creative commons
COMMONS DEED

저작자표시-비영리-변경금지 2.0 대한민국

이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.

변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

Abstract

Controlled Query Evaluation Enforcing Privacy-Policy for Safe and Efficient Data Sharing

Jo, Insoon

School of Computer Science and Engineering

The Graduate School

Seoul National University

With the growth in information access, comes the challenge of maintaining privacy and security on sensitive data in shared data storage. For instance, the Information Technology for Economic and Clinical Health (HITECH) Act's provisions penalize organizations who do not take measures to protect privacy of patient data even if the organization was unaware of such a duty. Thus, an efficient mechanism of fine-grained access control (FGAC) on such data should be considered. However, current techniques suffer from the possibility of revealing too much information or giving incorrect answers to aggregate queries. This dissertation targets data warehouse systems using SQL and aims for a generic approach to safeguard sensitive information stored in them while providing reasonably accurate query answers. It proposes improvements by considering properties of good security and defining levels of information revelation, and then develops an algorithm to evaluate user queries against a privacy policy. We assume a policy contains at least one rule and both rules and queries are written in SQL. A user query is evaluated against rules in the policy one after another. If the algorithm meets any rule with which the query is compliant, it stops and accepts the query as it is. Otherwise, it either rejects

or rewrites the query by the configuration. Given each rule in a policy, its attributes are classified into four categories, which represent different levels of information revelation to prevent inference attacks and used to decide a query's compliance with it. For a query to be compliant with a given rule, all attributes of the query should be allowed by the rule. Whether an attribute of the query is permitted by the rule or not is determined by the category which the attribute belongs to. If the algorithm fails to meet any rule with which the query is compliant (i.e. there is no rule in the policy to allow all attributes of the query), it either rejects or rewrites the query. For rewriting, it chooses a rule with which the query is more compliant than any other rule in the policy, and rewrites the query so as to be compliant with the chosen rule. We built prototypes of privacy-policy enforcement using two typical data warehouse systems: database management system (DBMS) and Hadoop-based query engine. Traditionally, DBMS has maintained a large amount of information and supported efficient data processing for it. However, the rapid growth of data sets being collected and analyzed has made it run into limitations on scalability and processing time. As a promising solution to efficiently process huge amount of data, cloud computing has come to the fore. Not only to provide a familiar programming model for existing users but to ease the programming burden for writing queries, data warehouse systems in the Cloud support SQL. Evaluation of prototype systems demonstrates that the overhead from our privacy-policy enforcement is small and scales well with typical query sizes.

Keywords: Privacy, Fine-Grained Access Control, Policy Enforcement, Query Evaluation, Database, Data Warehouse System for Hadoop
Student Number: 2004-23601

Contents

List of Tables

List of Figures

Chapter 1. Introduction

1.1 Necessity for access control

The rapid technological advances in network infrastructure hold promise to significantly reduce costs by changing the way data has been delivered. In hospital treatment, for instance, electronic health records are replacing patient charts. The total US expenditure on healthcare crossed $2 trillion in 2005, accounted for 17.3% of the national GDP by 2009, and continues to increase [1]. A large piece of this spending is on overheads including management of sensitive data. One focus in reducing costs and improving care is the electronic management of health records. Governmental support of easily retrievable health data has driven initiatives like the Nationwide Health Information Network (NHIN). Likewise, web-accessible personal health record (PHR) has attracted large players including Microsoft (HealthVault), Google (Google Health), Intel and Wal-Mart (Dossia).

These developments improve the widespread availability of medical data and consequently underscore the need for effective management of privacy and security. Multiple sources of security breaches exacerbate the issue (estimates suggest over 16 million people have had medical information compromised since 1996 [2]). Besides the loss of reputation for the hospital and financial repercussions, breaches also lead to apprehension and delay on the part of patients seeking medical care about sensitive conditions. Also affected are investigators who can lose funding and face sanctions from their Institutional Review Board (IRB). New legislation such as HITECH Act of 2009 lays out

strong punishments for organizations that did not apply due diligence in protecting patient data privacy.

In brief, the need for access control along with privacy guarantees has grown as data sets with requirements for distributed access have become common. This means that the steady and inevitable growth in ubiquitous access to critical data (e.g. medical or corporate information) comes coupled with a strong need for privacy mechanisms. For instance, not only medical data sets can involve a large number of users for treatment and research, but the correctness of controlled access to them is critical for legal or economic reasons. The key challenge is to ensure that adequate safeguards are in place, so that only authorized users can see confidential data.

This dissertation aims at providing a safe mechanism to ensure that privacy-policies are satisfied, while retaining efficiency of such policy enforcement and providing database administrators (DBAs) with convenience. In this dissertation, we focus on one of the techniques to restrict visible data based on who is accessing the system, namely fine-grained access control (FGAC) [3].

1.2 Deficiencies in existing access control mechanisms

Traditionally, the data warehouse system involving a large amount of data and a large number of users has been database. Common database access control model involves a combination of database views and role-based access control (RBAC). While effective in most scenarios, it has drawbacks in environments where a large number of roles are managed or access-control policies change frequently. In such cases, a view-based approach to FGAC not only becomes

cumbersome for DBAs but imposes overheads at DBMS level. As a solution, programmers bypass DBMS access control and manage security at application level. Even though widely seen, this approach suffers from the difficulties of pushing security and integrity functionalities to the interface (e.g. increased complexity of application code and probability of missed policies by some applications). Some of the difficulties can be alleviated by DBMS vendor enhancements such as Oracle Virtual Private Database (VPD) [4]. To restrict values users can see, VPD permits predicates to be automatically appended to `WHERE` clauses of user queries. Even though scaling better than defining customized views for each user, vender enhancements are not very expressive. And more importantly, problems remain in the correctness of such query rewriting approach when data privacy is of concern [5]. For instance, Oracle mechanically appends conditions without checking if they widen the visible data.

