A Data Level Database Inference Detection System

By

Raymond Wai-Man Yip

B.Sc. (Chinese University of Hong Kong) 1988

M.Phil. (Chinese University of Hong Kong) 1990

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Professor Karl Levitt, Chair

Professor Matthew Bishop

Professor Richard Walters

Committee in Charge

1998

A Data Level Database Inference Detection System

by

Raymond Wai-Man Yip

Abstract

A Data Level Database Inference Detection System

by

Raymond Wai-Man Yip

Doctor of Philosophy in Computer Science

University of California at Davis

Professor Karl Levitt, Chair

Inference is a technique a user can employ to defeat access control mechanisms in a database system. It poses a confidentiality threat to a database system, making it difficult to control access to sensitive information. An inference detection system is needed to determine if users can use legitimately accessed data to infer sensitive information. The design of an inference detection system is a trade-off among soundness, completeness, accessibility of the database, and efficiency of the inference detection process. We describe six inference rules to determine if an adversary has collected enough data to perform inference, namely split query, subsume, unique characteristic, overlapping, complementary, and functional dependency. We prove our detection system is sound, thus it will not object to legitimate queries. Schema-based inference detection systems, which detect inference using functional dependencies, are unsound and incomplete, that is, they can generate false positives and negatives. Our system makes use of the database contents to detect inferences, making it more complete than schema-based inference detection systems. In this respect, our inference detection system detects a known inference attack called Tracker, not detectable by the schema-based approach. Our detection system can be inefficient, as we need to keep track of all queries issued by users, and perform inference detection using them. A performance evaluation of our prototype shows that the system performance is affected by the size of the database, the amount of duplication of data in the database, the number of projected attributes and conjuncts in queries, and the number of return tuples from queries. We show that the system could be practically employed for certain realistic types of databases and queries, for example, a database with large number of attributes. Experimental results

also show a strong correlation between the information inferrable by an adversary and the amount of overlapping among return data and also the fraction of the database that has been accessed by a user. To report on an adversary approaching an inference, we investigate the detection of approximate inference. Our main results are for static inference, but we also investigate the detection dynamic inference.

Professor Karl Levitt
Dissertation Committee Chair

To my parents.

Contents

List of Figures

List of Tables

Acknowledgements

I am indebted to Dr. Karl Levitt for his guidance as my thesis advisor. He introduced computer security to me, spent hours in discussing research ideas with me, encouraged me to pursuit research in database security, and provided me with financial support through his research fundings.

I am indebted to Dr. Richard Walters and Dr. Matt Bishop for being in my dissertation committee and giving me valuable comments in improving this dissertation.

I would like to thank each member in the Computer Security Laboratory at UC Davis for broadening my understanding in various computer security research areas, especially through the weekly seminars. David O'Brien took care of my computer resource needs when I implemented the prototype, ran the experiments, and wrote the dissertation. Christopher Wee answered many of my system administration questions. Steven Cheung spent time to chat with me about research ideas. Dr. Calvin Ko acted as my mentor during my first year in the laboratory. I would also like to thank the staffs in the Computer Science Department for providing me with administrative support.

I am indebted to my parents. Without their support, I would not even think about furthering my study. Last but not least, I am indebted to Hilary Chen who gave me the most needed support and encouragement while I wrote this dissertation.

Chapter 1

Introduction

Modern database systems allow multiple users access to data. When users are not to be allowed accesses to every item of data in the database, an access control system is needed. An access control system has two components: the access control policy and the access control mechanism. The access control policy specifies the accesses that are allowed or disallowed for each user in the database system. The access control mechanism enforces the policy. A mechanism is *sound* with respect to a policy if it allows accesses that are allowed by the policy, and disallow accesses that are not allowed by the policy. The mechanism is *complete* with respect to a policy if it addresses all accesses as specified in the policy.

Each user accesses the database system using queries. For each query issued to the database system, the access control system determines if the query is allowed by the database system. The allowed queries are processed by the database system, and the results are returned to the user. The disallowed queries can be handled in various ways. For example, the user may simply be notified that the query violates the access control policy and is not processed by the database system, or the database system intentionally returns incorrect responses to the user in order to protect the data. The invalid accesses might also be recorded for further investigation.

There are two types of access control systems: mandatory and discretionary. In mandatory access control systems, each piece of data in the database is given a classification level. Each user is assigned a clearance level. It is the clearance level of a user u, the classification level of a data item d, and the type of the access operation that determine whether the user u can access the data d. Common mandatory access control policies

include the Bell-LaPadula [BL73], and Biba policies [Bib77], addressing confidentiality and integrity respectively.

In discretionary access control systems, the accesses to data are explicitly granted to or denied to users. The access control policy can be expressed in two forms: 1) a user is allowed to access all data unless it is prohibited by the policy, or 2) a user is not allowed to access any data unless it is granted by the policy. A typical discretionary access control policy is expressed in a 3-tuple: *(user, data, operation)*. If *(u, d, o)* is in the policy, then subject u is allowed (or disallowed) to perform the operation o on the data d.

Inference is known as a way to defeat an access control mechanism. It poses a confidentiality threat to a database system. Consider the following database with two tables: *NSJ* and *JS*.

NSJ: (Name [U], Salary [C], Job [U])
JS: (Job [U], Salary [U]).

Table *NSJ* stores data about the employee names, their salaries and job titles. Table *JS* stores data about the job titles and their salaries. Attributes followed by an *'U'* are *unclassified*. Attributes followed by an *'C'* are *confidential*. Consider a mandatory access control system that enforces the Bell-LaPadula policy. It states that a user can read a piece of data if the clearance level of the user dominates the classification level of the data. Suppose the access control policy requires that only users with *confidential* clearance can access the salaries of employees. A naive way to enfore this policy is to classify the attribute Salary in table *NSJ* as *confidential*, while leaving other attributes as *unclassified*. In this way, an unclassified user cannot directly access the salaries of employees from table *NSJ*. However, suppose job titles functionally determine salaries; that is, each job title is associated with one and only one salary. With this property among the data, an unclassified user can infer the salaries of employees. This is done by discovering the job J of an employee E from table *NSJ*, and then the salary S of the job J from table *JS*. Then, the user can infer that the salary of the employee E is S.

Inference can also occur in discretionary access control systems. Suppose the policy specifies that a user U is not allowed to access the salaries of employees. Again, a naive way to enforce this policy is to reject any query issued by the user U that accesses data about both the employee names and salaries. Using a similar method as discussed above,

if a user can find out the job title J of an employee E, and the salary of the the job title J is S, the user can infer the salary of the employee E.

The inference problem is defined as follows.

Definition 1 *An inference occurs when a user is able to use legitimately accessed data to infer data that the user is not allowed to access according to an access control policy.*

In mandatory access control systems where the Bell LaPadula policy is employed, the legitimate data that a user U can access are those whose classification levels are dominated by the clearance level of the user U. In discretionary access control systems, legitimate data that a user U can access are those that the user is allowed to access according to the access control policy. The existence of the inference problem defeats the access control mechanisms, making them neither sound nor complete. In general, the set of data that can be inferred is determined by the structure of the database, and the data stored in the database.

A multilevel database system is a database system that enforces the Bell LaPadula policy. Early work on inference detection in multilevel database systems employed a graph to represent functional dependencies among attributes in the database schema. Each node in the graph corresponds to an attribute in the database schema. An edge from node A to node B in the graph indicates that the attribute corresponding to node A functionally determines the attribute corresponding to node B. An inference path is detected when there are two or more paths found in the graph that connect one node to another, and the paths are labeled at different classification levels [Hin88, Bin93b, QSK+93]. The classification level of a path is the least upper bound of the classification levels of the attributes corresponding to the nodes on the path. For example, consider the *NSJ* and *JS* tables. We can construct a graph as shown in Figure 1.1 to represent the database schema.

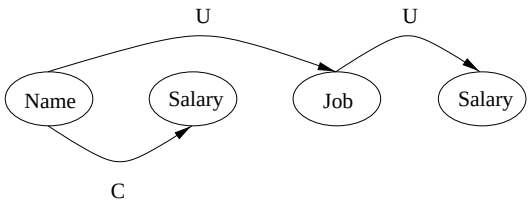

Figure 1.1: An Example on the Schema-based Inference Detection Approach.

Each node in the graph corresponds to an attribute in the database. The node 'Job'

corresponds to the attribute Job in both tables. Each edge is labeled with the the least upper bound of the classification levels of the attributes connected to the edge. An inference exists in this database as there are two paths with different classification levels. They are the unclassified path connecting the 'Name' to 'Job' which in turn is connected to 'Salary', and the classified path connecting the 'Name' and 'Salary'.

Lunt [Lun89] points out that some inference problems can be avoided by redesigning the database schema, and classifying the attributes properly. However, redesigning the database schema results in data duplication which leads to update anomalies. It also requires modifications to existing application programs.

Inference paths can also be eliminated by upgrading attributes along the paths [SO87, Sti94]. However, upgrading makes the database less accessible to users. Consider the inference as found in the *NSJ* and *JS* tables. This inference can be blocked by classifying the attribute Salary in table *JS* as *confidential*, as shown in the following new database schema,

NSJ: (Name [U], Salary [C], Job [U])
JS1: (Job [U], Salary [C]).

In this way, unclassified users cannot access any data about salaries, and hence cannot perform the inference. However, this makes the database system less useful to users. For example, it prevents unclassified users from accessing the salaries of job titles, not prohibited by the policy. Knowing the salaries of job titles does not necessary always lead to inference of salaries of employees. If an unclassified user only accesses job and salary data in table *JS1* but does not access table *NSJ*, then the user cannot infer the salaries of employees. However, with the new schema, such accesses are denied. Therefore, the new schema prevents inference at the expense of decreasing the accessibility of data to unclassified users. An ideal access control system should on the one hand prevent inference, and on the other hand provide maximum accessibility of the database system to users.

In discretionary access control systems, it is not an obvious task to grant users with all of the access rights they need. In some cases, users are simply given more access rights than they need in order not to hinder their work. User activities should be monitored to make sure that they are not misusing their privileges. A simple way to monitor user accesses is to reject any query that performs unauthorized access. However, a user may issues a series of authorized queries to infer data, using the similar inference techniques as

in multilevel database systems. Motro *et al.* address a similar problem, but their work focuses on detecting aggregation instead of inference attacks [MMJ94]. In the statistical database security community, various techniques have been proposed to protect individual records, for example, query-set-size control, cell suppression, and data perturbation [AW89]. However, these techniques are not suitable for detecting inference attacks employing general purpose queries.

As noted by SRI researchers, monitoring user activities may lead to detecting more inference than that can be detected by using schema-based approach which yeilds both false positive and false negative reports [SGLQ94]. Consider the following database.

Name	Salary	Resident
Robert	45K	San Francisco
Phil	50K	San Francisco
Jack	60K	Palo Alto

Resident	Salary
San Francisco	45K
San Francisco	50K
Palo Alto	60K

In this database, the attribute Resident does not functionally determine attribute Salary, as both Robert and Phil live in San Francisco but they earn different salaries. As a result, schema based inference detection systems do not report any inference threat in this database. However, if a user knows that Jack is the only employee who lives in Palo Alto, the user can infer the salary of Jack by querying the database to find the salary of the employee who lives in Palo Alto in the second table. This example illustrates that simply examining the database schema to detect inference is not sufficient, and taking the data in the database into consideration can lead to the detection of more inferences. We term the approach that detects inference with the consideration of the data in the database as the *data level inference detection approach.*

In this dissertation, we describe our effort to develop a data level inference detection system. We have identified six inference rules that users can use to infer data: *'split query'*, *'subsume'*, *'unique characteristic'*, *'overlapping'*, *'complementary'*, and *'functional dependency'* inference rules. These rules essentially cover the set-subset, intersection, and difference relationships among the return tuples of queries. The functional dependency inference rule is introduced to simulate the schema-based inference detection approach.

Hence, our system can detect more inferences than that detected by systems using the schema-based approach. The detection system can apply these rules any number of times, and in any order to detect inference. These inference rules are sound but not necessarily complete. Although we have no example that demonstrates incompleteness, more research effort is needed to determine if they are complete. We employ a rule-based approach so that when a new inference rule is discovered, it can be incorporated into the inference detection system. Our work confirms the inadequacy of the schema-based inference detection approach.

We have developed a prototype of the inference detection system to study its performance. The experimental results show that the system performance is affected by the characteristics of the database and queries. In general, the inference detection system performs better with a larger number of attributes in the database, more duplication of attribute values in the database, smaller number of tuples returned by the queries, smaller number of attributes projected by the queries, and larger number of conjuncts in the queries. Our approach would be most useful in detecting subtle inference attacks.

Most existing approaches to inference detection make the worst case assumptions and put more restrictions on the uses of the database. They focus more on making the access control mechanism complete rather than sound. On the other hand, our approach detects if an inference could possibly occur, and hence our system allows the maximum accessibility of the database to the users.

This dissertation is organized as follows. In Chapter Two, we describe previous work in inference detection and elimination. In Chapter Three, we introduce our rule-based approach to data level inference detection. We illustrate the effectiveness of our inference detection system by using the inference rules to detect the well known TRACKER inference attacks on database systems. In Chapter Four, we describe our prototype of the inference detection system, and present the experimental results. The results show how the characteristics of databases and queries affect the system performance. In Chapter Five, we explore issues in developing a realistic inference detection system. We discuss the detection of approximate inference and dynamic inference, and the uses of multiple tables in the database and nested queries. In Chapter Six, we discuss possible future work and provide a conclusion.

Chapter 2

Background

In general, users perform inference in two stages: 1) data collection, and 2) reasoning with the collected data. The data being collected include data in the database that are accessible to users, integrity constraints that hold in the database (including the functional dependencies among attributes), and real-world knowledge (which is not represented in the database) on the application domains. The data collection and the reasoning stages are performed repeatedly by the adversary until the intended inference is achieved or he gives up. The data that adversaries want to infer include the existence of certain item in the database, or the associations among data. For example, if a user wants to infer if a project is about developing some nuclear weapons (perhaps an association not explicitly represented in the database), the user might try to find out if there exists a team member who is a nuclear weapons expert. Also, if a user wants to find out the salary S of an employee E, the user not only needs to find out that there exists such an employee E and a certain salary amount S in the database, but also needs to find out that the salary S is associated with the employee E. In some cases, instead of inferring the exact data values in the database (*precise inference*), users may be content with a set of possible data values (*imprecise inference* or *approximate inference*). For example, when a user can infer that an employee earns either 60K or 61K, the user has practically inferred the employee's salary.

To tackle the inference problem, we should explicitly represent all information available to users, and mimic the reasoning strategies of users. These are major challenges to the inference problem. For more than a decade, researchers have proposed numerous approaches, however, up to the present, there is no single approach that can completely prevent the occurrence of inference without seriously sacrificing the accessibility of the

database systems. Note that one simple way to block any inference in a multilevel database system is to classify all data at the highest classification levels. In this case, obviously, no inference occurs, but the database becomes completely inaccessible to users with low clearance levels.

In the following sections, we review the characterizations of inference and existing approaches to detect and eliminate inference. We illustrate the inadequacy of existing approaches and, hence, demonstrate the need for a more sophisticated approach to the problem as provided by our data level inference detection system.

2.1 Characterizations of Inference

There are two types of inference: *static* and *dynamic* [Den86, Hin89]. Static inference is performed with respect to a snapshot of a database, that is, the approach assumes there is no change to the database while the inference is performed. Dynamic inference is performed when there are changes to the database. The objects to be changed include data stored in the database (by the update, insert, and delete operations), classification levels of data (for example, due to sanitization), and the database schema. The changes may be initiated by adversaries so as to facilitate inference, or they may result from normal operations on the database system. It is possible for a user to perform inference by observing the changes to the database. For example, when a new employee arrives, his/her data are added into various tables in the database. By querying the database before and after the new employee's records are added, a user can determine the set of newly added records that are accessible to the user. The dynamic inference problem is more difficult to handle than the static inference problem as we need to consider the effects of update operations. Most research efforts to date focus on tackling the static inference problem. In Chapter 5, we explore the issues of dynamic inference by discussing the effects of update operations on our data level inference detection system.

There have been three reasoning strategies in inference: deductive, abductive, and approximate reasoning strategies [GLS91, GL92]. Deductive reasoning uses *modus ponens* to deduce information. Given that a statement $S1$ is true, and the rule *'if the statement S1 is true, then the statement S2 is also true'*, it is concluded that the statement $S2$ is true. The challenge in deductive reasoning is the development of the complete set of rules for inference.

In abductive reasoning, users infer the possible causes of an observation. Given that the statement $S2$ is true, and the rule *'if the statement S1 is true, then the statement S2 is also true'*, the user may infer that the statement $S1$ is true. When there are more than one causes to the observation, the user might need to estimate the probabilities of occurrences of the causes given that the observation has occurred. The user could then conclude that the most plausible cause occurs, or conclude that one of the causes has occurred.

In approximate reasoning, probabilities are involved in inference. Morgenstern provides a fundamental framework for studying the logical inference problem [Mor88]. He quantifies inference by defining an inference function called *INFER* in an information theoretic sense. $INFER(x \rightarrow y)$ is a value between 0 and 1. The value denotes the reduction in uncertainty about y when x is known. This function is used to construct a sphere of inference which represents the set of data that can be inferred from another set of data. This framework sparked a decade of research in the inference problem.

Buczkowski developed a Database Inference Controller [Buc90]. He uses a probabilistic model to determine the probabilities of events. An inference network is used to represent the logical dependencies among classified parameters. The probabilities are then propagated along the inference network. If events A and B are independent, then the probability of *(A OR B)* is $(P(A) + P(B) \Leftrightarrow P(AB))$, where $P(X)$ is the probability that event X occurs. If the two events are dependent, then *P(A OR B)* becomes $MAX(P(A), P(B))$. This is a very simple probabilistic model. This model provides a way to derive estimated values about the probabilities that sensitive data are leaked. However, its accuracy is bound by the accuracy of the estimations of the probabilities of individual events.

Chang *et al.* apply the Bayesian methods to the inference problem [CM98]. The data to be inferred are treated as missing data in the database. Inference is then made based on data accessible to users. It assumes that the missing data follow a distribution as indicated by the accessible data. When there is more then one value to be inferred, the Bayesian Network approach is employed. This is proposed as a new approach to the inference problem. The complexity of the solution still remains to be analyzed, especially when the number of missing data in the database and the size of the data domains are large.

Most research efforts to date use the deductive approach. Not much effort has been done using the abductive approach because it is difficult to enumerate the complete set of causes for each observation. There are a handful of papers on the approximate reasoning approach. The difficulties are on the proper assignment of probabilities to events, and the

reasoning with probabilistic events. Also, it is difficult to assess the levels of confidence of the conclusions. In the following section, we discuss the uses of the deductive approach in inference detection and elimination.

2.2 Inference Detection

Most existing work in inference detection focuses on how to detect inference in a multilevel relational database systems. Hinke pioneered the use of a graph to represent the associations among attributes in a relational database [Hin88]. Each attribute in the database is represented by a node in the graph. A directed edge is added in the graph from a node representing attribute A_1 to another node representing A_2 if there is a one-to-one or one-to-many association between A_1 and A_2. An inference is said to occur when there are two paths going from one attribute to another in the graph, and these paths are classified at different classification levels. The classification level of a path is the least upper bound of the classification levels of attributes corresponding to the nodes on the path.

Binns constructs an inference path by joining tables that have common attributes [Bin93b]. He quantifies the potential of having an inference by estimating the probability that a value of the attribute at one end of an inference path can identify the value of the attribute at the other end of the path. He argues that the longer the inference path, the less likely users can recognize the inference path. By doing so, he attempts to reduce the complexity of the inference detection problem at the expense of generating false negative reports.

DISSECT is an inference detection tool developed at SRI. Inference paths are constructed based on relationships involving primary keys and foreign keys in the database [QSK+93, GLQS93]. They only detect inference paths from attribute X to attribute Y when an attribute value of X uniquely identifies a single attribute value of Y. Hence, their approach is more restrictive than that proposed by Hinke [Hin88]. Burns describes an integration of DISSECT and the Software Through Pictures (StP) that has a graphical interface for detecting inference [Bur95].

Hinke *et al.* take into consideration the cardinalities of associations to detect inference [HDW97]. Instead of simply using functional dependencies among the attributes to identify inference paths, they suggest using associations with low cardinality. An association between two attributes X and Y has a low cardinality if given an attribute value of X, there

are only a few possible corresponding Y attribute values. This method can be used to infer a set of possible inferred values.

As discussed in Chapter 1, schema-based inference detection systems are not complete. Such systems are not sound either; that is, they may produce false positive reports. This is because the existence of an inference path in the database schema does not necessarily imply the path also exists in the data of the database. For example, consider the following database for dental care records,

PJ: (Part-time-ID, Job)
JD: (Job, Dental-care-allowance)

Table PJ stores the identification number of the part-time employees and their job titles. Table JD stores job titles and the corresponding dental care allowances. Suppose attribute Part-time-ID functionally determines attribute Job which in turn functionally determines attribute Dental-care-allowance. Using the schema-based inference detection approach, there is an inference path going from attribute Part-time-ID to attribute Job, and then to attribute Dental-care-allowance. However, suppose part-time employees are not entitled to have dental care allowances. In this case, the inference path does not exist in the database. Measures that block this 'inference path' (for example, by restricting accesses to table JD) may result in making the database less accessible to users. A way to detect such false inference path is to issue queries to determine if the inference path exists in the database [Hin88, Bin93b]. The queries can be issued periodically or when there are changes to the database. However, this method is effective only if all the inference paths are detected in the first place.

The inference problem becomes more difficult to handle when users employ real-world knowledge to perform inference. For example, the fact that a flight carries bombs is sensitive. Suppose a user knows that a flight carries some cargos that weight more than 10 tons. With the real-world knowledge that cargos that weight more than 10 tons are probably bombs, the user can infer that the flight carries bombs. In order to protect the fact that the flight carries bombs, we should also protect the weight of the cargos on the flight. The challenge in handling real-world knowledge is the representation of all relevant information, and determining how this information relates to data in the database.

Thuraisingham uses a multilevel semantic net to represent a multilevel database schema [Thu92]. Two semantic associations are supported: *ISA* (membership relationships)

and *AKO* (set-subset relationships). Rules are developed to assign classification levels to objects in the semantic net. An inference occurs when the classifications of data violate the constraints as specified in the semantic net.

Hinke *et al.* developed an inference analysis tool that factors domain knowledge into the inference detection system [HDC94, HDW95, DH96]. They group inference relevant information into three layers: entity layer, activity layer, and the entity-activity relationship layer. Each layer is further subdivided into facets (or associations) to represent knowledge on that layer. Example facets include *is-a*, *part-of* , *temporal*, and *used-for* facets. The inference detection tool searches for inference paths through the associations on the same layer or across layers. Such a scheme is useful only if all the relevant knowledge is explicitly represented, and the reasoning strategies that users employ to process the knowledge are identified.

Another approach to deal with inference using real-world knowledge is to represent the knowledge as a part of the database called *catalytic relations* [Hin88, HS97]. The inference detection system is then run against the database together with the catalytic relations. Hale *et al.* employ imprecise and fuzzy relations to form the catalytic relations [HS97]. The effectiveness of such approach depends on the abilities of capturing all relevant real-world knowledge into the catalytic relations.

2.3 Inference Elimination

There are four inference elimination approaches reported in literature: upgrading attributes, withholding query results, polyinstantiation, and schema redesign. There are other inference elimination methods that are suitable for statistical database systems, including data perturbation, cell suppression, and random sample queries [AW89]. In statistical databases, users can only query statistics about the database. In this dissertation, we discuss the inference problem in general purpose database systems. Hence, we do not further discuss the inference elimination techniques that apply to statistical databases only.

Su and Ozsoyoglu point out that users can make use of the functional dependencies among attributes to perform inference [SO87, SO91]. They have developed algorithms that remove inference by upgrading attributes involved in functional dependencies and multivalued functional dependencies. In their solution, for any functional dependency where X functionally determines Y, the resulting classification level of X dominates or equals

that of Y. They prove that the problem of finding the minimum number of changes to classification levels of attributes is an NP-complete problem on the number of attributes in the database.

However, it is not necessary to maintain such relationships among the classification levels of attributes in all functional dependencies. For example, consider the inference path where X functionally determines Y, and Y functionally determines Z. In Su and Ozsoyoglu's solution, the classification level of X dominates or equals that of Y which in turn dominates or equals that of Z. Yet, if we only want to prohibit the user from inferring attribute values of Z, we can simply classify Y at a level that dominates or equals that of Z, and X can be assigned with any classification level. Su and Ozsoyoglu's conservative attribute upgrading scheme results in overclassifying attributes. Attributes are overclassified if they are classified at levels that are higher than necessary to eliminate inference, making the database less accessible to users.

Stickel tackles the same upgrading problem [Sti94]. He formulates the problem as a set of formulas that specify the constraints on classifying the attributes. He then uses the Davis-Putnam theorem-proving procedure to obtain an optimal solution for the classification problem. He has not addressed the performance issues of this system.

The LOCK Data View project eliminates inference by rejecting suspicious user queries [HOST90]. The system maintains a history file of previously answered user queries. Each new user query issued to the database is checked with the previously answered user queries to determine if they together reveal sensitive information. Violating requests are simply denied. Lunt [Lun89] comments that it is difficult to assure the correctness of the access control mechanism in such a scheme as it is needed to be included in the Trusted Computing Base.

Another way to eliminate inference is to use polyinstantiation (or cover stories) [DLS$^+$87, DLS$^+$88, JS90]. In a polyinstantiated database, each classification level has its own view of the database. It is possible to have data associate with one classification level different from the corresponding data at different classification level. For example, if knowing the expertise of an employee is in nuclear Physics leads to inferring classified information, then the system can eliminate inference by telling unclassified users that the employee's expertise is in, say, Computer Science. Chen and Sandhu have developed a model to handle update operations in polyinstantiated database [CS95]. Binns [Bin93a] noted that polyinstantiation should be done with care, otherwise the inference problem can

still exist. For example, the second tuple in the following database is added to cover up the classified information about John's salary. However, if there is another table that reveals the salary of legal consultant to be 80K, then a user can still infer John's salary.

Name	Job	Salary	Tuple Classification Level
John	legal consultant	80K	Classified
John	legal consultant	45K	Unclassified

Lunt points out that some inference problems can be solved by schema redesign, instead of developing any special access control mechanism [Lun89]. Suppose we need to classify the salaries of job titles while the individual attribute values of attributes Job and Salary are unclassified. Lunt suggests creating the following three tables,

(Job_code, Salary_code) [C],
(Job, Job_code) [U], and
(Salary, Salary_code) [U].

Attributes Job_code and Salary_code are unique identifications of attribute values of attribute Job and Salary respectively. The first table stores data about the associations between attribute Job and Salary. It is classified. The second and third table store data about individual attribute values of Job and Salary, and they are unclassified. Hence, unclassified users can still access the individual attribute values of Job and Salary, but they cannot access the associations between them.

However, modifying the database is an expensive operation. First, we need to adjust application programs that access the modified database. Second, creating separate tables to store protected associations lead to duplication of data. This introduces update anomaly problems. Also, extra join operations are needed to retrieve the protected associations. Although the join operations can be done efficiently with appropriate index structures, maintaining the index structures is costly.

Blocking one inference path by upgrading some attributes might introduce new inference paths [SO87]. Binns notices that some of these new inference paths are false inferences [Bin94]. He suggests maintaining two classification levels for each attribute: the intended and the actual classification levels. Only the intended classification levels are used to detect inference.

There are also research efforts in developing a database system that is free of

inference. Lin defines a lattice model where the classification level of each view in the database is the least upper bound of the classification levels of its elements [Lin93]. He claims that when a database satisfies such model, there is no inference. However, he ignores the fact when two related items (for example, related by functional dependencies) are classified at different levels, inference can occur. A counter-example is shown in [Bin93b].

Qian identifies classes of integrity constraints that make a database free of both static and dynamic inference [Qia94]. These constraints are sufficient but not necessary conditions to have an inference free database. However, more research is needed to determine if there exists some other less tight integrity constraints that can also result in inference free guarantees, and at the same time make the database more accessible to users.

Marks considers queries in a specific form where the selection condition is a conjunction of "$A = a$", where A is an attribute, and a is an attribute value [Mar96]. He develops an efficient algorithm that determines if a query poses an inference threat. This is the case when the query projects attributes in a *protected view*, a view of the database whose data are to be protected. The detection system is designed to make the database completely free of inference. He takes a conservative approach, making the database system less accessible to users. The method is effective only if the complete set of protected views is specified. However, the cause of most inference problems is the failure to identify the complete set of data to be protected. Marks' method can be useful in eliminating inference once the inference paths are identified, and hence the protected views are defined.

2.4 A Summary of Existing Work

Most existing inference detection approaches rely on examining the database schema to discover inference paths. Other approaches attempt to extend the database schema with knowledge found in the application domain (as catalytic relations). The database schema provide a general picture of the data stored in the database. However, inference paths found in the database schema do not necessarily occur in an extension of the schema. Most approaches make the worst case assumption, and prohibit any accesses that might lead to inference. That is, users are penalized for being suspected of making inference. Although existing approaches try to make the system as complete as possible, there are inference paths that are not found in the schema but exist in an extension of the schema.

This dissertation describes our effort in developing a data level inference detection

system. We aim at determining if a user has collected sufficient data to perform inference. Not only does our system detect inferences that can be detected by previous approaches, it also detects inferences that could not be detected previously. In particular, our system can detect inferences that do not exist at the database schema level. Hence, conventional schema-based inference detection systems fail to identify such inferences. Our inference detection system is sound, and hence it allows the database to be more accessible to users. However, as our system needs to monitor every user query, it has a higher operational cost than schema-based detection scheme. We have developed a prototype of the system, and evaluated its performance. We show that for certain types of database and queries, our system can still be practical to be employed.

