULLs and use the fact that $NULL \neq NULL$, and $NULL \neq c$, where c is a constant, are unsound.

3.1.2 Secure

Wang et al. [5] argue that "a policy P defines an equivalence relation, represented as $\equiv_P$, among database states". They claim that two database states D and D' are equivalent, i.e. $D \equiv_P D'$, if on applying policy P to both D and D' results in disclosing the same information in order to answer a query Q. This ensures that the user, by issuing a query Q and analyzing the result returned by the query evaluation engine, cannot determine if the database state is D or D'. More formally, they claim that a query evaluation algorithm is secure if and only if the following condition holds.

$$\forall_P \forall_Q \forall_D \forall_{D'} \left[(D \equiv_P D') \rightarrow (A(D, P, Q) = A(D', P, Q)) \right]$$

They refer to this condition as "weak security", since this just ensures that a single user cannot gain knowledge about information not disclosed by a policy by issuing a single query. In practice, there are scenarios where a single user issues a sequence of queries or many malicious users might collude to gain information that they are not supposed to know. The query evaluation algorithm enforcing access control should provide resistance from such attacks too. This, being a tedious requirement, Wang et al. refer to as "strong security".

More formally they argue that, if the policies for the colluding users are $P_1, P_2, P_3 \cdots P_n$ and the two database states D and D' are equivalent under each of these policies, then it should not be possible for the colluding users to differentiate D from D'.

Wang et al. [5] model the notion of strong security using an adversary argument as follows. The adversary knows that the database is either in state D or D'. He tries to identify which of the two is the current state of the database by issuing queries on behalf of the colluding users. Upon issuing queries and collecting the query results, he computes a function f based on the results, and tries to use this function to identify whether the current state is either D or D'. If it is ensured that the adversary fails always regardless of what queries he issue or what function he computes, then strong security has been achieved. Wang et al. define the notion of strong security more formally as follows.

$$\begin{aligned}
&\forall_{i\in[1,n]}(D \equiv_{P_i} D') \\
&\quad \rightarrow \forall_{f\in F}(f(A(D,P_1,Q_1),\cdots,A(D,P_n,Q_n)) \\
&\qquad\quad = f(A(D',P_1,Q_1),\cdots,A(D',P_n,Q_n)))
\end{aligned}$$

where F is the set of all possible functions the adversary can compute; $Q_1, Q_2 \cdots Q_n$ are the queries that the adversary issues on behalf of the colluding users, and the policies $P_1, P_2 \cdots P_n$ are the policies, corresponding to each of the n queries, that restricts access to data.

Wang et al. have proved that weak security is equivalent to strong security and hence weak security is a sufficient condition upon satisfying which an algorithm can be considered secure. The proof is summarized below.

It is quite obvious that on substituting $n = 1$ and $f(R) = R$ in the condition for strong security, we arrive at the condition of weak security, which means an algorithm has the weak security property if it possesses the strong security property. To prove the other way round, we make use of the condition for weak security yet again. Wang et al. argue that an algorithm satisfies weak security requirement if and only if

$$\forall_P \forall_Q \forall_D \forall_{D'} \left[(D \equiv_P D') \rightarrow (A(D,P,Q) = A(D',P,Q))\right].$$

Now, since $D \equiv_{P_i} D'$, we have $A(D, P_i, Q_i) = A(D', P_i, Q_i)$.

This further means that

$$\forall_{i \in [1,n]} A(D, P_i, Q_i) = A(D', P_i, Q_i)$$

At this juncture, it is easy to observe that, in the condition for strong security, the inputs to the function f on both sides of equality are the same, which means that f gives the same output, and hence the equality holds. Thus, weak security implies strong security. It is sufficient for an algorithm to possess weak security to be considered secure.

3.1.3 Maximal

This criterion requires that the query evaluation algorithm returns as much information as possible while satisfying the sound and secure criteria. Wang et al. [5] argue that an algorithm that returns no information still preserves soundness and security, but is certainly not a useful answer. They however point out that it is difficult to achieve the maximal criterion. Let us consider the Student relation shown in Figure 3.1(a). Let the query issued be "SELECT Student ID, Name from Student where CGPA $\geq$ 3.0 AND CGPA $\leq$ 3.0". Though Linda's CGPA is not to be disclosed, it doesn't hurt to include Linda in the result since the Where clause is a tautology. As pointed out by Wang et al. there are some clauses that involve integer multiplication and equality that have been proved to be undecidable [17].

3.2 A Labeling Mechanism for Masking a Database

We have seen that by replacing unauthorized values by NULL the resulting query evaluation algorithm would become unsound, and thus produce incorrect results. In addition , as Wang et al. point out, replacing unauthorized values by NULL would lead to information loss. For example, let us suppose we have the following two relation schemas.

Employee (EmpID, Name, Age, Salary)

Department (EmpID, DeptName, Manager)

EmpID is used to link the two relations Employee and Department. Let us assume that an employee considers his EmpID to be sensitive, but does not mind letting others know his manager and the department he is working for. In this scenario, by replacing EmpID with NULL, the linking between the Employee and the Department relations would be lost, as a result of which it is not possible to find out the manager and the department name for a given employee, which is not considered to be sensitive information. Thus, we need an alternative approach to represent unauthorized information. Wang et al. have introduced two different kinds of variables, which they call "type-1" and "type-2", in order to represent unauthorized cells.

A type-1 variable is represented by an alphabet. If v_1 and v_2 are two different type-1 variables, they claim that "$v_1 = v_1$" is true, but it is not known if "$v_1 = v_2$" and "$v_1 = c$" are true, where c is any constant. A type-2 variable is represented as a pair (α, d), where α is the name of the variable and d is the domain of the variable. If α, β are two different alphabets and d_1, d_2 are two different doamins, they claim that "$(\alpha, d_1) = (\alpha, d_1)$" and "$(\alpha, d_1) \neq (\beta, d_1)$" are true, but "$(\alpha, d_1) = (\beta, d_2)$", "$(\alpha, d_1) = v_1$" and "$(\alpha, d_1) = c$" are not known, where v_1 is a type-1 variable and c is any constant. Using these two kinds of variables, Wang et al. [5] have proposed a two-pass labeling algorithm to mask the unauthorized cells in a database.

In the first pass, two different cases are considered. The first case considers all those tables T in the database that are either not linked with any other table, or the tables that have been linked with T have not yet been masked. Let attribute A be the primary key of T, and let the linking between T and other tables be through attribute A. In such a scenario, all the unauthorized cells of the attribute A are replaced by new type-2 variables of the same domain. The second case considers all those tables T in the database that are linked to any table T_1, T_1 has already been masked, which means that the attribute A over which the linking takes place has already been masked in T_1. In such a scenario, for every cell in T_1 of attribute A, replace the corresponding cell of A in T using the same type-2 variable. Upon doing this, if there still exist cells of A in T_1 that are not yet but need to be masked, use new type-2 variables of the same domain.

The second pass deals with attributes declared as foreign keys. For all tables T in the database, for each attribute B in T that has been declared as a foreign key pointing to the primary key A of T_1, Wang et al. suggest that the cells in B of T that are unauthorized be replaced by the same type-2 variable that was used in the corresponding cell of A in T_1. Also, all the unauthorized cells of the remaining attributes are replaced with new type-1 variables.

An example similar to the one provided by Wang et al. [5], based on the two pass labeling mechanism, is shown in Figure 3.2. As can be observed from Figure 3.2, the EmpID attribute that has been used to link the Employee and Department relations has been replaced by type-2 variables. As a result, the linking between the two relations is still preserved in spite of hiding the unauthorized cells. The unauthorized cells in other attributes such as Age and Salary have been replaced by new type-1 variables.

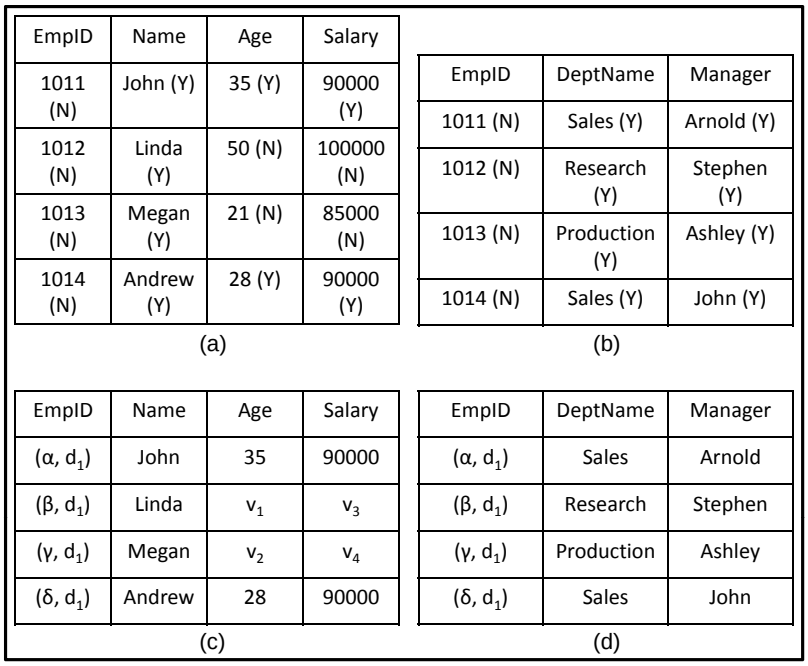

Figure 3.2: Example to illustrate labeling mechanism

We would like to point out to the readers that in the rest of this thesis we assume that such a masked version of a database is readily available and that this mechanism proposed by Wang et al. has been discussed here to show to the readers that it is indeed possible to

come up with such a masked version of a database.

3.3 A Sound and Secure Approach to Query Evaluation

Let Q be a single query, i.e. a query with no sub-queries within it. Let Q be issued over the masked version of the database D, say mask(D). Wang et al. [5] argue that for query evaluation algorithm to be sound, Q must return only those tuples that are *definitely correct*, for which the unauthorized cells are treated in a conservative fashion. They term this as low evaluation of Q, represented as Q_{-}. If on the other hand Q is of the form $Q = Q_1 - Q_2$, then for the whole query Q to be sound, it is required that Q_2 return not only those tuples that are definitely correct, but also those that are *possibly correct*. Wang et al. term this as high evaluation of Q_2, represented as Q_2^{-}. Before analyzing how *possibly correct* tuples arise, let us briefly discuss about tuple compatibility.

Let $t_1 = (x_1, x_2 \cdots x_n)$ and $t_2 = (y_1, y_2 \cdots y_n)$ be two tuples of the same schema. Wang et al. claim that the tuples t_1 and t_2 are not compatible if either, x_i and y_i are two different constants, or x_i and y_i are both different type-2 varibales of the same domain. The other possibilities are listed below.

- x_i is a type-1 variable and y_i is a constant or vice versa
- x_i and y_i are different type-1 variables
- x_i is a type-2 variable and y_i is a type-1 variable or vice versa
- x_i is a type-2 variable and y_i is a constant or vice versa
- x_i and y_i are both type-2 variables of different domains

In all the above cases, the comparison between x_i and y_i would lead to an unknown result. If $Q = Q_1 - Q_2$ is to be evaluated in low-mode, then we do not include those tuples in the result that would involve the above listed comparisons, which lead to an unknown value, thus returning only *definitely correct* tuples. If on the other hand $Q = Q_1 - Q_2$ is to be evaluated in high mode, then we include those tuples in the result that would involve comparisons, which lead to an unknown value, thus returning *possibly correct* tuples.

The sound and secure query evaluation algorithm proposed by Wang et al. is shown in Figure 3.3. Wang et al. define a function $IsUn(x)$, which return $TRUE$ if x evaluates to an unkown value. We would like to refer the readers to [20] for the proof of soundness and security of the algorithm in Figure 3.3.