Enforcing security on shared data is also one of the key challenges in cloud environments. As the size of data sets being collected and analyzed rapidly grows, traditional data warehouse solutions run into limitations on scalability and manageable processing time. Cloud computing is revolutionizing how data is stored and managed. MapReduce [6] is a programming model for processing large data sets and Hadoop [7] [8] is the most popular implementation of MapReduce in the Cloud. While well-suited for distributed programming tasks, directly using Hadoop to query and manipulate data is not easy in terms of programmer productivity. Thus, query engines or data

warehouse systems in cloud environments are often used on top of Hadoop, including Apache Hive [9], Pig Latin [10], HBase [11], Hypertable [12], CloudBase [13], and jaql [14]. All of these introduce familiar and expressive programming models to Hadoop and ease the programming burden for writing queries. Though well-suited for processing large data sets at a reasonable cost, the revolution comes with security problems. Hadoop lacks access control mechanisms to prevent data breaches as its authentication and authorization schemes are not sufficiently secure [15], and likewise data warehouse systems and query engines based on it. Though it has had prominent users like Yahoo! Inc., Amazon, IBM, and Sun since 2008, its security has not been considered intensively. Data leakage has been the biggest concern preventing industries from introducing cloud computing [16] [17] [18]. A few attempts [19] [20] have been made to address this issue of security and controlled access, but the solutions are yet to come.

1.3 Motivational example

Query rewriting approach (e.g. Oracle VPD) can be a good way of controlled access. However, problems remain in the correctness of such mechanism when data privacy is of concern. Considering the limitations of query rewriting to correctly handle complex privacy requirements, identified two problems include susceptibility to incorrect aggregations [5] and inference attacks [21].

We give brief examples of incorrect aggregations and inference attacks below. Consider the data in Table 1.1 and a privacy rule: "doctors can see the diseases of patients whose ages are 18 and over". In Table 1.1, `zip` and `BP` denote zip code and blood pressure, respectively.

Table 1.1 Example `patients` data set

recordID	patientID	doctor	age	zip	disease	BP
23	$patient_1$	doc_1	17	52241	dis_1	131.25
24	$patient_2$	doc_2	19	52246	dis_2	122.70
25	$patient_3$	doc_1	66	52242	dis_1	127.88
26	$patient_3$	doc_2	66	52242	dis_2	127.88
27	$patient_4$	doc_3	45	52246	dis_3	100.34
28	$patient_5$	doc_4	32	52245	dis_4	122.50
29	$patient_6$	doc_1	50	52245	dis_1	144.38
30	$patient_6$	doc_3	50	52245	dis_3	144.38
31	$patient_6$	doc_4	50	52245	dis_4	144.38

There are two possible ways of translating this policy into an SQL expression, as shown in Table 1.2. The difference between two rules is that $Rule_1$ only shows the diseases of adult patients whereas $Rule_2$ also shows the associated ages, i.e. only the latter permits users to learn the disease-age association.

Table 1.2 $Rule_1$ and $Rule_2$ with age $\geq$ 18

(a) $Rule_1$: disease only	(b) $Rule_2$: disease and age
`SELECT` disease	`SELECT` disease, age

FROM patients	FROM patients
WHERE age >=18	WHERE age >= 18

Suppose Dr. Alice's query is "SELECT disease, avg(BP) FROM patients GROUP BY disease". As we mentioned in Chapter 1.2, typical FGAC frameworks such as Oracle VPD enforce security by mechanically appending conditions from rules to user queries. By $Rule_1$, this query is rewritten as "SELECT disease, avg(BP) FROM patients WHERE age >= 18 GROUP BY disease". The rewritten version would return the wrong average of dis_1 as $patient_1$ is excluded from the calculation. This incorrect aggregation is a known problem of the prior FGAC frameworks [5].

Also, suppose Dr. Bob's query is "SELECT disease FROM patients WHERE age >= c", where c is a constant. FGAC rewriting by $Rule_1$ changes the query to "SELECT disease FROM patients WHERE age >= c AND age >= 18". Although this seems to be equivalent to $Rule_1$, a problem arises when Bob issues a sequence of queries with c = {18, 19, ···, 66}. Bob can learn the disease-age association not permitted by $Rule_1$ by manipulating the results and in effect gets to learn everything allowed by $Rule_2$. This leads to an implicit violation, called inference attack. In this work, we differentiate these two (classes of) rules explicitly. The default assumption is that users should not be allowed to associate data across arbitrary attributes. For instance, if the $Rule_1$ is in effect, Bob's query should be allowed only when c = 18.

1.4 Overview and contribution

This dissertation meet the challenge of safeguarding sensitive information stored in data warehouse systems while providing reasonably accurate query answers. It proposes improvements by considering properties of good security and defining levels of information revelation, and then develops an algorithm to evaluate user queries against a privacy-policy.

We assume a policy contains at least one rule and both rules and queries are written in SQL. A user query is evaluated against rules in the policy one after another. If the algorithm meets any rule with which the query is compliant, it stops and accepts the query as it is. Given each rule in a policy, its attributes are classified into four categories: `FA` (Fully Accessible), `IL` (Implicitly Limited), `EL` (Explicitly Limited), and `RA` (Restriction Attribute). These categories represent levels of information revelation and used to decide a query's compliance with the rule. For a query to be compliant with a given rule, all attributes of the query should be allowed by the rule. Whether an attribute of the query is permitted by the rule or not is determined by the category which the attribute belongs to. Only when all attributes of the query are allowed by the rule, the query is accepted by the rule. For instance, Rule_1 in Table 1.2 has an `IL` (`disease`) and a `RA` (`age`) attributes. For Dr. Bob's query, `disease` and `age` of the query belong to `IL` and `RA` attributes, respectively. If c does not equal 18, this query is rejected by Rule_1 because `age` is not permitted by the rule. To prevent inference attacks, the values of `RA`

attributes should not be shown to users. By rejecting queries to specify arbitrary conditions on RA attributes, the proposed algorithm forbids users to learn the specific values of RA attributes.