Chapter 3

The Design and Theoretical Basis of a Data Level Inference Detection System

In this chapter, we discuss the design of a data level inference detection system. We present the six inference rules that we have developed: 'split_query', 'subsume', 'unique characteristic', 'overlapping', 'complementary', and 'functional dependencies' inference rules. These rules are developed to mimic the reasoning strategies employed by a human. We provide examples to illustrate the rules. We also demonstrate the effectiveness of the inference rules by using them to detect a known database inference attack called *Tracker* [DDS79, DS80]. Any inference elimination method reported in the literature (including upgrading attributes, withholding query results, polyinstantiation, and schema redesign) can be used to eliminate inferences detected by our system. The inference elimination problem indeed is orthogonal to the detection problem, and we do not further discuss the inference elimination problem in this dissertation. Some results in this chapter have been reported in [YL98a].

3.1 Design Criteria

An inference detection system is evaluated according to the following criteria:

- soundness.

- completeness.
- accessibility of the database.
- efficiency.

An inference detection system is sound if it only reports inferences that exist. It is complete if it reports all inferences that exist. The degree of accessibility of a database is measured by the amount of data that are legitimately accessible to users. The more data a user can access from the database, the higher the accessibility of the database to the user. An unsound inference detection system leads to decrease in accessibility of the database. This is because queries that do not lead to any inference might be restricted, making the database less accessible to users. Similarly, the more complete the detection system, the lower the accessibility of the database if users are restricted to access the detected inference paths. For schema-based detection system, efficiency of the detection system may not be an issue, as the detection system is run once – at the database design time. For detection system that detect inference using user queries, efficiency can be an issue when the system needs to detect inference in real-time.

We note that the existence of an inference does not necessarily imply that the user is aware of it. An inference exists means that there is sufficient data available for a user to draw an inference. However, unless a user confesses that he/she has drawn the inference, it is impossible to determine if the user indeed has made use of the inference. For example, a user finds out the sum of the salaries of all employees is X_1, and the sum of the salaries of all employees excepts software engineers is X_2. Suppose there are n software engineers, and their salaries are about the same. The user can infer that the salary of a software engineer is about $\frac{X_1 - X_2}{n}$. A user may not be aware of this inference unless he/she intentionally does the calculation. Taking a conservative approach to inference detection, we assume the user is aware of any possible inference. We could track a user who has sufficient data to draw an inference about critical data to determine what he does with the inferred information.

Ideally, we should develop an inference detection system that is sound, complete, allows high accessibility of the database, and efficient. However, to date no such inference detection system exists. Schema-based inference detection systems tackle the problem by developing efficient detection algorithms. However, as shown in Chapter 2, such an approach is neither sound nor complete. Other researchers extend the database schema by including

background knowledge as a part of the database, for example, in the form of catalytic relations (a part of the database that represents external knowledge). This brings their system closer to a complete detection system. Marks attempts to develop a complete detection system [Mar96]. However, his approach generates numerous false positive reports as any query that accesses any part of a sensitive association is considered to be able to lead to inference.

The design of an inference detection system is a trade-off among soundness, completeness, accessibility of the database, and efficiency. If we want to have a secure system, then we should develop a complete detection system. If we aim at providing higher accessibility of data to users, we should develop a sound detection system. Efficiency is a desirable characteristic, and should be achieved whenever the soundness, completeness, or accessibility of the database is not sacrificed. In this chapter, we describe our data level inference detection system, which is sound and can detect more inferences than schema-based detection systems. The system can also be extended to include background knowledge to detect more inferences. Most existing inference detection systems are designed for multilevel database system. Our detection system is used in both multilevel and discretionary database systems.

3.2 The Policy

A database contains information about the existences of data values, and the associations among data values. Consider the following relational database about employee salaries,

Name	Salary
Tom	45K
Ann	60K

The existence of the four data values provides the following information,

- there is an employee named Tom
- there is an employee named Ann
- there is an employee who earns 45K
- there is an employee who earns 60K

With the semantics of the relational data model, data values appear in the same tuple belong to the same entity. Hence, we have two additional pieces of information about this database,

- Tom earns 45K
- Ann earns 60K

This illustrates that there are two types of information stored in a database: the data values, and the associations among the data values. Therefore, there are two aspects in protecting the confidentiality of data in a database: the protection of the existences of data values, and the protection of the associations among the data values.

The protection of the existences of data values can be done on a query-by-query basis. For example, if we want to protect the information about the fact that there is an employee who earns 45K, then we can simply reject any query that accesses the data value 45K. The protection of the associations is more subtle. Obviously, one should reject any query that directly accesses a protected association. However, it is insufficient, as users can make use of inference to infer associations.

The goal of our data level inference detection system is to detect if a user can use a series of queries to infer associations in the database. The associations to be protected are specified in a policy. A policy is a list of statements, each of which is a 3-tuple of the following form:

$$(\mathsf{U};\ A_1,\ \ldots,\ A_n;\ \mathsf{E})$$

U is the user that the policy statement applies to. $A_1, \ldots, A_n$, where $n > 1$, are the attributes of the associations to be protected. E is a logical expression that selects the tuples whose associations are to be protected. For example, consider a simple personnel database with the following schema,

$$(\mathsf{Name},\ \mathsf{Job},\ \mathsf{Salary})$$

.

If we want to detect if a user U can infer the salaries of managers, the policy is specified as follows:

$$(\mathsf{U};\ \mathsf{Salary},\ \mathsf{Job};\ \mathsf{Job} = \text{`manager'})$$

The policy to detect if a user U can infer the salary of a particular manager called Tom is specified as follows,

(U; Salary, Job; Job = 'manager' $\wedge$ Name = 'Tom')

A policy statement is similar to a query to the database. In fact, the policy defines a view on the associations to be protected. This is similar to view based access control systems [GW77, Qia96].

The policy can be extended to handle aggregation problems. For example, we can attach a threshold number, N, to a policy statement as follows.

(U; A_1, ..., A_n; E[N])

When the number of protected tuples that are revealed is greater than N, a policy violation occurs. For example, the following policy,

(U; Name, Phone_number; Department = 'Intelligence'[50])

states that the user U cannot know more than 50 phone numbers in the Intelligence Department. This is similar to the National Security Agency phone book problem where a user can learn about a few phone numbers in the phone book, but a significant number of phone numbers in the phone book.

The policy statement can also be applied in role-based system where accesses are determined based on the roles taken by users. In such a system, we replace the user field with a role field, and the policy statement is applied to users taking on that role. For example, this policy,

(bank-teller; Customer_id, Loan_info; true)

states that any user taking the role of a bank teller cannot access loan information of customers.

3.3 An Overview of the Data Level Inference Detection System

Figure 3.1 shows the overall system architecture of the data level inference detection system. The system can be deployed in real-time or post-mortem manner. In a real-time inference detection system, each user query and its return result are captured by the detection system. The system determines if the user query together with previously issued queries can access sensitive information as specified in the security policy. Any inference detected may result in real-time responses, for example, withholding return result from users. For a post-mortem inference detected system, a batch of user queries are input

to the detection system. It then determines if together they lead to any inference.

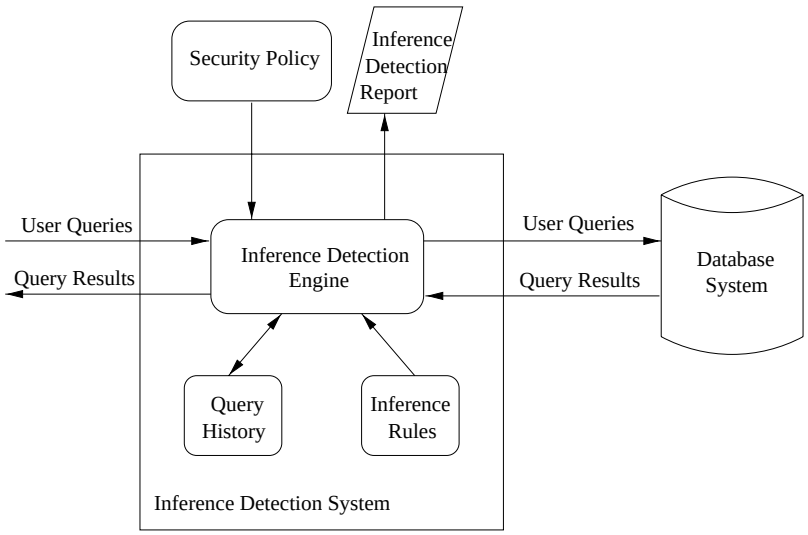

Figure 3.1: Overall System Architecture.

We have developed six inference rules that users can use to draw inferences. Each query result is a set of return tuples. Unless otherwise stated, a set of return tuples is indeed a multiset of return tuples. That is, duplicated return tuples are not removed. Each query and its set of return tuples allow users to learn about a part of the database. Users can learn more about the database by relating the return tuples from different queries. Consider the following database called personnel,

Name	Age	Salary	Job
Albert	28	50K	Receptionist
Betty	25	60K	Engineer
Calvin	35	70K	Engineer

Suppose a user issues the following SQL query,

```
select  Name, Age
from    Personnel
where   Age < 30;
```

which returns two tuples: (Albert, 28) and (Betty, 25). Users can learn that there are two employees who are younger than 30. One is Albert who is 28 and the other one is Betty who is 25. Suppose another query is issued as follows,

```
select  Age, Salary
from    Personnel
where   Age = 28;
```

which returns a single tuple: (28, 50K). By knowing that Albert is the only employee who is age 28 (from the first query), and that the employee who is 28 earns 50K (from the second query), users can infer that Albert earns 50K. We say that the tuple (Albert, 28) *relates* to the tuple (28, 50K), as the two tuples are projected from the same tuple in the database. If we want to protect the information about the associations between employee names and salaries, users should not be allowed to access the results of both queries.

In the remainder of this section, we provide the intuition behind five inference rules: split query, unique characteristic, subsume, overlapping, and complementary. The functional dependency inference rule is introduced to simulate the schema-based approach. We do not discuss it in this section. Each tuple in a database represents information about an *entity* in the application domain. Without background knowledge, users cannot identify the entities to which the return tuples of a query belong when 1) the query does not project the unique identifications, for example the primary key, of the selected entities, or 2) the query does not specifically select certain entities. For example, consider the following query to the above personnel database,

```
select   Job, Age
from     Personnel
where    Age > 24;
```

which returns three tuples ('Receptionist', 28), ('Engineer', 25), ('Engineer', 35). Without other information about the employees in the database, users cannot determine the employees that the return tuples belong to. However, if a return tuple contains some information that is known to be unique to an entity, users can make use of this information to draw inference. Continuing with the above example, if a user knows that there is only one receptionist in the company, then the user can infer that the receptionist is 28 year old. The unique characteristic inference rule handles this situation.

There are two possible relationships between two sets of return tuples. One possible relationship is that for each return tuple t_1 of a query, there is a return tuple t_2 of the other query, such that t_1 relates to t_2. The subsume inference rule handles this situation. For example, a user finds out that there are two consultants who work in the San Jose office of a company. Their information is shown as follows,

Name	Department
Bill	Production
Jack	Research

Also, the user finds out that there are three consultants who work in California, and their information is shown as follows,

Department	Salary
Production	80
Research	70
Research	60

By knowing that any employee who works in San Jose also works in California (San Jose is a city in California), the user can infer that each tuple in the first query relates to a tuple in the second query. In particular, as Bill's information can only be related to the first return tuple of the second query, the user can infer that Bill earns 80K.

The other possible relationships between two sets of return tuples is that only some return tuples of a query relate to some return tuples of another query. The split query and overlapping inference rules handle this situation by identifying the return tuples that are selected by both queries. For example, a user finds out the following information about a company,

1. John and Dan are the only two employees who are younger than 25.
2. The three employees who work in the Research Department earn 40K, 50K, or 60K.
3. There is only one employee who is younger than 25 and works in the Research Department.

The last piece of information implies that either John or Dan works in the Research Department, and hence either one of them earns 40K, 50K, or 60K.

If two sets of return tuples have a set-subset relationship between them, then by removing the same number of tuples from these two sets, the set-subset relationship still hold between them. The complementary inference rule handles this situation. For example, a user issues four queries and finds out the following information,

1. John and Bill are the two consultants who work in San Jose

2. There are three consultants who work in California; one of them earns 50K, one of them earns 60K, and one of them earns 70K.

3. Bill and Cathy are the two consultants who are younger than 30 and work in either San Jose or Austin.

4. There are two consultants who are younger than 30 and work in California or Texas; one of them earns 60K, and the other one earns 65K.

The first two queries imply that John and Bill earn 50K, 60K, or 70K. The third and fourth query imply that Bill and Cathy earn 60K or 65K (note that Austin is a city in Texas). Since 60K is the only value that appears in both results of the second and fourth queries, the user can conclude that Bill earns 60K. The user can further infer from the first two queries that John earns either 50K or 70K.

Once the related return tuples between two queries are identified, a user can generate *inferred queries*. An inferred query is a query whose return tuples can be determined by users without directly issuing the query to the database. For example, the user can infer a new query with returns tuples that are selected by two queries, or infer a new query that returns tuples from one query but not from another query. The user can also combine several queries into a single query. Essentially, the inference rules identify the set-subset, intersection, difference, and union relationships among return tuples of queries.

When a user issues a query, the inference detection system compares it with previously queries issued by the user, and applies inference rules to them when appropriate. The results of an application of the inference rules include: 1) modifications of the existing queries, for example by combining two related return tuples; and 2) generations of new inferred queries. The changes may trigger further applications of other inference rules. Hence, the inference rules are applied repeatedly until there is no new inference. This is a terminating process as the number of inferences that can occur is bounded by the size of the database. When two users are suspected of cooperating in drawing inference, we can run the inference detection system against their combined set of queries. For example, two users who issue queries in an interleaving manner are suspicious in collaborating in performing inference.

3.4 Preliminaries

We consider inference detection in a relational database with a single table. A database with multiple tables can be transformed into a universal relation as suggested in [Mar96]. We further discuss the issues of detecting inference in multiple tables in Chapter 5. We assume that the only way users learn about the data in the database is by issuing queries to it. That is, users do not rely on real-world knowledge to draw inference. Such knowledge might be added to the database as 'catalytic relation' as suggested in [Hin88].

Given a relational table, A_i denotes an attribute in the table, and a_i denotes an attribute value from the domain of A_i. $t[A_i]$ denotes the attribute value of a single tuple t over the attribute A_i. A query is represented by a 2-tuple: *(attribute-set; selection-criterion)*, where *attribute-set* is the set of attributes projected by the query, and *selection-criterion* is the logical expression that is satisfied by each return tuple of the query. No aggregation function (for example, maximum and average) is allowed in the attribute-set. In general, Q_i refers to the query $(AS_i; SC_i)$. $|Q_i|$ denotes the number of return tuples of Q_i. $\{Q_i\}$ denotes the set of return tuples of Q_i. For each query Q_i, AS_i is expanded with an attribute A_i when '$A_i = a_i$' appears in SC_i as a conjunct. A query Q is a *'partial query'* if the user can determine $|Q|$, but not all return tuples of Q. '$\cap$', '$\cup$', and '$\setminus$' stand for the set intersection, union, and difference operation respectively.

We introduce several notions that are used throughout this dissertation.

Definition 2 *A tuple t over a set of attributes AS 'satisfies' a logical expression E if E is evaluated to true when each occurrence of A_i in E is instantiated with $t[A_i]$, for all A_i in AS. t 'contradicts' with E if E is evaluated to false.*

For example, the tuple (35, 60K) that is projected over the attributes Age and Salary satisfies $E = (Age > 30 \wedge Salary < 70K)$; while the tuple (25, 50K) projected over the same set of attributes contradicts with E. The tuple (45K, Manager) projected over the attributes Salary and Job neither satisfies nor contradicts E. This is because after the instantiation, E becomes (Salary $<$ 70K) whose truth value is undetermined.

Definition 3 *Given two queries, Q_1 and Q_2, we say that Q_1 is 'subsumed' by Q_2, denoted as $Q_1 \sqsubset Q_2$, if and only if*

1. $SC_1 \Rightarrow SC_2$; *or*

2. *for each tuple* t_1 *in* $\{Q_1\}$, t_1 *satisfies* SC_2.

where $\Rightarrow$ is the logical implication. '$\sqsubset$' is a reflexive, anti-symmetric, and transitive relation. A return tuple t_1 *'relates'* to another return tuple t_2 if the two tuples are selected from the same tuple in the database. Hence, $Q_1 \sqsubset Q_2$ implies that for each return tuple t_1 of Q_1, there is a return tuple t_2 of Q_2, such that t_1 relates to t_2.

When evaluating a logical implication, we need to consider the integrity constraints that hold in the database. Consider the following implication,

$$(\mathsf{age} > 18 \wedge \mathsf{age} < 35) \Rightarrow (\mathsf{age} > 20 \wedge \mathsf{age} < 50),$$

which is false. Suppose the youngest person in the database is 22 years old. By adding this constraint to both sides of the implication, it becomes,

$$((\mathsf{age} > 18 \wedge \mathsf{age} < 35) \wedge (\mathsf{age} \geq 22)) \Rightarrow ((\mathsf{age} > 20 \wedge \mathsf{age} < 50) \wedge (\mathsf{age} \geq 22)) \equiv$$
$$(\mathsf{age} \geq 22 \wedge \mathsf{age} < 35) \Rightarrow (\mathsf{age} \geq 22 \wedge \mathsf{age} < 50),$$

which is true. The user could issue queries to reveal integrity constraints. For example, if a query Q_i returns no tuple, then $\neg SC_i$ is an integrity constraint satisfied by each tuple in the database. For the simplicity of the presentation, we do not include integrity constraints in checking logical implications.

We introduce the notion of *'indistinguishable'* as follows.

Definition 4 *A return tuple* t_1 *of* Q_1 *is 'indistinguishable' from a return tuple* t_2 *of* Q_2 *if and only if*

1. *for all* A_i *in* $(AS_1 \cap AS_2)$, $t_1[A_i] = t_2[A_i]$;

2. t_1 *does not contradict with* SC_2; *and*

3. t_2 *does not contradict with* SC_1.

t_1 *is 'distinguishable' from* t_2 *if* t_1 *is not indistinguishable from* t_2.

Intuitively, t_1 is indistinguishable from t_2 if it is not possible to conclude that t_1 and t_2 are selected from two different tuples in the database. Two tuples that relate to each other are indistinguishable from each other, while two tuples that are indistinguishable from each other does not imply that they relate to each other.

The attributes that are equivalent in semantics should be treated as the same attributes. For example, if the relational table contains two attributes: 'Salary' and

'Monthly_Salary' which are semantically equivalent, then for each tuple t in the table, $t[Salary] = t[Monthly\_Salary]$. Also, users can make inference based on the algebraic relationships among attributes. For example, consider these three attributes: Total_salary, Basic_salary, and Commission. If for each tuple t, $t[Basic\_salary] + t[Commission] = t[Total\_salary]$, then when the user reveals the Basic_salary and the Commission of a tuple, the user can reveal the Total_salary of the tuple. In this case, we can develop an inference rule saying that whenever a user knows the attribute values of any two of the three attributes of a tuple, the user knows the attribute values of the other attribute of the tuple.

Our system detects inference of the "correct" instances of the data. Consider the following table,

Name	Job	Age	Salary
John	Engineer	29	60K
Paul	Engineer	31	60K

Suppose a user knows that John is an engineer, and that there is an engineer who is 31 years old and earns 60K. A naive user may conclude that John earns 60K, assuming that John's age is 31. Although the user correctly infers the salary of John, this is not the correct instance of the salary of John. In fact, when the user learns that John is indeed 29 years old, the user will revoke this inference. In our system, we assume users are skeptical and do not make such hasty inferences.

Name	Job	Age	Salary	Department	Office
Alice	Manager	35	60K	Marketing	2nd Floor
Bob	Secretary	35	45K	Marketing	2nd Floor
Charles	Secretary	40	40K	Production	1st Floor
Denise	Manager	45	65K	Sales	2nd Floor

Figure 3.2: A Sample database.

3.5 Inference Rules

In this section, we present the six inference rules. We illustrate the inference rules using the sample database as shown in Figure 3.2. This database contains data about the names, job titles, ages, salaries, departments, and office locations of four employees. Name is the primary key in the database. The security policy is as follows,

(*; Name, Salary; true)

That is, no one (except, of course, privileged users) is allowed to find out the association between attributes Name and Salary for any tuple in the database. Any query that retrieve data about both name and salary is rejected. However, users are still allowed to access the name and salary data if the users cannot infer the association among them. Unless otherwise stated, all queries appear in the inference rules are not partial queries.

3.5.1 Split Queries

In this section, we discuss the situation where a query can be split into two smaller inferred queries with respect to another query. A query Q_i can be split into two smaller queries when the user can identify the return tuples of Q_i that relate to some other query.

Inference Rule 1 (Split Queries) Given two queries Q_1 and Q_2. Express SC_2 in disjunctive normal form. If there exists a disjunct of SC_2 such that the set of attributes appear in the disjunct is a subset of AS_1, then generate two inferred queries: 1) $Q_{11} = (AS_1; SC_1 \wedge SC_2)$; and 2) $Q_{12} = (AS_1; SC_1 \wedge \neg SC_2)$. The return tuples of Q_{11} are the return tuples of Q_1 that also satisfy SC_2. The return tuples of Q_{12} are the return tuples of Q_1 that do not satisfy SC_2.

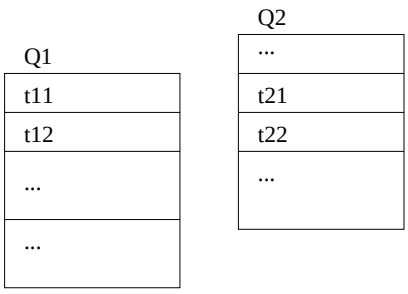

Figure 3.3: An example on splitting query.

When Q_1 projects attributes that appear in a conjunct of SC_2, a user is able to determine the return tuples of Q_1 that satisfy SC_2. Hence, the user is able to divide the return tuples of Q_1 into two sets: those that satisfy both SC_1 and SC_2, and those that satisfy SC_1 but not SC_2. Note that when no return tuple of Q_1 relates to the return tuples of Q_2, no inferred query is generated. Figure 3.3 illustrates the splitting of queries in a graphical form. Each rectangle represents a set of return tuples of a query. The rectangles are drawn in such a

way that return tuples that are selected from the same tuple in the database are aligned horizontally. For example, t_{11} and t_{21} correspond to the same tuple in the database, so do t_{12}, and t_{22}.

Example 1 Consider the following two queries.

Q_1 = (Name; Age = 35), and

Q_2 = (Age; Job = 'Secretary').

Q_1 returns two tuples (Alice), and (Bob). Q_2 returns two tuples (35), and (40). As the attribute Age in SC_1 is in AS_2, we generate two inferred queries from Q_2: 1) Q_{21} = *(Age; Job = 'Secretary'* $\wedge$ *Age = 35)* which has the return tuple (35); and 2) Q_{22} = *(Age; Job = 'Secretary'* $\wedge$ *Age* $\neq$ *35)* which has the return tuple (40). Note that $Q_{21} \sqsubset Q_1$. When the subsume inference rule in Section 3.5.2 is applied, the user can infer that the Secretary who is 35 years old is either Alice or Bob.

3.5.2 Subsume Inference

In this section, we describe inferences resulted from the '$\sqsubset$' relationships among queries.

Inference Rule 2 (Subsume) Given two queries Q_1 and Q_2, such that $Q_1 \sqsubset Q_2$.

SI1 If there is an attribute A in $(AS_2 \setminus AS_1)$, such that all return tuples of Q_2 take the same attribute value a over A, then for each return tuple t_1 of Q_1, $t_1[A] = a$. Q_1 may be a partial query.

SI2 If a return tuple t_1 of Q_1 is indistinguishable from exactly one return tuple t_2 of Q_2, then t_1 relates to t_2. Q_1 may be a partial query.

SI3 Let S be the set of return tuples of Q_2 that are distinguishable from the return tuples of Q_1. If $|S| = (|Q_2| \Leftrightarrow |Q_1|)$, generate two inferred queries from Q_2: 1) $Q_{21} = (AS_2;\ SC_2 \wedge \neg\ SC_1)$ with S as the set of return tuples; and 2) $Q_{22} = (AS_2;\ SC_2 \wedge SC_1)$ with $(\{Q_2\} \setminus \text{S})$ as the set of return tuples. If $|S| < (|Q_2| \Leftrightarrow |Q_1|)$, generate an inferred partial query: $Q_{23} = (AS_2;\ SC_2 \wedge \neg\ SC_1)$ with S as the partial set of return tuples, and $|Q_{23}| = (|Q_2| \Leftrightarrow |Q_1|)$.

Figure 3.4 illustrates the subsume inference rule in a graphical form.

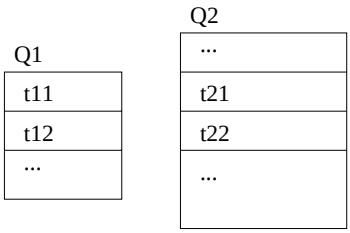

Figure 3.4: An example on subsume inference.

$Q_1 \sqsubset Q_2$ implies that for each return tuple t_1 of Q_1, there is a return tuple t_2 of Q_2 such that t_1 relates to t_2. *SI1* says that when all return tuples of Q_2 share a common attribute value, say a, over an attribute A, the user can infer that each return tuple of Q_1 also takes the attribute value a over the attribute A. This is because for each return tuple t_1 of Q_1, no matter which return tuple t_2 of Q_2 that relates to t_1, $t_2[A] = a$. Hence, $t_1[A]$ must be equal to a.

Example 2 Consider the following two queries.

$Q_1 = (\mathsf{Age};\ \mathsf{Name} = \text{'Alice'})$,

$Q_2 = (\mathsf{Department};\ \mathsf{Age} < 40)$.

Q_1 returns a single tuple (35) which says that Alice is 35 years old. Q_2 returns two tuples ('Marketing') and ('Marketing') which say that all employees at the age less than 40 work in the Marketing department. By *SI1*, Alice works in the Marketing department.

SI2 says that if a return tuple t_1 of Q_1 is indistinguishable from exactly one return tuple t_2 of Q_2, then t_1 relates to t_2. This is because $Q_1 \sqsubset Q_2$ implies that there is at least one return tuple of Q_2 that is indistinguishable from each return tuple of Q_2. Now, if t_1 of Q_1 is indistinguishable from one and only one tuple t_2 of Q_2, then we can conclude that t_1 relates to t_2.

Example 3 Consider the following two queries.

$Q_3 = (\mathsf{Age};\ \mathsf{Name} = \text{'Charles'})$,

$Q_4 = (\mathsf{Age},\ \mathsf{Salary};\ \mathsf{Age} \geq 40)$.

Q_3 returns a single tuple $t_3 = (40)$ which says that Charles is 40 years old. Q_4 returns two tuples (40, 40K) and (45, 65K) which say that there are only two employees who are at the age greater than or equal to 40. As $Q_3 \sqsubset Q_4$ (since (40) satisfies SC_4) and (40, 40K) is the only return tuple of Q_4 that is indistinguishable from t_3, by *SI2*, Charles earns 40K.

SI3 says that if a user identifies all the return tuples of Q_2 that relate to the return tuples of Q_1, then the user can infer these two queries from Q_2: *(AS_2; $SC_1 \wedge SC_2$)* which includes return tuples of Q_2 that relate to the return tuples of $Q1$, and *(AS_2; $SC_2 \wedge \neg SC_1$)* which includes return tuples of Q_2 that do not relate to the return tuples of Q_1.

Example 4 Continue from Example 3. After the application of *SI2*, the user can generate the following two inferred queries:

Q_{21} = (Age, Salary; Name = 'Charles' $\wedge$ Age $\geq$ 40),

Q_{22} = (Age, Salary; Name $\neq$ 'Charles' $\wedge$ Age $\geq$ 40).

Q_{21} returns a single tuple (40, 40K), and Q_{22} returns a single tuple (45, 65K). The two inferred queries together contains more information than Q_2. In particular, Q_{22} says that the employee who is at the age of 45 and earns 65K must be someone other than Charles.

3.5.3 Unique Characteristic Inference

A logical expression E is a unique characteristic of a tuple t if and only if t is the only tuple in the database that satisfies E. For example, if Alice is the only manager at the age of 35, then *(Job = 'Manager' $\wedge$ Age = 35)* is the unique characteristic of Alice in the database.

Inference Rule 3 (Unique Characteristic)

UC1 Given a tuple t_1 with unique characteristic C_1, and another tuple t_2 with unique characteristic C_2. If $C_1 \Rightarrow C_2$, $C_2 \Rightarrow C_1$, or $C_1 \Leftrightarrow C_2$ (that is $C_1 \Rightarrow C_2$ and $C_2 \Rightarrow C_1$), then t_1 relates to t_2.

UC2 Given there is a tuple with unique characteristic C. If both t_1 and t_2 satisfy C, then t_1 relates to t_2.

For example, the query

(Salary; Job = 'Manager' $\wedge$ Age $\leq$ 40)

returns a single tuple (60K). This query together with the above unique characteristic of Alice implies Alice earns 60K. *UC1* is a special case of the subsume inference. Suppose *(AS_1; UC_1)* returns a single tuple t_1, and *(AS_2; UC_2)* returns a single tuple t_2. Then, UC_1 is the unique characteristic of t_1, and UC_2 is the unique characteristic of t_2. If $UC_1 \Rightarrow UC_2$, $UC_2 \Rightarrow UC_1$, or $UC_1 \Leftrightarrow UC_2$ holds, then by *SI2*, t_1 relates to t_2.