Given a query Q:

- if $Q = R$, where R is a relation in a database, then $Q_- = Q^- = R'$, where R' is the masked version of R.
- if $Q = \sigma_c Q'$, then $Q_- = \sigma_c Q'_-$ and $Q^- = \sigma_{c \vee IsUn(c)} Q'^-$
- if $Q = \pi_{a_1,\ldots,a_n} Q'$, then $Q_- = \pi_{a_1,\ldots,a_n} Q'_-$ and $Q^- = \pi_{a_1,\ldots,a_n} Q'^-$.
- if $Q = Q_1 \times Q_2$, then $Q_- = Q_{1-} \times Q_{2-}$ and $Q^- = {Q_1}^- \times {Q_2}^-$.
- if $Q = Q_1 \cup Q_2$, then $Q_- = Q_{1-} \cup Q_{2-}$ and $Q^- = {Q_1}^- \cup {Q_2}^-$.
- if $Q = Q_1 - Q_2$, then Q_- contains all tuples $t \in Q_{1-}$ such that there exists no tuple in ${Q_2}^-$ that is compatible with t (we call the operation to compute Q_- *aggressive minus*, denoted as $-_a$), and Q^- contains all tuples $t \in {Q_1}^-$ such that $t \notin Q_{2-}$ (we call the operation to compute Q^- *conservative minus*, denoted as $-_c$).

Figure 3.3: Sound and Secure Query Evaluation Algorithm [5]

As pointed out in chapter 1, there are two approaches to implement access control in relational databases, namely query modification approach and query evaluation engine modification approach. Wang et al. have opted for a query modification approach to implement the algorithm outlined in Figure 3.3. Their approach is outlined in Figure 3.4. Since existing DBMSs do not have provisions for type-1 and type-2 variables, the unauthorized cells are masked using NULLs. As a result, it might not be possible to return the maximal result, but it is possible to guarantee soundness and security of the algorithm.

The significant difference between this approach proposed by Wang et al. and the one proposed by LeFevre et al. [6] lies in the way the minus operator has been handled, which arises as a result of having two modes of evaluation, namely, low and high. As a result, the performance of the query modification approach of Figure 3.4 was analyzed using queries

of the form $Q = Q_1 - Q_2$ on tables of varying sizes. Experimental results, which have been reported in chapter 5, clearly show that even for nominal table sizes of 50000 and 100000 tuples, this approach takes 8 to 32 seconds to complete evaluation of query. Also, this approach does not scale well with larger table sizes. One of the main reasons for this rather bad performance lies in the complexity of the join conditions, shown in Figure 3.4, when evaluating a query involving minus operator. One way to improve the performance is to modify the query evaluation engine to identify the modes of evaluation of sub-queries within a query, differentiate aggressive minus from conservative minus, and modify the semantics of the minus operator based on the notion of tuple compatibility discussed above. This approach, which is our contribution, is discussed in detail in chapter 4.

Input: a query Q

Output: Q_-, i.e. the low evaluation of Q

- Case $Q = T$, where T is a table:

  $$Q_- = Q^- = mask(T)$$

  where $mask(T)$ replaces unauthorized cells in T with NULL.

- Case $Q =$ "SELECT $a_1, \cdots, a_n$ FROM $Q_1, \cdots, Q_m$ WHERE *conds*":

  $$Q_- = \text{SELECT } a_1, \cdots, a_n \text{ FROM } Q_{1-}, \cdots, Q_{m-} \text{ WHERE } conds$$

  $$Q^- = \text{SELECT } a_1, \cdots, a_n \text{ FROM } Q_1^-, \cdots, Q_m^- \text{ WHERE } addNull(conds)$$

  where $addNull(conds)$ adds "IS NULL" predicates to selection condition *conds*. For instance, let *conds* = "Age $\geq$ 25 AND Salary $<$ 80000". We have, $addNull(conds)$ = "(Age $\geq$ 25 OR Age IS NULL) AND (Salary $<$ 80000 OR Salary IS NULL)".

- Case $Q =$ "Q_1 MINUS Q_2", where Q_1 and Q_2 select attributes $a_1, \cdots, a_n$:

  $$Q^- = Q_1^- \text{ MINUS } Q_c$$

  where Q_c = SELECT $T_1.a_1, \cdots, T_1.a_n$ FROM Q_{1-} $T_1, Q_2{}^-$ T_2 WHERE $(T_1.a_1 = T_2.a_1)$ AND $\cdots$ AND $(T_1.a_n = T_2.a_n)$

  $$Q_- = Q_{1-} \text{ MINUS } Q_a$$

  where Q_a is as follow
  SELECT $T_1.a_1, \cdots, T_1.a_n$ FROM $Q_{1-}T_1$,$Q_2{}^-T_2$WHERE ((T_1.a_1=T_2.a_1) OR (T_1.a_1IS NULL) OR (T_2.a_1IS NULL)) AND $\cdots$ AND ((T_1.a_n=T_2.a_n) OR (T_1.a_nIS NULL) OR (T_2.a_nIS NULL))

Figure 3.4: Query Modification Approach [5]

Chapter 4

A Database Level Implementation to Enforce Fine Grained Access Control

In this chapter we discuss the various algorithms, data structures, and optimizations that we have introduced to implement the sound and secure query evaluation algorithm outlined in Figure 3.3 to efficiently enforce fine grained access control at the database level. We would like to reiterate the fact that we have replaced unauthorized cells using the NULL value since existing DBMSs do not have the provisions to use type-1 and type-2 variables. In order to overcome this limitation of the DBMSs we use the following two rules.

- To compute *definitely correct* tuples, we use the rule that NULL matches any value, i.e. $NULL = NULL$ and $NULL = c$ for any constant c
- To compute *possibly correct* tuples, we use the rule that NULL does not macth any value, i.e. $NULL \neq NULL$ and $NULL \neq c$ for any constant c

This is pretty obvious since the NULL value is now equivalent to the type-1 and type-2 variables used in the labeling mechanism discussed in section 3.2. In order to implement the algorithm shown in Figure 3.3 at the database level, firstly we modify the query evaluation engine of the DBMS to assign low and high modes of operation to sub-queries. Secondly we modify the manner in which the minus operator is implemented in existing

DBMSs to incorporate the semantics of aggressive and conservative minus. Since the modifications have to be made at the database level, we opt for POSTGRESQL, which is an open source DBMS. However, the algorithms that we present in this chapter are generic to all DBMSs. POSTGRESQL does not provide a *minus* operator as such, but the *except* operator that it provides is similar to what the *minus* operator does in Oracle. The terms *except* and *minus* are used interchangeably and mean *set difference*.

4.1 Assigning Modes of Evaluation to Sub-Queries

As discussed earlier, when the query Q is of the form $Q_1 - Q_2$, then we would need to evaluate Q_1 in low mode (Q_{1-}) and Q_2 in high mode (Q_2^-) in order to ensure that we evaluate Q in a sound fashion. Now, the query Q can be as complex as $(Q_1 - (Q_2 - (Q_3 \bigcup Q_4)))$. To evaluate Q in a sound fashion, we would have to evaluate the sub-query Q_1 in low mode (Q_{1-}) and the sub-query $Q_2 - (Q_3 \bigcup Q_4)$ in high mode, which further requires that we evaluate Q_2 in high mode (Q_2^-) and the sub-query $Q_3 \bigcup Q_4$ in low mode ($Q_{3-} \bigcup Q_{4-}$). Thus, it is required to parse the input query Q to differentiate sub-queries that need to be evaluated in low mode from those that need to be evaluated in high mode. For those sub-queries that need to be evaluated in high mode, we add predicate(s) to its WHERE clause in order to include those tuples in the result that are *possibly correct*. We accomplish this task by modifying the query parser of the DBMS, as shown in Algorithm 1.

Suppose Q is the query issued by the user, the initial call to the rewriteSelectStmt procedure would pass the parse tree of Q and 0 (or low mode) as the parameters. This ensures that the query Q is evaluated in low mode and thus the result returned is sound. When the operator is a minus, the left sub-query is evaluated in the same mode as its parent query, and the right sub-query is evaluated in the alternative mode. For other operators, both left and right sub-queries are evaluated in the same mode as their parent query. It can be observed that if the mode of a sub-query is found to be 1 (or high mode), predicate(s) is added to include those tuples in the result that are *possibly correct*. We would also like to point out to the readers that while we assign modes of evaluation to sub-queries, we also determine whether the *except* present in the query needs to be treated *aggressively* or *conservatively*.

Algorithm 1 rewriteSelectStmt(*stmt*, *mode*): Takes as input the parse tree of SELECT query ($SelectStmt * stmt$)and mode (integer) of evaluation and outputs modified parse tree

if $stmt \rightarrow op = MINUS$ **then**
  rewriteSelectStmt($stmt \rightarrow larg$, *mode*)
  $stmt \rightarrow mode = mode$
  rewriteSelectStmt($stmt \rightarrow rarg$, $1 - mode$)
else if ($stmt \rightarrow op = UNION$) OR ($stmt \rightarrow op = INTERSECT$) OR ... **then**
  rewriteSelectStmt($stmt \rightarrow larg$, *mode*)
  rewriteSelectStmt($stmt \rightarrow rarg$, *mode*)
else if $mode = 0$ **then**
  return
else
  rewrite *stmt* to include tuples that are *possibly* correct
  return
end if

Having assigned modes of evaluation to queries while parsing them, the next step is to redefine the semantics of the minus operator to implement aggressive and conservative minus.

4.2 Implementing Aggressive and Conservative Minus

For the convenience of the readers, we restate the definition of aggressive and conservative minus, which has already been provided in Figure 3.3. Let the query Q be of the form $Q = Q_1 - Q_2$. Aggressive minus, which is the operation to compute Q_- and represented as $Q_1 -_a Q_2$, contains all tuples $t \in Q_{1-}$ such that there exists no tuple in Q_2^- that is compatible with t. Conservative minus, which is the operation to compute Q^- and represented as $Q_1 -_c Q_2$, contains all tuples $t \in Q_1^-$ such that $t \notin Q_{2-}$ [5].

In today's DBMS, a query $Q_1 - Q_2$ is typically handled through a process similar to merge join, in which tuples are sorted by treating NULL as the smallest value. However, while computing aggressive and conservative minus, we regard NULL as a special value that matches every other value. Thus, existing implementation of MINUS needs to be revised to implement aggressive and conservative minus. A straightforward approach to implement aggressive and conservative minus has been shown in Algorithm 2 and Algorithm 3 respec-

tively, but has been found experimentally to be inefficient. Hence we have implemented the aggressive and conservative minus in an efficient way by using an index structure, which has been discussed in detail in section 4.2.2.

4.2.1 Straightforward Approach to Implement Aggressive and Conservative Minus

From Algorithm 2 and Algorithm 3 it can be observed that the straightforward approach involves scanning all tuples in the result of Q_2 for every tuple in the result of of Q_1. While computing aggressive minus, if we encounter tuples T_1 in the result of $Q_1$and T_2 in the result of Q_2, with one having a NULL value in attribute i and another with a NULL or constant value in the same attribute, then we do not include T_1 in the result, provided the same situation occurs corresponding to all other attributes or values in the corresponding attributes are the same constants. On the other hand, while computing conservative minus, we include T_1 in the result if the value in any attribute of T_1 is NULL or if we cannot find any compatible tuple T_2 in the result of Q_2 using the same procedure that we used to check for compatibility while computing aggressive minus.

Experimental results suggest that this straightforward approach is not efficient. Hence we need a better approach to compute aggressive and conservative minus.

We can clearly observe that checking for compatibility to compute aggressive and conservative minus is almost the same, except for the difference in the way we treat NULL value in each case. Thus, it is easy to comprehend that performing the tuple compatibility check is the major operation while computing aggressive and conservative minus. In the next section we have introduced a data structure, which we call the "Bucket" structure to perform this tuple compatibility check in an efficient manner.