If the algorithm fails to meet any rule with which the query is compliant, it either rejects or rewrites the query by DBA's configuration. For rewriting, it finds the rule with which the query is compliant as much as possible. To find the most compliant rule, for each rule in the policy, it finds common attributes in the query and the rule. For each common attribute, it measures how much its value ranges from both overlap. The most compliant rule is the one that has the most number of attributes in common with the query and the degree of overlap of those attributes is highest. After choosing such rule, it rewrites the query so as to be compliant with the chosen rule. From the query, it first removes attributes not permitted by the rule (i.e. attributes that do not appear in the rule). For each remaining attribute of the query, it is set to some value permitted by the rule. In the example in Chapter 1.3, if c (the lower-bound of age attribute) from Bob's query does not equal 18, this query violates $Rule_1$. To allow this query by $Rule_1$, our algorithm rewrites it by setting c to 18. The accepted/rewritten query is sent to the target data warehouse system.

The success of evaluation mechanisms depends on providing efficient, correct, light-weight and convenient means of checking user queries. About the query evaluation mechanism, this dissertation makes the following contributions:

1) Query evaluation must be efficient to support real-time queries and

scale with large data sets. By evaluating queries at schema level, we can reject certain queries before actually running them, and thus save processing time.

2) Correctness is another critical requirement and our algorithm satisfies it as described in Chapter 4.
3) We chose to extend FGAC query rewriting schemes instead of using views. This lightweight mechanism minimizes the need for creating or editing a view each time a rule is added or changed.
4) Finally, the likelihood of acceptance by practitioners is increased if the proposed mechanism uses SQL, a language they are familiar with, rather than a mathematically rich language such as first-order logic. We allow them to specify policies in SQL, and also support rewriting incompliant queries to a compliant form whenever possible.

1.5 Dissertation outline

To provide an efficient and correct query evaluation, FGAC algorithms should come up with some desiderata. In Chapter 2, we briefly summarize these desiderata, noting how our approach differs from prior work. Chapter 3 covers policy specification and system goals. In Chapter 4, we present our algorithms and explain how they satisfy desired goals. Time complexity analysis and experimental results show that the performance overhead is very small in Chapter 5. We conclude with a summary of this dissertation and future directions (Chapter 6).

Chapter 2. Related Work

2.1 Privacy definitions

The first and obvious step in anonymization before publishing any sensitive data (e.g. medical records) is to remove personally identifiable information such as social security numbers, names, and phone numbers. De-identifying data, however, provide no guarantee of anonymity. Released information often contains other data that can be linked to publicly available information to re-identify individuals and to infer information that was not intended for release.

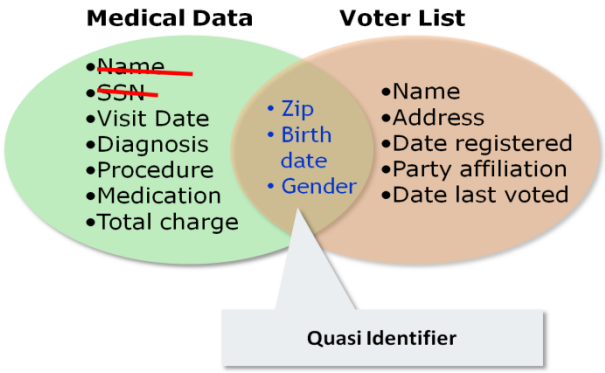


Figure 2.1 Example of linkage attack

Fig. 2.1 represents a real word example of data breaches. As shown in this figure, zip code, gender and birth date were used to identify the health record of Massachusetts governor in the public health data [22]. More recent works [23] [24] have pointed out that location data from mobile devices (e.g. smart phones and GPS-enabled cars) enable inference of users' privacy such as

habits, home locations, and even their names. These auxiliary data that may identify a person together are called a quasi-identifier (`QI`). An attack that uses `QI` from one data source to compromise privacy in different data source (e.g. personal data in public databases or social network sites) is frequently called a linkage attack [25].

k-anonymity [22], *l*-diversity [26], and *t*-closeness [27] use partitioning and aggregation to anonymize sensitive information against linkage attacks. *k*, *l* and *t* are parameters representing how much privacy is desired in the anonymization process. As the number gets higher, the higher guarantee of anonymity is desired.

k-anonymity is the first of many privacy definitions in this line of work such as *l*-diversity and *t*-closeness, etc. To prevent linkage attacks, it takes an approach requiring that, each record in the published table should be indistinguishable from at least *k*-1 others on its `QI`. To guarantee this requirement, *k*-anonymity focuses on two techniques: generalization and suppression. Generalization refers to replacing `QI`s with less specific values until getting *k* identical values. The records indistinguishable with respect to their `QI` form an equivalence class. However, generalization may cause too much information loss because outliers belong to very small equivalence classes. Merging them into a bigger one effectuates more loss in information which is often not acceptable to users. Thus, suppression was introduced as an additional method to moderate the amount of generalization necessary to

satisfy *k*-anonymity constraint. It substitutes occurrences of some values with a special value such as "?", indicating that any value can be placed instead. Although it is easy to check if an anonymized view satisfies *k*-anonymity, it may be difficult to compute a *k*-anonymous table, which is known to be NP-hard for $k \geq 4$ [28].