Denote $sum(A; SC)$ as the sum of $t[A]$, for each tuple t that satisfies SC; and $sum(A; true)$ as the sum of $t[A]$, for each tuple t in the database. When all inferred queries are identified, unique characteristics are determined as follows.

1. if Q_i returns all but one tuple t in the database, then the unique characteristic of t is $(\neg SC_i)$. For each attribute $A \in AS_i$, if $sum(A, true)$ is known, then $t[A] = sum(A, true) - sum(A, SC_i)$.

2. if Q_i has only one return tuple t that relates to a return tuple of Q_j, then t has the unique characteristic $(SC_i \wedge SC_j)$.

3. if Q_i returns one more tuple t than Q_j, then the unique characteristic of t is $(SC_i \wedge \neg SC_j)$. For each attribute $A \in AS_i \cap AS_j$, $t[A] = sum(A; SC_i) - sum(A; SC_j)$.

where both Q_i and Q_j are not partial queries. When the tuple t is not identified, the user can still conclude that there exists a tuple that has the mentioned unique characteristic. This conclusion can be used in *UC2* to draw inference. Determination of the overlapping tuples among queries is discussed in the Section 3.5.4.

3.5.4 Overlapping Inference

In this section, we describe the overlapping inference rule.

Inference Rule 4 (Overlapping)

OI1 Given $Q_1 \sqsubset Q_2$, and $Q_1 \sqsubset Q_3$. Let S_2 be the set of return tuples of Q_2 that are indistinguishable from the return tuples of Q_3. If $|S_2| = |Q_1|$, and a return tuple t_2 of Q_2 is indistinguishable from exactly one return tuple t_3 of Q_3, then t_2 relates to t_3. Similarly, let S_3 be the set of return tuples of Q_3 that are indistinguishable from the return tuples of Q_2. If $|S_3| = |Q_1|$, and a return tuple t_3 of Q_3 is indistinguishable from exactly one tuple t_2 of Q_2, then t_3 relates to t_2. Note the related return tuples between Q_1 and Q_2, and between Q_1 and Q_3 can be identified using the subsume inference rule. Q_1 may be a partial query.

OI2 Given a query Q_1, and a set of queries, $QS = \{Q_2, \ldots, Q_n\}$, where $n \geq 3$. If 1) for each Q_i in QS, $Q_i \sqsubset Q_1$; and 2) the number of distinguishable tuples in $QS = |Q_1|$, then any pair of indistinguishable tuples in QS relate to each other.

OI3 When *OI1* is applied with $|S_2| = |Q_1|$, generate the following two inferred queries: 1) $Q_{21} = (AS_2; SC_2 \wedge \neg SC_3 \wedge \neg SC_1)$ with $\{Q_2\} \backslash S_2$ as the set of return tuples; and 2) $Q_{22} = (AS_2; SC_2 \wedge SC_3 \wedge SC_1)$ with S_2 as the set of return tuples. When *OI1* is applied with $|S_3| = |Q_1|$, generate the following two inferred queries: 1) $Q_{31} = (AS_3; SC_3 \wedge \neg SC_2 \wedge \neg SC_1)$ with $\{Q_3\} \backslash S_3$ as the set of return tuples; and 2) $Q_{32} = (AS_3; SC_3 \wedge SC_2 \wedge SC_1)$ with S_3 as the set of return tuples. If $|S_2|$ ¿ $|Q_1|$, generate this inferred partial query: $(AS_2; SC_2 \wedge \neg SC_3)$. If $|S_3|$ ¿ $|Q_1|$, generate this inferred partial query: $(AS_3; SC_3 \wedge \neg SC_2)$. Similarly, when *OI2* is applied, generate inferred queries for each pair of queries that have overlapping return tuples.

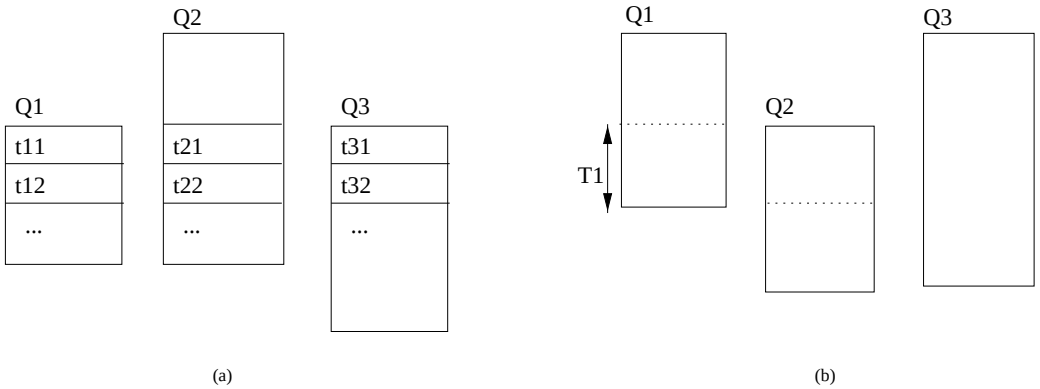

Figure 3.5: Examples on overlapping inference.

Figure 3.5(a) illustrates *OI1*. Given that $Q_1 \sqsubset Q_2$ and $Q_1 \sqsubset Q_3$, the number of return tuples of Q_2 that relate to return tuples of Q_3 must be at least $|Q_1|$. *OI1* identifies the cases where the user can infer the related return tuples among the three queries. When Q_1 implies three or more queries, *OI1* is applied to Q_1 and two of them at a time.

Example 5 We illustrate *OI1* using the following three queries,

Q_1 = (Name; Job = 'Manager' $\wedge$ Age = 35),

Q_2 = (Salary; Job = 'Manager'),

Q_3 = (Salary; Age = 35).

Q_1 returns a single tuple (Alice) which says that Alice is the only manager at the age of 35. Q_2 returns two tuples (60K) and (65K). Q_1 and Q_2 together implies that the salary of Alice is either 60K or 65K. Q_3 returns two tuples *(60K)* and *(45K)*. Q_1 and Q_3 together implies that the salary of Alice is either 60K or 45K. As $Q_1 \sqsubset Q_2$ and $Q_1 \sqsubset Q_3$, and there is only

one return tuple of Q_2 that is indistinguishable from the return tuples of Q_3, namely the tuple (60K). Hence, by *OI1*, Alice earns 60K.

Figure 3.5(b) illustrates *OI2* using three queries, Q_1, Q_2, and Q_3, where $Q_1 \sqsubset Q_3$ and $Q_2 \sqsubset Q_3$. T_1 is the set of return tuples of Q_1 that relate to returns tuples in Q_2. We show by contradiction the correctness of *OI2* for these three queries as follows. Let N be the number of distinguishable tuples in Q_1 and Q_2. That is, all return tuples of Q_1 and Q_2 are selected from N tuples in the database. As $Q_1 \sqsubset Q_3$ and $Q_2 \sqsubset Q_3$, each return tuple of Q_1 or Q_2 relates to a return tuple of Q_3. Hence, $N \leq |Q_3|$. When the number of distinguishable tuples in Q_1 and Q_2 equals $|Q_3|$, each distinguishable tuple of Q_1 and Q_2 relates to a return tuple of Q_3, and vice versa. Suppose there are two return tuples, t_1 of Q_1 and t_2 of Q_2, such that t_1 is indistinguishable from t_2, but t_1 does not relate to t_2. As t_1 does not relate to t_2, there must exist two return tuples of Q_3, say t_{31} and t_{32}, such that t_1 relates to t_{31} and t_2 relates to t_{32}. However, this contradicts with the above implication that each distinguishable tuple relates to one return tuple of Q_3. Therefore, all indistinguishable tuples relate to each other. That is, users can infer that for each return tuple t_1 of Q_1 that is indistinguishable from a return tuple t_2 of Q_2, t_1 relates to t_2.

Example 6 We illustrate *OI2* using the following three queries,

Q_1 = (Salary; Department = 'Marketing' $\wedge$ Office = '2nd Floor'),

Q_2 = (Salary; Job = 'Manager' $\wedge$ Office = '2nd Floor'),

Q_3 = (Name; Office = '2nd Floor').

Q_1 returns two tuples (60K) and (45K) which say that the two employees who work in the Marketing department on the 2nd floor earn either 60K or 45K. Q_2 returns two tuples (60K) and (65K) which say that the two managers who work on the 2nd floor earn either 60K or 65K. Q_3 returns three tuples (Alice), (Bob), and (Denise) which say that Alice, Bob and Denise all work on the 2nd Floor. We have 1) $Q_1 \sqsubset Q_3$; 2) $Q_2 \sqsubset Q_3$; and 3) there is only one return tuple of Q_1 that is indistinguishable from a return tuple of Q_2, namely the tuple (60K); that is, the number of indistinguishable tuples in both Q_1 and Q_2 is $3 = |Q_3|$. By *OI2*, the tuple (60K) of Q_1 relates to the tuple (60K) of Q_2. That is, the user can infer that the marketing manager who works on the 2nd floor earns 60K.

We show by construction that the two conditions in *OI2* are necessary. Without loss of generality, we consider three queries Q_1, Q_2, and Q_3. We list the two conditions as follows:

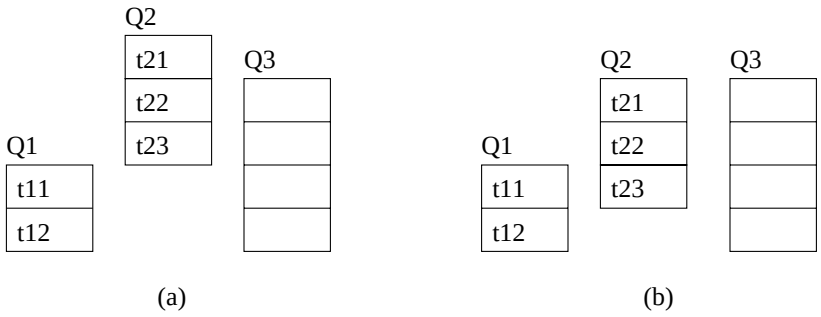

Figure 3.6: Counter examples on overlapping inference.

1. $Q_1 \sqsubset Q_3$ and $Q_2 \sqsubset Q_3$.

2. the number of distinguishable tuples in Q_1 and Q_2 equals $|Q_3|$.

Figure 3.6(a) shows an example where condition (1) does not hold, as $Q_2 \not\sqsubset Q_3$. Suppose tuple t_{11} is indistinguishable from tuple t_{21}, and the number of indistinguishable tuples in Q_1 and Q_2 is 4 which is equal to $|Q_3|$; that is, condition (2) holds. In this example, there is a pair of indistinguishable tuples that are not related to each other, namely t_{11} and t_{21}. Therefore, *OI2* cannot be applied.

Figure 3.6(b) shows an example where condition (1) holds. Suppose tuple t_{12} is indistinguishable from tuple t_{21}, and the number of distinguishable tuples is 3 which is less than $|Q_3|$; that is, condition (2) does not hold. *OI2* cannot be applied as t_{11} and t_{21} is a pair of indistinguishable tuples that are not related to each other.

3.5.5 Complementary Inference

The complementary inference rule allows a user to draw inference by eliminating tuples that do not relate to one another.

Inference Rule 5 (Complementary Inference) Given four queries, Q_1, Q_2, Q_3, and Q_4, where $Q_1 \sqsubset Q_2$, and $Q_3 \sqsubset Q_4$. Also, the return tuples of Q_1 that relate to the return tuples of Q_3 are identified (for example using the overlapping inference rule), and the return tuples of Q_2 that relate to the return tuples of Q_4 are identified. If one of the following three conditions holds,

1. for each return tuple t_1 of Q_1 that does not relate to any return tuple of Q_3, t_1 is distinguishable from all return tuples of Q_4,

2. $Q_4 \sqsubset Q_3$, or

3. $|Q_3| = |Q_4|$,

then $Q_1' \sqsubset Q_2'$, where $Q_1' = (AS_1;\ SC_1 \wedge \neg\ SC_3)$, and $Q_2' = (AS_2;\ SC_2 \wedge \neg\ SC_4)$. The return tuples of Q_1' is the set of return tuples of Q_1 that do not relate to any return tuple of Q_3, and the return tuples of Q_2' is the set of return tuples of Q_2 that do not relate to any return tuple of Q_4.

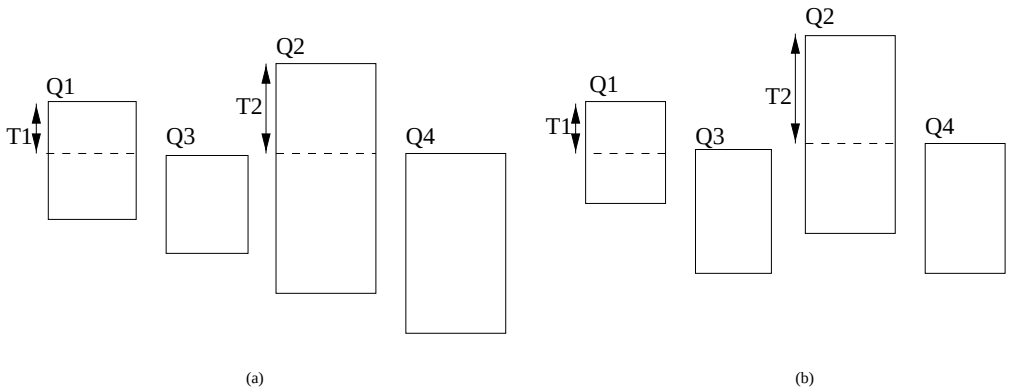

Figure 3.7: Examples on complementary inference.

Figure 3.7(a) illustrates the case where condition (1) holds. Let T_1 be the set of return tuples of Q_1 that do not relate to any return tuple of Q_3, and T_2 be the set of return tuples of Q_2 that do not relate to any return tuple of Q_4. As $Q_1 \sqsubset Q_2$ and $\{Q_1'\} \subset \{Q_1\}$, each return tuple of Q_1' relates to a return tuple of Q_2. Condition (1) says that each return tuple of Q_1' does not relate to any return tuple of Q_4. Hence, each return tuple of Q_1' relates to a return tuple of Q_2'. Figure 3.7(b) illustrates the case where condition (2) or (3) holds. Condition (2) or (3) implies that $Q_3 \sqsubset Q_4$ and $Q_4 \sqsubset Q_3$. By removing from Q_1 and Q_2 the "same" set of return tuples, we have $Q_1' \sqsubset Q_2'$.

Example 7 Consider the following four queries,

Q_1 = (Name; Department = 'Marketing'),

Q_2 = (Salary; Department = 'Marketing' $\vee$ Office = '2nd Floor'),

Q_3 = (Name; Job = 'Secretary'),

Q_4 = (Salary; Job = 'Secretary').

Q_1 returns two tuples (Alice) and (Bob). Q_2 returns three tuples (60K), (45K), and (65K). As $Q_1 \sqsubset Q_2$, both Alice and Bob earn either 60K, 45K, or 65K. Q_3 returns two tuples (Bob) and (Charles). Q_4 returns two tuples (45K) and (40K). As $SC_3 = SC_4$, both Bob and Charles earn either 45K or 40K. We have $Q1 \sqsubset Q2$, $Q_3 \sqsubset Q_4$, and $Q_4 \sqsubset Q_3$, (Bob) is the only related tuple between Q_1 and Q_3, (45K) is the only related tuple between Q_2 and Q_4 (it is the only indistinguishable tuple between Q_2 and Q_4). By the complementary inference rule, $Q'_1 \sqsubset Q'_2$, where

$Q'_1 =$ (Name; Department = 'Marketing' $\wedge$ Job $\neq$ 'Secretary')

$Q'_2 =$ (Salary; (Department = 'Marketing' $\vee$ Office = '2nd Floor') $\wedge$ Job $\neq$ 'Secretary')

Q'_1 returns a single tuple (Alice), as it is the tuple returned by Q_1 but not by Q_3. Q'_2 returns two tuples (60K) and (65K), as they are the tuples returned by Q_2 but not by Q_4. Therefore, the user can infer that Alice earns either 60K or 65K.

It should be noted that in some cases, an inference as obtained from the complementary inference rule can also be obtained from the overlapping inference rule. For example, consider the four queries Q_1, Q_2, Q_3, and Q_4 as shown in Figure 3.7(a), where $Q_1 \sqsubset Q_2$, and $Q_3 \sqsubset Q_4$. Suppose the overlapping inference rule can be applied to identify the related tuples between Q_1 and Q_3, and between Q_2 and Q_4. These result in the generation of two inferred queries: 1) $Q'_1 = (AS_1; SC_1 \wedge \neg SC_3)$; and 2) $Q'_2 = (AS_2; SC_2 \wedge \neg SC_4)$. If $SC_1 \wedge \neg SC_3 \Rightarrow SC_2 \wedge \neg SC_4$, then we have $Q'_1 \sqsubset Q'_2$ which is the same result as obtained by applying the complementary inference rule to the four queries. However, $SC_1 \Rightarrow SC_2$ and $SC_3 \Rightarrow SC_4$ does not necessarily imply $SC_1 \wedge \neg SC_3 \Rightarrow SC_2 \wedge \neg SC_4$. When this implication does not hold, the complementary inference rule is needed to obtain the inference.

3.5.6 Functional Dependency Inference

The functional dependency inference rule employs the functional dependencies among the attributes to draw inference. It simulates the uses of functional dependencies in schema-based inference detection systems. A similar rule can be constructed for multivalue functional dependencies.

Inference Rule 6 (Functional Dependency) Given that attribute A_1 functional determines attribute A_2, and there exists a tuple t, such that $t[A_1] = a_1$ and $t[A_2] = a_2$. If there is a tuple t_i, such that $t_i[A_1] = a_1$, then $t_i[A_2] = a_2$. The same applies when A_1 or A_2 is a composite attribute (that is, a group of attributes).

Example 8 If it is known that the attribute Department functionally determines the attribute Office, and in particular the Marketing department is located on the 2nd Floor. Then, whenever a user knows a person works in the Marketing department, the user knows the office of that person is located on the 2nd Floor.

3.6 Inference with Union Queries

In this section, we discuss the use of a union of queries in inference. Consider the following three queries,

$Q_1 = (\mathsf{Job};\ \mathsf{Age} < 50 \wedge \mathsf{Age} > 40)$,

$Q_2 = (\mathsf{Job};\ \mathsf{Age} > 45 \wedge \mathsf{Age} < 60)$, and

$Q_3 = (\mathsf{Job};\ \mathsf{Age} > 30 \wedge \mathsf{Age} \leq 45)$.

since the following implication holds,

$(\mathsf{Age} < 50 \wedge \mathsf{Age} > 40) \Rightarrow ((\mathsf{Age} > 45 \wedge \mathsf{Age} < 60) \vee (\mathsf{Age} > 30 \wedge \mathsf{Age} \leq 45))$,

$Q_1 \sqsubset (Q_2 \cup Q_3)$ holds. The inference rules can be applied by treating $(Q_2 \cup Q_3)$ as a single user query. We call such a union of queries a *'union query'*. A user query is called a *'simple query'*. If Q_u is a union query that consists Q_i, ..., and Q_j, then $AS_u = (AS_i \cap \ldots \cap AS_j)$, and $SC_u = (SC_i \vee \ldots \vee SC_j)$. Note that AS_u can equal $\emptyset$. Any overlapping tuple among the simple queries in a union query should have been identified. The applications of the unique characteristic and functional dependency inference rules on a union query are similar to their applications on the simple queries of the union query. Hereafter, we only discuss the applications of the subsume, overlapping, and complementary inference rules on union queries.

3.6.1 Subsume Inference Rule on Union Queries

Consider the applications of the subsume inference rule on union queries when the union queries are subsumed by other queries. Let $Q_u = \{Q_i, \ldots, Q_j\}$ be a union query, and $Q_u \sqsubset Q_1$. We show that inference obtained by applying the subsume inference rule on $(Q_i \cup \ldots \cup Q_j) \sqsubset Q_1$ can also be obtained by applying the subsume inference rule on $Q_i \sqsubset Q_1$, ..., and $Q_j \sqsubset Q_1$.

Consider the applications of *SI1*. If there is an attribute A in $(AS_1 \backslash AS_u)$, such that all return tuples of Q_1 take the same attribute value a over A, then for each return tuple t_u of Q_u, $t_u[A] = a$. This implies that for each return tuple t of a simple query of

Q_u, $t[A] = a$. This is the same as if *SI1* is applied to Q_i and Q_1, where $Q_i \sqsubset Q_1$, for each simple query Q_i of Q_u.

Consider the applications of *SI2*. If there exists a tuple t_u in Q_u that is indistinguishable from exactly one return tuple t_1 of Q_1, there exists at least one simple query Q_i of Q_u such that t_u relates to a return tuple t_i of Q_i. Now, t_i is indistinguishable from t_1 of Q_1. Hence, when *SI2* is applicable to infer that t_u of Q_u relates to t_1 of Q_1, it is also applicable to infer that t_i of Q_i relates to t_1 of Q_1.

Consider the applications of *SI3*. When all the related tuples between Q_u and Q_1 are identified, two inferred queries are generated from Q_1: 1) $Q_{u1} = (AS_1; SC_1 \wedge \neg SC_u)$; and 2) $Q_{u2} = (AS_1; SC_1 \wedge SC_u)$. We show that these two queries can also be generated from the simple queries of Q_u and Q_1. Note that when all the related tuples between Q_u and Q_1 have been identified, all related tuples among the simple queries of Q_u are also identified. Without loss of generality, suppose $Q_u = \{Q_2, Q_3\}$. The application of *SI3* on Q_1 and Q_2 generates two inferred queries: 1) $Q_{21} = (AS_1; SC_1 \wedge \neg SC_2)$; and 2) $Q_{22} = (AS_1; SC_1 \wedge SC_2)$. Similarly, the application of *SI3* on Q_1 and Q_3 generates two inferred queries: 1) $Q_{31} = (AS_1; SC_1 \wedge \neg SC_3)$; and 2) $Q_{32} = (AS_1; SC_1 \wedge SC_3)$. Now, Q_{21} and Q_{31} are both generated from Q_1, and we can generate the following inferred query for their related tuples: $(AS_1; SC_1 \wedge \neg SC_2 \wedge \neg SC_3)$ which equals Q_{u1}. Q_{22} and Q_{32} are both generated from Q_1, and we can identify the related tuple between them. The union of these two queries is $(AS_1; SC_1 \wedge (SC_2 \vee SC_3))$ which equals Q_{u2}. Therefore, we do not need to consider the applications of the subsume inference rule when the union query is subsumed by other queries.

Consider the case where union queries subsume other queries, say $Q_1 \sqsubset Q_u$. *SI1* is applied as follows. If for each return tuple t of any simple query of Q_u, $t[A] = a$, then $t_1[A] = a$ for each return tuple t_1 of Q_1. *SI2* is applied as follows. If there is a return tuple t_1 of Q_1 that is indistinguishable from a set of return tuples S from the simple queries of Q_u, where all tuples in S relate to one another, then t_1 relates to each tuple in S. *SI3* is applied similarly. Note that the subsume inference rule can still be applied when the simple queries of Q_u have no common projected attribute.

3.6.2 Overlapping and Complementary Inference Rule on Union Queries

Consider the applications of *OI1*. Given three queries, Q_1, Q_2, and Q_u, where Q_u is a union query. Suppose $Q_u \sqsubset Q_1$ and $Q_u \sqsubset Q_2$. If *OI1* is to be applied to identify the related return tuples among Q_2 and Q_3, $|Q_u|$ must be known. That is, the number of related tuples, if any, among the simple queries are identified. Suppose $Q_1 \sqsubset Q_u$ and $Q_1 \sqsubset Q_2$. If *OI1* is to be applied to identify the related return tuples between Q_u and Q_2, then the user must has already identified those related tuples among the simple queries in Q_u. Also, the user has to identify the return tuples of Q_u that are indistinguishable from the return tuples of Q_2, and the number of these return tuples equals $|Q_1|$.

Consider the applications of *OI2*. Suppose there is a set of queries $QS = \{Q_2, \ldots, Q_n, Q_u\}$ such that for each query $Q_i \in QS$, $Q_i \sqsubset Q_1$. *OI2* is applicable when the related tuples among the queries in QS are identified. That is, the related return tuples, if any, between Q_u and other queries in QS have to be identified. *OI3* is applied similar to the case with simple queries. Note that the overlapping inference rule can still be applied when $AS_u = \emptyset$. For example, let $Q_u = \{Q_{u1}, Q_{u2}\}$. If $SC_{u1} \wedge SC_{u2} = false$, the user can conclude that there is no related return tuple between Q_{u1} and Q_{u2}, and $|Q_u| = |Q_{u1}| + |Q_{u2}|$.

Consider the applications of the complementary inference rule on the union queries. Suppose there are four queries Q_1, Q_2, Q_3, and Q_u, where Q_u is a union query, $Q_1 \sqsubset Q_2$, and $Q_3 \sqsubset Q_u$. To apply the complementary inference rule on these four queries, the related return tuples among the simple queries in Q_u that also relate to return tuples of Q_2 must have been identified. Similarly for the case when Q_1, Q_2, or Q_3 is a union query.

3.7 Detection of Tracker Attacks

We illustrate the effectiveness of our detection system by detecting the *Tracker*, a known inference attack method that has driven research in the statistical database security community [DDS79, DS80]. As our system does not allow queries with statistical functions, we replace the statistical functions by general queries. Let $count(SC)$ be the function that counts the number of tuples satisfying the condition SC, and $sum(A; SC)$ be the sum over an attribute A for all tuples satisfying the condition SC. In our system, these two functions are transformed into the query *(A; SC)*. We consider three types of tracker attacks: individual, general and double tracker attacks.

3.7.1 Individual Tracker Attack

Suppose a tuple t is known to have a unique characteristic C. Let $C = C_1 \wedge C_2$, and $T = C_1 \wedge \neg C_2$. The following individual tracker can determine if the tuple t also has the unique characteristic D.

$$count(C \wedge D) = count(T \vee C_1 \wedge D) \Leftrightarrow count(T)$$

Note that '$\wedge$' takes precedence over '$\vee$'. If $count(C \wedge D) = 0$, there is no tuple that satisfies the condition $C \wedge D$, and hence the tuple t does not have the unique characteristic D. If $count(C \wedge D) = 1$, then there is one and only one tuple, namely t, that satisfies both C and D, and the tuple t also has a unique characteristic D. We transform the two count queries into the following two queries,

Q_1: $(AS_1; T \vee C_1 \wedge D)$ and

Q_2: $(AS_2; T)$.

where AS_1 and AS_2 are arbitrary sets of attributes. They do not necessarily have set-subset or intersection relationships. As $T \Rightarrow (T \vee C_1 \wedge D)$, we have $Q_2 \sqsubset Q_1$. Q_1 returns one more tuple than Q_2 which implies that there is a tuple, say t_1, with the unique characteristic $(SC_1 \wedge \neg SC_2)$. Now,

$$(T \vee C_1 \wedge D) \wedge \neg T$$
$$= C_1 \wedge D \wedge \neg T$$
$$= C_1 \wedge D \wedge \neg(C_1 \wedge \neg C_2)$$
$$= C_1 \wedge D \wedge C_2 = C \wedge D.$$

As $(C \wedge D) \Rightarrow C$, by the unique characteristic inference rule, t relates to t_1; that is, t has the unique characteristic $C \wedge D$. Hence, t also has the unique characteristic D.

The individual tracker can also employ sum functions to find out attribute values of individual tuples. The attribute value over A of the tuple t that has the unique characteristic C is found as follows,

$$sum(A; C) = sum(A; C1) \Leftrightarrow sum(A; C_1 \wedge \neg C_2)$$

The two sum functions are transformed into the following two queries,

Q_3: $(A; C_1)$ and

Q_4: $(A; C_1 \wedge \neg C_2)$.

As $C_1 \wedge C_2 \Rightarrow C_1$, we have $Q_4 \sqsubset Q_3$. $|Q_3| = |Q_4| + 1$ implies that there is a tuple returned by Q_3 but not by Q_4. Hence, there is a tuple with the unique characteristic $(SC_3 \wedge \neg SC_4)$ which equals $(C_1 \wedge C_2) = C$. That is Q_3 returns only one tuple more than those returned

by Q_4. Therefore, users are able to obtain the value of $t[A]$.

3.7.2 General Tracker

The general tracker can be used to determine the existence of a tuple with certain unique characteristic. Suppose a user wants to find out if there exists a tuple having the unique characteristic C. The user can do this by using three count functions as follows,

$count(C) = count(C \vee T) + count(C \vee \neg T) \Leftrightarrow count(true)$.

If $count(C) = 1$, then there is a tuple having the unique characteristic C. If $cond(C) = 0$, then no such tuple exists. The three count functions are transformed into the following general queries,

$Q_1 : (AS_1; C \vee T)$,

$Q_2 : (AS_2; C \vee \neg T)$, and

$Q_3 : (AS_3; true)$.

AS_1, AS_2, and AS_3 do not necessarily have any set-subset or intersection relationships. Q_3 selects all tuples in the database. Q_3 can be replaced by several smaller queries, for example, by these two queries: $(AS_3; E)$ and $(AS_3; \neg E)$ for any logical expression E. As $Q_1 \sqsubset Q_3$, $Q_2 \sqsubset Q_3$, $((C \vee T) \vee (C \vee \neg T)) \Leftrightarrow SC_3$, and $|Q_1| + |Q_2| = |Q_3| + 1$, a user can conclude that there is only one overlapping tuple between Q_1 and Q_2 and the tuple has the unique characteristic of $((C \vee T) \wedge (C \vee \neg T)) = C$.

Another form of general tracker is shown as follows,

$count(C) = 2count(true) \Leftrightarrow count(\neg C \vee T) \Leftrightarrow count(\neg C \vee \neg T)$.