4.2.2 An Indexed Approach to Implement Aggressive and Conservative Minus

In this section we discuss an index structure, which we call the "Bucket" structure, that we use to efficiently compute aggressive and conservative minus. It is similar to the trie structure used in prefix matching [21]. Its generic structure is shown in Figure 4.1. From

Algorithm 2 aggressiveExcept(Q_1, Q_2): Input is a query of the form $Q = Q_1 - Q_2$, and output is Q_-, which is the low evaluation of Q

```
RESULT = NULL
for each tuple T1 in the result of Q1 do
  for each tuple T2 in the result of Q2 do
    compatible = FALSE
    for i = 1 to number of columns in the schema do
      x_i = value at column i in T1
      y_i = value at column i in T2
      if (x_i = NULL) OR (y_i = NULL) OR (x_i = y_i) then
        if i is the last column in the schema then
          compatible = TRUE
          break
        end if
      end if
    end for
    if compatible = TRUE then
      break
    end if
  end for
  if compatible = FALSE then
    Add T1 to RESULT
  end if
end for
```

Algorithm 3 conservativeExcept(Q_1, Q_2): Input is a query of the form $Q = Q_1 - Q_2$, and output is Q^-, which is the high evaluation of Q

```
RESULT = NULL
for each tuple T1 in the result of Q1 do
  for each tuple T2 in the result of Q2 do
    compatible = FALSE
    for i = 1 to number of columns in the schema do
      x_i = value at column i in T1
      y_i = value at column i in T2
      if (x_i = NULL) then
        break
      end if
      if x_i = y_i then
        if i is the last column in the schema then
          compatible = TRUE
          break
        end if
      end if
    end for
    if compatible = TRUE then
      break
    end if
  end for
  if compatible = FALSE then
    Add T1 to RESULT
  end if
end for
```

Figure 4.1 we can observe that each level of the structure corresponds to an attribute of the relation for which the "Bucket" structure is built. We discuss subsequently, in detail, how we build this index structure for a given relation and how we make use of this to efficiently compute aggressive and conservative minus.

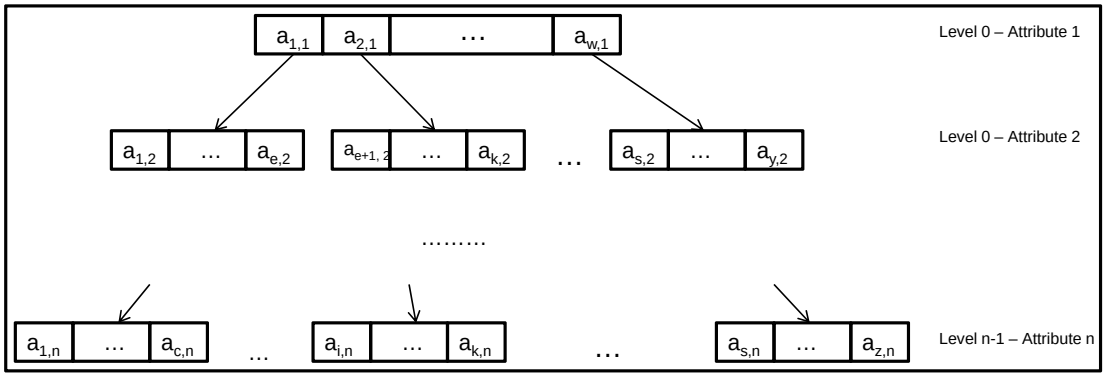

Figure 4.1: Generic Form of the "Bucket" structure

Let the query under consideration be of the form $Q = Q_1 - Q_2$. Let T_1 correspond to the result of Q_1 and T_2 correspond to the result of Q_2. To implement the tuple compatibility check more efficiently we build the "Bucket" structure corresponding to T_2. The algorithm for building the "Bucket" structure is outlined in Algorithm 4.

The "struct Bucket", which we refer to as simply bucket, holds the value of a cell in a given column and points to an array of buckets associated with this bucket. This bucket forms the building block for the "Bucket" structure that we intend building. The way in which we build the "Bucket" structure is as follows. The initial call to the buildIndex procedure would pass T_2 (table for which the "Bucket" structure is to be built) and 1 (indicating that we start with first column). We first group the tuples of the table based on the values in attribute 1. Let $R_1, R_2 \cdots R_n$ be the resulting groups. All tuples within a given group, say R_i ($\forall i \in [1..n]$), have the same value corresponding to attribute 1. We now build a bucket for each group of tuples R_i corresponding to attribute 1, and recurse on the second attribute to build the list of sub-buckets. We do this for each attribute within a given group and for all groups. The resulting structure would be as shown in Figure 4.1.

Algorithm 4 buildIndex(R, colNumber): Takes as input sorted Relation R for which the "Bucket" structure is to be built and a $columnNumber$ that is initially 1, and the output is the "Bucket" structure for relation R

struct Bucket
{
  int $value$;
  struct Bucket $*subBuckets$;
};
Group tuples of R into $R_1, R_2 \cdots R_n$ such that all tuples within each R_i ($\forall i \in [1..n]$) have the same value at $columnNumber$
Initialize Bucket array, $Buckets[n]buckets$
Initialize $index = 0$
for each group of tuples S in $R_1, R_2 \cdots R_n$ **do**
  $buckets[index].value$ = value at $columnNumber$ in tuples of S {All tuples in S have the same value in column $columnNumber$}
  if $columnNumber$ is the last column in the schema **then**
    $buckets[index].subBuckets = NULL$
  else
    $buckets[index].subBuckets = buildIndex(S, columnNumber + 1)$
    $index = index + 1$
  end if
end for
return buckets

The example shown in Figure 4.2 illustrates Algorithm 4 clearly. Figure 4.2(a) shows the unsorted table for which we intend to build the "Bucket" structure. Figure 4.2(b) shows the "Bucket" structure corresponding to the sorted version of the table shown in Figure 4.2(a) that was built using the algorithm outlined in Algorithm 4. From Figure 4.2(b) it is clear that there are 4 buckets corresponding to the column VA with values NULL, 1, 2 and 3. By bucket, we mean the $(value, sub-buckets)$ pair. There is only 1 sub-bucket corresponding to $VA = NULL$ that has a value $VB = 2$, which in turn has only 1 sub-bucket with value $VC = NULL$. Now, there are 2 sub-buckets corresponding to $VA = 1$. The values in both these sub-buckets are $NULL$ (which in turn has 2 sub-buckets with values $VC = NULL$ and $VC = 1$) and 3 (which has only 1 sub-bucket with value $VC = 2$). The remainder of the "Bucket" structure is to be interpretted in a similar fashion.

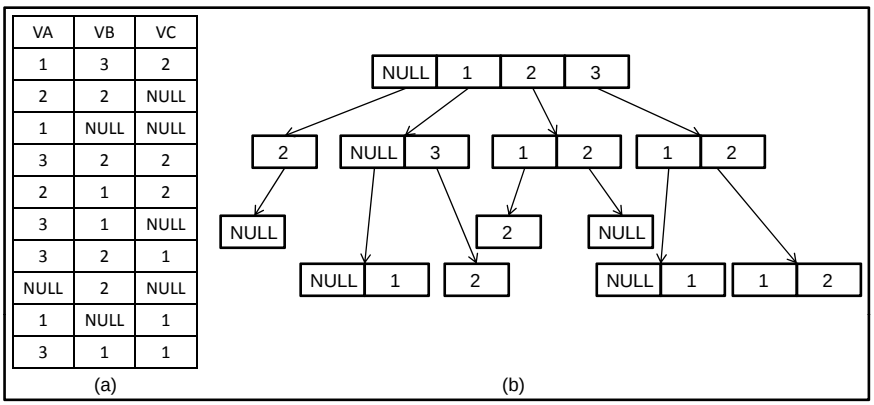

Figure 4.2: Example to demonstrate building of "Bucket" structure

Having discussed the algorithm to build the "Bucket" structure, we now present and discuss algorithms to compute aggressive and conservative minus using the "Bucket" structure.

Upon encountering a query with a SQL set difference operation we invoke Algorithm 5. The inputs to Algorithm 5 include the query Q of the form $Q_1 -_a Q_2$ or $Q_1 -_c Q_2$, and the mode in which query Q has to be evaluated. The mode of evaluation of query Q is decided by the query parser, as shown in Algorithm 1. Algorithm 5 invokes Algorithm 4 to build the "Bucket" structure for the result of Q_2, after sorting it. Then, depending on whether the mode is aggressive or conservative, Algorithm 5 invokes Algorithm 6 or Algorithm 7 for each tuple T in the result of Q_1. If the invoked algorithm returns $FALSE$, then Algorithm 5

adds the tuple T to the result of query Q.

Algorithm 5 except(Q_1, Q_2, *mode*): Takes as input query $Q = Q_1 - Q2$ and *mode* of except, whether aggressive or conservative, and then invokes Algorithm 6 or Algorithm 7 accordingly

```
Let relation R be the result of executing Q_2
Sort the tuples of R by each column
Bucket[] buckets = buildIndex(R, 0)
RESULT = NULL
if mode is aggressive then
   for each tuple T in the result of Q_1 do
      if aggressiveExcept(T, 0, buckets) returns FALSE then
         add T to RESULT
      end if
   end for
else if mode is conservative then
   for each tuple T in the result of Q_1 do
      if conservativeExcept(T, 0, buckets) returns FALSE then
         add T to RESULT
      end if
   end for
end if
```

We now discuss the indexed approach to compute aggressive minus . We would like to remind the users that we treat $NULL = NULL$ and $NULL = c$ for any constant c, while computing aggressive minus. Algorithm 6 shows the way we compute aggressive minus using the "Bucket" structure. The inputs to Algorithm 6 include the tuple T (for which a decision has to be made whether or not include it in the result of the evaluation of aggressive minus), the column number (which is initially 1) and the "Bucket" structure. For a given column number (or attribute) i we look at the buckets at level $i-1$. If the value of tuple T in the i^{th} attribute (column) is $NULL$, then we search in the sub-buckets of every bucket in the level $i-1$. If the value of tuple T in the i^{th} attribute is not $NULL$, but there exists a bucket in level $i-1$ with value $NULL$, then we first search in the sub-buckets of that bucket. If we still do not succeed in finding a tuple compatible to T or if there is no bucket in level $i-1$ with value $NULL$, then we search in the buckets in level $i-1$ to find a bucket

having value same as the value in tuple T in the i^{th} attribute. If the search is successful, we proceed to the next level (attribute). If the search is unsuccessful we declare that there is no tuple in the "Bucket" structure compatible to tuple T. We begin this entire process with attribute 1 of tuple T.

We now elaborate, through the use of an example, the way in which aggressive minus is computed using Algorithm 6. Let T_1 be the table shown in Figure 4.3(a). Let T_2 be the table shown in Figure 4.3(b). Let Q_1 be "Select * from T_1;" and Q_2 be "Select * from T_2;". We are interested in computing $Q_1 -_a Q2$.