Except lack of efficient computational procedures, *k*-anonymity has security drawbacks. Machanavajjhala et al. [26] identified that a *k*-anonymized table is susceptible to privacy violations when the values of a sensitive attribute in equivalence class lack diversity or attackers have background knowledge. To alleviate such privacy breaches, they proposed the model of *l*-diversity which obtains anonymization with an emphasis on the diversity of sensitive attribute values on a *k*-anonymous equivalence class. *l*-diversity requires that the values of sensitive attributes should be "almost" evenly distributed in their respective equivalence classes.

Further work presented by Li et al. [27] showed that *l*-diversity is also susceptible to certain attacks. It does not prevent attribute disclosure when the overall distribution of the values of sensitive attributes is skewed or they are semantically similar. To this effect, they emphasize having the *t*-closeness property that maintains the same distribution of sensitive attribute values in an equivalence class as is present in the entire table, with a tolerance level of *t*. However, *t*-closeness deals with no identity disclosure scenarios and thus was proposed to be used in conjunction with *k*-anonymity. Moreover, it lacks

computational procedures that can force *t*-closeness with minimum data utility loss [29].

2.2 FGAC and query rewriting

The methods introduced above guarantee privacy by anonymization through generalization and suppression. Therefore, there is a trade-off between privacy and utility even though it varies with anonymization models and parameters [30] [31] [32].

Hippocratic databases [33] [34] proposed design principles for better medical information privacy in databases. These papers promoted limited disclosure of sensitive information as needed. Also, they proposed that such disclosure should be achieved by access control rather than anonymization. For the compliance with complicated privacy policies, protection at the individual data item level was desired [35].

As explained in Chapter 1, data warehousing systems have increasingly stored information for large and sophisticated applications. Data stored in them may be confidential or vital and thus need to be kept from unauthorized parties. Access control, in general, provides such protection to ensure that data are revealed or modified only by authorized users. It consists of three essential elements: users, data, and actions. Users are humans or applications accessing the data warehouse systems, data are their tables, views, or other entities, and

actions include read, update, and others. One way to implement access control is to model it as a matrix [36] [37]. The matrix has three dimensions, where X, Y, and Z axes correspond to users, data, and actions, respectively. Each element in the matrix defines a mapping from a user, a data item, and an action to a decision on accessibility. For instance, relational DBMSs have traditionally enforced access control by such access control matrix. Each element in the matrix has a flag indicating whether the user of that element can read/update the corresponding table/view. However, this traditional access control mechanism using a three-dimensional matrix is too coarse-grained to specify access control at a fine data granularity.

Suppose an employee data set with name, SSN (Social Security Number), salary, and email. Confidential information (e.g. SSN and salary) should be allowed only to their owners, but others not. Thus, users should access a subset of the whole data set (i.e. their virtual private tables). FGAC was proposed for a restricted access at a cell level [3]. DBMSs provide varying degree of FGAC support. Support for FGAC is mostly based on query rewriting as the Truman model (TM) systems [5] and DBMSs do. A user query is transparently rewritten by either substituting the target table with the user's private view or appending conditions to the `WHERE` clause of the original query. Even though user queries should be answered from the subset which he/she is allowed to access, creating per-user views is difficult to administer. Thus, PostgreSQL supported a parameterized view, where `CREATE VIEW` statement contains filter expressions with parameters rather than

explicit values. The parameter accepts values that can be supplied later. For instance, the following example creates a parameterized view that selects blood pressures of patients whom the current user treats.

An example statement for creating parameterized views

```
CREATE view MyPatients as
SELECT  BP FROM patients WHERE doctor = System_Function(MyRole)
```

However, creating per-user views is very expensive. Not only should private views as many as users should be created, but redefinitions that change the view schema require dropping the views and redefining them. Thus, Oracle devised its own security model (i.e. VPD), which works as a wrapper on database. In VPD [38], security policies are user-defined functions that return conditions for SQL `WHERE` clause, and multiple functions can be attached to the same table. Not to create private views while limiting query answers to each user's allowed data set, security functions are executed each time the table is accessed. As Figure 2.2 depicts, VPD transparently rewrites a user query by appending security functions to the query's `WHERE` clause.

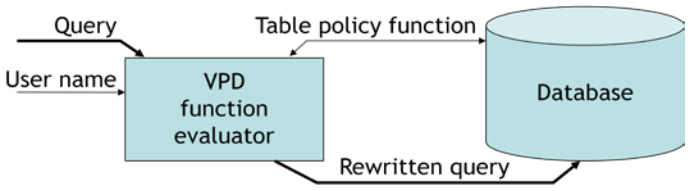


Figure 2.2 How Oracle VPD works

Even though Oracle VPD is more simple and secure than creating private

views, it has deficiencies as a FGAC mechanism. A good FGAC approach should be so expressive that users can define privacy policies at fine data granularity. Also, it should be so secure as to prevent linkage attacks. However, Oracle VPD does not satisfy these requirements. First, it is not flexible at defining privacy policies. Oracle VPD is row-level security rather than column-level security, and hiding the whole columns using it is very cumbersome. The actual problem is that it is vulnerable to inference attacks. It does not prevent users from drawing inferences about sensitive attribute values as long as queried data is within the range allowed by the privacy policies. Moreover, Rizvi et al. [5] addressed limitation with this query rewriting approach by substituting the target table with the user's private view or appending conditions to the `WHERE` clause of the original query. Rewriting may cause incorrect aggregations, because it transparently shrinks the target data set of a user query to the subset which the corresponding user is allowed to access. To this effect, they emphasize the Non-Truman model (NTM) [5], which prefers to accept/reject an original query. Instead of modifying queries, they proposed two types of query validness: conditional and unconditional. The former incorporates information from integrity constraints, and the latter means that a query can be answered within a view defined by given rules. Unconditional validness is conceptually similar to the conjunctive query (CQ) containment problem, determining if a given query is "contained" in another query. Thus, it can be checked in the same way to test on CQ containment. However, CQ containment has been proven to be NP-complete [39], and thus hard to check in general. Further, it is difficult to solve CQ containment under