If $count(C) = 1$, then there is a tuple with the unique characteristic C. If $count(C) = 0$, there is no tuple satisfies C. The three count functions are transformed as follows,

$Q_1 : (AS_1; \neg C \vee T)$,

$Q_2 : (AS_2; \neg C \vee \neg T)$, and

$Q_3 : (AS_3; true)$.

Suppose the tuple that satisfies C also satisfies T. In this case, Q_1 returns all tuples from the database, and Q_2 returns all but one tuples from the database. Hence, a user can infer that the tuple that is not returned by Q_2 has unique characteristic $\neg(SC_1 \wedge SC_2)$ $= \neg((\neg C \vee T) \wedge (\neg C \vee \neg T)) = \neg(\neg C) = C$. Similarly for the case where the tuple that satisfies C does not satisfy T. In this case, Q_1 returns all but one tuples from the database, and Q_2 returns all tuples from the database. A user can infer the existence of a unique

characteristic C.

3.7.3 Double Tracker

The general form of a double tracker is as follows, where T and U are logical expressions, and $T \Rightarrow U$,

$$count(C) = count(U) + count(C \vee T) \Leftrightarrow count(T) \Leftrightarrow count(\neg(C \wedge T) \wedge U)$$

If $count(C) = 1$, then there is a tuple, say t, having the unique characteristic C. If $count(C) = 0$, then there is no tuple satisfies C. Users can determine $t[A]$ using the following expression,

$$sum(A; C) = sum(A; U) + sum(A; C \vee T) \Leftrightarrow sum(A; T) \Leftrightarrow sum(A; \neg(C \wedge T) \wedge U)$$

The four count functions are transformed into the following four queries,

$Q_1 : (AS_1; U)$,

$Q_2 : (AS_2; C \vee T)$,

$Q_3 : (AS_3; T)$, and

$Q_4 : (AS_4; \neg(C \wedge T) \wedge U)$.

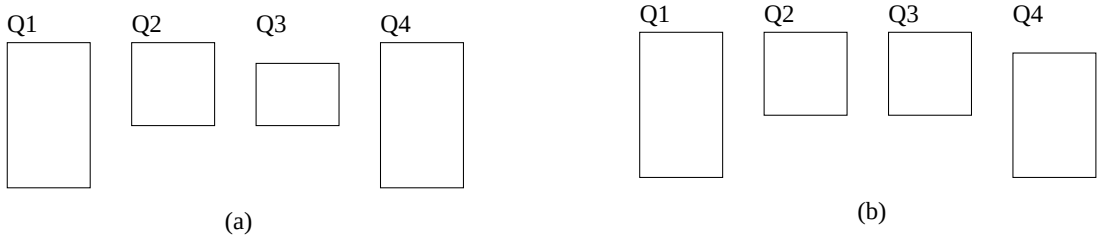

Figure 3.8: An Example on Double Tracker Attack.

Let t be the tuple that satisfies C. There are two cases to be considered: t satisfies T or t does not satisfy T. Suppose t satisfies T. Figure 3.8(a) shows four sample queries in this case. Q_2 and Q_3 return the same number of tuples. As $T \Rightarrow (C \vee T)$, a user can infer that any tuple satisfies C also satisfies T. As $T \Rightarrow U$, any tuple satisfies C also satisfies U. Now, $SC_4 \Rightarrow SC_1$, and Q_1 returns one more tuple than Q_4. This implies that there exists a tuple with the unique characteristic $SC_1 \wedge \neg SC_4 = U \wedge \neg(\neg(C \wedge T) \wedge U) = U \wedge T \wedge C$. As shown above, any tuple that satisfies C also satisfies both U and T. That is, $U \wedge T \wedge C = C$. Hence, there is a tuple with unique characteristic C.

Consider the case where t does not satisfy T. Figure 3.8(b) shows four sample queries in this case. Q_1 and Q_4 return the same number of tuples. This is because $(C \wedge T)$

becomes false, and hence $SC_4 = \neg(C \wedge T) \wedge U = U = SC_1$. $SC_4 \Rightarrow SC_1$ and $|Q_1| = |Q_4|$ imply that any tuple that satisfies U also satisfies $(\neg C \vee \neg T)$. As $T \Rightarrow U$, any tuple satisfies T also satisfies $(\neg C \vee \neg T)$. Obviously, a tuple that satisfies T does not satisfies $\neg T$. Hence, any tuple that satisfies T also satisfies $\neg C$. That is, any tuple satisfies C also satisfies $\neg T$. Now, $SC_3 \Rightarrow SC_2$, and Q_2 returns one tuple more than Q_3 which imply that there is a tuple with unique characteristic $((C \vee T) \wedge \neg T) = (C \wedge \neg T)$. As any tuple satisfies C also satisfies $\neg T$, $C \wedge \neg T = C$. This is, the user can conclude that there is a tuple with the unique characteristic C.

Another form of a double tracker is shown as follows,

$count(C) = count(\neg U) \Leftrightarrow count(\neg C \vee T) + count(T) + count(\neg(\neg C \wedge T) \wedge U)$

If $count(C) = 1$, then there is a tuple with the unique characteristic C. If $count(C) = 0$, no such tuple exists. The four count functions are transformed into the following four queries,

$Q_1 : (AS_1; \neg U)$,

$Q_2 : (AS_2; \neg C \vee T)$,

$Q_3 : (AS_3; T)$, and

$Q_4 : (AS_4; \neg(\neg C \wedge T) \wedge U)$.

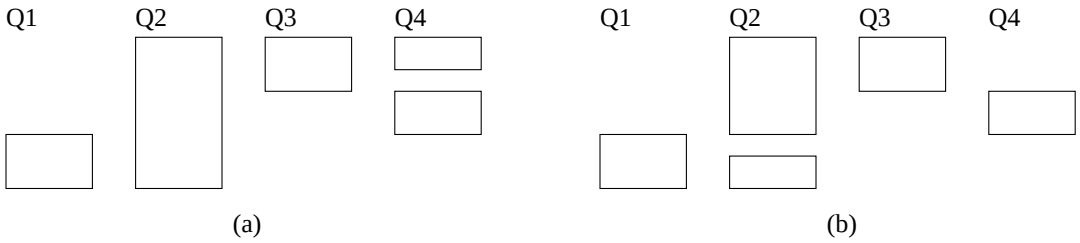

Figure 3.9: Another Example on Double Tracker Attack.

There are three cases to be considered: C logically implies T, C logically implies $\neg T$, and C does not logically imply T or $\neg T$. Suppose C logically implies T; that is, $C \Rightarrow T$. Figure 3.9(a) shows four sample queries in this case. That is, $\neg C \vee T = \mathit{true}$, hence Q_2 returns all tuples from the database. Now, $SC_1 \vee SC_3 \vee SC_4 = (\neg U) \vee T \vee (\neg(\neg C \wedge T) \wedge U)$ $= T \vee \neg U \vee \neg T \vee C = \mathit{true}$ implies that Q_1, Q_3, and Q_4 together returns all tuples from the database. $(SC_1 \wedge SC_4) = \neg U \wedge (\neg(\neg C \wedge T) \wedge U) = \mathit{false}$ implies that there is no return tuple of Q_1 that relates to a return tuple of Q_4. Similarly, $(SC_1 \wedge SC_3) = (\neg U \wedge T)$ $= \mathit{false}$ implies that there is no return tuple of Q_1 that relates to a return tuple of Q_3. $|Q_1| + |Q_3| + |Q_4| = |Q_2| + 1$ implies that Q_3 and Q_4 have one overlapping return tuple.

Hence, there is a tuple with the unique characteristic $(SC_3 \wedge SC_4) = T \wedge (\neg(\neg C \wedge T) \wedge U)$ $= T \wedge (C \vee \neg T) = T \wedge C = C$.

Suppose C logically implies $\neg T$; that is, $C \Rightarrow \neg T$. Figure 3.9(b) shows four sample queries in this case. $SC_1 \vee SC_3 \vee SC_4 = (\neg U \vee T \vee (\neg(\neg C \wedge T) \wedge U)) = \mathit{true}$ implies that Q_1, Q_3, and Q_4 together returns all tuples from the database. As $T \Rightarrow U$, $SC_3 \wedge SC_1 = T \wedge \neg U = \mathit{false}$, and hence no return tuple of Q_1 relates to a return tuple of Q_3. $SC_1 \wedge SC_4$ $= \neg U \wedge \neg(\neg C \wedge T) \wedge U = \mathit{false}$, hence no return tuple of Q_1 relates to a return tuple of Q_4. As $C \Rightarrow \neg T$, $SC_3 \wedge SC_4 = T \wedge (C \vee \neg T) \wedge U = T \wedge \neg T \wedge U = \mathit{false}$, and hence no return tuple of Q_3 relates to a return tuple of Q_4. Therefore, Q_1, Q_3, and Q_4 partition the database. As $|Q_2| = |Q_1| + |Q_3| + |Q_4|$ - 1, Q_2 returns all but one tuple. Hence, there is a tuple with the unique characteristic $\neg(\neg C \vee T) = (C \wedge \neg T) = C$.

Suppose C does not logically imply T or $\neg T$. $SC_1 \vee SC_3 \vee SC_4 = \mathit{true}$ implies that Q_1, Q_3, and Q_4 together returns all tuples from the database. There are two cases to be considered: Q_1, Q_3, and Q_4 partition the database, and they do not. First, consider the case where the three queries partition the database. As $|Q_2| + 1 = |Q_1| + |Q_3| + |Q_4|$, Q_2 returns all but one tuples from the database, and hence there is a tuple with the unique characteristic $\neg SC_2 = \neg(\neg C \vee T) = C \wedge \neg T$. Since the three queries partition the database, no tuple satisfies $SC_3 \wedge SC_4 = T \wedge (C \vee \neg T) \wedge U = T \wedge (C \vee \neg T) = T \wedge C$. That is, any tuple satisfies C does not satisfy T. Therefore, the unique characteristic $C \wedge \neg T$ is reduced to C. Now, consider the case where Q_1, Q_3, and Q_4 do not partition the database. That is, there is at least one overlapping return tuple among the three queries. Two queries have an overlapping return tuple if there is a return tuple of one query relates to a return tuple of another query. Let N be the number of tuples in the database. $N \geq |Q_2|$, and $N < |Q_1| + |Q_3| + |Q_4|$. As $|Q_2| + 1 = |Q_1| + |Q_3| + |Q_4|$, we have $N = |Q_2|$. Hence, the number of overlapping return tuple among the three queries is exactly one. $SC_1 \wedge SC_3 = \mathit{false}$, hence Q_1 and Q_3 do not have overlapping return tuple. $SC1 \wedge SC_4 = \mathit{false}$, hence Q_1 and Q_4 do not have overlapping return tuple. Therefore, Q_3 and Q_4 have one overlapping return tuple which satisfies $SC_3 \wedge SC4 = T \wedge (C \vee \neg T) \wedge U = T \wedge C$. As Q_2 returns all tuples from the database, any tuple that satisfies C also satisfies T. Therefore the unique characteristic $T \wedge C$ is reduced to C.

3.8 Summary

In this chapter, we present the six inference rules that we have developed based on a set-theoretic approach to the inference problem. They are the split query, subsume, unique characteristic, overlapping, complementary, and functional dependency inference rules. Each query returns a set of tuples. The possible relationships among them are set-subset, intersection, difference, and union. We develop the inference rules to simulate the use of these set relationships among query results to infer data. All the inference rules are sound. However, more research is needed to determine if they are complete. We can achieve completeness if we strictly limit the release of any part of sensitive information as suggested by Marks [Mar96], but at a cost of decreasing the accessibility of the database. We have shown that our inference rules can detect the known database inference attacks called the Tracker. This further assures the effectiveness of our detection system.

Chapter 4

Inference Detection Algorithms and Implementation

In this chapter, we present the inference detection algorithms, provide complexity analysis on them, and describe a prototype of the data level inference detection system. We carried out experiments to evaluate the performance of the inference detection system. Although in theory the detection algorithms are NP-complete, our experimental results show that the system could be practical for databases and user queries with certain characteristics. Some of the results in this chapter have been reported in [YL98b]

4.1 Data Structures

The detection algorithms maintain a data structure called *USER_VIEW(U)*. It represents the set of data that a user U learns about the database. USER_VIEW(U) is represented by a set of tables. Whenever a user U issues a query, the result of the query is added into USER_VIEW(U). Also, any data that the user U can infer are added into USER_VIEW(U) as well. We illustrate the construction of a USER_VIEW by an example. Consider the following sample database as used in Chapter 3.

Name	Job	Age	Salary	Department	Office
Alice	Manager	35	60K	Marketing	2nd Floor
Bob	Secretary	35	45K	Marketing	2nd Floor
Charles	Secretary	40	40K	Production	1st Floor
Denise	Manager	45	65K	Sales	2nd Floor

Suppose Bill issues the following query to the sample database,

Q_1: (Name, Job; Name = 'Alice')

which returns a single tuple ('Alice', 'Manager'). Assuming Bill has not accessed any data from the database before, USER_VIEW(Bill) contains a single table as follows,

Name	Job
Alice	Manager

which indicates that Bill only knows about the result of Q_1. Then, Bill issues the second query as follows,

Q_2: (Job, Salary: Job = 'Manager')

which returns two tuples ('Manager', 60K) and ('Manager', 65K). Bill does not know the employees to which these two tuples belong to, hence there is no inference among the first two queries. USER_VIEW(Bill) now contains the following two tables,

Name	Job
Alice	Manager

Job	Salary
Manager	60K
Manager	65K

Note that $SC_1 \Rightarrow SC_2$ (that is, SC_1 logically implies SC_2), and hence $Q_1 \sqsubset Q_2$. That is, Bill can tell that one of the two return tuples from Q_2 belongs to Alice. Bill issues the third query as follows,

Q_3: (Name, Job, Age: Name = 'Denise')

which returns this tuple ('Denise', 'Manager', 45). Again no inference occurs. USER_VIEW(Bill) is expanded with the third table as follows,

Name	Job
Alice	Manager

Job	Salary
Manager	60K
Manager	65K

Name	Job	Age
Denise	Manager	45

Also note that the return tuple of Q_3 satisfies SC_2, hence $Q_3 \sqsubset Q_2$. That is, one of the two return tuples of Q_2 belongs to Denise. Bill issues the fourth query,

Q_4: (Age, Salary: Age = 45)

which returns a single tuple (45, 65K). As there is only one return tuple, Bill can infer that

(Age = 45) is a unique characteristic of a tuple in the database. From Q_3, Bill knows that Denise is 45 years old, hence Bill can infer that the tuple (45, 65K) relates to the return tuple of Q_3. The tuple ('Denise', 'Manager', 45) is expanded with the tuple (45, 65K). USER_VIEW(Bill) is updated accordingly as follows,

Name	Job
Alice	Manager

Job	Salary
Manager	60K
Manager	65K

Name	Job	Age	Salary
Denise	Manager	45	65K

As $Q_3 \sqsubset Q_2$, and the tuple ('Denise', 'Manager', 45, 65K) is indistinguishable from exactly one return tuple of Q_2, namely ('Manager', 65K), the two tuples relate to each other. The tuple ('Manager', 65K) is then expanded with the tuple ('Denise', 'Manager', 45, 65K). After the expansion, the table that contains the single tuple ('Denise', 'Manager', 45, 65K) can be removed. USER_VIEW(Bill) becomes,

Name	Job
Alice	Manager

Name	Job	Age	Salary
	Manager		60K
Denise	Manager	45	65K

Also, as $Q_1 \sqsubset Q_2$ and the return tuple ('Alice', 'Manager') of Q_1 is indistinguishable from the return tuple ('Manager', 60K) of Q_2 only, these two tuples relate to each other. The tuple ('Manager', 60K) is expanded with the tuple ('Alice', 'Manager'). USER_VIEW(Bill) becomes,

Name	Job	Age	Salary
Alice	Manager		60K
Denise	Manager	45	65K

From the USER_VIEW(Bill), we can conclude that Bill can infer both Alice's and Denise's salaries.

We formally define a USER_VIEW(U) for a user U as follows:

- a USER_VIEW(U) is a set of tuples. Each tuple t is associated with a selection criterion SC, such that t satisfies SC.
- for each return tuple t of a query Q_i issued by the user U, t is a tuple in the

USER_VIEW(U). Set the selection criterion of t as SC_i.

- Given two tuples t_1 and t_2 in a USER_VIEW(U). If t_1 and t_2 relate to each other, then expand t_1 using t_2 and vice versa. Let the selection criterion of t_1 be SC_1 before the expansion, and that of t_2 be SC_2. After the expansion, the selection criteria of t_1 and t_2 are both $SC_1 \wedge SC_2$.

The USER_VIEW(U) is used to determine if the user U violates a policy. Recall in Chapter 3 that a policy is expressed as a 3-tuple as follows,

$$(\mathsf{U};\ A_1,\ \ldots,\ A_n;\ \mathsf{E})$$

which says that the user U is not allowed to access the association among attributes A_1, ..., and A_n for tuples satisfying the logical expression E. The user U violates this policy if there exists a tuple t_i in the $USER\_VIEW(U)$ with selection criterion SC_i, such that t_i satisfies E or $SC_i \Rightarrow E$, and t_i projects all the attributes A_1, ..., and A_n. For example, consider the following policy,

$$(\mathsf{Bill};\ \mathsf{Name},\ \mathsf{Salary};\ \mathsf{Job} = \text{'Manager'})$$

which says that Bill is not allowed to access salaries of managers. With respect to the USER_VIEW(Bill) after Q_4 is processed, both tuples in the USER_VIEW(Bill) satisfy the expression (Job = 'Manager'), and they both project the attributes Name and Salary. Therefore, we can conclude that Bill violates the policy.

As the inference rules are sound, any inference detected using the inference rules is correct and will not be revoked afterwards. In other words, any information added into a USER_VIEW is not removed. Hence, a USER_VIEW has a *monotonic* property which is stated as follows. The information content of a USER_VIEW(U) is monotonically increasing as the user U accesses the database and performs inference using the sound inference rules as presented in Chapter 3.

An implication of this monotonic property of the USER_VIEW is that if an inference rule is applicable at a state of the USER_VIEW, the rule is also applicable in successive states of the USER_VIEW. It is stated as follows. Given two states of a USER_VIEW, S_1 and S_2. If state S_2 is a superset of the state S_1, then any inference rule that is applicable at state S_1 is also applicable at state S_2.

We can treat an inference rule as a function that maps from a state of a USER_VIEW to another state of the USER_VIEW. That is,

$$R : S_1 \rightarrow S_2$$

where R is an inference rule, and S_1 and S_2 are states of a USER_VIEW. Sound inference rules have the following properties:

- idempotent: $R_1(R_1(S)) = R_1(S)$
- commutative: $R_2(R_1(S)) = R_1(R_2(S))$
- associative: $R_3(R_2(R_1(S))) = R_1(R_3(R_2(S)))$

where R_1, R_2, and R_3 are inference rules, and S is a state of a USER_VIEW. These properties simplify the design of the detection algorithms, as we do not need to maintain any order of applications of inference rules.

4.2 Inference Detection Algorithms

In this section, we present the inference detection algorithms, and provide complexity analysis for them. The basic operation in the complexity analysis is the comparison operation between two attribute values in the database. In the algorithms, query and inference results for a user U are added to USER_VIEW(U), as discussed in the above section. For simplicity, we omit the updating of the USER_VIEW(U) in the algorithms.

4.2.1 Function INFERENCE

Figure 4.1 shows the main function *INFERENCE(U, Q_i)*, which is called each time a user U issues a query Q_i to the database. The function maintains two data structures: GEN and EXP. GEN is initialized with the user issued query Q_i, and is subsequently added with inferred queries generated by the inference rules, at lines 16, 19-21, and 24-27. Note that the unique characteristic inference rule does not generate an inferred query. From lines 15 through 26, each query in GEN is compared with queries previously issued or inferred by user U (denoted as *PREV_QUERY(U)*) to determine if inference rules are applicable to them. EXP is the set of tuples that are expanded during the applications of the inference rules. After a tuple is expanded, the query that returns the expanded tuple might trigger further applications of inference rules. The query is identified at line 11, and is processed as other queries in GEN.

In each call to the INFERENCE function, all queries in GEN are processed before the expanded tuples in EXP. This avoids repeatedly processing the same tuple which is

```
INFERENCE (U, Q_i):
1.    initialize GEN with Q_i;
2.    EXP ← ∅;
3.    GEN_Q ← ∅;
4.    EXP_Q ← ∅;
5.    while (GEN ≠ ∅ or EXP ≠ ∅) do
6.       if GEN ≠ ∅ then
7.          Q_j ← a query in GEN;
8.          remove Q_j from GEN
9.          GEN_Q ← GEN_Q ∪ {Q_j};
10.      else if  EXP ≠ ∅ then
11.         Q_j ← a query that returns a tuple in EXP;
12.         EXP_Q ← EXP_Q ∪ {Q_j};
13.      ts ← return tuples of Q_j in EXP;
14.      remove return tuples of Q_j from EXP;
15.      for each Q_k ∈ PREV_QUERY(U) do
16.         GEN ← SPLIT_QUERY(Q_j, Q_k, GEN);
17.         if Q_j ⊏ Q_k then
18.            EXP ← UNIQUE(Q_j, Q_k, ts, EXP);
19.            (GEN, EXP) ← SUBSUME(Q_j, Q_k, GEN, EXP);
20.            (GEN, EXP) ← OVERLAP(U, Q_j, Q_k, GEN, EXP);
21.            GEN ← COMPLEMENTARY(U, Q_j, Q_k, GEN);
22.         else if Q_k ⊏ Q_j then
23.            EXP ← UNIQUE(Q_k, Q_j, ts, EXP);
24.            (GEN, EXP) ← SUBSUME(Q_k, Q_j, GEN, EXP);
25.            (GEN, EXP) ← OVERLAP(U, Q_k, Q_j, GEN, EXP);
26.            GEN ← COMPLEMENTARY(U, Q_k, Q_j, GEN);
27.   (GEN, EXP) ← UNION_QUERIES(U, GEN_Q, EXP_Q);
```

Figure 4.1: The function INFERENCE.

expanded when processing different queries in GEN. For example, there is a query Q_j in GEN, and a return tuple t_j of Q_j is in EXP. Suppose after Q_j is processed, t_j is expanded. If the system processes t_j in EXP before the Q_j in GEN, then t_j is inserted into EXP after the Q_j is processed. This means that the system needs to process the tuple t_j twice. On the other hand, if the Q_j in GEN is processed before t_j in EXP, then the system only needs to process the tuple t_j once. At the end of the function, the *UNION_QUERIES* function is called to handle inference involving union queries.

SPLIT_QUERY(Q_i, Q_j, GEN):
1. **if** Q_i and Q_j have no related return tuples **then**
2. **return** GEN;
3. **if** the attributes in a conjunct of SC_j is a subset of AS_i,
and Q_i is not a partial inferred query **then**
4. generate an inferred query $(AS_i; SC_i \wedge SC_j)$ for return tuples of Q_i that also relate to return tuples of Q_j;
5. generate an inferred query $(AS_i; SC_i \wedge \neg SC_j)$ for return tuples of Q_i that do not relate to return tuples of Q_j;
6. **else if** the attributes in a conjunct of SC_i is a subset of AS_j,
and Q_j is not a partial inferred query **then**
7. generate an inferred query $(AS_j; SC_j \wedge SC_i)$ for return tuples of Q_j that also relate to return tuples of Q_i;
8. generate an inferred query $(AS_j; SC_j \wedge \neg SC_i)$ for return tuples of Q_j that do not relate to return tuples of Q_i;
9. **return** $GEN \cup$ {newly generated inferred queries};

Figure 4.2: The function SPLIT_QUERY.

4.2.2 Function SPLIT_QUERY

Figure 4.2 shows the function *SPLIT_QUERY*. It splits a query Q_i if there exists another query Q_j such that some tuples from Q_i must also be returned by Q_j. This occurs when the set of attributes in a conjunct of SC_j is a subset of AS_i. This function only generates inferred queries for future applications of the inference rules. Note that the system does not generate an inferred query that is equivalent to a query in $PREV\_QUERY(U)$ to avoid duplication. Once an inferred query Q_1 is generated from another query Q_2, a link is maintained between the two queries. Whenever a tuple in Q_1 is expanded, the corresponding related tuple in Q_2 is also expanded. Let N be the number of tuples in the database, and A be the number of attributes in the database. Line 1 runs in $O(N)$ time by checking for each return tuple of Q_i if there is a related tuple returned by Q_j, assuming that it takes a single step to determine if a query returns a particular tuple. The generation of inferred queries, that is line 4, 5, 7, or 8, runs in $O(N)$ time. The running time for line 3 or 6 is $O(A)$, assuming it takes a single step to find out if an attribute appears in a selection criterion. Hence, the *SPLIT_QUERY* function runs in $O(A + N)$ time.

UNIQUE(Q_i, Q_j, ts, EXP)
1. **if** $|Q_i| = 1$ **then**
2. let t_i be the tuple returned by Q_i;
3. t_i has the unique characteristic of SC_i;
4. **else if** $|Q_i| = N \Leftrightarrow 1$, where N is the number of records in the database **then**
5. there is a tuple with unique characteristic of $\neg SC_i$;
6. **if** $|Q_j| = |Q_i| + 1$ **then**
7. there is a tuple with unique characteristic of $SC_j \wedge \neg SC_i$;
8. **for** each tuple $t_i \in ts$ **do**
9. **for** each unique characteristic, uc **do**
10. **if** t_i satisfies uc **then**
11. **for each** tuple t_j that has the unique characteristic uc **do**
12. **EXPAND**(t_i, t_j);
13. **return** $EXP \cup$ {newly expanded return tuples};

Figure 4.3: The unique function.

4.2.3 Function UNIQUE

Figure 4.3 shows the function *UNIQUE*. It checks if a user can identify unique characteristics in the database. Lines 1 through 3 check if the input query Q_i, which is either a user issued query or an inferred query, returns a single tuple; and lines 4 through 5 check if Q_i returns all but one tuple. If this is the case, the only returned tuple of Q_i or the only tuple not returned by Q_i has a unique characteristic. Lines 6 through 7 check if Q_j returns one more tuple than Q_i. As $Q_i \sqsubset Q_j$, the tuple that is returned by Q_j but not by Q_i has the unique characteristic $SC_j \wedge \neg SC_i$. Note that the existence of a tuple having a unique characteristic is useful in inference. For example, it is known that there exists a tuple t_1 that has the unique characteristic UC. If there is another tuple t_2 that also has the unique characteristic UC, then t_1 and t_2 relate to each other. Note that both Q_i and Q_j can be partial inferred queries.

The *UNIQUE* function is also input with a set of expanded tuples resulting from applications of inference rules, denoted as ts. If a tuple t in ts satisfies a unique characteristic UC, then t is expanded with any other tuple that also satisfies UC. The function *EXPAND*(t_1, t_2) performs the following operation, for each attribute A projected by t_2 but not by t_1, set $t_1[A]$ to be $t_2[A]$, and for each attribute $A\prime$ projected by t_1 but not by t_2, set $t_2[A\prime]$ to be $t_1[A\prime]$.

We provide the complexity analysis of the *UNIQUE* function. Lines 1 through 7 run in $O(1)$ time. ts is the set of tuples returned by a query, hence, the number of

tuples in ts is bound by N, where N is the number of tuples in the database. That is, the number of iterations at line 8 is $O(N)$. Lines 9 through 11 can be optimized as follows. Suppose the auditing system can identify the tuples that are related to each other. For each tuple t_i that is identified at line 8, we only consider its related tuple t_j which has a unique characteristic uc. If t_i satisfies uc, then expand t_i with t_j. Hence, the number of iterations runs from lines 9 through 11 is bound by the maximum number of unique characteristic of a tuple, which is 2^A, where A is the number of attributes in the database. This is because each unique characteristic consists a subset of the A attributes. The running time for the *EXPAND* function is $O(A)$, assuming it takes a single step to determine if a query projects a certain attribute. That is, the running time from lines 8 through 12 is $O(NA2^A)$. Hence, the complexity of the *UNIQUE* function is $O(NA2^A)$. Although in theory the function runs in exponential time of the number of attributes, in practice, there are a few unique characteristics identified for each tuple. This is supported by the results of the experiments that we have carried out. Let U be the maximum number of unique characteristic identified for a tuple in the database. The running time of the function is $O(NAU)$, and U is expected to be a small number.

4.2.4 Function SUBSUME

Figure 4.4 shows the function *SUBSUME*. Given that $Q_i \sqsubset Q_j$, this function performs two checks. First, it checks if all return tuples of Q_j project the same attribute value a_j over some attribute A_j. If this is the case, each return tuple t_i of Q_i is expanded with the attribute A_j and $t_i[A_j]$ is set to be a_j. Second, the function checks if a return tuple of Q_i is indistinguishable from only one return tuple of Q_j. If this is the case, the two return tuples are expanded with each other. The number of projected attributes is bound by A, hence the number of iterations at line 2 is bound by $O(A)$. Line 3 runs in $O(N)$ time, as the number of tuples of each query is bound by N. Line 4 runs in $O(N)$ time. Hence, the running time from lines 2 through 5 is $O(AN)$. The number of iterations at line 6 is $O(N)$. Line 7 runs in $O(NA)$ time. Line 8 runs in $O(A)$ time. Hence, lines 6 through 8 run in $O(N(NA+A)) = O(N^2A)$ time. Each inferred query is generated using $O(N)$ time. The running time from lines 9 through 13 is $O(N)$. Hence, the total running time of the *SUBSUME* function is $O(AN^2)$.