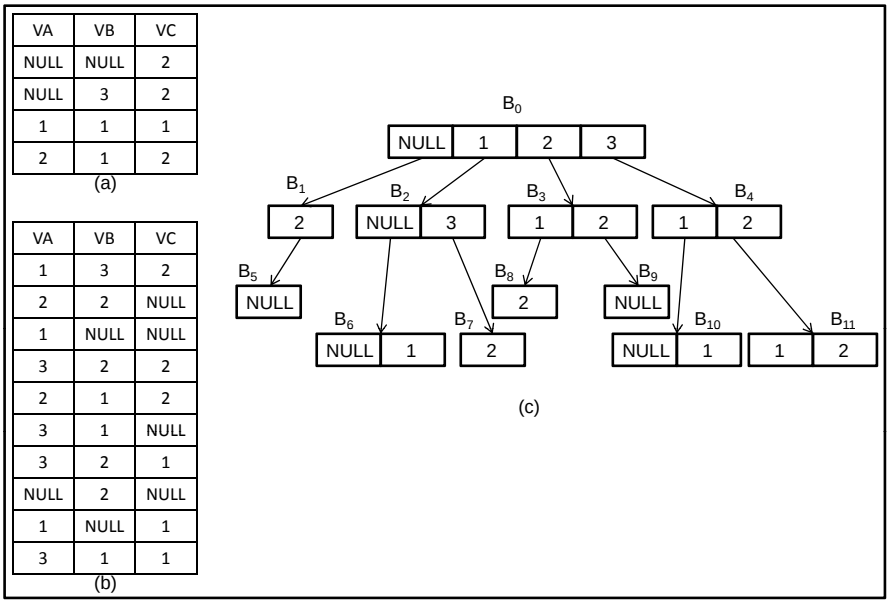

Figure 4.3: Example to illustrate computation of aggressive minus

We first sort the tuples of T_2 and build the "Bucket" structure, which is shown in Figure 4.3(c). For convenience, we have labeled each node of the "Bucket" structure. We now consider the first tuple of T_1, call it t_1. We can see that the value in attribute VA in t_1 is NULL. Since $NULL$ matches any value, we should search in the sub-buckets corresponding to every bucket in B_0, which includes B_1 through B_4, before we can declare that t_1 has to be included in the result. We begin with B_1. The value in t_1 in attribute VB is also $NULL$. So we search the sub-bucket B_5. We can see that there is only one bucket in B_5 with value $NULL$. Since $NULL$ matches anything, we have found that there is a match in

Algorithm 6 aggressiveExcept(T, $columnNumber$, $buckets$): Takes as input the tuple T from the result of Q_1, the column number, and the "Bucket" structure for the result of Q_2, and returns $TRUE$ if T is in the result of $Q_1 -_a Q_2$

x = value at $columnNumber$ in tuple T
if $x = NULL$ **then**
  if $columnNumber$ is the last column in the schema **then**
    return $TRUE$
  else
    for each $bucket$ in $buckets$ **do**
      $aggressiveExcept(T, columnNumber + 1, bucket.subBuckets)$
    end for
  end if
end if
if $buckets[0].value = NULL$ **then**
  if $columnNumber$ is the last column in the schema **then**
    return $TRUE$
  else
    $aggressiveExcept(T, columnNumber + 1, buckets[0].subBuckets)$
  end if
end if
perform binary search to search for x in $buckets[0].value, buckets[1].value \cdots buckets[n].value$ and store the return value in $index$
if $index = -1$ **then**
  return $FALSE$
else
  if $columnNumber$ is the last column in the schema **then**
    return TRUE
  else
    $aggressiveExcept(T, columnNumber + 1, buckets[index].subBuckets)$
  end if
end if

T_2 for t_1. Thus, we return $TRUE$ for t_1.

We now consider the second tuple in T_1, call it t_2. Since value in t_2 in attribute VA is $NULL$, we would have to search the sub-buckets of every bucket in B_0. This includes B_1 through B_4. The value in t_2 in attribute VB is 3. We can observe that the search in B_1 is not fruitful. Now we retract to B_0 and search in the sub-buckets B_2 of the bucket in B_0 with value 1. In B_2 there is a bucket with value $NULL$. So we search in its sub-bucket B_6. B_6 has a bucket with value $NULL$, which allows us to stop the search and declare that we have found a tuple in T_2 compatible to t_2.

We now proceed to the third tuple of T_1, let's call it t_3. Since there is a bucket in B_0 with value $NULL$, we search in its sub-buckets, i.e. B_1, though value in t_3 in attribute VA is 3. This is because, $NULL$ macthes any value. The value in t_3 in attribute VB is 1. We can observe that the search in B_1 is not fruitful. Thus, we retract to B_0. Now, we perform a binary search in B_0 and arrive at the bucket with value 1. Now, we search in its sub-buckets B_2. In B_2 there is a bucket with value $NULL$. So we search in its sub-bucket B_6. B_6 has a bucket with value $NULL$, which allows us to stop the search and declare that we have found a tuple in T_2 compatible to t_3.

We now proceed to the fourth tuple of T_1, let's call it t_4. Value in t_4 in attribute VA is 2. As in t_3, search in the sub-buckets B_1 of the bucket with value $NULL$ in B_0 is not fruitful. So we perform a binary search in B_0 and arrive at the bucket with value 2. So we proceed with the search in its sub-buckets, i.e. B_3. The value in t_4 in attribute VB is 1. In B_3 there is no bucket with $NULL$ value. So performing a binary search leads us to the bucket with value 1. So we continue the search in its sub-buckets, i.e. B_8. The value in t_4 in attribute VC is 2. In B_8 there is no bucket with $NULL$ value. So performing a binary search leads us to the bucket with value 2, which apparently is the only bucket in B_8. Since VC is the last attribute and we have found a match in it, we declare that t_4 has a compatible tuple in T_2. Thus, we find that there is some tuple in T_2 that is compatible with every tuple in $T1$. Hence, the result contains 0 rows.

Having discussed in detail the computation of aggressive minus we now focus on the computation of conservative minus using our "Bucket" structure. We would like to remind

the readers that, while computing conservative minus, $NULL$ does not match any value, i.e. $NULL \neq NULL$ and $NULL \neq c$ for any constant c. Algorithm 7 shows the way we compute conservative minus using the "Bucket" structure. The inputs to Algorithm 7 include the tuple T, for which a decision has to be made whether or not include it in the result of the evaluation of conservative minus, the column number that is initially 0 and the "Bucket" structure. For a given column number (or attribute) i we look at the buckets at level $i-1$. If the value of tuple T in the i^{th} attribute (column) is $NULL$, then we stop our search and declare that T has no compatible tuple in the "Bucket" structure. If the value of tuple T in the i^{th} attribute is not $NULL$, then we search in the buckets in level $i-1$ to find a bucket having value same as the value in tuple T in the i^{th} attribute. If the search is successful, we proceed to the next level (attribute). If the search is unsuccessful we declare that there is no tuple in the "Bucket" structure compatible to tuple T. We begin this entire process with attribute 1 of tuple T. Since Algorithm 7 to compute conservative minus is quite similar to the Algorithm 6 to compute aggressive minus, we believe that the readers would be able to comprehend it without an example.

Algorithm 7 conservativeExcept(T, *columnNumber*, *buckets*): Takes as input the tuple T from the result of Q_1, the column number, and the "Bucket" structure for the result of Q_2, and returns $TRUE$ if T is in the result of $Q_1 -_c Q_2$

x = value at *columnNumber* in tuple T
if $x = NULL$ **then**
  return $FALSE$
end if
perform binary search to search for x in $buckets[0].value, buckets[1].value \cdots buckets[n].value$ and store the return value in *index*
if $index = -1$ **then**
  return $FALSE$
else
  if *columnNumber* is the last column in the schema **then**
    return TRUE
  else
    $conservativeExcept(T, columnNumber + 1, buckets[index].subBuckets)$
  end if
end if

So far we have assumed that the entire "Bucket" structure that we build fits into the main memory. This may well not be the case for large tables. In this scenario we need to find a way to build the "Bucket" structure in disk and find efficient ways to retrieve it while computing aggressive and conservative minus. The way we build the "Bucket" structure in the disk is pretty much similar to what we do in main memory, except that we flush the buckets into the disk when there is no more space in the main memory. Also, we ensure that each disk block contains buckets of a single attribute (column). In addition, buckets corresponding to a given column may span multiple blocks. The logical view of the "Bucket" structure in disk is shown in Figure 4.4. As in the "Bucket" structure in memory, the "Bucket" structure in disk has the buckets corresponding to each attribute in a separate level, with the buckets corresponding to the first attribute at the root. If the buckets of an attribute cannot be fit into a single disk block, we pack the extra buckets into an overflow block and make it accessible from the original disk block. This is the reason for the "links" among blocks at the same level in Figure 4.4.

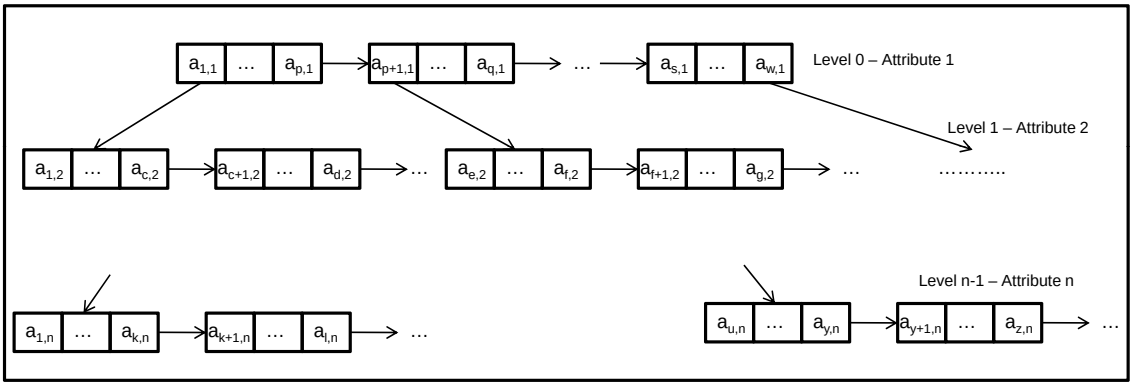

Figure 4.4: Logical View of "Bucket" Structure in disk

We now illustrate, with an example, exactly how we store the "Bucket" structure in the disk. Let's revisit the table shown in Figure 4.2(a). Figure 4.5 shows the "Bucket" structure built in the disk for that table. Each bucket, which originally held a value and main memory location of its sub-buckets, now includes a block number in which its sub-buckets are located in the disk instead of the actual main memory location. In the context of the "Bucket" structure stored in disk, bucket means the (value, block number) pair.

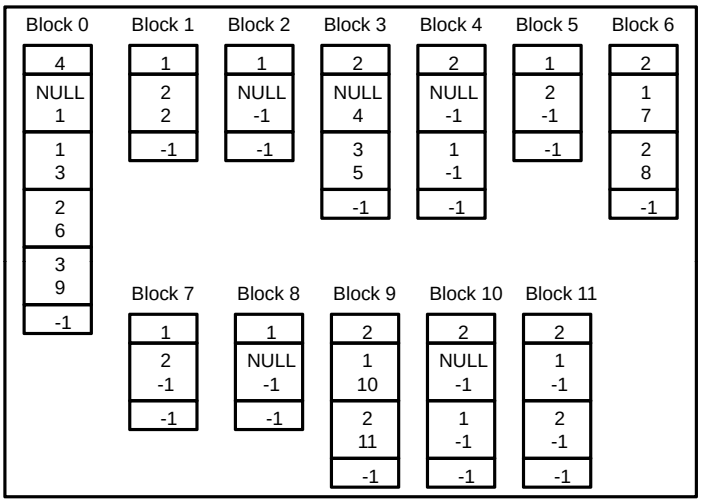

Figure 4.5: Example illustrating storage of "Bucket" structure in disk

In Figure 4.5 the block numbers are used to identify each block of the "Bucket" structure. This is a number that we assign for our reference. All the buckets corresponding to the attribute VA are written in Block 0. The first integer in each block represents the number of buckets stored in that block. The number of buckets corresponding to the attribute VA is 4, and hence the first integer in Block 0 is 4. The same holds true for other blocks too. Following the number of buckets are (value, block-number) pairs. The value represents the value in the bucket and the block-number represents the block number in which its sub-buckets can be found. For example, in Block 0, $(NULL, 1)$ means that the value in that bucket is NULL and its sub-buckets can be found in Block 1. Suppose in a bucket, the block-number in the (value, block-number) pair is -1, it means that there are no sub-buckets associated with this bucket. This is usually true for buckets corresponding to the last column. The last integer in any block, say block i, is used to represent the block number in which the buckets that overflow from block i are stored. The overflow occurs due to the limitation of the block size in disk, which is usually 1 KB or 2 KB or 4 KB. If the last integer in the block is -1, then it means that there are no overflow buckets, as in this example.

These blocks can be written to a file so that they can be loaded into main memory, in parts, at a later point in time to evaluate queries. The file, which we call as index file, can be named after the table for which it has been created to avoid name conflicts. The same file can be used whenever the underlying table is involved in the computation of aggressive

or conservative minus. However, we need to build the index file again if the underlying table is modified.

Now that we have built the "Bucket" structure in disk, we need to load it into main memory to compute aggressive or conservative minus. We have come up with two loading strategies, namely, `pre-fetching` and `on-demand`. In the `pre-fetching` scheme we try to take advantage of as much main memory as available and try to load as many of the blocks from the disk into main memory as possible. In the `on-demand` scheme we load blocks from the disk into main memory as and when required.

The `on-demand` scheme might seem pretty obvious, but to help the readers visualize the `pre-fetching` scheme we use the "Bucket" structure shown in Figure 4.5. Let us assume that the main memory can accommodate 13 buckets. In the `pre-fetching` scheme, we would load blocks 0 through 6 into main memory initially. More generally, we use a depth first loading strategy with respect to the buckets in each block. The reason for this is an optimization we have come up with, which we discuss subsequently using an example.