bag semantics [40], which SQL evaluation is based on. SQL uses a multi-set/bag semantics rather than a set semantics. In bag semantics, both queries and databases are bags, i.e. duplicate rows are retained unless explicitly requested. Contrary to this, in set semantics, the results of queries are sets as well as the databases. Evaluation by bag semantics is originally for an efficiency reason to save time of eliminating duplicates. Jayram et al. [41] has proven that CQ containment is not decidable in bag semantics. Hence, the NTM is impractical to implement.

Query rewriting is not only for FGAC. Rewriting techniques have been popularly adopted for query optimization and data integration [42] [43] [44]. Query optimization is to find a more efficient execution plan, and thus the rewritten query should be equivalent to the original one. Contrary to this, data integration aims at reducing calls to the federated database, and tries to find a maximally contained query in a local materialized cache.

2.3 Limitation of prior FGAC frameworks

Augmenting existing literature, we highlight a series of desired properties for FGAC frameworks. Some properties are from the recent work [45], but their "security" property is replaced with "compliance". Security in their definition is onerous to verify in large data sets since it depends on comparing result set differences with changes between database states. Instead, we propose to use compliance based on containment of revealed information.

1) `Aggregation:` FGAC framework should return correct results of aggregate functions such as min, max, and average.

2) `Anti-inference:` From a series of queries, attackers may draw inference about a relationship between values of multiple attributes, which is not allowed by the policies. This attack is frequently called an inference attack. FGAC framework should be robust against these attacks.

3) `Soundness:` FGAC framework must return results consistent with those without access control, i.e. rewriting should cause no harm to the correctness of the results.

4) `Compliance:` FGAC framework should return results that only include information allowed by one or more policies. This means that any released information should be allowed explicitly by one policy or implicitly by a combination of multiple policies.

5) `Maximality:` FGAC framework should return results that include as much information as is allowed. Our algorithm supports it partially (denoted by "`p`") and Wang's does relatively (denoted by "`r`"). In general, due to the ambiguity in policy and limited expressiveness of policy specification languages, it is hard to achieve maximality. We feel that our *4*-level classification (explained in Chapter 3.2) removes

some ambiguity and helps establish basic thresholds for information revelation.

6) *k*-anonymity: FGAC framework should support the model of *k*-anonymity.

7) Efficiency: FGAC framework should be able to carry out a query evaluation (i.e. a decision on its acceptance) within polynomial time. By evaluating queries at schema level, we can reject certain queries before actually running them, and thus save processing time.

5) Convenience: When describing policies, practitioners must prefer familiar languages (e.g. SQL) to mathematically rich languages (e.g. first-order logic). Likelihood of acceptance by them is increased if the proposed FGAC framework uses SQL.

Table 2.1 Comparison of access control models

	Ours	**TM**	**NTM**	**Wang**
Aggregation	Yes	No	Yes	No
Anti-inference	Yes	No	No	No
Soundness	Yes	No	Yes	Yes
Compliance	Yes	Yes	Yes	Yes
Maximality	Yes(p)	No	No	Yes(r)
***k*-anonymity**	Yes	No	No	No
Efficiency	Yes	No	No	No

Convenience	Yes	No	No	No

Table 2.1 summarizes how ours and prior works satisfy the desired properties of FGAC framework. Prior works have drawbacks that they failed to consider formal security criteria or identify levels of information revelation (and thus determine what users can see). These drawbacks may cause them to allow data breaches or return incorrect information. Also, unlike ours, they present neither algorithmic details nor experimental validations. In Chapter 4.7, we will return to the discussion of these desiderata in order to describe how our approach satisfies them in detail.

2.4 Limitation of prior privacy enforcement frameworks in cloud environments

As the size of data sets being collected and analyzed rapidly grows, traditional data warehouse solutions (e.g. relational DBMSs) have run into limitations on processing time. MapReduce [6] has been considered the most promising programming model that can provide scalable processing for these rapidly growing data sets, and Hadoop [7] [8] is its most popular implementation in the Cloud.

Although data leakage has been the biggest concern preventing industries from introducing cloud computing, very few attempts [19] [20] have been made to address this issue of security. Moreover, the proposed solutions are limited in their application, unattractive in performance overhead, or

susceptible to privacy violations. Airavat [19] aimed at secure MapReduce to provide confidentiality and privacy assurances for sensitive data. To enforce security, it added differential privacy [25] and mandatory access control to Hadoop. To guarantee differential privacy, access to a statistical database should not enable users to learn anything about an individual that cannot be learned without access. Informally, given the output of a computation/query, attackers should not be able to tell whether any particular value was in the input data set or not. Thus, ε-differential privacy will be achieved by adding random noise whose magnitude is chosen as a function of the largest change a single input element can make to the output. This quantity is referred as the sensitivity of the function. For instance, Δcount, i.e. the sensitivity of a query to calculate the number of records in some data set is less than or equal to one.

Definition. *For $f:D \rightarrow R^d$, the L1-sensitivity of f is*

$$\Delta f = \max_{D1,D2} \| f(D_1) - f(D_2) \|_1$$

for all D_1, D_2 differing in at most one element.