SUBSUME(Q_i, Q_j, GEN, EXP):
1. return if Q_j is a partial inferred query;
2. **for** each attribute A of AS_j **do**
3. **if** $t_j[A] = a$ for all return tuples t_j of Q_j **then**
4. **for each** return tuple t_i of Q_i **do**
5. $t_i[A] \leftarrow a$;
6. **for** each return tuple t_i of Q_i **do**
7. **if** t_i is indistinguishable from exactly one return tuple, t_j, of Q_j only **then**
8. **EXPAND**(t_i, t_j);
9. **if** all tuples in Q_j that relate to Q_i are identified **then**
10. generate an inferred query $(AS_j; SC_j \wedge SC_i)$ for tuples returned by both Q_j and Q_i;
11. generate an inferred query $(AS_j; SC_j \wedge \neg SC_i)$ for tuples returned by q_j but not Q_i;
12. **else**
13. generate a partial inferred query $(AS_j; SC_j \wedge \neg SC_i)$ for those return tuples of Q_j that are distinguishable from return tuples of Q_i;
14. **return** $GEN \cup$ {newly inferred queries}, $EXP \cup$ {newly expanded return tuples};

Figure 4.4: The function SUBSUME.

4.2.5 Function OVERLAP

Figure 4.5 shows the function *OVERLAP*. It checks if the overlapping inference rule is applicable. It consists two parts. The first part, from lines 1 through 19, detect inference using *OI1*. Given three queries, Q_i, Q_j, and Q_k, where $Q_i \sqsubset Q_j$ and $Q_i \sqsubset Q_k$, the function checks if the overlapping tuples between Q_j and Q_k can be identified. Lines 3 and 4 determine if the return tuples of Q_j that overlap with Q_k are identified. If this is the case, lines 5 and 6 expand the related tuples between Q_j and Q_k. Similarly, lines 7 and 8 determine if the return tuples of Q_k that overlap with Q_j are identified. If this is the case, lines 9 and 10 expand the related tuples between Q_j and Q_k. Lines 11 through 13 expand the return tuples of Q_i that relate to the overlapping return tuples of Q_j or Q_k. When $|Q_i|$ = 1, a user can infer that there is a tuple with the unique characteristic of $(SC_j \wedge SC_k)$.

The second part of the function *OVERLAP*, from lines 20 through 29, detects inference using *OI2*. That is, it finds a set of queries SQ, such that each query in SQ is subsumed by Q_j, and the number of distinguishable return tuples in SQ equals $|SQ|$. When such a set of queries is found, any indistinguishable tuples among queries in SQ relate to one another. When all the related tuples between two queries, Q_j and Q_k, in SQ are identified, at most four inferred queries can be generated: the set of return tuples of Q_j that are not

function **OVERLAP**(U, Q_i, Q_j, GEN, EXP):

1. return GEN, EXP if Q_j is a inferred partial query;
2. **for** each query $Q_k \in PREV\_QUERY(U)$, where $Q_i \sqsubset Q_k$, and Q_k is not a partial inferred query **do**
3. $S_j \leftarrow \{t_j$: $t_j \in \{Q_j\}$, there exits a t_k, $t_k \in \{Q_k\}$, t_j is indistinguishable from $t_k\}$;
4. **if** $|Q_i| = |S_j|$ **then**
5. **for** each return tuple t_j of Q_j that is indistinguishable from t_k of Q_k **do**
6. **EXPAND**(t_j, t_k);
7. $S_k \leftarrow \{t_k$: $t_k \in \{Q_k\}$, there exists a t_j, $t_j \in \{Q_j\}$, t_k is indistinguishable from $t_j\}$;
8. **if** $|Q_i| = |S_k|$ **then**
9. **for** each return tuple t_k of Q_k that is indistinguishable from t_j of Q_j **do**
10. **EXPAND**(t_k, t_j);
11. **if** $|S_j| = |S_k|$ **then**
12. **for** each return tuple t_i of Q_i that is indistinguishable from a return tuple t_l of Q_j or Q_k **do**
13. **EXPAND**(t_i, t_l);
14. **if** all related tuples between Q_j and Q_k are identified **then**
15. generate four inferred queries between Q_j and Q_k;
16. **else**
17. generate partial queries;
18. **if** $|Q_i| = 1$ **then**
19. there is a tuple with unique characteristic $(SC_j \wedge SC_k)$;
20. $SS \leftarrow$ **FIND_OVERLAP_SET(**U, Q_j**)**;
21. **for** each set of queries SQ in SS **do**
22. **if** t_i of a query in SQ is indistinguishable from a tuple t_k of another query SQ **then**
23. **EXPAND**(t_i, t_k);
24. **if** all indistinguishable tuples be between two queries are identified **then**
25. generate four inferred queries between them;
26. **else**
27. generate partial queries;
28. **if** Q_i has only one tuple indistinguishable with tuples of Q_k **then**
29. there is a tuple with unique characteristic $(SC_i \wedge SC_k)$;
30. **return** $GEN \cup$ {newly inferred queries}, $EXP \cup$ {newly expanded return tuples};

Figure 4.5: The function OVERLAP.

returned by Q_k, the set of return tuples of Q_j that are also returned by Q_k, the set of return tuples of Q_k that are not returned by Q_j, and the set of return tuples of Q_k that are also returned by Q_j.

function **FIND_OVERLAP_SET**(U, Q_i):
1. $S \leftarrow \emptyset$;
2. **for** each query $Q_j \in$ *PREV_QUERY(U)*, where $Q_j \sqsubset Q_i$, and Q_j is not a partial inferred query **do**
3. $S \leftarrow S \cup \{Q_j\}$;
4. return {} if there exists a return tuple t_i of Q_i such that no return tuple of queries of S relates to t_i;
5. return $\{S\}$ if the number of distinguishable return tuples in $S = |Q_i|$;
6. $SQ \leftarrow \emptyset$;
7. **for** each subset $S\prime$ of S **do**
8. **if** the number of distinguishable tuples in $S\prime = |Q_i|$ **then**
9. $SQ \leftarrow SQ \cup \{S\prime\}$;
10. return SQ;

Figure 4.6: The function FIND_OVERLAP_SET.

Figure 4.6 shows the function *FIND_OVERLAP_SET* which is called at line 20 of the function *OVERLAP*. *FIND_OVERLAP_SET* find all the set of queries such that *OI2* is applicable. Lines 1 through 3 find a set of queries S such that for each query Q in S, Q is subsumed by Q_i. When there is a return tuple of Q_i that does not relate to a return tuple of queries in S, then the number of distinguishable return tuples in S must less than $|Q_i|$. For this case, at line 4, the function returns an empty set to indicate that *OI2* cannot be applied. At line 5, when the number of distinguishable return tuples in S equals $|Q_i|$, the function returns the set S to indicate that *OI2* can be applied to queries in S and Q_i. Note that we do not need to apply *OI2* to queries in a subset $S\prime$ of S. This is because overlapping return tuples identified in any pair of queries in $S\prime$ can also be identified in S.

When the number of distinguishable return tuples in S is less than $|Q_i|$, *OI2* may be applicable to a subset of S. Lines 6 through 9 find out all these subsets of S. The algorithm is exponential in the number of queries in S. Figure 4.7 shows five queries, where $S = \{Q_2, Q_3, Q_4, Q_5\}$, and each query in S is subsumed by Q_1. We need to consider the following subsets of S for possible applications of *OI2*: 1) Q_2 and Q_4; 2) Q_2 and Q_5; 3) Q_3 and Q_4; and 4) Q_3 and Q_5. This example illustrates that in some cases the number of subsets to be considered is at least $O(2^{\frac{n}{2}})$, where n is the number of queries in S. If n is small, we can afford to search exhaustively for all possible subsets of S. When n is large,

we might improve performance at the expense of accuracy. For example, we might report any indistinguishable tuples as related tuples, which might lead to false positive reports.

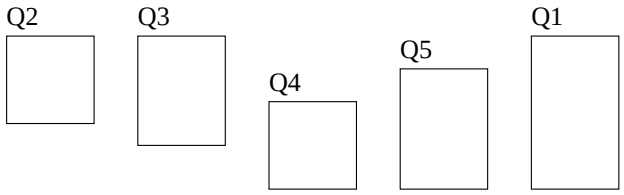

Figure 4.7: An Example on Finding Overlapping Set.

Consider the complexity of the function *OVERLAP*. Let Q be the number of queries issued or inferred by a user, that is, the size of *PREV_QUERY(U)*. The number of iterations in line 2 is Q. Line 3 or 7 runs in $O(N^2A)$ time. Lines 5 and 6, 9 and 10, or 12 and 13 run in $O(NA)$ time. Each inferred query is generated using $O(N)$ time. Hence, lines 2 through 19 run in $O(QN^2A)$ time. Consider the complexity of the function *FIND_OVERLAP_SET*. Let the number of sets of queries found in function *FIND_OVERLAP_SET* be S_1, line 2 has $O(Q)$ number of iterations. Line 4 runs in $O(QN)$ time by checking if each return tuple in S relates to a return tuple of Q_i. Line 5 runs in $O(Q^2N^2A)$ time as it need to consider $O(Q^2)$ pairs of queries in S, and find out the number of distinguishable tuples (in $O(N^2A)$ time) in each pair of them. Line 7 has $O(2^Q)$ number of iterations, and line 8 runs in $O(Q^2N^2A)$ time as in line 5. Hence, the running time of the function *FIND_OVERLAP_SET* is $O(Q^2N^2 2^Q)$. Lines 21 and 22 in function *OVERLAP* are similar to lines 7 and 8 in function *FIND_OVERLAP_SET*, and they both run in the same time. Line 23 runs in $O(A)$ time. Lines 25 and 27 both run in $O(N)$ time. Hence, the total running time of the function *OVERLAP* is $O(Q^2N^2A2^Q)$.

4.2.6 Function COMPLEMENTARY

Figure 4.8 shows the function *complementary*. It checks if the complementary inference rule can be applied. If this is the case, it generates at most six inferred queries: the set of tuples returned by Q_i but not by Q_k, the set of tuples by Q_i and also by Q_k, the set of tuples returned by Q_k but not by Q_i, and similarly the three inferred queries generated from Q_j and Q_l. Two of them have subsume relationships among each other. There are $O(Q^2)$ pairs of queries that satisfy the $\sqsubset$ relations, hence line 1 runs in $O(Q^2)$

function **COMPLEMENTARY**(U, Q_i, Q_j):
1. **for** each pair of queries Q_k and Q_l in *PREV_QUERY(U)*, where $Q_k \sqsubset Q_l$, the related tuples among return tuples
   of Q_i and Q_k, and Q_j and Q_l are identified **do**
2. **if** (each t_i of Q_i that does not relate to Q_k are distinguishable from all tuples of Q_l) or ($Q_l \sqsubset Q_k$) or ($|Q_k| = |Q_l|$) **then**
3. generate an inferred query $Q_1 = (AS_i;\ SC_i \wedge \neg\ SC_k)$;
4. generate an inferred query $Q_2 = (AS_j;\ SC_j \wedge \neg\ SC_l)$;
5. set $Q_1 \sqsubset Q_2$;
6. generate possibly four inferred queries;
7. **return** $GEN \cup$ {newly inferred queries};

Figure 4.8: The function COMPLEMENTARY

time. Line 2 runs in $O(N^2A)$ time. Line 3, 4 or 6 runs in $O(N)$ time. Therefore, the function total running time is $O(Q^2N^2A)$.

4.2.7 Function UNION_QUERIES

Figure 4.9 shows the function *UNION_QUERIES*. It begins with two sets of queries: S_1 and S_2. S_1 is the set of queries generated in the *INFERENCE* function. S_2 is the set of queries whose return tuples have been expanded in the *INFERENCE* function. For each of these newly generated queries Q_i, the function (from lines 2 through 14) checks if there is a union query that subsumes or is subsumed by Q_i. Also, the function checks if there is a union query, including Q_i, that subsumes or is subsumed by other queries. The function maintains a data structure IMP which stores pairs of queries that have subsume relationships. Lines 15 through 22 check if each query in S_2 have subsume relationship with some union queries. If this is the case, the pair of queries are added to IMP. As the queries in S_2 are not newly generated inferred queries, it is not necessary to create any new union query that includes them. Lines 23 through 26 apply inference rules to each pair of queries in IMP.

Figure 4.10 shows the function *FIND_SUBSUMED_QUERY(U, Q_i, Q_j)*. It finds all the union queries, each of which includes Q_i, that are subsumed by Q_j. The running time of this algorithm is exponential in the size of the set S which is bound by $O(Q)$, as line 4 generates $2^{|S|}$ number of union queries. Figure 4.11 shows the function *FIND_SUBSUME_QUERY(U, Q_i, Q_j)*. It finds all union queries, each of which includes Q_i, that subsumes Q_j. The running time of this algorithm is also exponential in $|S|$. For

```
function UNION_QUERIES(U, S1, S2):
1.    IMP ← ∅;
2.    for each query Qi ∈ S1 do
3.       for each query Qj in PREV_QUERY(U) that has tuples relate to those of Qi do
4.          if there is a union query UQ including Qj then
5.             IMP ← IMP ∪ (Qi, UQ) if Qi ⊏ UQ;
6.             IMP ← IMP ∪ (UQ, Qi) if QU ⊏ Qi;
7.          SU ← FIND_SUBSUMED_UNION(U, Qi, Qj);
8.          for each union query UQ in SU do
9.             create a new union query UQ if it does not exist;
10.            IMP ← IMP ∪ (UQ, Qj);
11.         SU ← FIND_SUBSUM_UNION(U, Qi, Qj);
12.         for each union query UQ in SU do
13.            create a new union query UQ if it does not exist;
14.            IMP ← IMP ∪ (Qj, UQ);
15.   for each query Qi ∈ S2 do
16.      for each union query UQ do
17.         IMP ← IMP ∪ (Qi, UQ) if Qi ⊏ UQ;
18.         IMP ← IMP ∪ (UQ, Qi) if UQ ⊏ Qi;
19.      for each union query that contains Qi do
20.         for each query Qj ∈ PREV_QUERY(U) do
21.            IMP ← IMP ∪ (Qj, UQ) if Qj ⊏ UQ;
22.            IMP ← IMP ∪ (UQ, Qj) if UQ ⊏ Qj;
23.   for each (Qi, Qj) in IMP do
24.      SUBSUME(Qi, Qj);
25.      OVERLAP(U, Qi, Qj);
26.      COMPLEMENTARY(U, Qi, Qj);
27.   return inferred queries generated and expanded tuples;
```

Figure 4.9: The function UNION_QUERIES.

example, let S be $\{Q_1, Q_2, \ldots, Q_n\}$, that is, $|S| = n$. Suppose for each query Q_k in S, there is one and only one query Q_{k+1} in S such that Q_k and Q_{k+1} do not have an overlapping tuple that relates to return tuples of Q_j. Then, a union query is formed by taking either Q_k or Q_{k+1}, for $k = 1, 3, 5, \ldots, \frac{k}{2}$, Hence, the number of union queries is $2^{\frac{|S|}{2}}$. For example, consider the five queries as shown in Figure 4.12. We want to find the union queries formed from Q_1, Q_2, Q_3, and Q_4 that subsume Q_0. Each of these four queries has two parts: the one that overlaps with Q_0, and the one that does not. From these four queries, we can form the following union queries that subsume Q_0: $\{Q_1, Q_3\}$, $\{Q_1, Q_4\}$, $\{Q_2, Q_3\}$, and $\{Q_2, Q_4\}$. That is, the number of union queries found $= 2^{\frac{|S|}{2}} = 2^{\frac{4}{2}} = 4$. In our implementation, we assume all union queries have at least one projected attributes. This reduces the num-

function **FIND_SUBSUMED_UNION**(U, Q_i, Q_j):
1. $S \leftarrow \emptyset$;
2. **foreach** Q_k in $PREV\_QUERY(U)$, $Q_k \sqsubset Q_j$, and Q_k and Q_i do not have overlapping return tuple **do**
3. $S \leftarrow S \cup Q_k$;
4. $SS \leftarrow \emptyset$;
5. **for** each subset $S\prime$ of the set of queries S, the queries in $S\prime$ is subsumed by Q_j **do**
6. $SS \leftarrow SS \cup S\prime$;
7. return SS;

Figure 4.10: The function FIND_SUBSUMED_UNION.

function **FIND_SUBSUM_UNION**(U, Q_i, Q_j):
1. $S \leftarrow \emptyset$;
2. **foreach** Q_k in $PREV\_QUERY(U)$, Q_k and Q_j has related tuples, and related tuples between Q_k and Q_i do not have related return tuple in Q_j **do**
3. $S \leftarrow S \cup Q_k$;
4. return {}if all queries in S does not has a related tuple in Q_j;
5. $SS \leftarrow \emptyset$;
6. **for** each subset $S\prime$ of the set of queries S, the queries in $S\prime$ subsums Q_j **do**
7. $SS \leftarrow SS \cup S\prime$;
8. return SS;

Figure 4.11: The function FIND_SUBSUM_UNION.

ber of possible union queries, but it introduces false negative reports. Both the function *FIND_SUBSUMED_QUERY* and *FIND_SUBSUME_QUERY* run in $O(QN+2^Q)$ time. The function *FIND_UNION* has $O(Q^2N + Q2^Q))$ running time.

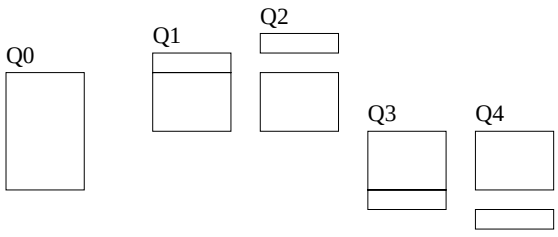

Figure 4.12: An Example on Finding Union Queries.

The problem of finding union queries that subsume another query can be transformed into a graph problem. Given a set of queries S, where each query in S has a return tuple related to a return tuple of Q_0. We can form a graph such that each node in the graph

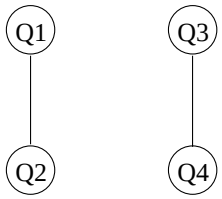

Figure 4.13: An Example on Using a Graph to Find Union Queries.

corresponds to a query in S. An edge connects a node A, corresponding to a query Q_A, to another node B, corresponding to a query Q_B, if the related tuples between Q_A and Q_B do not have related return tuples in Q_0. Let V be the set of vertices in the graph such that after removing all vertices in V from the graph, all edges in the graph are removed. Then, the queries corresponding to the vertices in V form a union query. For example, Q_1, Q_2, Q_3, and Q_4 in Figure 4.12 can be transformed into a graph as shown in Figure 4.13. The nodes are named with their corresponding queries. A totally disconnect graph is formed when any one of the following sets of vertices is removed from the graph $\{Q_1, Q_3\}$, $\{Q_1, Q_4\}$, $\{Q_2, Q_3\}$, and $\{Q_2, Q_4\}$. We can form the four union queries as discussed above using queries corresponding to the remaining vertices.

4.3 Implementation and Experimental Results

We have developed a prototype of the inference detection system in about 4,000 lines of Perl code. We have implemented the split_query, subsume, unique characteristic, overlapping (without *OI2*), and complementary inference rules. We have also implemented the applications of the inference rules in union queries. We ran our experiments with randomly generated tables and user queries. Each table has N_{attr} number of attributes, and N_{rec_num} number of records. The primary key of the table is a single attribute. All attributes are of integer types. Each attribute value in the table is uniformly distributed between 0 and $(N_{data\_dist} \times N_{rec\_num})$, where $0 < N_{data\_dist} \leq 1$. We also randomly generate N_{query_num} number of user queries. Each query projects N_{proj} number of attributes from the table. The selection criterion of each query is a conjunction of N_{cond} number of conjuncts. Each conjunct is of the form 'A_i *op* a_i', where A_i is an attribute from the table, *op* is one of the relational operations ($>$, $\geq$, $\leq$, $<$, and $=$), and a_i is an attribute value. We only consider queries in which the number of return tuples falls between $(N_{ret\_tuple} - 20)$ and

(N_{ret_tuple} + 20). In our experiments, the average number of return tuples turned out to fall between N_{ret_tuple} − 1 and N_{ret_tuple} + 1. Appendix A shows a sample database and queries that are generated by our prototype.

The evaluation of an implication between two logical expression is an NP-hard problem [SKN89]. We approximate the evaluation of a logical implication $C_i \Rightarrow C_j$ by checking if the tuples selected by C_i is also selected by C_j, and that the set of attributes appears in C_j is a subset of those appear in C_i. Also, we only consider those union queries that have simple queries with at least one common projected attribute.

The prototype ran on a a Sun SPARC 20 workstation running Solaris. We collected the following data to measure the system performance,

- average number of seconds used to process one query.
- number of inferred queries generated.
- number of times the inference rules are applied.
- *percentage of database retrieved.*
- *percentage of database revealed.*

The *percentage of database retrieved* is the percentage of the database that is retrieved by queries in each experiment. In our experiments, we assume the policy is to protect the data about individual records. We can specify the policy as follows,

$$(\mathsf{U};\ A_1,\ \mathsf{*};\ \mathsf{true})$$

where A_1 is the primary key of the database. With this policy, we define the *percentage of database revealed* as the ratio (expressed in percentage) between the number of attribute values of individual tuples that have been revealed to a user (either by directly accessing them using queries or by inference) and the total number of attribute values in the database. For example, consider the following two queries that are issued to the sample database in Figure 3.2, Chapter 3,

Q_1 = (Age; Name = 'Alice'),

Q_2 = (Department; Age < 40).

Q_1 returns a single tuple (35) which says that Alice is 35 years old. Q_2 returns two tuples ('Marketing') and ('Marketing') which say that all employees at the age less than 40 work in the Marketing department. By subsume inference rule *SI1*, a user can infer that Alice works

N_{attr}	Avg. Query Processing Time (sec.)	Number of Inferences	Percentage of DB Revealed	Number of Inferred Queries	Percentage of DB Retrieved	Query Overlapping Ratio
40	8.010	2697p, 41s	16.52	375	90.08	2.770
60	3.452	1839p, 2s	9.50	110	80.10	2.097
80	2.414	1060p	4.55	52	70.99	1.768
100	2.092	1156p	4.13	20	62.99	1.596
120	1.976	1036p	3.08	12	56.19	1.482
140	1.684	354p	0.98	4	51.11	1.406

Table 4.1: Experiment Results for Experiment 1 with $N_{data\_dist} = 50\%$.

in the Marketing department. Hence, the number of attribute values revealed to the user is three, namely Alice's name, age, and department. Note that although Q_2 returns two tuples, the user cannot determine to which employees these two tuples belong; hence, they are not included as the attribute values that are revealed to the user. The total number of attribute values in the sample database is 24 — there are four records, each with 6 attribute values. Hence, the percentage of database revealed by Q_1 and Q_2 with respect to the sample database is (3 / 24) $\times$ 100%, or 12.5%.

We define a *query overlapping ratio* to describe the amount of overlapping among queries. It is defined as follows:

$$\textit{query overlapping ratio} = \frac{\text{`}\textit{total number of return tuples}\text{'} \times N_{proj}}{\text{`}\textit{percentage of database retrieved}\text{'} \times N_{rec\_num} \times N_{attr}}$$

When the query overlapping ratio is 1, there is no overlapping among the queries. The higher the ratio, the larger the amount of overlapping among queries. Appendix B shows a sample session on a run of the prototype.

We ran six experiments to determine how the characteristics of the database and the queries affect the system performance. For the database, we considered the following characteristics: 1) the number of tuples in the database; 2) the number of attributes in the database; and 3) the amount of duplication of the data values. For the queries, we considered the following characteristics: 1) the number of attributes projected by the queries; 2) the number of conjuncts in the selection conditions; 3) the number of queries being issued; and 4) the number of tuples returned by the queries.

Experiment 1 investigated the effect of the number of attributes in the database

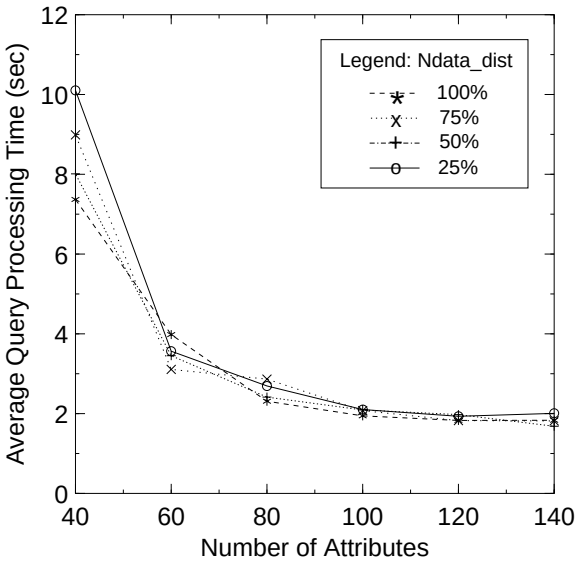

Figure 4.14: The Effect of the Number of Attributes and Amount of Data Duplication on System Performance.

on the system performance. We chose the following parameter values in this experiment: $N_{rec\_num} = 1000$, $N_{ret\_tuple} = 50$, $N_{proj} = 4$, $N_{cond} = 3$, $N_{query\_num} = 500$. N_{attr} takes the values of 40, 60, 80, 100, 120, and 140. N_{data_dist} takes the values of 25%, 50%, 75%, and 100%. Figure 4.14 shows the results in a graph plotted with the average query processing time (in seconds) against the number of attributes in the database. The four lines in the graph corresponds to different values of N_{data_dist}.

Table 4.1 show the detail results of the experiment for $N_{data\_dist} = 50\%$. The third column is the "Number of Inferences". The number before an 'p' stands for the number of times the unique characteristic inference rule is applied using the primary key of the database as the unique characteristic. The number before an 's', an 'o', and an 'c' stand for the number of times the subsume, overlapping, and complementary inference rules are applied respectively. There was no overlapping or complementary inference detected in this experiment.

Consider individual lines in Figure 4.14. It shows that the system in general runs faster as N_{attr} increases from 40 to 140. The larger the number of attributes in the table, the lesser the amount of overlapping among the return tuples of queries. This is reflected

N_{data_dist}	Avg. Query Processing Time (sec.)	Number of Inferences	Percentage of DB Revealed	Number of Inferred Queries	Percentage of DB Retrieved	Query Overlapping Ratio
25%	10.106	3546p, 34s, 1o	21.44	382	90.50	2.778
50%	8.010	2697p, 41s	16.52	375	90.08	2.770
75%	8.988	2660p, 24s	17.75	368	90.81	2.748
100%	7.368	2800p, 19s	17.49	314	90.37	2.759

Table 4.2: Experiment Results for Experiment 2 with $N_{attr} = 40$.

in Table 4.1 by the decreasing values of query overlapping ratios as N_{attr} increases. The smaller amount of query overlapping results in a lesser chance that the subsume relationships hold among queries, and hence the smaller number of inferences. This is also reflected in Table 4.1 by the decreasing number of inferences as N_{attr} increases. The system performance reaches a plateau when $N_{attr} \geq 100$. When N_{attr} increases to a certain value, the amount of overlapping among queries becomes steady. For $N_{data\_dist} = 50\%$, the percentage decreases of the query overlapping ratio as N_{attr} increases are 24.3%, 15.7%, 9.7%, 7.1%, and 0.5%. Similar pattern occurs for other values of N_{data_dist}.

Experiment 2 investigated the effect of the change of N_{data_dist} on the system performance. Table 4.2 shows the results for $N_{attr} = 40$. In general, the system performs better as N_{data_dist} increases. Intuitively, the lower the value of N_{data_dist}, the more the amount of duplication among data in the database. This results in a lesser chance that a return tuple is distinguishable from others, and hence the smaller number of occurrences of inferences. This is reflected in Table 4.2, where the number of subsume inferences decreases as N_{data_dist} increases (with an exception when $N_{data\_dist} = 50\%$).

Experiment 3 investigated the effect of the number of return tuples of queries on the system performance. Figure 4.15 shows the results for $N_{rec\_num} = 1000$, $N_{data\_dist} = 50\%$, $N_{proj} = 4$, $N_{cond} = 3$, and $N_{query\_num} = 500$. N_{ret_tuple} takes the values of 50, 100, 150, 200, and 250. N_{attr} takes the values of 80 and 120. The result shows that the system runs slower as N_{ret_tuple} increases. First of all, the larger the number of return tuples, the longer it takes for the system to process them. Also, the more the number of tuples returned by the queries, the larger the amount of overlapping among queries, and hence, the more the number of occurrences of inferences. For $N_{attr} = 80$, the query overlapping ratios are 1.768, 2.740, 3.849, 5.052, 6.271 as N_{ret_tuple} increases from 50 to 250, and the total number

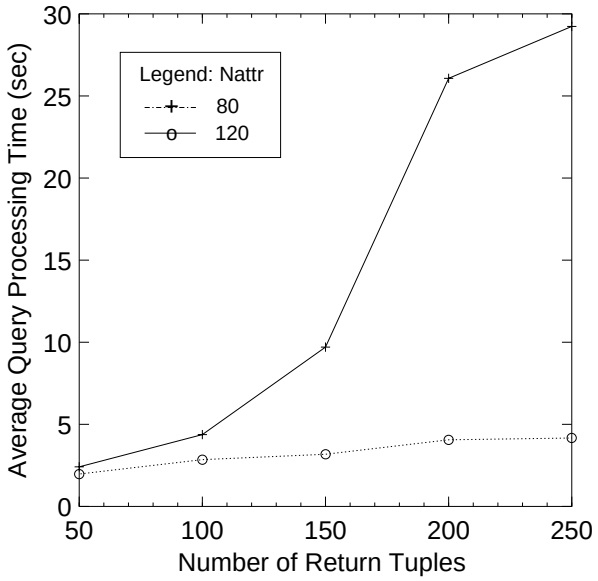

Figure 4.15: The Effect of the Number of Return tuples on System Performance.

of inferences are 1060, 2751, 3769, 6037, and 8091 as N_{ret_tuple} increases.