Let the query that we are trying to evaluate be $Q = Q_1 -_a Q_2$, where Q_1 is "Select * from T_1;", and Q_2 is "Select * from T_2;". Let the schemas of T_1 and T_2 consist of three attributes VA, VB, and VC. Let us assume that we have already built the index file ("Bucket" structure) for T_2. To evaluate the query Q, we need to scan the index file of T_2, for each tuple of T_1. Let us further assume that, in the index file of T_2, there are two blocks of buckets corresponding to the attribute VA, each block holding 100 buckets. If the tuples of T_1 are such that the first tuple has a VA value that is present in the first block corresponding to attribute VA of the "Bucket" structure, second tuple has a VA value that is present in the second block corresponding to attribute VA of the "Bucket" structure, third tuple has a VA value that is present in the first block corresponding to attribute VA of the "Bucket" structure and so on, then the blocks are scanned back and forth multiple times. Also, in this scenario we can never claim that a block of the "Bucket" structure is never going to be used in evaluating the aggressive minus because we might always encounter a tuple of T_1 with $VA = NULL$, which requires us to scan in the index file all the blocks corresponding to VA. In other words, if the tuples in T_1 are in some random order, then the disk accesses turn out to be very expensive.

Obviously we cannot guarantee a perfect sequential access of the index file, but we can definitely reduce the back and forth scans of blocks by sorting the tuples of T_1. It should now seem pretty obvious why, in the pre-fetching scheme, we perform depth-first loading with respect to the buckets in each block. Since the tuples in T_1 are sorted and the "Bucket" structure of T_2 is built by sorting the tuples of T_2, we would intuitively expect the attributes of the first few tuples of T_1 to have a match in the first few blocks of the "Bucket" structure.

We now provide an example to illustrate both our loading strategies. As before, let us suppose we are evaluating the query $Q = Q_1 -_a Q_2$, and Q_1 = "Select * from T_1;" and Q_2 = "Select * from T_2;". T_1, T_2 and the "Bucket" structure for T_2 are shown in Figure 4.6(a), Figure 4.6(b) and Figure 4.6(c) respectively. We now process the first tuple of T_1, call it t_1. In the on-demand scheme, we would load the first (root) block B_0 from the disk. Since it has a bucket with value $NULL$ we proceed with our search in its sub-buckets no matter what the value in t_1 corresponding to attribute VA is. So we load B_1 from the disk and search for bucket with value 2. On performing binary search we arrive at the bucket with value 2. So we load its sub-buckets, i.e. block B_3, from the disk and search for bucket with value 5. On doing a binary search in B_3 we find a bucket with value 5. So we do not include t_1 in the result. We now consider the next tuple of T_1, call it t_2. We search in B_0, which is already in main memory. That leads us to block B_1, which is also already in main memory. Performing a binary search in B_1 leads us to the bucket with value 1, which is what we are looking for. We then load its sub-buckets, i.e. B_2, from the disk and perform a binary search to arrive at the bucket with value 2. Since we find a tuple in T_2 that is compatible with t_2, we do not include t_2 in the result.

On the other hand, in the prefetching approach we load as much "Bucket" structure as possible into the main memory. In the best case, if we have enough main memory to load the entire "Bucket" structure we do so. In this scenario, we load blocks B_0, B_1, B_2 and B_3 into the main memory. Having loaded all the blocks into the main memory, we compute the aggressive minus as illustrated previously. What do we do if the main memory cannot accomodate the entire structure? We discuss this scenario, with an example, subsequently.

Let us revisit the example in Figure 4.3. Let T_1 be the table shown in Figure 4.3(a). Let T_2 be the table shown in Figure 4.3(b). Let Q_1 be "Select * from T_1;" and Q_2 be

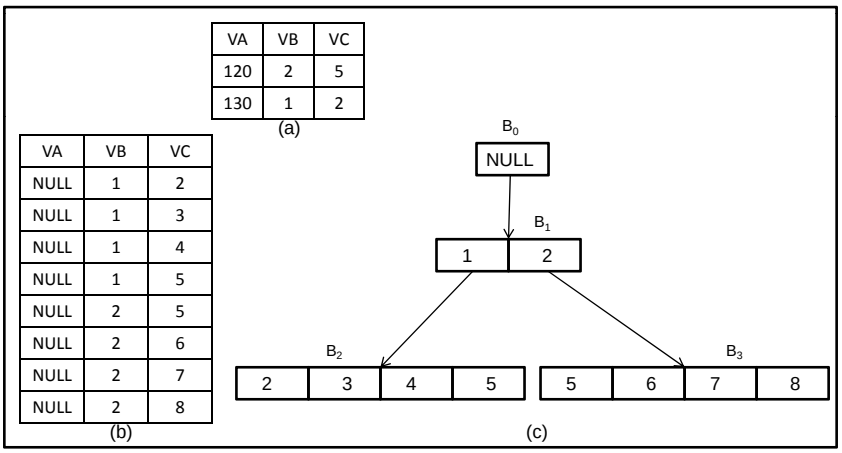

Figure 4.6: Example to illustrate performance of on-demand and prefetching schemes

"Select * from T_2;". We are interested in computing $Q1 -_a Q2$. Let us assume that we have already constructed the "Bucket" structure for T_2, shown in Figure 4.3(c), in the disk. Let us further assume that the main memory is sufficient to load only 11 buckets.

In this scenario, in our prefetching scheme we initially load blocks B_0, B_1, B_2, B_5, B_6 and B_7. We can observe that, in order to process the first 3 tuples of T_1 the blocks we have already loaded into main memory suffice. To process the fourth tuple of T_1, we require blocks B_3 and B_8. To load these blocks into main memory we need to deallocate some of the blocks currently in main memory. One key observation we make here is that we are currently processing a tuple in T_1 with $VA = 2$. Since the tuples in T_1 are sorted we are sure not to further encounter any tuple in T_1 with $VA = 1$. This suggests that the bucket in B_0 with value 1 and its sub-buckets in blocks B_2, B_6 and B_7 that are currently in main memory would not be used any further in evaluating the query. Hence this observation allows us to deallocate these buckets from the main memory. Now, we load the required blocks B_3 and B_8 and continue with the query evaluation.

Having discussed in detail the building of the "Bucket" structure and elaborating on how we use it to compute aggressive and conservative minus, we would like to briefly discuss why we opted for the "Bucket" structure and not used an existing data structure such as a B+ tree. To start with, we cannot use the existing B+ tree search algorithm as such in trying to check for tuple compatibility. In the existing B+ tree search algorithm, at each point we

follow a single branch upon comparing with the key value. This would not be sufficient in our query evaluation algorithm. Since NULL matches any value, we would have to modify the B+ tree search algorithm to search multiple branches.

Secondly, B+ trees are usually built on key values corresponding to a single attribute. In our algorithm, we would require an index on all attributes, since all attributes in a relation can be deemed sensitive in the worst case. As a result, if we use a B+ tree, then the B+ tree needs to be indexed on all attributes. As a result, each key value in the B+ tree node would be replaced by values of all attributes. Typically, the size of the B+ tree node is chosen such that it would fit into a single disk block. Let's assume that the disk block size is 1 KB. Let us further assume that we have n attributes, all integers. Thus, the effective number of keys stored in the B+ tree node is (1024/4n), i.e. 256/n. We have 256/n values in the node corresponding to the first attribute. In contrast, in our "Bucket" structure, each block contains values corresponding to a single attribute, which means a single disk block contains 256 values corresponding to the first attribute. This implies that the coverage for a given attribute is greater in the "Bucket" structure than in B+ trees. As a result, we used the "Bucket" structure instead of B+ trees.

Chapter 5

Experimental Results and Analysis

In this chapter we discuss the experiments that we perform to analyze and compare our database level implementation to enforce access control with the query modification approach proposed by Wang et al. [5] as well as the existing approach to evaluate queries without access control. We restrict ourselves to queries involving the SQL set difference operation.

In order to measure performance we use the following parameters.

Parameter	**Description**
Table Size	Number of tuples (records) in the table
Execution Time	Time to execute a query of the form $Q_1 - Q_2$
Sensitivity	Number of attributes considered sensitive
Disclosure Probability	Probability with which a sensitive attribute's value is authorized to be disclosed
Range	Set of values an attribute can take
Main Memory Available	Amount of main memory available to hold the "Bucket" structure

We first discuss the experimental setup in which we elaborate on the data set and the type of queries we used for our experiments, and also the configuration of the computer in which we performed the experiments. We then discuss the various experiments, their intent and analyze the results. Our experiments were targetted to observe how well our database level implementation scales with table size. We study the difference in behavior between our

Attribute	Description
ID1 (number)	Primary key, sequential order
ID2 (number)	Candidate key, random order
VA (number)	Value 0-k, k = TabSize/Uniformity
VB (number)	Value 0-k, sensitive attribute
VC (number)	Value 0-k, sensitive attribute
Select-25 (number)	Values 0-1 (25% = 1)
Select-50 (number)	Values 0-1 (50% = 1)
Select-75 (number)	Values 0-1 (75% = 1)
Disclose-50-B (number)	Values 0-1 (50% = 1), governs VB
Disclose-75-B (number)	Values 0-1 (75% = 1), governs VB
Disclose-90-B (number)	Values 0-1 (90% = 1), governs VB
Disclose-50-C (number)	Values 0-1 (50% = 1), governs VC
Disclose-75-C (number)	Values 0-1 (75% = 1), governs VC
Disclose-90-C (number)	Values 0-1 (90% = 1), governs VC

Figure 5.1: Description of benchmark datasets.

database level implementation and the query modification approach [5] by varying Range and Disclosure Probability parameters. We then compare the performance of the pefetching and the on-demand schemes. We finally analyze how our prefetching scheme performs by varying the main memory made available to load the "Bucket" structure.

5.1 Experimental Setup

The tables used in our experiments were generated based on the Wisconsin benchmark [18], the description of which is shown in Figure 5.1.

All the experiments were run on the POSTGRESQL database management system. The machine on which the experiments were performed has a system memory of 512 MB, clock speed of 1.80 GHz, cache size of 256 KB, bus speed of 100 MHz, and a block size of 1 KB.

All the queries in the experiments had the form $Q_1 - Q_2$. Neither Q_1 nor Q_2 contain sub-queries. One or two of the three attributes (Va, Vb and Vc) are sensitive. We avoid having all three attributes sensitive in order to avoid the situation where one tuple in the result of Q_2 contains all unauthorized values, in which case every tuple of Q_1 would then

match this tuple of Q_2. When loading the "Bucket" structure from the disk into the main memory we disable the kernel readahead in order to avoid any caching of the index file.

We use the following terminology while plotting graphs.

- SoundDB - The database level implementation proposed in this thesis
- SoundQM - The query modification approach Wang et al. [5]
- Unmodified - The original algorithm with no sensitive attributes
- Prefetching - The approach where the "'Bucket"' structure is loaded in the main memory in a greedy fashion
- On-Demand - The approach where buckets are loaded into main memory as and when they are necessary

5.2 Results and Analysis

The first set of experiments were performed in order to compare the performance of SoundDB, SoundQM and Unmodified approaches.The results are reported in Figure 5.2.

We would like to point out to the readers that the execution time reported for the SoundDB approach in Figure 5.2 includes both the index building time and the query evaluation time. From Figure 5.2, we can observe that the performance of SoundDB is better compared to the performance of SoundQM. This is because the overhead of computing the join operation in SoundQM has been eliminated in SoundDB approach by modifying the query evaluation engine of POSTGRESQL. Many optimizations targetted specifically at access control, such as building the "'Bucket"' structure to efficiently search for compatible tuples, make the performance of SoundDB better compared to SoundQM. For moderately sized tuples, with 100000 records, the performance of SoundDB is comparable to that of Unmodified.