In discretionary access control mechanisms such as Linux file permission, an individual user can set the policy. Contrary to this, in mandatory access control mechanisms such as memory protection mechanism, the security policy is built into the system and individuals are not allowed to modify it. Airavat mainly focused on differential privacy than mandatory access control. Output perturbation (the main mechanism of differential privacy) has an inherent deficiency of data utility. Even minimum noise to prevent adversary's

learning may interfere with user's learning anything useful from the perturbed outputs [47]. Besides, Airavat incurs such severe overhead that it slowed down up to 32% compared to the computational overhead caused from the unmodified Hadoop.

Secure Cloud Repository (SCL) aimed at access control on Apace Hive [9]. However, its table-level (or view-level) access control is too coarse-grained to be practical. Data providers may require security policies at fine granularity, but SCL cannot support them in masking sensitive rows or columns.

Basescu et al. [48] proposed a different approach from the above two. Whereas Airavat and SCL focus on protecting "data" themselves, they aim at protection of the "resource utilization" of storage systems. Attack patterns submitted by data providers are considered security policies. Their framework monitors user activities and detects accesses matching the predefined attack patterns. Except behavior-based detection is vulnerable to unforeseen attacks, FGAC is not a matter of concern for them.

While well-suited for distributed programming tasks, directly using Hadoop to query and manipulate data is not easy in terms of programmer productivity. Thus, query engines or data warehouse systems in cloud environments are often used on top of Hadoop. As we explained above, Hadoop lacks access-control mechanisms to prevent data breaches, and likewise data warehouse systems and query engines based on it. As far as we know, there is no FGAC

framework for query engines or data warehouse systems on Hadoop. SCL is the only access control framework on Hadoop, but its table/view level access control is too coarse-grained to be practical.

Chapter 3. Policy Specification and Properties

We assume that a policy `P` consists of a set of rules and each rule is defined as

---

`SELECT` Data View `FROM` One or More Tables

`WHERE` Restriction Predicates

(Optionally `GROUP BY` Attributes

`HAVING` Restriction Predicates)

---

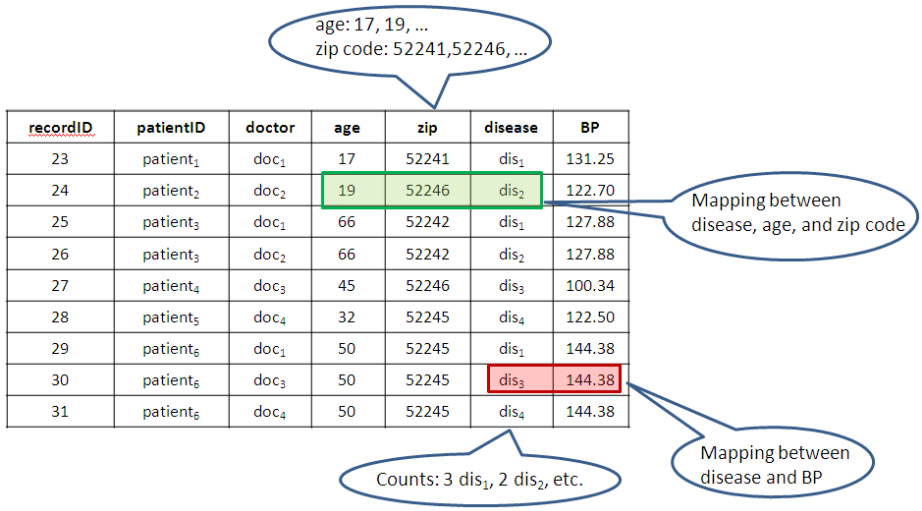
1) Data View: A set of accessible attributes or expressions (simply, "attributes" from now on). These attributes are also used to link information across columns from one or more tables.

2) Restriction Predicates: Conditions specified in `WHERE` or `HAVING` clauses. Users should be allowed to access the attributes in "Data View" only from the rows that satisfy these conditions. Currently, we only consider predicates in the form of $A\theta c$, where A is an attribute, $\theta \in \{=, \geq, \in\}$, and c is a constant.

For the rest of this dissertation, we omit the optional `GROUP BY` part of the query for the sake of brevity. Note that we support sub-queries in Restriction Predicates. In Chapter 3.4, we will discuss this support further.

3.1 Four levels of information revelation

We consider that a data set can reveal three types of information: values of attributes, their counts, relationship between attributes. Here, non-sensitive and sensitive attributes are not identity information which directly identifies a person. Non-sensitive attributes are `QI`s, and sensitive attributes are private information that should not be shown to unauthorized users. As shown in Fig. 3.1, the example `patients` data set reveals attribute values and their counts. Also, it reveals relationship between non-sensitive and sensitive attributes (e.g. mapping between age and disease) and that between sensitive attributes (e.g. mapping between disease and BP).



recordID	patientID	doctor	age	zip	disease	BP
23	patient$_1$	doc$_1$	17	52241	dis$_1$	131.25
24	patient$_2$	doc$_2$	19	52246	dis$_2$	122.70
25	patient$_3$	doc$_1$	66	52242	dis$_1$	127.88
26	patient$_3$	doc$_2$	66	52242	dis$_2$	127.88
27	patient$_4$	doc$_3$	45	52246	dis$_3$	100.34
28	patient$_5$	doc$_4$	32	52245	dis$_4$	122.50
29	patient$_6$	doc$_1$	50	52245	dis$_1$	144.38
30	patient$_6$	doc$_3$	50	52245	dis$_3$	144.38
31	patient$_6$	doc$_4$	50	52245	dis$_4$	144.38

Figure 3.1 Four types of information a data set reveals

Based on this observation, we discuss four levels of information revelation. As we explained, a data set can reveal three types of information. Some rules

allow users to know all types of information, but others let users know part of them. According to the amounts of information a rule reveals, we classify it into four categories: by-value, by-value set, by-range, and by-range set. For each category, we provide a template and some examples on the `patients` data set in Table 1.1.