Experiment 4 investigated the effect of the number of projected attributes in queries on the system performance. Figure 4.16 shows the results for $N_{rec\_num} = 1000$, $N_{query\_num} = 500$, $N_{data\_dist} = 50\%$, $N_{attr} = 80$, and $N_{ret\_tuple} = 50$. N_{proj} takes the values of 4, 5, 6, 7, and 8. N_{cond} takes the values of 4, 5, 6, and 7. Table 4.3 shows the result for $N_{cond} = 4$. It shows that the system runs slower as N_{proj} increases. This is because the higher the number of attributes projected by the queries, the larger the amount of overlapping among the return tuples of queries, and hence the larger the number of inferences. As

N_{proj}	Avg. Query Processing Time (sec.)	Number of Inferences	Percentage of DB Revealed	Number of Inferred Queries	Percentage of DB Retrieved	Query Overlapping Ratio
4	2.066	1609p	8.82	4	80.63	2.062
5	3.158	2148p	14.81	24	87.01	2.380
6	3.182	2162p, 4s	17.42	26	91.25	2.767
7	4.422	2582p, 11s, 1c	22.73	57	93.69	3.091
8	8.894	4153p, 65s, 2c, 3o	36.47	167	95.18	3.489

Table 4.3: Experiment Results for Experiment 4 with $N_{cond} = 4$.

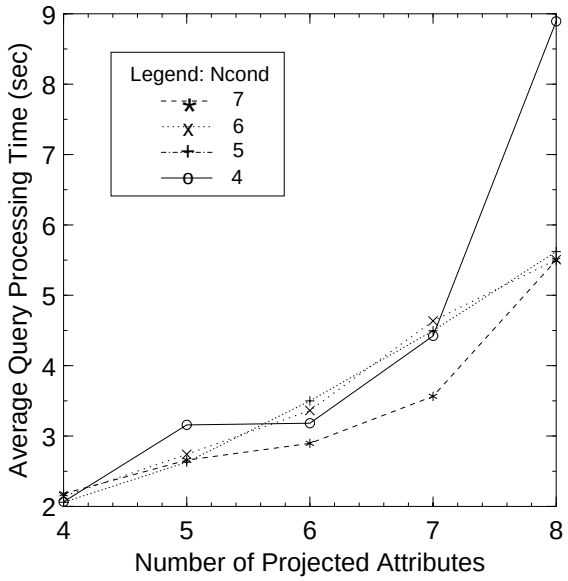

Figure 4.16: The Effect of the Number of Projected Attributes in Queries on System Performance.

shown in Table 4.3, the query overlapping ratio and the number of inferences increase as N_{proj} increases.

Experiment 5 investigated the effect of the number of conjuncts in the selection criteria on the system performance. Figure 4.17 shows the results for $N_{rec\_num} = 1000$, $N_{query\_num} = 500$, $N_{data\_dist} = 50\%$, $N_{attr} = 80$, and $N_{ret\_tuple} = 50$. N_{cond} takes the values of 3, 4, 5, 6, and 7. N_{proj} takes the values of 4, 5, 6, and 7. Table 4.4 shows the results for $N_{proj} = 5$. It shows that the system runs faster as N_{cond} increases. This is because the larger the number of conjuncts in the selection criteria of the queries, the lesser the

N_{cond}	Avg. Query Processing Time (sec.)	Number of Inferences	Percentage of DB Revealed	Number of Inferred Queries	Percentage of DB Retrieved	Query Overlapping Ratio
3	6.270	2897p, 33s	13.39	242	87.16	2.416
4	3.158	2148p	14.81	24	87.01	2.380
5	2.628	2236p	15.35	0	86.87	2.366
6	2.74	2187p	15.00	0	87.05	2.366
7	2.658	2144p	14.79	0	87.23	2.364

Table 4.4: Experiment Results for Experiment 5 with $N_{proj} = 5$.

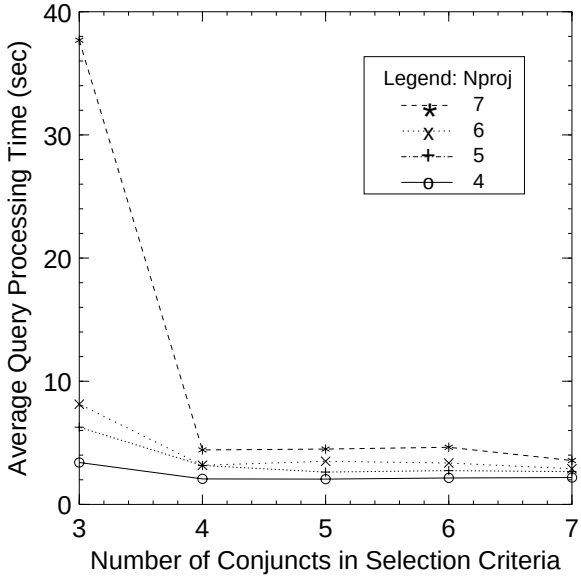

Figure 4.17: The Effect of the Number of Conjuncts in Selection Criteria on System Performance.

chance that the subsume relationships hold among the queries, and hence the smaller the number of occurrences of inferences. However, the effect is not significant when $N_{cond} \geq 4$. As shown in Table 4.4, although the query overlapping ratios and the number of inferences decrease as N_{cond} increases, it becomes steady for $N_{cond} \geq 4$.

Experiment 6 investigated the effect of the number of tuples in the database on the system performance. Figure 4.18 shows the results for $N_{data\_dist} = 50\%$, $N_{attr} = 80$, $N_{ret\_tuple} = 50$, $N_{query\_num} = 500$, $N_{proj} = 4$, and $N_{cond} = 3$. N_{rec_num} takes the following values: 1000, 2500, 5000, 7500, and 10000. Table 4.5 shows the data of the experiment. It

N_{rec_num}	Avg. Query Processing Time (sec.)	Number of Inferences	Percentage of DB Revealed	Number of Inferred Queries	Percentage of DB Retrieved	Query Overlapping Ratio
1000	2.414	1060p	4.55	52	70.99	1.768
2500	2.330	1889p	3.28	41	39.12	1.292
5000	1.912	1060p	0.99	18	21.93	1.141
7500	1.854	1108p	0.73	8	15.24	1.100
10000	1.988	1305p	0.64	17	11.70	1.073

Table 4.5: Experiment Results for Experiment 6.

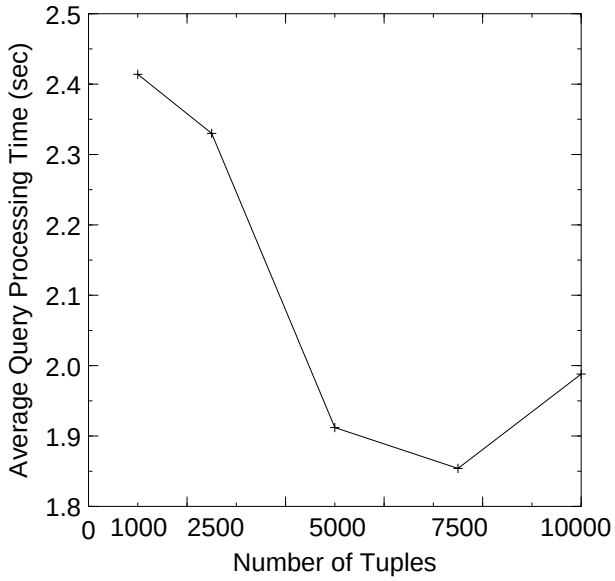

Figure 4.18: The Effect of the Number of Tuples in Database on System Performance.

shows that the system runs faster as the number of tuples of the database increases. As the size of the database increases, the possible amount of overlapping among the queries decreases, and hence the lesser the number of inferences. This is reflected by the decreasing value of the query overlapping ratios as N_{rec_num} increases. For $N_{ret\_tuple} = 10000$, the set of queries happen to generate more inferences and inferred queries than the case for $N_{ret\_tuple} = 5000$ or 7500, and this accounts for the longer running time.

Experiment 7 investigated the effect of the number of queries processed on the system performance. Figure 4.19 shows the results for $N_{rec\_num} = 1000$, $N_{data\_dist} = 50\%$, $N_{attr} = 80$, $N_{ret\_tuple} = 30$, $N_{proj} = 4$, and $N_{cond} = 3$. N_{rec_number} takes the values of 200, 400, 600, 800, 1000, and 1200. It shows that the system runs slower as the number of queries to be processed increases. This is because the more the number of queries, the more the number of inferences. Also, as each user query needs to be compared with previously issued queries to determine their subsume relationships, the more the number of queries, the longer it takes to determine all possible subsume relationships.

Figure 4.20 shows a graph plotted with the percentage of database revealed against the average query processing time for all experiments we have carried out, except three points, (30.9, 37.686), (21.39, 26.072), and (25.84, 29.232) that fall out of the graph. The figure shows that there is a correlation between the percentage of database revealed with

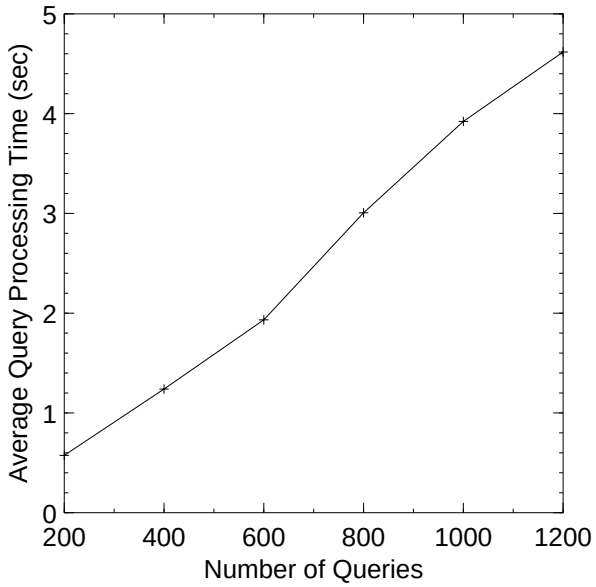

Figure 4.19: The Effect of the Number of Queries Processed on System Performance.

the system performance. The greater the amount of the database being revealed, the lower the system performance. Except for the eight points that lie on the upper half of the graph, most points on the graph lie on a straight line with a slope which increases about 1 second in the average query processing time for every 10% increases in the percentage of database revealed. Four of the eight points are obtained from experiment 1 with $N_{attr} = 40$. Two of them are obtained from experiment 4 with $N_{cond} = 3$. One of them is obtained from experiment 5 with $N_{proj} = 8$ and $N_{cond} = 4$. One of them is obtained from experiment 3 with $N_{ret\_tuple} = 150$ and $N_{attr} = 80$. For the three excluded points, two of them obtained from experiment 3 with $N_{attr} = 80$ and $Nret\_tuple = 200$ and 250, and one of them is obtained from experiment 5 with $N_{cond} = 3$ and $N_{proj} = 7$. We can say that the eleven points are obtained under the extreme conditions in the experiments, namely small values of N_{attr}, small values of N_{cond}, high values of N_{proj}, and high values of N_{ret_tuple}. This suggests that, other than in the extreme conditions, the system performance is rather predictable, decreasing steadily with the increase of the amount of database revealed.

From the experimental results, we note that there is a high correlation between the percentages of database revealed and the query overlapping ratios. In Figure 4.21 we plot a graph with the percentage of database revealed against the query overlapping ratio for all experiments that we have carried out. The eight points on the right lower half of the

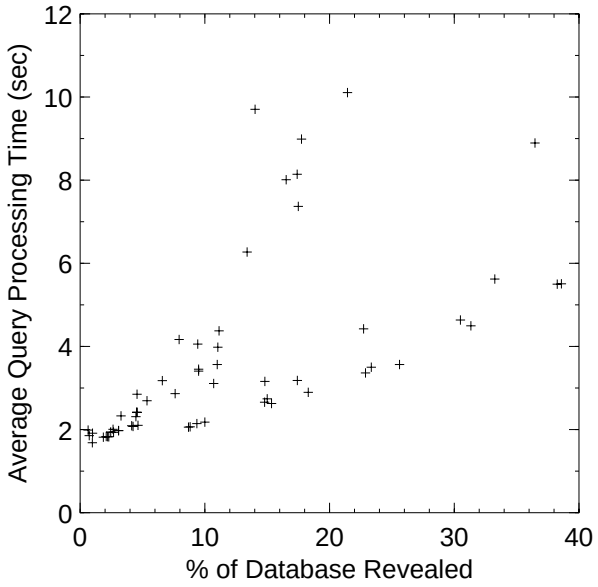

Figure 4.20: The Correlation between the Percentage of Database Revealed and System Performance.

graph are obtained from experiment 3 with N_{ret_tuple} equals 100, 150, 200, and 250. The query overlapping ratio is a statistic that can be easily calculated. Hence, one might use the ratio as an indicator of the amount of data in the database that have been revealed to users. The higher the value of the query overlapping ratio, the larger the amount of data might have been revealed to users. Our experiment results show that the use of the query overlapping ratios to estimate the amount of database revealed does not generate false negative reports, though it does generate false positive reports. That is, with low values of the query overlapping ratios, the values of percentages of database revealed are always low. However, with high values of the query overlapping ratios, the values of percentages of database revealed are high in most cases, but there are cases where the values are indeed low (specifically, the eight points mentioned above). For nonextreme conditions, we expect the query overlapping ratio to be a good indicator of the amount of database being revealed to users.

The percentage of database retrieved is another easy calculated statistic that we might use to estimate the amount of the database revealed. Figure 4.22 shows a graph plotted with the percentage of database revealed against the percentage of database retrieved for all experiments that we have carried out. Again, there are eight points on the lower right hand side of the graph that correspond to results obtained under extreme conditions,

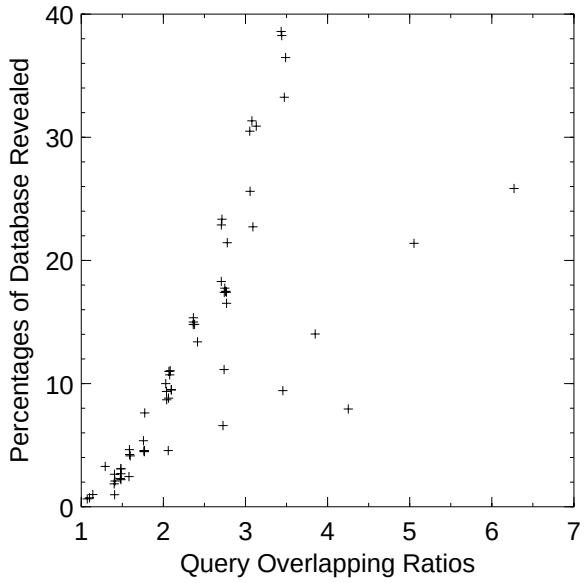

Figure 4.21: The Correlation between the Percentages of Database Revealed and the Amount of Query Overlapping.

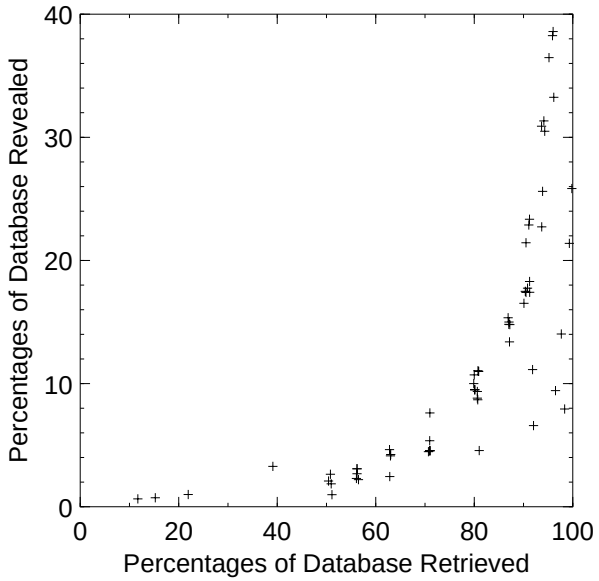

Figure 4.22: The Correlation between the Percentages of Database Revealed and the Percentage of Database Retrieved.

namely in experiment 3 with N_{ret_tuple} equals 100, 150, 200, and 250. The graph also indicates that there is a exponential relationship between the percentage of database revealed and the percentage of database retrieved.

4.4 Summary

In this chapter, we present the data level inference detection algorithms. In theory, the algorithms are exponential in the number of queries issued. For example, we found that identifying the union queries that subsume or are subsumed by other queries, and the evaluation of logical implications run in exponential time. In the former case, we assume all simple queries in an union query share at least one projected attribute. This simplification generates false negative reports. In the latter case, we approximate the evaluation of the implication $C_1 \Rightarrow C_2$ by checking if the set of tuples selected by C_1 is a subset of the set of tuples selected by C_2, and the set of attributes in SC_2 is a subset of the set of attributes in SC_1. This approximation generates false positive reports.

We have developed a prototype to investigate the system performance under various characteristics of the database and queries. The results show that, in general, the inference detection system performs better with a larger number of attributes in the database, more amount of duplication of attribute values in the database, smaller number of tuples returned by the queries, smaller number of attributes projected by the queries, and larger number of conjuncts in the queries. Therefore, although the inference detection system can be inefficient (in fact, it is an NP-complete problem), in some cases, the system could be practically employed. For example, the system performance decreases steadily with an increasing amount of the database revealed to users. We also discovered that it is possible to use the query overlapping ratio and percentage of database retrieved as indicators of the amount of the database revealed to users. In general, the higher the values of the query overlapping ratios or the percentage of database retrieved, the higher the amount of the database revealed to users. These ratios can be easily calculated, and hence they can act as a rough indicator of the amount of data revealed to users.

Chapter 5

Towards a Realistic Inference Detection System

In this chapter, we discuss several extensions to the data level inference detection system described in Chapters 3 and 4. In particular, we discuss the detection of approximate inferences, the effects of update operations on inference rules, the processing of nested queries, and the uses of multiple tables in the database. The results from this chapter can improve the performance of the detection system, and allow the system to be operated with more realistic assumptions.

5.1 Approximate Inference

In chapters 3 and 4, we discussed the development of an exact inference detection system. By exact inference, we mean that an inference occurs when a user can infer exactly the protected associations. For example, if we want to protect the salaries of employees, then an inference occurs only if a user can infer the exact salary of an employee. Even if the user can deduce that an employee earns either 60K or 61K, we still say that no exact inference has occurred. However, if the purpose of protecting this association is to protect the privacy of employee personal information, we do not need to protect the exact salaries of employees. A user knowing the salary of an employee is either 60K or 61K has more or less deduced the salary of the employee. We can make the detection system more practical by extending it to detect approximate inferences as well.

In approximate inference, an inference occurs when a user can infer protected

associations within certain "deviations". The policy specifies the conditions under which approximate inferences are deemed to occur. We extend the policy as described in Chapter 3 as follows,

$(U; A_1, \ldots, A_n; E; E_{appr})$

where U is a user, A_1, ..., A_n are the attributes of the protected association, E is a logical expression that selects the tuples to be protected, and E_{appr} is a logical expression called the *approximate inference expression*. We define two tuple variables that are useful in specifying the approximate inference expression. We define a *protected tuple variable* to be a tuple variable representing a tuple in the database that satisfies E. We also define an *inferred tuple variable* to be a tuple variable representing a tuple that a user infers about a tuple in the database. For example, we want to protect John's salary which is 60K. We can create a protected tuple variable t_p to represent John's information, where $t_p[Name]$ = 'John' and $t_p[Salary]$ = 60K. Suppose a user can infer that John's salary is either 60K or 61K. We can create two inferred tuple variables, t_{i1} and t_{i2}, to represent this information, where $t_{i1}[Name]$ = 'John', $t_{i1}[Salary]$ = 60K, $t_{i2}[Name]$ = 'John', and $t_{i2}[Salary]$ = 61K.

We define formally the occurrences of approximate inferences with respect to the above extended policy as follows,

Definition 5 *Given a policy $(U; A_1, \ldots, A_n; E; E_{appr})$, an approximate inference occurs if there exists a tuple t_p, t_p satisfies E, and a set of inferred tuples S, such that there exists an inferred tuple t_i in S, t_i relates to t_p, and for each inferred tuple t_i about t_p, t_i satisfies E_{appr}.*

Note that t_p is a tuple from the database, while t_i might not exist in the database. A user might believe that an inferred tuple exists in the database based on information available to the user. For example, from these two return tuples ('Bill', 'Manager') and ('Manager', 60K), a user might believe that there exists a tuple ('Bill', 'Manager', 60K) in the database, although the two return tuples do not relate to each other. We discuss the specification of the approximate inference expression in the following section.

5.1.1 Specifying Approximate Inference Policies

There are various ways to specify the expression E_{appr}. The policy may specify a range of values such that an approximate inference occurs when the inferred value falls within the range. For example, consider the following policy,

$(U; Name, Salary; Name = \text{`}John\text{'}; t_i[Name] = \text{`}John\text{'} \wedge t_i[Salary] > 75K \wedge t_i[Salary] < 80K)$

where t_i is an inferred tuple variable. This policy says that an approximate inference occurs when the user U can infer that John's salary is between 75K and 80K. The policy can also specify the range using the protected attribute values. For example, the following policy,

$(U; Name, Salary; true; \frac{|t_i[Salary] - t_p[Salary]|}{t_p[Salary]} \leq 0.05)$

specifies that an approximate inference occurs when the user U can infer that an employee's salary falls in the range $[s \times (1.05),\ s \times (0.95)]$, where s is the salary of the employee. In some cases, we might want to detect exact inference in one situation, and detect approximate inference in another. For example, we want to protect information about the number of felony charges that a person has. There is a significant difference between knowing a person has none or one felony charge; while there might not be a significant difference between knowing a person has ten or eleven felony charges. Hence, we might specify the policy as follows,

$(U; Name, FC\_num; true; (t_i[FC\_num] = t_p[FC\_num] \wedge t_p[FC\_num] < 4) \vee ((|t_i[FC\_num] \Leftrightarrow t_p[FC\_num]|) \leq 1 \wedge t_p[FC\_num] \geq 4)$

where *FC_num* stands for the attribute storing the number of felony charges of a person. The policy says that if a person has less than four felony charges, then we detect if the user U can infer the exact number of felony charges that the person has. However, if a person has four or more felony charges, then the system detects if the user U can infer the number of felony charges that the person has, with a plus or minus one unit of error.

For categorical data, data items can be classified into categories. We can specify a policy such that an approximate inference occurs if a user can infer that an attribute value belongs to certain category. For example, there is a database containing information about the cities and the counties where weapons are located. Suppose we need to protect the information about the cities where the weapons are located, so as to avoid adversaries from attacking the arsenal. In general, each county includes several cities, and letting a user know the counties where the weapons are located does not give out enough information about which cities they are located in. However, for counties that cover a small area, the counties that the weapons are located should also be protected. We can specify the policy as follows,

$$(U; Weapon\_id, city; true; t_i[County] = t_p[County] \wedge City\_num(County) \leq 2)$$

where $City\_num$ is a function that maps a county name to the number of cities in the county. The policy says that an approximate inference occurs when the user U knows about a county where the weapons are located if the county has two or fewer cities.

Consider the specifications of approximate inferences in protecting one-to-many associations. Two attributes A_1 and A_2 have an one-to-many association among them if, given one attribute value of, say, A_1, there can be zero, one, or more than one associated attribute values of A_2. Given an association $(A_1, \ldots, A_n)$, each tuple from the database that is projected over these n attributes is an instance of the association. We can protect an instance of an association between A_1 and A_2 using the following policy,

$(U; A_1, A_2; true)$

For example, we can protect employees' children information using this policy,

$(U; Employee\_name, Child\_name; true)$

which says that the user cannot find out any child's name of an employee. However, in some cases, we allow a user to know some instances of an association, but we want to prohibit the user from knowing all instances of the association. For example, consider the association between product name and ingredients. Given one product name, there are several product ingredients. It is not suspicious that a user might infer a small number of ingredients of a product, but it becomes suspicious when the user can infer most of the ingredients. We can specify the following policy to protect such an association,

$$(U; Product, Ingredients; Product = X[count(Product = X)];$$
$$\frac{|count(t_i[Product]=X \wedge defined(t_i[Ingredients])) - count(t_p[Product]=X)}{count(t_p[Product]=X)} \geq 80\%)$$

where X is a variable storing a product name, and $count(E)$ is a function returning the number of tuples that satisfy the expression E. Recall from Chapter 3 that a number n in a square bracket indicates that an inference occurs when the user can infer n number of protected tuples. The function $defined(t_i[Ingredients])$ is a boolean function which returns a true value if $t_i[Ingredients]$ is instantiated with a value. Hence, the function $count(t_i[Product] = X \wedge defined(t_i[Ingredients]))$ returns the number of ingredients that a user can infer about the product X. The policy says that an exact inference occurs when the user U can infer all ingredients of a product, and an approximate inference occurs when the user can infer more than 80% of the ingredients of a product.

The protection of a many-to-many association is equivalent to the protection of two one-to-many associations. Suppose we want to protect the association between projects and the employees who work on the projects. This is a many-to-many association. That is, each project has one or more project members, and each employee works in one or more projects. The protection of this many-to-many association can be achieved by protecting the two one-to-many associations: the one from projects to employees, and the other from employees to projects.

5.1.2 Enforcement of Approximate Inference Policies

To determine if an approximate inference occurs with respect to a policy, we need to perform the following two tasks:

1. For each protected tuple, determine the set of inferred tuples S such that the protected tuple must relate to an inferred tuple in S.

2. Determine if the set of inferred tuples satisfy the approximate inference expression E_{appr} of the policy.

The first task can be done by keeping track of the subsume relationships among queries. This is because if $Q1 \sqsubset Q2$, each return tuple t_1 of Q_1 relates to a return tuple t_2 of Q_2. Hence, for each attribute A_1 in AS_1 and attribute A_2 in AS_2, the set of associated attribute values for $t_1[A_1]$ is $\{t_2[A_2]$: t_2 is a return tuple of $Q_2\}$. For example, consider the following policy,

$$(U; Name, Salary; Name = John; |t_i[Salary] \Leftrightarrow t_p[Salary]| \leq 2K)$$

which says that we need to protect John's salary within a 2K error. Suppose the user issues the following two queries,

$$Q_1 : (Name, Age; Age \leq 30)$$

$$Q_2 : (Age, Salary; Age \leq 35)$$

The results to these two queries are as follows,

Name	Age
John	30
Bill	29

Salary
45K
43K
46K

Suppose the protected tuple is ('John', 45K); that is, John earns 45K. As $Q_1 \sqsubset Q_2$, the user can infer that John's salary is either 43K, 45K or 46K. That is, the set of inferred tuples for this protected tuple is {('John', 45K), ('John', 43K), ('John', 46K)}. Task 1 as stated above is achieved, as one of these three inferred tuples must relate to the protected tuple ('John', 45K). Task 2 checks if any of these three inferred tuples satisfies E_{appr}. As the three inferred salaries of John fall between the range [43, 47], we conclude that an approximate inference occurs.

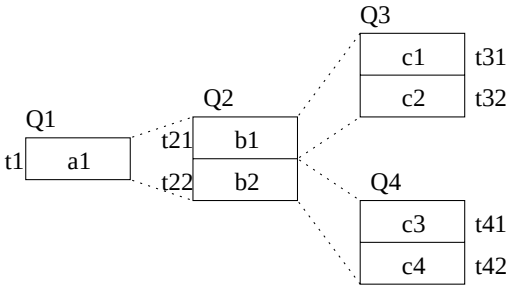

Figure 5.1: An Example on the Propagation of Approximate Inference.

The set of inferred tuples can be obtained through propagation along the subsume relationships. For example, consider the four queries shown in Figure 5.1. $Q_1 \sqsubset Q_2$. Suppose $t_1[A] = a_1$ for the return tuple t_1 of Q_1. Also, $t_{21}[B] = b_1$ and $t_{22}[B] = b_2$ for the two return tuples t_{21} and t_{22} of Q_2. That is, $\{b_1, b_2\}$ is the set of associated attribute values over attribute B for the attribute values a_1. Further, suppose that t_{21} relates to one of the two return tuples, t_{31} and t_{32}, of Q_3, and t_{22} relates to one of the two return tuples, t_{41} and t_{42}, of Q_4. If $t_{31}[C] = c_1$, $t_{32}[C] = c_2$, $t_{41}[C] = c_3$, and $t_{42}[C] = c_4$, then we can conclude that $\{c_1, c_2, c_3, c_4\}$ is the set of associated attribute values over attribute C for the attribute value a_1.

In general, when we need to detect approximate inferences for an association between attribute A_1 and A_n, we need to find out if there exists a sets of associated attribute values, S_1, ..., S_n, such that an attribute value a_1 of A_1 has a set of associated attribute values in S_1 (over certain attributes), and the attribute values in S_1 have a set of associated attribute values in S_2, and so forth. Hence, a_1 has a set of associated attribute values in S_n. This is shown in Figure 5.2. Note that for each pair of consecutive sets S_i and S_{i+1}, where $1 \leq i \leq n \Leftrightarrow 1$, each attribute value in S_i must associate with one of the attribute values in S_{i+1}. For example, refer to the previous example as shown in Figure 5.1. If the

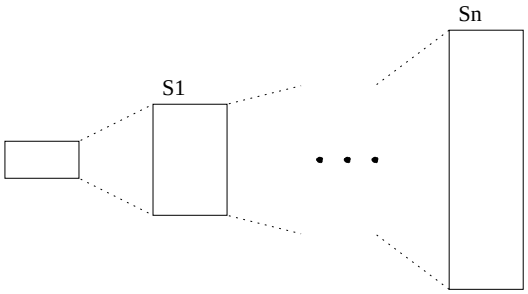

Figure 5.2: An Example on the General Propagation of Approximate Inference.

user cannot determine b_2's associated attribute values over attribute C is $\{c_3, c_4\}$, then the user would not conclude that the set of associated attribute values of a_1 is simply $\{c_1, c_2\}$.