Figure 5.3 compares the performance of SoundDB and Unmodified for larger table sizes. Figure 5.3 also reports the break-up of SoundDB into index building and query evaluation. For tables with upto 300000 tuples, building the bucket structure on-the-fly for the

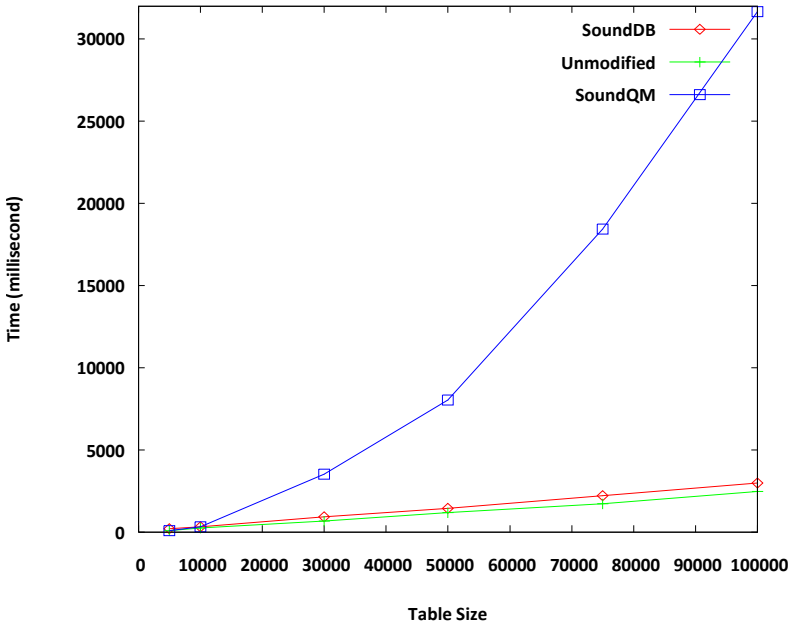

Figure 5.2: Comparison of SoundDB, SoundQM and Unmodified approaches. Other parameters: Range = 1000, Sensitivity = 2, Disclosure Probability = 75%

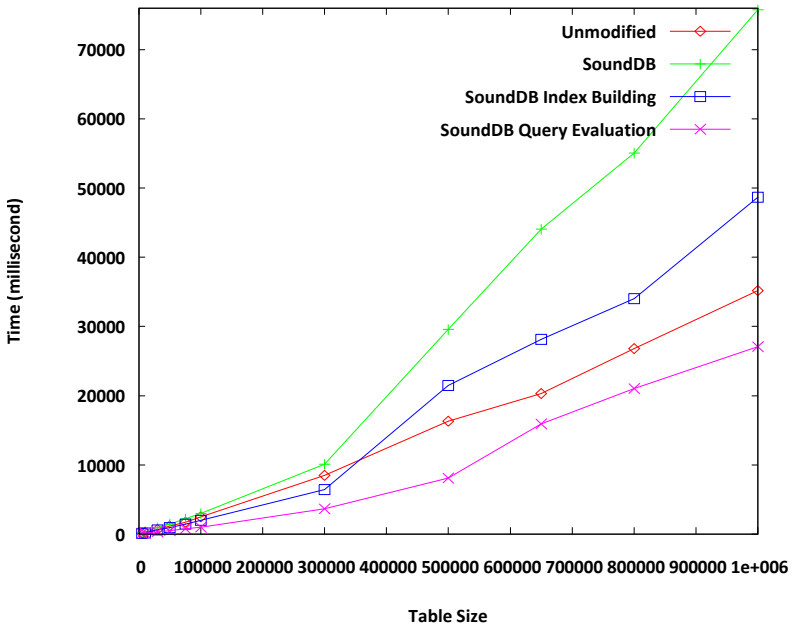

Figure 5.3: Break up of SoundDB into index building and query evaluation. Other parameters: Range = 1000, Sensitivity = 2, Disclosure Probability = 75%

SoundDB approach does not affect the performance much. For larger tables, it is the index building time that dominates over the query evaluation time in SoundDB approach. In

such scenarios it might be wise to make the index building a one time task. Once the index structure is built, the query evaluation does not take much time and its performance is even slightly better than Unmodified. This is because, in POSTGRESQL, the except operator is implemented in a manner similar to merge join. Since we have already built the "Bucket" structure, the time spent on sorting, which is the major bottleneck in merge join, is avoided when evaluating the query using SoundDB.

Our next set of experiments was targetted at observing the manner in which the SoundQM and SoundDB approaches performed with different Ranges. Figure 5.4 reports the behavior of SoundQM and Figure 5.5 reports the behavior of SoundDB with varying Range values.

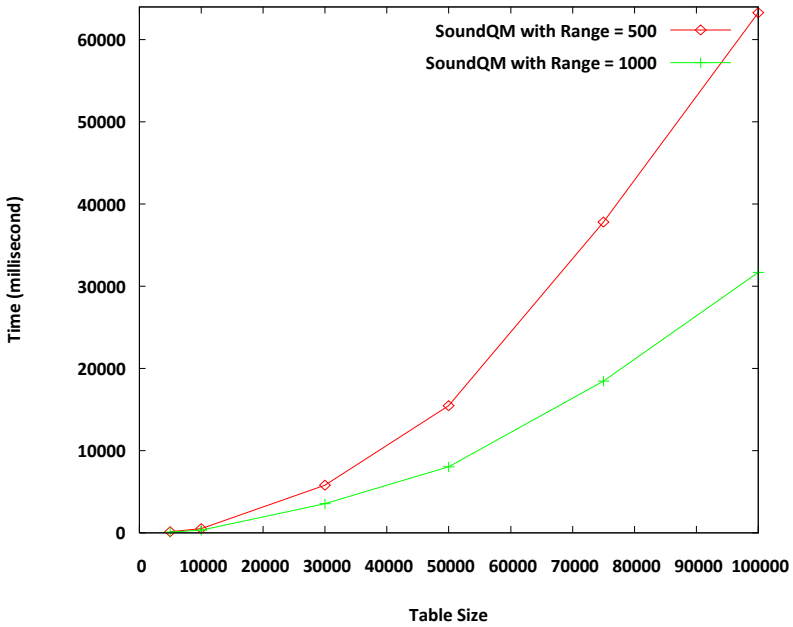

Figure 5.4: Behavior of SoundQM with varying Range. Other parameters: Sensitivity = 2, Disclosure Probability = 75%

From Figure 5.4 we can observe that the execution time of SoundQM increases with decrease in Range. This is because, for a given table size, when the range is 500, the number of tuples having the same value in an attribute is more compared to the number of tuples having the same value in an attribute when the range is 1000. As a result, the join operation in SoundQM takes a longer time to execute since each tuple in the result of Q_1 has to be compared to more number of tuples in the result of Q_2. In contrast, we can observe

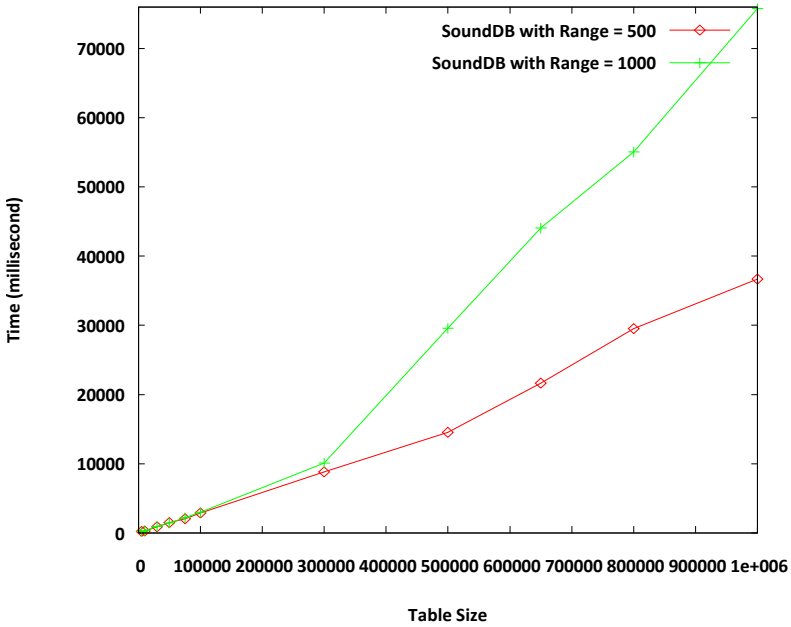

Figure 5.5: Behavior of SoundDB with varying Range. Other parameters: Sensitivity = 2, Disclosure Probability = 75%

from Figure 5.5 that the execution time for SoundDB increases with increase in Range. This is because, the number of buckets corresponding to a column is larger when range is 1000, than when the range is 500. As a result, the search space for each column increases with increase in Range. Due to this reason, the performance of SoundDB is reduced, in other words the execution time increases, when Range increases.

Our next set of experiments were conducted to analyze the behaviour of SoundQM and SoundDB when the disclosure probability and the number of sensitive attributes are varied. Figure 5.6 and Figure 5.7 report the results for these experiments.

From Figure 5.6, we observe that SoundQM performs better when the number of sensitive attributes are less, or in other words, SoundQM performs better when more cells can be disclosed. This is because, we use the rule that NULL matches any vaue, and existing DBMSs do not have optimization for this. When the disclosure probability of an attribute, say a_1, is 100%, the join condition simply becomes $(T_1.a_1 = T_2.a_1)$ instead of $((T_1.a_1 = T_2.a_1)OR(T_1.a_1 isNULL)OR(T_2.a_1 isNULL))$. In contrast, SoundDB performs significantly better when there are more sensitive attributes, or in other words, SoundDB performs better

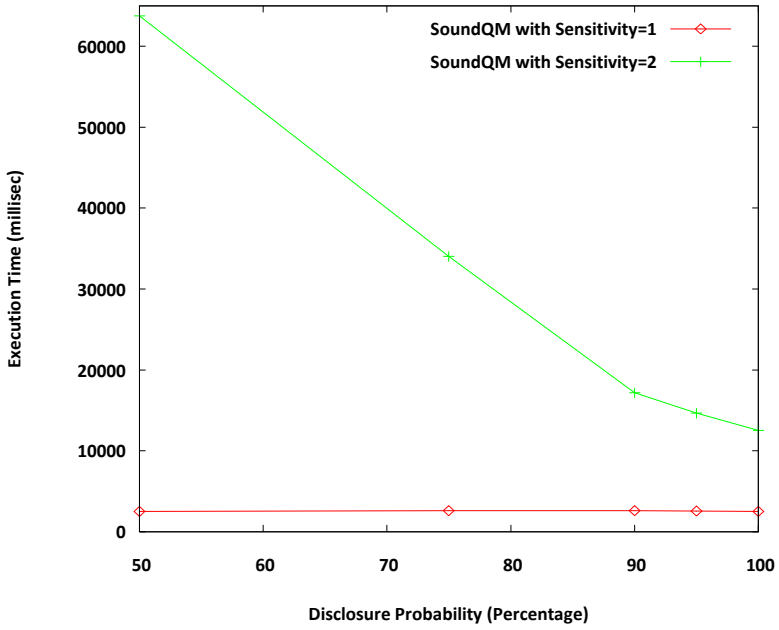

Figure 5.6: Behavior of SoundQM with varying Disclosure Probability. Other parameters: Sensitivity = 2, Table Size = 100000, Range = 1000

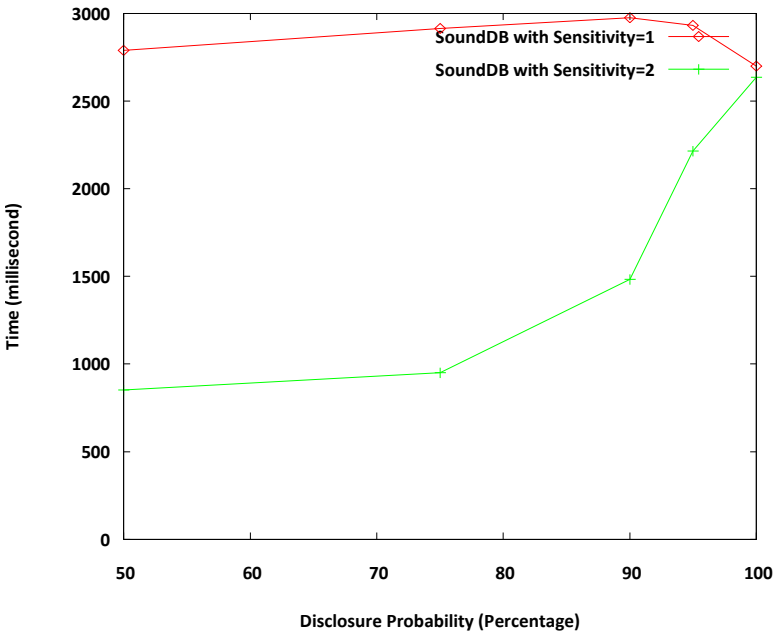

Figure 5.7: Behavior of SoundDB with varying Disclosure Probability. Other parameters: Sensitivity = 2, Table Size = 100000, Range = 1000

when less number of cells can be disclosed, which is evident from Figure 5.7. The intuition behind this is that, when more number of cells in the result of Q_2 are unauthorized the more

number of tuples of Q_1 have a match in the result of Q_2, and this match is obtained quickly since the unauthorized cells, which are represented using NULL values, occur earlier in the "Bucket" structure compared to other values. The evidence for this argument is presented in Figure 5.8.