1) By-value rules:

`SELECT` attributes `FROM` one or more tables
`WHERE` conditions using = or $\in$ or any conditions on Data View

By-value rules are the closest to the traditional FGAC among these four categories. They allow users to see attribute values and therefore their counts from the rows satisfying Restriction Predicates. Also, users are allowed to learn the correlation between attributes. The following is an example of a by-value rule. This rule allows users to see the diseases and ages of patients as long as their ages are 20 and over. It also allows users to learn the relationship between disease and age.

An example by-value rule
`SELECT` disease, age `FROM` patients `WHERE` age >= 20

Table 3.1 Information revealed by the example by-value rule

age	disease	Count
66	dis_1	1

66	dis_2	1
45	dis_3	1
32	dis_4	1
50	dis_1	1
50	dis_3	1
50	dis_4	1

2) By-value set rules:

By-value set rules are similar to by-value rules, except that `SELECT` now becomes `SELECT DISTINCT`. In a data set, some of the attributes may contain duplicate values, and the `DISTINCT` clause eliminates these duplicates from a result. Thus, by-value set rules disallow users to know the number of each attribute value. The following is an example of a by-value set rule. By this rule, users can see the distinct ages and diseases of patients whose ages are 20 and over. However, they are not allowed to know the number of each disease that corresponds to the same age.

An example by-value set rule
`SELECT DISTINCT` disease, age `FROM` patients `WHERE` age >= 20

Table 3.2 Information revealed by the example by-value set rule

age	**disease**
66	dis_1, dis_2

45	dis_3
32	dis_4
50	dis_1, dis_3, dis_4

3) By-range rules:

`SELECT` attributes `FROM` one or more tables

`WHERE` conditions using >= or < or any conditions on Data View

A by-range rule allows users to see attribute values and their counts from the matching rows, but not the relationship between attributes. For instance, the following rule reveals the information in Table 3.3. From this result, users cannot learn the relationship between age and disease.

An example by-range rule

`SELECT` disease `FROM` patients `WHERE` age >= 20

Table 3.3 Information revealed by the example by-range rule

age	disease	Count
age >= 20	dis_1	2
age >= 20	dis_2	1
age >= 20	dis_3	2
age >= 20	dis_4	2

By this rule, range queries whose boundaries are strictly contained in the rule's range should be rejected or rewritten. If such queries are allowed, users may learn some association between attributes not permitted by the rule. For instance, the example rule above

should reject or rewrite a query "SELECT disease FROM patients WHERE age >= 30". Because this query can reveal the age-disease relationship this rule does not allow.

4) By-range set rules:

By-range set rules are similar to by-range rules, except that SELECT now becomes SELECT DISTINCT. These reveal the least amount of information among the four categories. They allow users to know the distinct values of attributes, but disallow them to know the number of duplicate values and relationship between attributes.

An example by-range set rule
SELECT DISTINCT disease FROM patients WHERE age >= 20

Table 3.4 Information revealed by the example by-range set rule

BP	**disease**
age >= 20	dis_1, dis_2, dis_3, dis_4

To summarize, a rule of each category reveals different amount information as shown in Table 3.5.

Table 3.5 Amount of information each rule type reveals

Rule type	Values of attributes	Relationship between attributes	Number of duplicate values
By-value	Yes	Yes	Yes
By-value set	Yes	Yes	No
By-range	Yes	No	Yes
By-range set	Yes	No	No

We explain the relationship between these four categories in terms of the amount of information they reveal. To describe such relationship, we define V(D, r) and $\sqsubseteq$. Given a data set D and a rule r, V(D, r) is the maximum amount of information about D revealed by r, which includes the results of evaluating user queries against r and executing them on D. Also, $\sqsubseteq$ denotes subsuming. For two rules r_1 and r_2, $r_2 \sqsubseteq r_1$ if and only if $V(D, r_2) \subset V(D, r_1)$. Intuitively speaking, if $r_2 \sqsubseteq r_1$, users can infer the information allowed by r_2 by manipulating the results of queries allowed by r_1. For instance, let r_1 be a rule "SELECT age, BP FROM patients WHERE age >= 18 AND age < 50". This is a by-value rule and V(patients, r_1) includes data in Table 3.6. Using aggregate functions such as min and max, users can find out minimum and maximum blood pressures in patients whose ages are between 20 and 30, between 30 and 40, or fall within any sub-range of [18, 50).

Table 3.6 A subset of V(patients, r_1)

age	BP
19	122.70
45	100.34
32	122.50

Consider another rule r_2, "SELECT BP FROM patients WHERE age >= 18 AND age < 50". This is a by-range rule and V(patients, r_2) includes {122.70, 100.34, 122.50}. Contrary to r_1, r_2 allows no aggregate function if users query blood pressures in patients whose ages fall within any proper sub-range of [18, 50). For instance, r_2 either rejects or rewrites a query asking the average blood pressure of patients whose ages are between 20 and 30. Thus, V(D, r_2)$\subset$V(D, r_1) and $r_2 \sqsubseteq r_1$. Let r_V, r_{VS}, r_R, and r_{RS} denote a by-value rule, a by-value set rule, a by-range rule, and a by-range set rule, respectively. If all four have the same Restriction Predicates, $r_{VS} \sqsubseteq r_V$ and $r_{RS} \sqsubseteq r_R$. Also, $r_R \sqsubseteq r_V$ and $r_{RS} \sqsubseteq r_{VS}$.