5.2 Effects of Updating on Inference Rules

The inference rules that we developed in Chapter 3 assume the database instances (or contents) and schema do not change while users perform inference. However, this may not be a practical assumption. The database instances and schema can be changed due to normal operations on the database, or the users might intentionally make changes to the database in order to perform inference. Suppose a user has already found out the following information about the database,

Name	Age
John	40
Bill	40

Age	Salary
40	50K
40	60K

That is, both John and Bill are 40 year old, and the two employees who are at the age of 40 earn either 50K or 60K. Suppose the user is authorized to remove an employee's record from the database. The user can infer the salaries of John and Bill as follows. The user first removes Bill's record from the database. After Bill's record is removed, (Age = 40) becomes a unique characteristic in the database. The user can then issue a query to ask for the salary of the employee whose age is 40. If this query returns 50K, then the user can infer that John earns 50K, and hence Bill earns 60K. Note that in case the user does not have the authority to remove the record, the user might trick other authorized users to perform the deletion. For example, the user might produce a false report so that an authorized user

thinks that Bill's record should be removed. In the following sections, we discuss the effects of modifications of the database on the inference rules.

5.2.1 Effects of Modifications of Database Schema on Inference Rules

First, we discuss the effects of modifying the database schema on the subsume, overlapping, and complementary inference rules; that is, determining whether these inference rules can still be applied after the database schema is modified. Changing the database schema by adding or remove attributes does not affect the subsume relationships among queries. A subsume relationship holds between two queries as long as the tuples in the two queries remain the same. The subsume relationships among queries are essential in determining if the subsume, overlapping, and complementary inference rules are applicable. Hence, if these three rules are applicable before a modification of the database schema, they are still applicable after the change.

However, the results of the applications of these three inference rules can be affected by modifications of the database schema. In particular, determining if two tuples are distinguishable can be affected by deleting or inserting attributes into the database schema. For example, a database contains the following information about two tuples,

Age	Sex	Department
35	Male	Marketing
35	Male	Sales

These two tuples are distinguishable. However, if the attribute Department is removed from the database, the two tuples become indistinguishable. If a user learns about these two tuples before the deletion of the attribute Department, then the user can distinguish between the two tuples, which might lead to inference. On the other hand, if the user learns about these two tuples after the deletion of the attribute, the user cannot distinguish between them. Similarly, after inserting new attributes into the database, two indistinguishable tuples may become distinguishable.

Changing the database schema affects applications of the unique characteristics inference rule. For example, a tuple t has this unique characteristic (Age = 30 $\wedge$ Sex = 'Female' $\wedge$ Department = 'Sales'). Suppose the attribute Sex is removed, and there are more than one employees in the sales department who are 30 year old. Then, (Age = 30 $\wedge$ Department = 'Sales') is no longer a unique characteristic of the tuple t. Similarly, inserting

new attributes may introduce new unique characteristics in the database.

Modification of the database schema has an effect on applications of the split query inference rule. Suppose there is a query Q_1 with the selection criteria $SC_1 = (A_1 = a_1 \wedge A_2 = a_2)$. If the attribute A_1 is removed from the database, then it is impossible to determine if some return tuples of a newly issued query, say Q_2, satisfy SC_1, as Q_2 cannot project the attribute A_1. Hence, the split query inference rule cannot be applied to split Q_2 with respect to Q_1. Modification of the database schema also affect applications of the functional dependency inference rule, as they may remove existing functional dependencies or introduce new functional dependencies.

Therefore, in general, inserting attributes into the database schema might lead to more inferences, while deleting attributes from the database schema might result in a smaller number of inferences.

5.2.2 Effects of Modifications of Database Instances on Inference Rules

In this section, we discuss the effects of changing database instances on the applications of inference rules. The database instances are changed using the insertion, deletion, and updating operations on tuples. Changing the database instances has no effect on the applications of the functional dependency inference rule. Functional dependencies are determined by the semantics of the application domains, and not by the contents of the database. If job title functionally determines salary, then this dependency holds no matter what kind of data are stored in the database. Hence, in the following sections, we only consider the effects of changing database instances on the applications of other inference rules.

Suppose an update operation is performed on a database. If a user is not aware of the update operation, the user might fail to apply inference rules to perform inference. However, if the inference detection system assumes the user is aware of the existence of the update operation, the detection system might produce false positive reports. Also, it is possible that if a user is not aware of the existence of an update operation, the user might falsely apply inference rules that are not applicable, hence introducing a false inference. The false inference might be revoked later as the user learns more about the database.

This shows that it is necessary for the detection system to determine if a user is able to identify update operations. Note that a user who can insert or delete a tuple does not imply the user can also access all attribute values of the tuple. It is possible that a

user is authorized to delete a tuple based on, say, the primary key of the tuple, while the user is not allowed to access other attribute values of the tuple. Similarly, a user might be granted the right to insert a tuple with primary key value provided. Other attributes could be inserted later by other users.

Given two queries Q_1 and Q_2, where Q_1 is issued to the database before Q_2. Suppose there is an insert, delete or update operation executed between the issues of these two queries. We can divide the tuples in these two queries into four groups:

1. tuples that are returned by both Q_1 and Q_2.
2. tuples that are returned by Q_1 but not by Q_2.
3. tuples that are returned by Q_2 but not by Q_1.
4. tuples that are not returned by Q_1 or by Q_2.

A new tuple can be inserted into one of these four groups of tuples. A tuple can be deleted from one of these four groups of tuples. A tuple can be modified so that it moves from one of these four groups to another, or the modified tuple might remain in the same group before or after the modification. We discuss each of these cases in the following sections. Modifying a tuple t into t' is equivalent to deleting the tuple t and inserting the tuple t'. Hence, we only discuss the effects of insertions and deletions of tuples.

The applications of the subsume, overlapping and complementary inference rules depend on the subsume relationships among queries. Hence, we only consider the effects of modifying database instances on the applications of the subsume inference rule. The effects of updating on the unique characteristic inference rule is simple. Suppose a tuple t_1 has a unique characteristic C_1. If an inserted tuple t also satisfies C_1, t_1 no longer has the unique characteristic C_1. If t_1 and t_2 are the only two tuples that satisfy the condition C, then after t_1 is deleted, t_2 has the unique characteristic C. The effects of updating on the split query inference rule is similar the effects on the subsume inference rule. We omit the discussion of the effects on the split query inference rule.

Effects of Insertion on Subsume Inference Rule

In this section, we consider the effects of inserting a new tuple t on the applications of the subsume inference rule. Note that if no subsume relationship holds between Q_1 and

Q_2 before the insertion of a tuple t, the subsume relationship does not hold after the insertion of t. This is because if, for example $Q_1 \not\sqsubset Q_2$ holds before the insertion of the tuple t, then there exists a return tuple t' of Q_1 and a return tuple t'' of Q_2, such that t' and t'' do not relate to each other. After the insertion of t, the tuples t' and t'' still exist, and hence $Q_1 \not\sqsubset Q_2$ still holds. That is, the subsume inference cannot be applied to these two queries before or after the insertion of t. Hereafter, we only consider the cases where the subsume relationship holds between two queries before the insertion of a tuple.

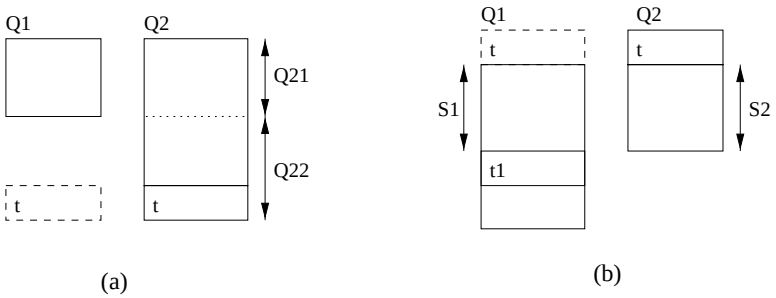

Figure 5.3: An Example on the Effect of Insertion on Subsume Inference Rule. The inserted tuple satisfies both SC_1 and SC_2.

Suppose the inserted tuple t satisfies both SC_1 and SC_2. As t is inserted after Q_1 is issued, t is returned by Q_2 but not by Q_1. There are two cases to be considered. In the first case, as shown in Figure 5.3(a), $Q_1 \sqsubset Q_2$ holds before the insertion of t. The dotted box in the figure indicates the inserted tuple that should be returned by Q_1 if it is issued after t is inserted. After the insertion of t, $Q_1 \sqsubset Q_2$ still holds, and the user can apply the subsume inference rule to identify the related tuples between Q_1 and Q_2. However, if the user does not notice that t should also be one of the return tuples of Q_1, the user might generate incorrect inferred queries. Specifically, the user might generate the following two inferred queries: 1) $Q_{21} = (AS_2; SC_1 \wedge SC_2)$ which does not include the inserted tuple t when it should; and 2) $Q_{22} = (AS_2; SC_2 \wedge \neg SC_1)$ which includes the inserted tuple t when it should not. Unless the system detects that the user has identified the inserted tuple, the detection system should not generate these two incorrect inferred queries.

In the other case, as shown in Figure 5.3(b), $Q_2 \sqsubset Q_1$ holds before the insertion of t. After the insertion of t, $Q_2 \sqsubset Q_1$ does not hold. However, if a user is not aware of the insertion of t, the user might still apply the subsume inference rule to Q_1 and Q_2. This might result in false inference. For example, if the inserted tuple t of Q_2 is indistinguishable

from a return tuple t_1 of Q_1, then the user might falsely conclude that the two tuples relate to each other. Other than this possible false inference, the user can still apply the subsume inference rule to identify the related tuples between Q_1 and Q_2, that is, between the sets of tuples as indicated by S_1 and S_2 in Figure 5.3(b).

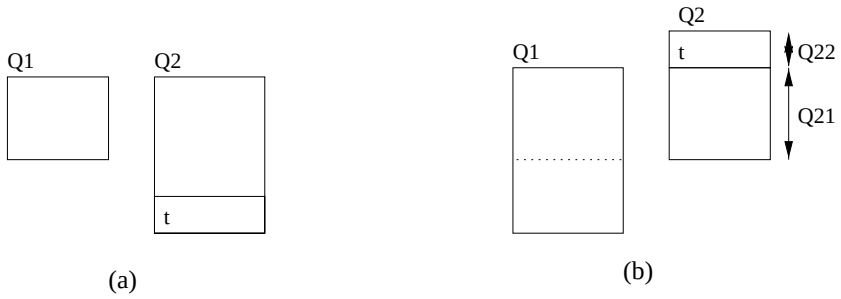

Figure 5.4: An Example on the Effect of Insertion on Subsume Inference Rule. The inserted tuple satisfies SC_2 but not SC_1.

Suppose the inserted tuple t does not satisfy SC_1, but satisfies SC_2. If $Q_1 \sqsubset Q_2$ holds before the insertion of t, as shown in Figure 5.4(a), then $Q_1 \sqsubset Q_2$ still holds after the insertion of the tuple t. The user can apply the subsume inference rule to these two queries. Consider the case where $Q_2 \sqsubset Q_1$ holds before the insertion of t, as shown in Figure 5.4(b). If $SC_2 \Rightarrow SC_1$, then it is impossible to have such an inserted tuple t that satisfies SC_2, but not SC_1. If $Q_2 \sqsubset Q_1$ holds because each return tuple of Q_2 satisfies SC_1, then it is possible to have such an inserted tuple t. In this case, the user can use the split query inference rule to generate two inferred queries: 1) Q_{21} which includes all return tuples of Q_2 except t; and 2) Q_{22} which includes t only. Now, $Q_{21} \sqsubset Q_1$, and the user can apply the subsume inference rule to Q_1 and Q_{21} to find out the related tuples among them.

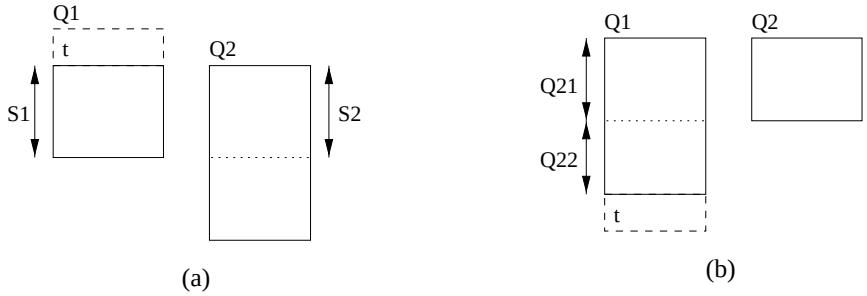

Figure 5.5: An Example on the Effect of Insertion on Subsume Inference Rule. The inserted tuple satisfies SC_1 but not SC_2.

Consider the case where the inserted tuple t satisfies SC_1, but does not satisfy SC_2. Suppose $Q_1 \sqsubset Q_2$ holds before the insertion of t. After the insertion of t, $Q_1 \sqsubset Q_2$ does not hold as there is a return tuple t of Q_1 that does not satisfy SC_2. This is shown in Figure 5.5(a). However, whether a user can identify the inserted tuple or not, the user can apply the subsume inference rule to Q_1 (without the inserted tuple t) and Q_2 to identify the related tuples between the return tuples of Q_1 (except t) and the return tuples of Q_2; that is, between the sets of tuples S_1 of Q_1 and S_2 of Q_2, as indicated in Figure 5.5(a). If $Q_2 \sqsubset Q_1$ holds before the insertion of t, $Q_2 \sqsubset Q_1$ still holds after the insertion. This is shown in Figure 5.5(b). The subsume inference rule can be applied as usual. However, if a user does not identify the tuple inserted to Q_1, the user might generate this incorrect inferred query $Q_{22} = (AS_1; SC_1 \wedge \neg SC_2)$ which does not include the inserted tuple t when it should be included.

For the case where t does not satisfy both SC_1 and SC_2, it does not affect the subsume relationship between Q_1 and Q_2, and hence there is no effect on the application of the subsume inference rule on them.

In summary, no matter how a tuple is inserted into a database, we can still apply the subsume inference rule to the queries issued before or after the insertion of the tuple. However, if the user does not identify the existence of the inserted tuple, the user might generate incorrect inferred queries.

Effects of Deletion on Subsume Inference Rule

In this section, we discuss the cases where a tuple t is deleted between the issues of two queries, Q_1 and Q_2.

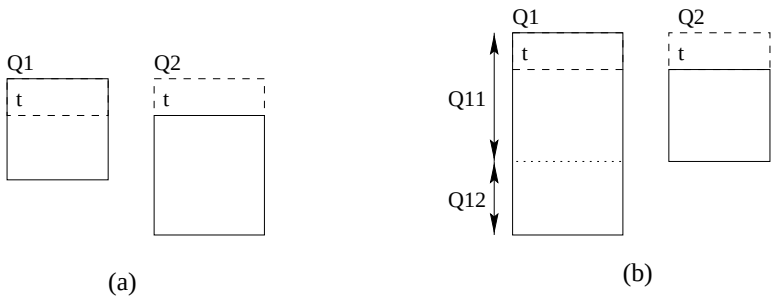

Figure 5.6: An Example on the Effect of Deletion on Subsume Inference Rule. The deleted tuple satisfies both SC_2 and SC_1.

Suppose the deleted tuple t satisfies both SC_1 and SC_2. If $Q_1 \sqsubset Q_2$ holds before the deletion of t, $Q_1 \sqsubset Q_2$ does not hold after t is deleted from Q_2 only. This is shown in Figure 5.6(a). The user needs to know that t should also be removed from the set of return tuples of Q_1, before the user can apply the subsume inference rule to the two queries. If $Q_2 \sqsubset Q_1$ holds before the deletion of t, $Q_2 \sqsubset Q_1$ still holds. This is shown in Figure 5.6(b). In this case, the subsume inference rule can be applied to Q_1 and Q_2. However, if the user does not identify the deleted tuple from Q_1, then the user might generate this incorrect inferred query $Q_{11} = (AS_1 : SC_1 \wedge SC_2)$ which includes the deleted tuple t when it should not.

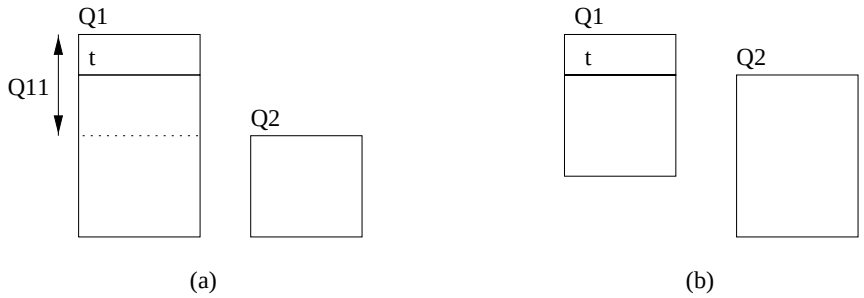

Figure 5.7: An Example on the Effect of Deletion on Subsume Inference Rule. The deleted tuple satisfies SC_1 but not SC_2.

Suppose the deleted tuple t satisfies SC_1 but not SC_2. If $Q_2 \sqsubset Q_1$ holds before the deletion of t, $Q_2 \sqsubset Q_1$ still holds after the deletion of t, and the subsume inference rule can be applied to Q_1 and Q_2. This is shown in Figure 5.7(a). However, if the user does not identify that t should be removed from the set of return tuples of Q_1, the user might generate this incorrect inferred query $Q_{11} = (AS_1; SC_1 \wedge \neg SC_2)$ which includes the deleted tuple when it should not. When such a deleted tuple t exists, $Q_1 \sqsubset Q_2$ can not be held before the deletion of t. However, if the user can identify the deleted tuple in Q_1, it is possible to have a subsume relationship between the two queries. For example, consider the two queries as shown in Figure 5.7(b). If the user can deduce that t is no longer a return tuple of Q_1 when Q_2 is issued, $Q_1 \sqsubset Q_2$ holds, and the user can apply the subsume inference rule to Q_1 and Q_2.

Suppose the deleted tuple t satisfies SC_2 but not SC_1. When a user issues the query Q_2, t has already been removed from the database, and t is not returned by Q_1. Hence, the deletion of the tuple t has no effect on the application of the subsume inference

rule to these two queries. If the deleted tuple t does not satisfy both SC_1 and SC_2, then there is no effect on the application of the subsume inference rule on Q_1 and Q_2.

As discussed in the last two sections, the inference detection system needs to detect if the user can identify the inserted or deleted tuple. There are several ways a user can deduce if a tuple is deleted or inserted. The user can issue a query twice and compare the results of the two queries. If at the first time the query returns a tuple that is not returned the second time, the tuple is deleted. If at the first time the query does not return a tuple that is returned at the second tiem, the tuple is inserted. Also, if a user knows that some tuples satisfying a condition C are deleted, the user can deduce if a tuple t is deleted by checking if t satisfies C. In a conservative approach, the detection system might assume users are aware of all the insertions or deletions of tuples that have occurred, and generate as many inferences as possible from a set of user queries.

5.3 Multiple Tables and Nested Queries

In the presentation of our inference rules in Chapter 3, we assume that there is only a single table in the database. This simplifies the development of the inference rules. However, in practice, most databases consist several tables. In this section, we discuss the modification to the inference detection system when a query accesses more than one table in the database. Queries access multiple tables using join operations or nested queries. In the following discussion, we denote T_i as a table in the database.

5.3.1 Join Operations

In this section, we consider a query that joins two tables together using equijoin operations. There are three cases to be considered: join columns from both tables are projected by the query, no join column is projected by the query, and join columns from one of the tables are projected by the query. Consider a query that performs an equijoin operation on two tables, T_1 and T_2, as follows,

```
select  T1.A11, T1.A12, T2.A22
from    T1, T2
where   T1.A11 = T2.A21
```

where both join columns, A_{11} and A_{21}, are projected by the query. In this case, we can treat the join query as two separate queries as follows,

```
select  A11, A12
from    T1
where   A11 IN (select  A21
                from    T2);
```

and

```
select  A21, A22
from    T2
where   A21 IN (select  A11
                from    T1);
```

The inference detection system can then process these two queries separately. The return tuples of the first query are added into a table, say T_{u1}, in USER_VIEW, and the return tuples of the second query are added into another table, say T_{u1}, in USER_VIEW. As the join column appears in both T_{u1} and T_{u1}, we can determine the related tuples among the tuples in the two tables, and, hence, reconstruct the tuples as returned by the original join query.

Consider the case where no join column is projected by the query, as in the following query,

```
select  T1.A12, T2.A23
from    T1, T2
where   T1.A11 = T2.A22
```

The inference detection system maintains a table, say T_u, in USER_VIEW, storing the return tuples of the query, together with the join condition of the query. With this information, the detection system can make the following inference. Whenever it is known that a tuple, say t, is found to be related to a tuple t_1 in T_1 with $t_1[A_{11}] = a_{11}$, then t also relates to a tuple t_2 in T_2 with $t_2[A_{22}] = a_{11}$.

Consider the case where the join query projects attributes from one of the two join tables, say T_1. In this case, simply based on the return tuples of this join query, the user cannot determine the tuples in T_2 that relate to the return tuples of the query. However, the user can deduce that there exist some tuples in T_2 that satisfy the join condition in the query and relate to the return tuples of the query. For example, consider the following query,

```
select  T1.A12, T1.A13
from    T1, T2
where   T1.A11 = T2.A21 and T2.A22 = 100, and T2.A23 = 50;
```

For each return tuple t_1 of this query, where $t_1[A_{11}] = a$, the user can infer that there exists a tuple t_2 in the table T_2, such that $t_2[A_{21}] = a$, $t_2[A_{22}] = 100$, and $t_2[A_{23}] = 50$. The tuple t_2 is added into USER_VIEW together with return tuples of the query. The existence of a tuple is useful in inference. For example, suppose the user finds out that there exists a tuple t_3 in T_2, where $t_3[A_{22}] = 100$, and that $(A_{22} = 100)$ is a unique characteristic in T_2, then the user can infer that $t_3[A_{23}] = 50$.

When a protected association includes attributes from multiple tables, the detection system needs to determine if the user can deduce the related tuples among the tables. This is done by checking if the protected association is revealed by joining the the corresponding tables in USER_VIEW. For example, consider this policy

$(U; Name, Salary; true)$

where the salaries of employees are protected. Suppose the database has two tables with the following schema: (Name, Job-title) and (Job-title, Salary). Let the USER_VIEW(U) be as follows,

Name	Job-title
John	Software Engineer
Bill	Sales Representative

Job-title	Salary
Software Engineer	60K
Administrative Assistant	40K

Suppose Job-title is the primary key in the table (Job-title, Salary). By joining these two tables over the attribute Job-title, the system can determine that the policy is violated.

5.3.2 Nested Queries

A nested query is a query containing a subquery. The *where* clause of a nested query is in the following BNF form,

expression [IN | NOT IN | EXISTS | op [ALL | ANY]]$^+$ subquery

where *op* is one of the comparison operators '<', '$\leq$', '=', '$\geq$', '>', or '<>'. For nested queries with the IN operators, they can be transformed into equijoin queries and can be processed as discussed in the above section. For example, the following two queries are equivalent [Kim82],

```
select   A11
from     T1
where    A12 IN (select   A21
                 from     T2
                 where    T1.A13 = T2.A22)
```

and

```
select   A11
from     T1, T2
where    T1.A12 = T2.A21 AND T1.13 = T2.A22
```

Nested queries with comparison operators can also be transformed into equijoin queries. The following two queries are equivalent [Kim82]. Note that in this form of nested queries, the subquery must return a single value.

```
select   A11
        from     T1
        where    A12 op (select   A21
                         from     T2
                         where    T1.A13 = T2.A22)
```

and

```
select   A11
from     T1, T2
where    T1.A12 op T2.A21 AND T1.A13 = T2.A22
```

Consider a subquery with the EXISTS operator. First, consider a nested query with no correlated conditions; that is, the subquery of the query does not refer the table in the query. Consider the following query,

```
select   A11
from     T1
where    EXISTS (select    *
                 from      T2
                 where     A21 = 100);
```

If this query has one or more return tuples, then the user can infer that there is at least one tuple in T_1 that satisfies the condition in the subquery. In this example, we can create a tuple in the USER_VIEW to indicate that there is a tuple t, where $t[A_{21}] = 100$. Note that there may be more than one tuple in T_2 that satisfies the subquery condition. If the query returns no tuple, the user can deduce that there is no tuple in T_2 that satisfies the where condition of the subquery. Consider a correlated subquery with the EXISTS operator.

```
select   A11
from     T1
where    EXISTS (select    *
                 from      T2
                 where     T1.A12 op T2.A21);
```

For each return tuple t_1 of this query, the user can infer that there exists a tuple t_2 from T_2 such that $t_1[A_{12}]$ *op* $t_2[A_{21}]$ holds. If there is no tuple returned by this query, then the user can infer that for each tuple t_1 of T_1 and a tuple t_2 of T_2, $t_1[A_{12}]$ $\neg op$ $t_2[A_{21}]$. We can update the USER_VIEW accordingly.

Consider the following query without a correlated subquery, with an '> ANY' operator in the where clause of the query,

```
select   A11
from     T1
where    A12 > ANY (select    A21
                    from      T2
                    where     A22 = 100);
```

If the user knows the results of the subquery, then the user can infer that for each return tuple t_1, $t_1[A_{12}]$ is greater than the minimum of the set of return values of the subquery. Suppose the result of the subquery is not know to the user. If $t_1[A_{12}]$ is known for each return tuple t_1 of the query, then the user can infer that there is a return tuple t_2 of the subquery such that $t_2[A_{21}] < t_1[A_{12}]$. This information can be used to perform inference. For example, if there are n number of return tuples of the subquery, and $t_2[A_{21}] \geq t_1[A_{12}]$ for each return tuple t_2 of the subquery except t_2', then the user can conclude that $t_2'[A_{21}]$ $< t_1[A_{12}]$ for each return tuple t_1 of the query.

Consider the following query with correlated subquery, with an '> ANY' operator in the where clause of the query,

```
select   A11
from     T1
where    A12 > ANY (select    A21
                    from      T2
                    where     T1.A13 = T2.A22);
```

For each return tuple t_1 of this query, there exists a tuple t_2 of T_2, such that $t_1[A_{13}] = t_2[A_{22}]$ and $t_1[A_{12}] > t_2[A_{21}]$. If the user knows about the values of $t_1[A_{13}]$ and $t_1[A_{12}]$ but not the subquery result, then the user can infer that there exists a tuple t_2 of T_2, such that $t_1[A_{13}] = t_2[A_{22}]$, and $t_1[A_{12}] > t_2[A_{21}]$. If the user knows about the value of $t_1[A_{13}]$ and the subquery result, then the user can infer that $t_1[A_{12}]$ is greater than one of the return values of the subquery. The cases are similar for subqueries with other comparison operators together with the 'ANY' operator.

Consider the following query without correlated subquery, with an '> ALL' operator in the where clause of the query,

```
select   A11
from     T1
where    A12 > ALL (select    A21
                    from      T2
                    where     A22 = 100);
```

If the subquery result is known to the user, then the user can deduce that for each return tuple t_1 of the query, $t_1[A_{12}]$ is greater than any return value of the subquery. On the other hand, if the subquery result is not known to the user, but the user knows the value of $t_1[A_{12}]$ for some return tuple t_1, then the user can deduce that $t_1[A_{12}]$ is greater than all return values of the subquery.

Consider the following query correlated subquery, with an '> ALL' operator in the where clause of the query.

```
select   A11
from     T1
where    A12 > ALL (select    A21
                    from      T2
                    where     T1.A13 = T2.A22);
```

For each return tuple t_1 of the query, if the user knows about the values of $t_1[A_{12}]$ and $t_1[A_{13}]$, but not the subquery result, then the user can infer that for each tuple t_2 in T_2, where $t_2[A_{22}] = t_1[A_{13}]$, $t_2[A_{21}] < t_1[A_{12}]$. If the user knows about the value of $t_1[A_{13}]$ and the subquery result, then the user can infer $t_1[A_{12}]$ is greater than all return values of the subquery.

5.4 Summary

In this chapter, we extended the security policy to include the specification of approximate inference. We discussed the detection of dynamic inference by propagations through subsume relationships. We investigated the effects of inserting and deleting tuples during a course of an inference. We found out that the detection system needs to determine if the user can identify the inserted or deleted tuples in order to detect all inferences. We also discussed the extension of the detection system to allow more than one tables in the database, and queries with subqueries.

Chapter 6

Conclusions and Future Work

Inference is known as a way to defeat access control mechanisms in database systems. It poses a confidentiality threat, making it difficult to control user accesses to sensitive information. An inference detection system is needed to determine if users can use legitimate data to infer sensitive information. The design of an inference detection system is a trade-off among soundness, completeness, accessibility of the database, and efficiency. Previous approaches have been developed to tackle the inference problem. Each approach is designed to achieve one or two aspects of the four design criteria: completeness, soundness, efficiency, and accessibility of the database.

The schema-based approach uses an efficient method to detect inference based on functional dependencies among attributes in the database schema. Schema redesign can be used to prevent inference, but it might decrease accessibility of the database because of attribute overclassification. The schema-based approach is not sound, as users might be able to make use of the dependencies that although do not occur at the schema level do occur among data in the database to perform inference. This deficiency of the schema-based approach is illustrated by the ability of our data level inference detection system to detect inferences that cannot be detected using the schema-based approach. Other researchers have investigated the extension of the database to include background knowledge in the application domains, making the detection system more complete in detecting inference. Marks [Mar96] developed an inference detection system that is complete. However, it is achieved by reporting inferences whenever a query accesses any portion of sensitive associations. Although the completeness is achieved, the system is not sound, and it leads to lower accessibility of the database.