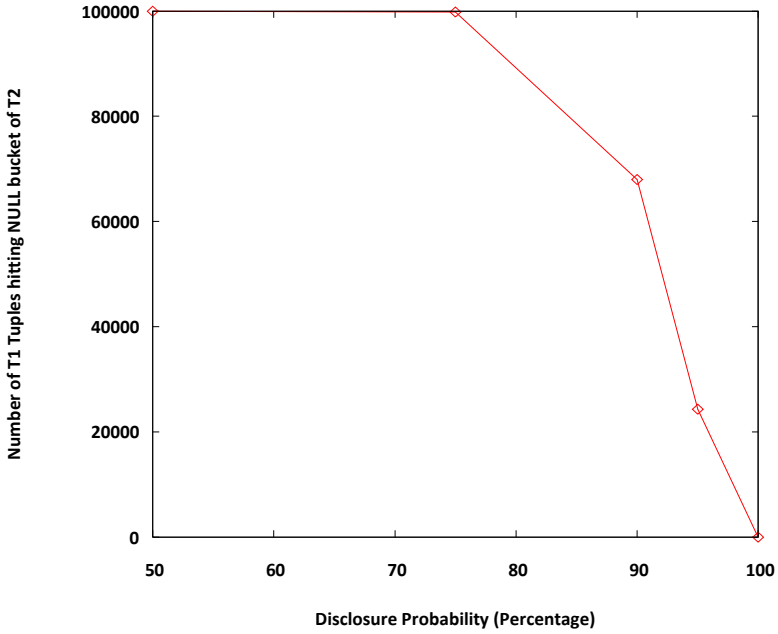

Figure 5.8: Number of Tuples in $T1$ having a match in sub-buckets corresponding to the NULL bucket in the "'Bucket"' structure of $T2$ vs Disclosure Probability. Other parameters: Sensitivity = 2, Table Size = 100000, Range = 1000

Figure 5.8 shows the number of tuples in the result of Q_1, say T_1, having a match corresponding to the NULL buckets in the "Bucket" structure built for the result of Q_2, say T_2. In T_2, for each distinct value of VA (between 0 and 999, since range is 1000) we have one tuple with VB and VC as NULL, when the disclosure probability is 50%. This is precisely why all 100000 tuples of T_1 hit the NULL buckets corresponding to the "Bucket"' structure of T_2. In T_2, even when the disclosure probability was 95% there were 229 values of VA which had VB and VC as NULL. For these 229 values of VA we need not go about doing a binary search like how we would do for non-NULL buckets (when disclosure probability is 100%). This is the reason why SoundDB performs better when disclosure probability is low.

In general, when disclosure probability decreases, there are more NULLs in the tables, which could lead to more matching and thus more comparison operations. For example, given two tuples (x_1, x_2, x_3) and (y_1, y_2, y_3), x_3 needs to be compared with y_3 only if x_1 matches y_1 and x_2 matches y_2. If there are more NULLs, then the probability that x_2 matches y_2 increases, which increases the expected number of times x_3 is compared with y_3 and leads to longer execution time. Our database level implementation is optimized to handle NULLs, through the "Bucket" structure, and it shows in the experimental results that this optimization indeed works.

Our next set of experiments were targetted at comparing the performance of the on-demand scheme and the prefetching scheme of SoundDB. Figure 5.9 reports the result that we obtained for this experiment.

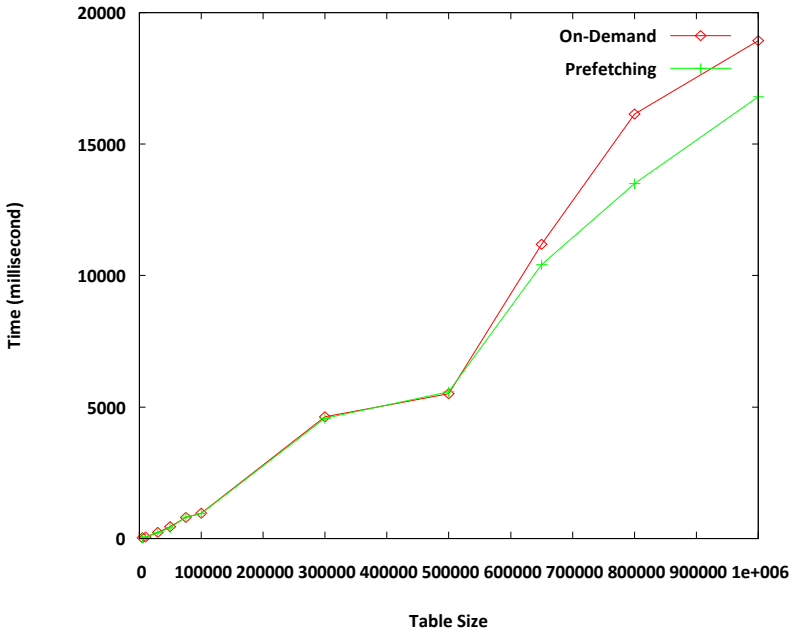

Figure 5.9: Comparison of prefetching and ondemand schemes of SoundDB approach. Other parameters: Sensitivity = 1, Disclosure Probability = 0%, Sensitive Attribute = VA, Range = 1000

We can see from Figure 5.9 that when the table size is less than 500000, the performance of the two schemes are comparable. When the table size increases beyond 500000, the performance gap becomes marginal, with prefetching scheme performing better than the on-demand scheme. In order to analyze this graph we would like to revisit the

example in Section 4.2.2 that we use to illustrate the working of our on-demand loading strategy (refer Figure 4.6). We can observe that the on-demand scheme leads to a non-sequential acess of the index file (i.e. "Bucket" structure in disk). For a given range, the number of tuples corresponding to a given value in an attribute is less when the table size is less and increases with increase in table size. Thus, the non-sequentiality access of the index file increases with increase in table size. This is precisely the reason why the performance of the on-demand scheme is marginally less compared to the prefetching scheme.

On analyzing both the schemes in detail, intuitively, we expect the prefetching scheme to out perform the on-demand scheme. Unfortunately, the results show that their performance is quite comparable. In this experiment the attribute VA was completely undisclosed. As a result, there is only 1 bucket corresponding to the attribute VA in the "Bucket" structure. So there is no appreciable non-sequentiality with respect to attribute VA in the on-demand scheme. As a result, we conduct experiments varying the disclosure probability of attribute VA and study the difference in performance between the two loading strategies. The performance of the prefetching and the on-demand schemes with varying disclosure probability are reported in Figure 5.10. Figure 5.11 reports the performance difference (i.e. On-demand execution time - Prefetching execution time) with varying disclosure probability.

We can see that the performance gap increases with increase in disclosure probability. The extent of non-sequentiality in the on-demand scheme increases with increase in disclosure probability. As we increase the disclosure probability we reduce the NULLs in the table, thus reducing the chances of finding a "match" (i.e. compatible tuple) early in the "Bucket" structure. As a result, the index file has to be scanned back and forth multiple times, once to search in the sub-buckets of bucket with NULL value and another for the bucket with the corresponding value. Also, even when the disclosure probability is 100%, the prefetching scheme performs better. This is because, the on-demand scheme leads to a linear search with respect to the blocks in the disk (although we do binary search within each block), whereas in the prefetching scheme we do a "perfect" binary search.

We would like to point out to the readers that the performance gap is actually less when the disclosure probability is 100%, than when it is 90% or 95%. This is because of the presence of NULLs. The sub-buckets of the bucket with value NULL get searched first. When a match is not found we then go about with our binary search process. Obviously, the

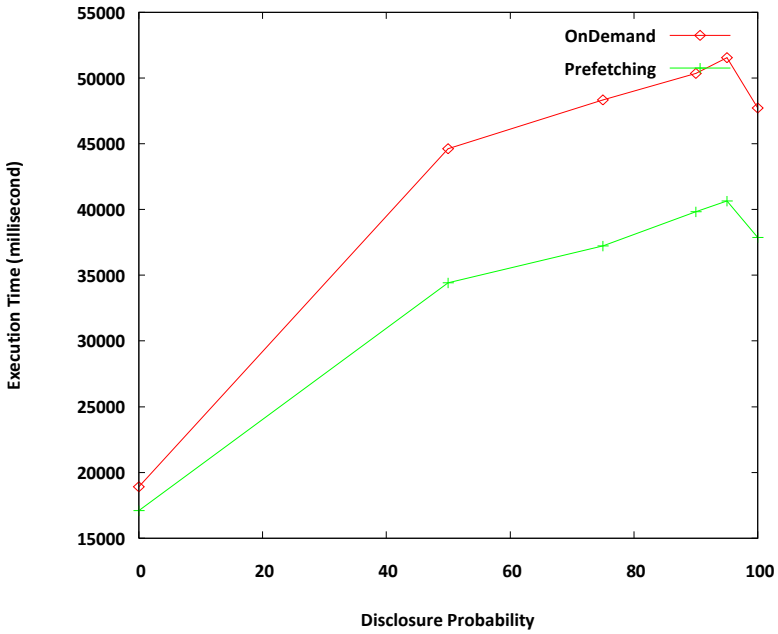

Figure 5.10: Comparison of prefetching and ondemand schemes of SoundDB approach. Other parameters: Sensitivity = 1, Table Size = 1000000, Sensitive Attribute = VA, Range = 1000

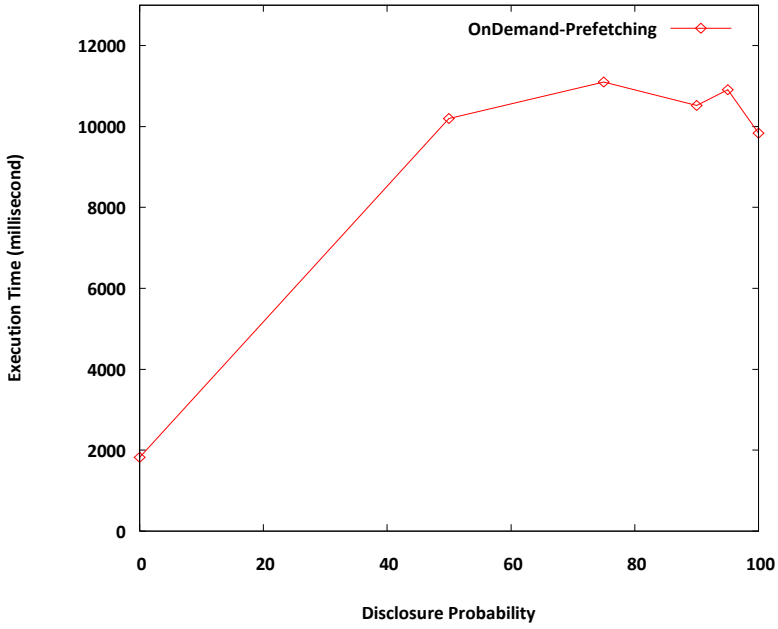

Figure 5.11: Performance difference between prefetching and ondemand schemes of SoundDB approach. Other parameters: Sensitivity = 1, Table Size = 1000000, Sensitive Attribute = VA, Range = 1000

search in sub-buckets of the bucket with NULL value is avoided when disclosure probability is 100%.