Most existing inference detection systems are designed for multilevel database systems, where maintaining data confidentiality is important while providing discretionary access control is desirable. Most researchers put emphasis on making the detection systems complete in protecting sensitive information. However, in practice, most database systems are not built using the multilevel secure model. A more important threat to these database systems is the misuse of the database by insiders, that is, by the authorized users in the database systems. A user misuses a database if the user accesses data not for the interests of the organization. For example, a payroll clerk is allowed to access employee salaries in order to generate paychecks for employees. If the clerk accesses salary data that are not for the purposes of generating paychecks, the clerk is misusing the database.

Ideally, we should specify exactly the set of data that a user is authorized to access. However, in most cases, it is almost impossible to specify exactly the set of data a user needs. In commercial database systems, providing high accessibility to the database is important, as unnecessary restrictions on the uses of the database might hinder the work of users. At the same time, we also want to make sure that users are not misusing the database. With such applications in mind, we develop an inference detection system to determine if users have collected enough data to perform inference. Our detection system is sound, and hence it provides high accessibility of the database to users. Our system makes use of the data in the database to detect inferences, making it more complete than the schema-based inference detection approach. In fact, our system can be extended to include the schema-based inference detection approach. We have shown the effectiveness of our data level inference detection system by using the inference rules to detect a known inference attack called Tracker. Such an attack cannot be detected using the schema-based inference inference detection approach.

The contributions we have made in this research include

1. Explored the uses of data in the database to detect subtle inference.
2. Developed sound inference rules to detect inference.
3. Developed inference detection algorithms for which we performed complexity analysis.
4. Implemented a prototype to prove the concept.
5. Identified the characteristics of the databases and queries under which the inference detection systems are practical.

6. Extended the inference detection system to detect approximate inference.

7. Developed an initial solution to the detection of dynamic inference.

Our detection system can be inefficient, as we need to keep track of all queries issued by users, and perform inference detection using these queries. We have carried out a performance evaluation on the prototype that we have developed. The experimental results show that the system performance is affected by the characteristics of the databases and queries. In general, the system runs faster with a larger number of attributes in the database, a smaller number of return tuples of the queries, a larger amount of duplication among data in the database, a smaller number of projected attributes from the queries, and a larger number of conjuncts in the queries. Hence, it shows that the system would be practically employed. We have also found out that it is possible to use simple statistics to estimate the number of inferences drawn by users. The possible statistics include the amount of overlapping among return data, and the number of database attributes that have been retrieved by users.

Different inference detection approaches solve different aspects of the inference problem. We might use different approaches for different security needs. For example, we could use an inference detection system that is complete when protecting the sensitive data is important, whereas we could use an inference detection system that is sound when providing high accessibility of the database is important. Alternatively, we can develop a system that has merits from all these approaches. Indeed our approach can be used together with other approaches. For example, we first apply the schema-based approach to make sure that all inferences detectable at the database schema are detected. Then, we extend the database to include background knowledge in the application domains so as to detect as many inferences as possible. After that, we can apply our data level inference detection system to determine if a set of user queries can lead to inference.

6.1 Future Work

In this section, we discuss three possible directions to extend our work. We can extend the inference detection system to perform misuse detection. For example, a user is suspicious if he performs the following actions,

- Accesses a portion of a sensitive association in the database.

- Accesses an area of the database which the user normally does not access.
- Accesses a large amount of data from the database.
- Issues the same query repeatedly, which might indicate that the user is interested in observing changes to the database.
- Issues refined queries to request more specific information.

Our inference detection system maintains the data structure USER_VIEW(U) to represent the set of information that a user U can infer about the database. This data structure can be used to detect the first three types of misuse behavior. We can check the last two types of misuse behavior by determining if user queries significantly overlap, or they have a large number of subsume relationships among them.

In misuse detection, we need to define the normal access pattern of users. This access pattern can then be used to determine if users' actions deviate from the normal behavior. The deviation might indicate that the user is intentionally misusing the database, or the user account has been compromised and an attacker is masquerading as the user to access the database. A way to obtain the normal access pattern is to have a supervised session on the uses of the database for each user. We input this set of supervised user queries into the inference detection system to generate a USER_VIEW that represents the normal access pattern of the user. During the normal operations of the database, we collect the user queries and generate a new USER_VIEW. A misuse might occur when the generated USER_VIEW is different from the one that corresponds to the normal access pattern.

We can also use heuristic approaches to detect inference. From our experiments we found that more inferences occur when the amount of duplication of the attribute values decreases. We can preprocess the database and determine the area of the database where the attribute values are relatively unique in the database. When a user accesses data in such an area of the database, it might be an indication that the user is trying to perform inference on those data. Suppose there are a few vice-presidents in a company. When a user is accessing data about the vice-presidents, there is a good chance that the user is attempting to infer information about the vice-presidents. The detection system might generate a report to indicate that it is worth doing a more exhaustive detection of inference using those suspicious user queries.

Another possible extension is to explore parallelism in applying the inference rules. We can speed up the detection system by parallelizing the inference detection process. Due to their monotonic property, the inference rules can be applied in any order. Indeed, we can apply the inference rules in parallel. In detecting inference, we need to apply repeatedly the inference rules until no new inference occurs. This is illustrated in Figure 6.1.

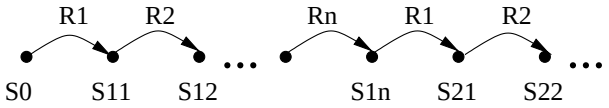

Figure 6.1: Applications of Inference Rules in Serial.

S_i are the states of the USER_VIEW. R_i are the inference rules. There are n inference rules. At state S_0, we apply the inference rule R_1 to change the state to S_{11}. R_2 changes the database state from S_{11} to S_{12}, and so forth. As the inference rules can be applied in any order, we can process them in parallel as shown in Figure 6.2.

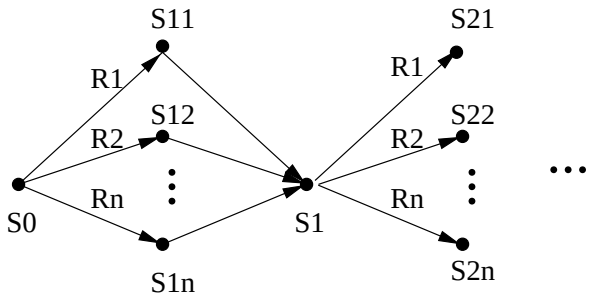

Figure 6.2: Applications of Inference Rules in parallel.

R_1 transforms the database from state S_0 to S_{11}. R_2 transforms the database from state S_0 to S_{12}, and so forth. After the n inference rules are applied to the state S_0, we combine all the resulting states S_{11}, S_{12}, ..., and S_{1n} into a single state S_1. This is done by taking a union of the USER_VIEWs from these n states. If S_1 is different from S_0, then the n inference rules are applied again in parallel. This process is repeated until the new state is the same as the previous state. Hence, there are two stages in this process: applications of the inference rules, and synchronization of the states resulting from the applications of the n inference rules.

The inference detection system that we have developed is designed for relational database systems. We could extend the system to handle databases that are constructed

using other data models such as hierarchical, network, and object-oriented data models. In most data models, there is the concept of associations among data. For example, in object-oriented data models, the attributes of an object associate with the unique identification of the object. Also, one object may have a domain-specific relationship with other objects. Each query reveals certain amount of associations in the database. Hence, it is possible to record the associations that each query reveals, and then determine the subsume relationships among the query results. Another way to approach this problem is to transform other data models into a relational model. We can transform both queries, and query results from one data model into the relational data model. Then, we can apply our inference detection system to the transformed data to detect inference.

Inference detection is a hard problem. On the one hand, it is difficult to represent all information available to users, and, on the other hand, it is difficult to simulate the complete reasoning strategies of users. However, this is an important problem as it defeats the access control system of a database system. This research developed techniques to detect subtle inferences using data in the database. More research is needed to develop, if possible, a complete, sound, and efficient detection system that also provides high accessibility of data to users.

Bibliography

[AW89] N. R. Adam and J. C. Wortmann. Security-control methods for statistical databases: A comparative study. *ACM Computing Surveys*, 21(4):515–556, December 1989.

[Bib77] K. J. Biba. Integrity considerations for secure computer systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Mass, April 1977.

[Bin93a] L. J. Binns. Inference and cover stories. In Bhavani M. Thuraisingham and Carl E. Landwehr, editors, *Database Security VI: Status and Prospects*, pages 169–178. North-Holland, 1993.

[Bin93b] L. J. Binns. Inference through secondary path analysis. In Bhavani M. Thuraisingham and Carl E. Landwehr, editors, *Database Security VI: Status and Prospects*, pages 195–209. North-Holland, 1993.

[Bin94] L. J. Binns. Implementation considerations for inference detection: Intended vs. actual classification. In Thomas F. Keefe and Carl E. Landwehr, editors, *Database Security VII: Status and Prospects*, pages 297–306. North-Holland, 1994.

[BL73] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, MITRE Corp., Bedford, Mass, May 1973.

[Buc90] L. J. Buczkowski. Database inference controller. In David L. Spooner and Carl E. Landwehr, editors, *Database Security III: Status and Prospects*, pages 311–322. Elsevier Science Pub. Co., 1990.

[Bur95] R. K. Burns. Inference analysis during multilevel database design. In Steven A. Demurjian David L. Spooner and John E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 301–316, 1995.

[CM98] L. Chang and I. S. Moskowitz. Bayesian methods applied to the database inference problem. In *Database Security XII: Status and Prospects*, 1998.

[CS95] F. Chen and R. S. Sandhu. The semantics and expressive power of the mlr data model. In *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, pages 128–142. IEEE Computer Society Press, 1995.

[DDS79] D. E. Denning, P. J. Denning, and M. D. Schwartz. The tracker: A threat to statistical database security. *ACM Transactions on Database Sysetms*, 4(1):76–96, March 1979.

[Den86] D. E. Denning. A preliminary note on the inference problem. In *Proceedings of the National Computer Security Center Invitational Workshop on Database Security*, June 1986.

[DH96] H. S. Delugach and T. H. Hinke. Wizard: A database inference analysis and detection system. *IEEE Transactions on Data and Knowledge Engineering*, 8(1):56–66, 1996.

[DLS+87] D. E. Denning, T. F. Lunt, R. R. Schell, M. Heckman, and W. Shockley. A multilevel relational data model. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 220–234. IEEE Computer Society Press, 1987.

[DLS+88] D. E. Denning, T. F. Lunt, R. R. Schell, W. R. Shockley, and M. Heckman. The seaview security model. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 218–233. IEEE Computer Society Press, 1988.

[DS80] D. E. Denning and J. Schlorer. A fast procedure for finding a tracker in a statistical database. *ACM Transactions on Database Sysetms*, 5(1):88–102, March 1980.

[GL92] T. D. Garvey and T. F. Lunt. Cover stories for database security. In Carl E. Landwehr and Sushil Jajodia, editors, *Database Security V: Status and Prospects*, pages 363–380. Elsevier Science Pub. Co., 1992.

[GLQS93] T. D. Garvey, T. F. Lunt, X. Qian, and M. E. Stickel. Toward a tool to detect and eliminate inference problems in the design of multilevel databases. In Bhavani M. Thuraisingham and Carl E. Landwehr, editors, *Database Security VI: Status and Prospects*, pages 149–167. North-Holland, 1993.

[GLS91] T. D. Garvey, T. F. Lunt, and M. E. Stickel. Abductive and approximate reasoning models for characterizing inference channels. In *The 4th Computer Security Foundations Workshop*, pages 118–126, 1991.

[GW77] P. P. Griffiths and B. W. Wade. An authorization mechanism for a relational database system. *ACM Transactions on Database Systems*, 1(3):242–255, September 1977.

[HDC94] T. H. Hinke, H. S. Delugach, and A. Chandrasekhar. Layered knowledge chunks for database inference. In Thomas F. Keefe and Carl E. Landwehr, editors, *Database Security VII: Status and Prospects*, pages 275–295. North-Holland, 1994.

[HDW95] T. H. Hinke, H. S. Delugach, and R. P. Wolf. Iliad: An integrated laboratory for inference analysis and detection. In Steven A. Demurjian David L. Spooner and John E. Dobson, editors, *Database Security IX: Status and Prospects*, pages 333–348, 1995.

[HDW97] T. H. Hinke, H. S. Delugach, and R. Wolf. A framework for inference-directed data mining. In Pierangela Samarati and Ravi S. Sandhu, editors, *Database Security X: Status and Prospects*, pages 229–239. Chapman and Hall, 1997.

[Hin88] T. H. Hinke. Inference aggregation detection in database management systems. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 96–106. IEEE Computer Society Press, 1988.

[Hin89] T. H. Hinke. Database inference engine design approach. In Carl E. Landwehr, editor, *Database Security II: Status and Prospects*, pages 247–262. Elsevier Science Pub. Co., 1989.

[HOST90] J. T. Haigh, R. C. O'Brien, P. D. Stachour, and D. L. Toups. The ldv approach to database security. In David L. Spooner and Carl E. Landwehr, editors, *Database*

Security III: Status and Prospects, pages 323–339. Elsevier Science Pub. Co., 1990.

[HS97] J. Hale and S. Shenoi. Catalytic inference analysis: Detection inference threats due to knowledge discovery. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 188–199. IEEE Computer Society Press, 1997.

[JS90] S. Jajodia and R. Sandhu. Polyinstantiation integrity in multilevel relations. In *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, pages 104–115. IEEE Computer Society Press, 1990.

[Kim82] W. Kim. On optimizing an sql-like nested query. *ACM Transactions on Database Systems*, 7(3):443–469, 1982.

[Lin93] T. Y. Lin. Inference secure multilevel databases. In Bhavani M. Thuraisingham and Carl E. Landwehr, editors, *Database Security VI: Status and Prospects*, pages 317–332. North-Holland, 1993.

[Lun89] T. F. Lunt. Aggregation and inference: Facts and fallacies. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 102–109. IEEE Computer Society Press, 1989.

[Mar96] D. G. Marks. Inference in mls database systems. *IEEE Transactions on Knowledge and Data Engineering*, 8(1):46–55, February 1996.

[MMJ94] A. Motro, D. G. Marks, and S. Jajodia. Aggregation in relational databases: Controlled disclosure of sensitive imformation. In *Proceedings of the Third European Symposium on Research in Computer Security*, pages 431–445, November 1994.

[Mor88] M. Morgenstern. Controlling logical inference in multilevel database systems. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pages 245–255. IEEE Computer Society Press, 1988.

[Qia94] X. Qian. Inference channel-free integrity constraints in multilevel relational databases. In *Proceedings of the 1994 IEEE Symposium on Security and Privacy*, pages 158–167. IEEE Computer Society Press, 1994.

[Qia96] X. Qian. View-based access control with high assurance. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 85–93. IEEE Computer Society Press, 1996.

[QSK+93] X. Qian, M. E. Stickel, P. D. Karp, T. F. Lunt, and T. D. Garvey. Detection and elimination of inference channels in multilevel relational database systems. In *Proceedings of the 1993 IEEE Symposium on Security and Privacy*, pages 196–205. IEEE Computer Society Press, 1993.

[SGLQ94] M. E. Stickel, T. D. Garvey, T. F. Lunt, and X. Qian. Inference channel detection and elimination in knowledge-based systems. Technical report, SRI International, Octobet 1994.

[SKN89] X. Sun, N. N. Kamel, and L. M. Ni. Processing implication on queries. *IEEE Transactions on Software Engineering*, 15(10):1168–1175, October 1989.

[SO87] T. A. Su and G. Ozsoyoglu. Data dependencies and inference control in multilevel relational database systems. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pages 202–211. IEEE Computer Society Press, 1987.

[SO91] T. A. Su and G. Ozsoyoglu. Controlling fd and mvd inferences in multilevel relational database systems. *IEEE Transactions on Data and Knowledge Engineering*, 3(4):474–485, 1991.

[Sti94] M. E. Stickel. Elimination of inference channels by optimal upgrading. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 168–174. IEEE Computer Society Press, 1994.

[Thu92] B. Thuraisingham. The use of conceptual structures for handling the inference problem. In Carl E. Landwehr and Sushil Jajodia, editors, *Database Security V: Status and Prospects*, pages 333–362. Elsevier Science Pub. Co., 1992.

[YL98a] R. Yip and K. Levitt. Data level inference detection in database systems. In *The 11th IEEE Computer Security Foundations Workshop*, pages 179–189, June 1998.

[YL98b] R. Yip and K. Levitt. The design and implementation of a data level database inference detection systems. In *The 12th Annual IFIP Working Conference on Database Security*, July 1998.

Appendix A

Sample Experimental Database and Queries

In this section, we show a sample database and sample queries that we used in running our experiments. We generate the following database, called personnel, using these three parameter values: $N_{rec\_num} = 15$, $N_{attr} = 8$, and $N_{data\_dist} = 50\%$.

SSN	Dept	Job	Year	Sale1	Sale2	Sale3	Sale4
1	3	6	2	2	1	6	6
2	3	4	0	0	4	5	0
3	1	6	6	5	2	3	0
4	4	4	3	2	2	5	5
5	3	0	6	0	6	2	5
6	3	3	2	4	0	4	4
7	5	0	4	6	2	2	6
8	5	5	4	0	0	0	0
9	3	2	4	6	0	5	0
10	1	1	1	5	0	1	2
11	2	4	1	2	1	0	3
12	1	0	7	6	3	2	3
13	6	4	7	2	0	0	6
14	2	6	4	7	2	1	0
15	7	6	3	6	5	6	3

$N_{rec\_num} = 15$ means that the database has 15 tuples. $N_{attr} = 8$ means that the database has 8 attributes. and $N_{data\_dist} = 50\%$ means that the data values (except the primary key values) in the database are uniformly distributed between 0 and $(N_{rec\_num} \times N_{data\_dist}) =$

$(15 \times 0.5) = 7.5$. Hence, all data values in the database, except those for attribute SSN, take values between 0 and 7. Note that all data values are of integer types. The data values for the attribute SSN are assigned consecutively starting from 1.

We generate the following five queries using these three parameter values: $N_{ret\_tuple} = 25$, $N_{proj} = 4$, $N_{cond} = 3$.

```
SELECT  Sale3, Year, Job, Sale2
FROM    personnel
WHERE   Dept <= 5 and Sale2 >= 0 and Sale3 >= 0;

SELECT  Dept, Sale2, SSN, Sale1
FROM    personnel
WHERE   Sale4 <= 6 and Sale2 < 3 and Job >= 3;

SELECT  Sale3, Sale4, Job, Sale1
FROM    personnel
WHERE   Year >= 3 and SSN <= 14 and Job >= 1;

SELECT  Sale3, SSN, Sale1, Job
FROM    personnel
WHERE   SSN < 14 and Sale3 >= 0 and Sale3 < 3;

SELECT  Sale3, SSN, Sale4, Dept
FROM    personnel
WHERE   Sale2 <= 7 and Dept >= 2 and SSN >= 6;
```

$N_{proj} = 4$ means that each query projects 4 attributes from the database. $N_{cond} = 3$ means that the where clause of each query consists 3 conjuncts. $N_{ret\_tuple} = 25$ means that the number of return tuples of each query falls between (N_{ret_tuple} - 20) and (N_{ret_tuple} + 20), or between 5 and 45. As $N_{rec\_num} = 15$, the range becomes between 5 and 15. The numbers of tuples returned by these five queries are 13, 8, 6, 7, and 8.

Appendix B

Sample Sessions with the Data Level Inference Detection System

In this section, we present sample sessions on running the prototype of our data level inference detection system. The queries in the sessions are issued to the following table called personnel,

TID	SSN	Name	Department	Job	Salary
1	10	John	3	10	86
2	20	Peter	2	20	80
3	30	Susan	1	20	80
4	40	Paul	2	40	89
5	50	Paul	1	40	86
6	60	Jack	2	50	82
7	70	Jenny	3	50	84
8	80	John	4	30	85
9	90	Dan	3	20	90
10	100	Susan	5	20	88
11	110	Jeff	2	50	94
12	120	Hilary	2	40	96

The database contains the following information about employees: social security numbers (SSN), names, departments (between 1 and 5), jobs between (10 and 50), and salaries. The attribute TID is added in order to assign a unique identification number to each tuple in the database. The TID attribute values are also appended to the query results.

B.1 Subsume Inference Rule

In this session, we illustrate the application of the subsume inference rule. Three queries are issued.

```
Q1: SELECT  SSN, Dept
    FROM    personnel
    WHERE   Salary >= 80 and Salary <= 82;
```

Q_1 returns the following table,

TID	SSN	Dept
2	20	2
3	30	1
6	60	2

The user can identify the department information for the three employees returned by Q_1.

```
Q2: SELECT  Job, Dept
    FROM    personnel
    WHERE   Salary = 80;
```

Q_2 returns the following table,

TID	Job	Dept
2	20	2
3	20	1

As $SC_2 \Rightarrow SC_1$, the subsume inference rule can be applied to Q_1 and Q_2. Tuple 3 of Q_2 is indistinguishable from exactly one return tuple of Q_1, that is tuple 3. Hence, the two return tuples can be expanded with respect to each other. The user can infer that the employee with SSN equals 30 has Job equals 20, and Salary equals 80.

```
Q3: SELECT  Job
    FROM    personnel
    WHERE   SSN = 60;
```

Q_3 returns the following table,

TID	Job
6	50

The return tuple belongs to the employee with SSN equals 60. Since this employee's information is also return in Q_1, the two return tuples can be expanded with respect to each other. That is, the user can infer that the employee with SSN equals 60 has Dept equals 2,

and Job equals 50. After Q_2 is issued, $Q_1 \sqsubset Q_2$, and tuple 2 of Q_2 is indistinguishable from both tuple 2 and tuple 6 of Q_2. After Q_3 is issued, tuple 2 of Q_2 becomes indistinguishable from only tuple 2 of Q_1. Hence, the two return tuples can be expanded with respect to each other. That is, the user can infer that the employee with SSN equals 20 has Dept equals 2, Job equals 20, and Salary equals 80. In this session, there are four inferences using the primary key (3 in Q_1 and 1 in Q_3), and two subsume inferences (1 after Q_2 is issued, and 1 after Q_3 is issued).

B.2 Unique Characteristic Inference Rule

In this session, we illustrate the application of the Unique Characteristic inference rule. Four queries are issued.

```
Q1: SELECT  SSN, Dept, Job
    FROM    personnel
    WHERE   Salary < 85;
```

Q_1 returns the following table,

TID	SSN	Dept	Job
2	20	2	20
3	30	1	20
6	60	2	50
7	70	3	50

From Q_1, the user can identify the department and job information for the four employees.

```
Q2: SELECT  SSN, Job, Dept
    FROM    personnel
    WHERE   Salary >= 92;
```

Q_2 returns the following table,

TID	SSN	Job	Dept
11	110	50	2
12	120	40	2

```
Q3: SELECT  Dept, Job, Salary
    FROM    personnel
    WHERE   Dept >= 2 and Job = 20;
```

Q_3 returns the following table,

TID	Dept	Job	Salary
2	2	20	80
9	3	20	90
10	5	20	88

tuple 2 has Salary equals 80 which satisfies SC_1. That is, tuple 2 must also be selected by Q_1. Tuple 2 of Q_3 is indistinguishable from tuple 2 of Q_1. Hence, the two tuples relate to each other. That is, the user can infer that the employee with SSN equals 20 has salary equals 80.

```
Q4: SELECT  SSN, Job, Dept
    FROM    personnel
    WHERE   Salary >= 84 and Salary < 93;
```

Q_4 returns the following table,

TID	SSN	Job	Dept
1	10	10	3
4	40	40	2
5	50	40	1
7	70	50	3
8	80	30	4
9	90	20	3
10	100	20	5

$SC_1 \vee SC_2 \vee SC_4 \equiv$ true. That is, the three queries together retrieves all the tuples from the database. All the three queries return the attributes Job and Dept. The user can infer that $(Job = 20 \wedge Dept = 3)$ and $(Job = 20 \wedge Dept = 5)$ are two unique characteristics in the database. As tuple 9 of Q_3 satisfies $(Job = 20 \wedge Dept = 3)$, it relates to tuple 9 of Q_4. Also, tuple 10 of Q_3 satisfies $(Job = 20 \wedge Dept = 5)$, and, hence, it relates to tuple 10 of Q_4. Therefore, the user can infer that the employee with SSN equals 90 has salary equals 90, and the employee with SSN equals 100 has salary equals 88.

B.3 Overlapping Inference Rule

In this session, we illustrate the application of the overlapping inference rule. Three queries are issued.

```
Q1: SELECT  SSN
```

```
        FROM    personnel
        WHERE   Salary = 84;
```

Q_1 returns the following table,

TID	SSN
7	70

SSN is the primary key in the database. Hence, the user knows that the employee with SSN equals 70 has salary equals 84.

```
    Q2: SELECT  Name
        FROM    personnel
        WHERE   Salary >= 84 and Salary <= 86;
```

Q_2 returns the following table,

TID	Name
1	John
5	Paul
7	Jenny
8	John

```
    Q3: SELECT  Name
        FROM    personnel
        WHERE   Salary <= 84;
```

Although $SC_1 \Rightarrow SC_2$, the return tuple of Q_1 is indistinguishable from all return tuples of Q_2. Hence, there is no inference occurs. Q_3 returns the following table,

TID	name
2	Peter
3	Susan
6	Jack
7	Jenny

As $SC_1 \Rightarrow SC_2$, and $SC_1 \Rightarrow SC_3$, the only return tuple of Q_1 must relate to a return tuple of Q_2, and a return tuple of Q_3. As Q_2 and Q_3 both project the attribute Name, by comparing the return values of Q_2 and Q_3 over Name, the user can determine that 'Jenny' must be the overlapping attribute value. Three queries are generated

$Q_4 : (Name; Salary \leq 84 \wedge \neg(Salary = 84))$

$Q_5 : (Name; Salary \geq 84 \wedge Salary \leq 86 \wedge \neg(Salary = 84))$

$Q_6 : (Name; Salary \leq 84 \wedge Salary = 84)$

Q_4 returns tuples 2, 3, and 6. Q_5 returns tuples 1, 5, and 8. Q_6 returns tuples 7. As $SC_6 \Rightarrow SC_1$, and both queries return a single tuple, the return tuple of Q_1 relates to the return tuple of Q_6. Expand the two return tuples with respect to each other. That is, the user can infer that the person with SSN = 70 has the name Jenny, and salary equals 84. In this session, there is one inference using the primary key (in Q_1), one subsume inference, and one overlapping inference.

B.4 Complementary Inference Rule

In this session, we illustrate the application of the complementary inference rule. Four queries are issued.

```
Q1: SELECT  SSN
    FROM    personnel
    WHERE   Name = Susan;
```

Q_1 returns the following table,

TID	SSN
3	30
10	100

The user can determine that the employee with SSN equal 30 and the employee with SSN equals 100 are both called Susan.

```
Q2: SELECT  Salary
    FROM    personnel
    WHERE   Name = Susan;
```

Q_2 returns the following table,

TID	Salary
3	80
10	88

As $SC_1 \equiv SC_2$, the user can infer that the two employees who called Susan both have salaries either 80 or 88.

```
Q3: SELECT  SSN
    FROM    personnel
    WHERE   Dept = 1;
```

Q_3 returns the following table,

TID	SSN
3	30
5	50

No new inference occurs after Q_3 is issued.

```
Q4: SELECT  Salary
    FROM    personnel
    WHERE   Dept = 1;
```

Q_4 returns the following table,

TID	Salary
3	80
5	86

The complementary inference rule can be applied to these four queries. This is because 1) $Q_1 \sqsubset Q_2$; 2) $Q_3 \sqsubset Q_4$; 3) Q_1 and Q_3 have common projected attribute, SSN, and the overlapping tuple is identified, namely tuple 3; and 4) Q_2 and Q_4 have common projected attribute, Salary, and the overlapping tuple can be identified, namely tuple 3. After the complementary inference rule is applied, the following two inferred queries are generated,

$Q_5 : (SSN; Dept = 1 \wedge \neg(Name = Susan))$

$Q_6 : (Salary; Dept = 1 \wedge \neg(Name = Susan))$

Q_5 returns the tuple selected by Q_3 but not by Q_1; that is, tuple 5. Q_6 returns the tuple selected by Q_4 but not by Q_2; that is, tuple 5. $SC_5 \equiv SC_6$, and both queries return a single return tuple, hence the two return tuples relate to each other. The user can then infer that the employee with SSN equals 50 has salary equals 86. After this inference, Q_3 and Q_4 become as follows, the table on the left represents Q_3 and the one on the right represents Q_4,

TID	SSN	Salary
3	30	
5	50	86

TID	SSN	Salary
3		80
5	50	86

As $SC_3 \equiv SC_4$, tuple 3 of Q_3 must relate to a return tuple of Q_4. From the above two tables, tuple 3 of Q_4 is indistinguishable from tuple 3 of Q_3, hence these two tuples relate to each other. The user can infer that the employee with SSN equals 30 has salary equals 80. After this inference, Q_1 and Q_2 become as follows, the one on the left is Q_1 and the one on the right is Q_2,

TID	SSN	Salary
3	30	80
10	100	

TID	Salary
3	80
10	88

As $SC_1 \equiv SC_2$, tuple 10 of Q_2 must relate to a return tuple of Q_1. Tuple 10 of Q_2 is indistinguishable with tuple 10 of Q_1, hence the two return tuples relate to each other. That is, the user can infer that the employee with SSN equals 100 has salary equals 88.