Our last experiment was conducted to analyze the performance of the prefetching scheme of SoundDB with varying amounts of system memory. Figure 5.12 reports the result of this experiment.

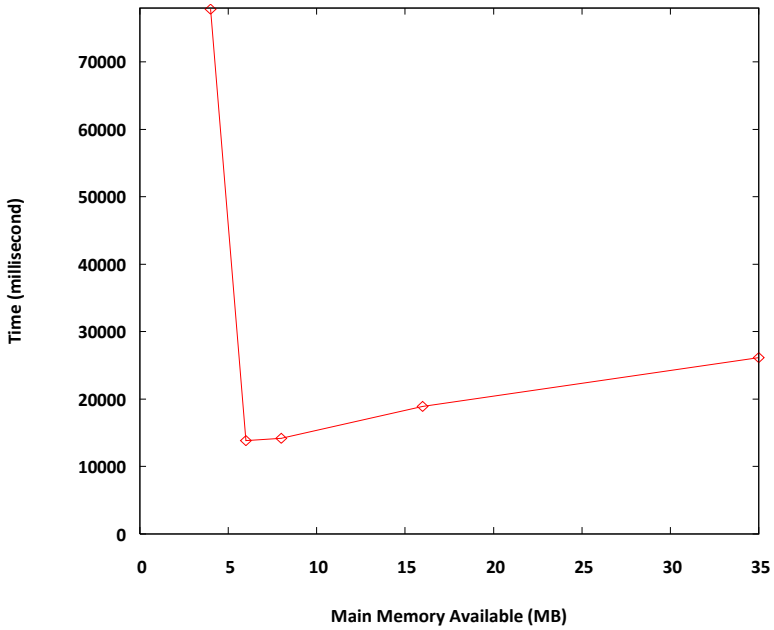

Figure 5.12: Performance of SoundDB prefetching scheme with varying amounts of system memory. Other parameters: Sensitivity = 2, Disclosure Probability = 75%, Table Size = 1000000, Range = 1000

The index size for the table T_2 was found to be approximately 32 MB. In this experiment, both VA and VB were considered to be sensitive attributes. The size of the index corresponding to the NULL bucket of VA was found to be around 4 MB. We can certainly say that the sub-buckets associated with the NULL bucket of VA would be used to answer almost every tuple of T_1, but we can not say much about the sub-buckets corresponding to the NULL bucket of Vb as they would occur corresponding to many buckets of VA, and also it is not guaranteed that they would be used for answering every tuple of T_1. So we focus our attention on the NULL buckets corresponding to the attribute VA.

When the system memory available is between $5MB$ and $10MB$ the performance of the prefetching scheme of SoundDB is best. When the memory is more, we try to greedily load buckets into the system memory, and as a result we end up loading many buckets that are not going to be used at all. The more the system memory available, the more the number of unwanted buckets we would load into memory and hence greater the execution time. When the system memory falls below 4 MB, which is the size of the index corresponding to the unauthorized value (NULL) of VA, the performance falls drastically. This is because, the sub-buckets of the NULL bucket of VA is swapped back and forth between main memory and disk, which turns out to be pretty expensive.

5.3 Discussion

On analyzing the process of building our "Bucket" structure, we find that we build redundant buckets. To make this point clearer, we elaborate with an example. Consider the table shown in Figure 5.13(a). The corresponding "Bucket" structure is shown in Figure 5.13(b).

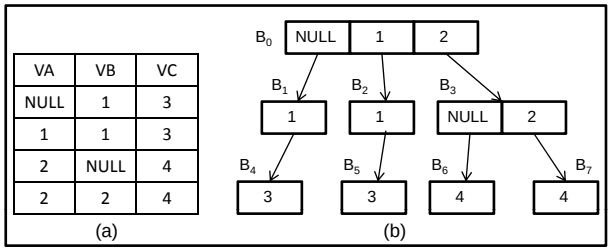

Figure 5.13: Example to illustrate redundancy in "Bucket" structure

We can observe that there are 3 distinct buckets (with values $NULL$, 1 and 2) corresponding to attribute VA in block B_0. Also, the sub-buckets of the buckets with values $NULL$ and 1 are the same. While evaluating the query $Q_1 -_a Q_2$, where Q_1 = "Select * from T_1;" and Q_2 = "Select * from T_2;" (T_2 is the table in Figure 5.13(a)), if a tuple in T_1 is not going to be compatible with the tuple $(NULL, 1, 3)$ of T_2, then it is obviously not going to be compatible with the tuple $(1, 1, 3)$ of T_2. This suggests that the bucket with value 1 in block B_0 and the buckets in blocks B_2 and B_5 are not required. Also, the bucket in block B_3 with value 2 and the bucket in block B_7 are not required for the same

reason. This further suggests that we need not have built those buckets while constructing the "Bucket" structure. How does this improve the performance of our approach? We discuss this subsequently.

By not constructing the non-required buckets (like those in the above example) we reduce the size of the index file. Thus, the time required to load the index file into main memory decreases as a result of a lesser seek time. This in turn improves the performance of the query evaluation. In addition, by getting rid of the non-required buckets, we make sure that "useful" buckets get loaded into the main memory while prefetching the index file from the disk in our prefetching scheme. This again suggests performance improvement while evaluating aggressive minus.

On the contrary, we need to preprocess the table for which we wish to construct the "Bucket" structure in order to remove the tuples that lead to non-required buckets (such as the ones provided in the above example). By doing so, we increase the time to build the "Bucket" structure. This is a trade-off that we need to consider. If the "Bucket" structure needs to be built on-the-fly, then it might not be worth to spend extra time on building it. On the other hand, if the aggressive minus is computed frequently and the table used is relatively stable, then spending the extra time in building the "Bucket" structure might be worth in the long run.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, we present a database level implementation to efficiently enforce fine grained access control. We modify the query evaluation engine of POSTGRESQL to evaluate queries in a sound and secure fashion. Though the query evaluation algorithm that we implement is not our contribution per se, we would like to bring to the notice of the readers that the database level implementation of the algorithm is certainly not a trivial task. We find, through experiments, that the most straightforward approach to implement the query evaluation algorithm is inefficient. As a result we introduce an index structure, which we call the "Bucket" data structure, and suggest optimizations to improve the performance of the database level implementation. We also address challenges in the form of main memory constraints. We introduce memory management schemes so that our implementation tries to make use of the main memory available in an adaptive fashion. We measure the performance of our implementation against various parameters, and prove experimentally that it performs better compared to the query modification approach of implementation proposed by Wang et al. [5]. Experimental results also suggest that the performance of our implementation to enforce fine grained access control is comparable to POSTGRESQL's implementation without access control, which shows that our implementation is practical.

6.2 Future Work

In our prefetching scheme, we prefetch blocks from the disk blindly, without making an effort to determine if the block we prefetch is actually going to be useful. As a result, we end up loading blocks from the disk that are never used for the query evaluation, which degrades the performance to a certain extent. We propose to improve the performance of the *prefetching* scheme by looking ahead tuples of the table on the left handside of the minus operator to determine the usefulness of a block, before it is prefetched from the disk, in answering the query.

As an alternative to our "Bucket" structure, we can use R-tree to compute aggressive minus. We can do so by treating each tuple in the relation as a single multidimensional attribute, i.e. a point in space. We propose to modify the insertion algorithm of R-tree to suit our semantics of NULL. It would be interesting to compare the performance that we would obtain using the R-tree with the performance we obtained with our "Bucket" structure. We would like to study this comparison sometime in the future.

Exisitng DBMSs do not have provisions for the type-1 and type-2 variables used in the labeling mechanism proposed by Wang et al. [5]. As a result we use NULLs to represent unauthorized cells, which does not guarantee the satisfaction of maximal criterion. Also, without type-2 variables, it is difficult to preserve the linking between tables if the attribute involved in linking is replaced by NULL. We propose to incorporate type-1 and type-2 variables in POSTGRESQL and implement the labeling mechanism proposed by Wang et al. sometime in the future. Currently, our implementation ensures the sound and secure enforcement of fine grained access control at the database level, assuming that policies for each query context are in place. Luo et al. [19] discuss ways to store XML access control policies in relational DBMSs. We propose to implement something similar in POSTGRESQL to allow policies specified in languages such as XACML to be stored in relational DBMSs. This together with implementing the labeling mechanism introduced by Wang et al. [5] in POSTGRESQL would make it a more complete access control system.

Bibliography

[1] L. Olson. Database Access Control Tutorial. Lecture Slides of Computer Security class. Available at www.cs.uiuc.edu/class/sp07/cs498cag/slides/463.5.1%20DB%20Access%20Control.ppt

[2] R.S. Sandhu, E.J. Coyne, H.L. Feinstein, C.E. Youman. Role Based Access Control Models. Computer. Vol. 29. No. 2, Feb 1996, pp 38-47

[3] R. Sandhu. INFS 762: Information Systems Security, Summer 2004, Lecture Notes. Available at www.list.gmu.edu/infs762/infs762su04ng/l1-access-control.ppt

[4] D. Bell and L. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, Mitre Corporation, Belford, MA, 1975.

[5] Q. Wang, T. Yu, N. Li. On the Correctness Criteria of Fine-Grained Access Control in Relational Databases. VLDB '07, September 2328, 2007, Vienna, Austria.

[6] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt. Limiting disclosure in Hippocratic databases. In Proceedings of the 30th International Conference on Very Large Data Bases (VLDB), Aug. 2004.

[7] Oracle Coperation. Oracle Database: Security Guide, December 2003. Available at www.oracle.com.

[8] M. Stonebraker and E. Wong. Access control in a relational database management system by query modification. In Proceedings of the 1974 Annual Conference (ACM/CSC-ER), pages 180-186. ACM Press, 1974.

[9] R. Bond, K. Y.-K. See, C. K. M. Wong, and Y.-K. H. Chan. Understanding DB2 9 Security. IBM Press; 1st edition, December 2006.

[10] Oracle. The Virtual Private Database in Oracle9iR2: An Oracle Technical Whi te Paper, Jan. 2002. Available at www.oracle.com/technology/deploy/security/oracle9ir2/pdf/VPD9ir2twp.pdf.

[11] S. Rizvi, A. Mendelzon, S. Sudarshan, P. Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. SIGMOD 2004, June 13-18

[12] IBM Corporation. DB2 Version 9 for Linux, Unix, Windows. Available at http://publib.boulder.ibm.com/infocenter/db2luw/v9/index.jsp?topic=/com.ibm.db2.udb.admin.doc/doc/c0021114.htm

[13] D. Chamberlin. A Complete Guide to DB2 Universal Database. Morgan Kaufmann, 1998.

[14] Patricia P. Griffiths and Bradford W. Wade. An authorization mechanism for a relational database system. ACM Trans. Database Syst. 1, 3 (Sep. 1976), Pages 242 - 255.

[15] R. Agrawal, P. Bird, T. Grandison, J. Kiernana, S. Logan and W. Rjaibi. Extending Relational Database Systems to Automatically Enforce Privacy Policies. IEEE International Conference on Data Engineering, 2005.

[16] R. Agarwal, J. Kiernan, R. Srikant, Y. Xu. Hippocratic Databases. Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

[17] Y. V. Matiyasevich. Hilbert's Tenth Problem. The MIT Press, 1993.

[18] D. J. DeWitt. The wisconsin benchmark: Past, present, and future. In The Benchmark Handbook. 1993.

[19] B. Luo, D. Lee, P. Liu. Pragmatic XML Access Control using off-the-shelf RDBMS. ESORICS 2007

[20] T. Yu, N. Li, Q. Wang, J. Lobo, E. Bertino, K. Irwin, and J.-W. Byun. On the correctness criteria of fine-grained access control in relational databases. Technical Report, Available at www.cs.purdue.edu/homes/wangq/papers/fgdb tr.pdf, March 2007.

[21] E. H. Fredkin. Trie Memory. Communications of the ACM, vol. 3, pp. 490–500, 1960