3.2 Privacy and accuracy by by-range rules

As we noted in Chapter 1, the existing FGAC frameworks enforce security by appending conditions from all privacy policies related to a queried data set to the query's WHERE clause. Such mechanical appending causes them not only to be vulnerable to linkage attacks but to give incorrect aggregate outputs. Consider a policy P_1 with two rules r_1 and r_2. Suppose a user doc_2 issues a

query q, “SELECT BP FROM patients WHERE age >= 20 AND doctor = doc2”.

P_1 r_1 and r_2	
SELECT age	SELECT BP
FROM patients	FROM patients
WHERE age >=18	WHERE doctor = System_Function(MyRole)

For this example case, some data warehouse systems such as Oracle would append “age >= 18 and doctor = System_Function(MyRole)” to q’s WHERE clause and then execute the rewritten query. By issuing q, doc_2 may learn that there is an adult patient whose blood pressure is 127.88. However, r_1 and r_2 only allow the information in Tables 3.7 and 3.8, respectively. Learning the information in Tables 3.7 and 3.8 separately does not reveal that blood pressure of some adult patient is 127.88.

Table 3.7 Information revealed by r_1 (output rotated)

age	19	66	66	45	32	50	50	50

Table 3.8 Information revealed by r_2 for doc_2

BP	**doctor**
122.70	doc_2
127.88	doc_2

Due to the deficiencies in query rewriting by mechanical appending, we are

against mechanical appending. Given a set of rules and a query, if the query inquires linkage information or may return incorrect aggregate output to users, we consider it compliant with no rule. By rejecting or rewriting incompliant queries, we try to overcome two weaknesses of the prior FGAC frameworks. For the example case above, we reject `q` or rewrite it so as to be compliant with either of r_1 or r_2. Similarly, given rules in Table 1.2 and Dr. Alice's query, we reject the query rather than return the incorrect average.

3.3 *k*-anonymity support

We also provide validation support for *k*-anonymity [22], one of the widely accepted privacy definitions in confidential data protection. By using `COUNT` (or `COUNT DISTINCT`) in SQL, we can express the *k*-anonymity requirement as shown below.

`SELECT` (`DISTINCT`) sensitive attributes

`FROM` tables

`WHERE` Restriction Predicates RP

`AND` (`SELECT COUNT`(*) `FROM` tables `WHERE` RP) >= *k*

Including a sub-query with `COUNT`(*) in the `WHERE` clause ensures that values of sensitive attributes would be released only if there are at least *k* records that satisfy Restriction Predicates. Consider an example policy P_2 with a single rule. This rule allows users to know diseases of patients only when there are two or more patients to satisfy age and zip code conditions (i.e. *2*-anonymity).

The information revealed by this *2*-anonymous rule is shown in Table 3.9.

P₂ A *2*-anonymous rule
SELECT disease FROM patients
WHERE age >= 18 AND age < 50 AND zip >= 52244 AND zip < 52246
AND (SELECT COUNT(*) FROM patients WHERE age >= 18 AND age < 50
AND zip >= 52244 AND zip < 52246) >= 2

Table 3.9 Information revealed by *2*-anonymous rule

age	**zip**	**disease**
[18; 50)	[52244; 52246)	dis_2
[18; 50)	[52244; 52246)	dis_3
[18; 50)	[52244; 52246)	dis_4

Note that our support of *k*-anonymity is for validation rather than computing *k*-anonymous tables. The attributes in the WHERE clause of a sub-query can be considered as QI (or dimensions to be generalized). Similarly, their range can be considered as categorical values of generalization nodes. The thing they are completely decided by policy writers provides flexibility to data owners.

Chapter 4. Design

In this chapter, we first describe how our query evaluation algorithm works. Then, we summarize how it achieves the desiderata introduced earlier. All rules and queries in Chapter 4 are used to retrieve data from the `patients` data set in Table 1.1.

4.1 Notation and assumptions

We use the following notations:

1) A policy `P` is a set of rules $\{r_1, r_2, \cdots, r_n\}$. Each r_i $(1 \le i \le n)$ is an SQL statement and equivalent to a data set restricted by its conditions. We assume that no rule in `P` is a subset of another rule in `P`. (i.e. there is no duplicate in `P`.)
2) `q` is a user query written in SQL.
3) $\texttt{SELECT}_r$ (respectively $\texttt{SELECT}_q$) is the set of attributes that appear in the `SELECT` clause of a rule `r` (respectively of `q`), i.e. the attributes whose values would appear in the answer.
4) $\texttt{WHERE}_r$ (respectively $\texttt{WHERE}_q$) is the set of attributes that appear in the `WHERE` clause of `r` (respectively of `q`). For each $a \in \texttt{WHERE}_r$, we classify it as continuous or discrete. If its conditions are range predicates (e.g. age of r_1 in Chapter 3.2), we call it a continuous attribute. If its conditions are equality predicates (e.g. doctor of r_2 in Chapter 3.2), we call it a discrete attribute.

5) For a continuous attribute a, rg(a) denotes the min and max boundaries defined by its range predicates. In the example of r_1 in Chapter 3.2, rg(age) is [18, ∞). We assume that rg(a) is left-closed, right-open without loss of generality.
6) Similarly, for a discrete attribute a, s(a) denotes the set of values specified by its equality predicates. In the example of r_2 in Chapter 3.2, sg(doctor) is the value set that System_Function(MyRole) can take.

4.2 Attribute classification

We assume that a rule r takes the form "SELECT a_0, ⋯, a_n FROM V WHERE $b_1\theta c_1$ AND/OR $b_2\theta c_2$ … AND/OR $b_m\theta c_m$", where θ refers to any operator allowed in WHERE clauses (e.g. = and <). Each attribute $a \in SELECT_r \cup WHERE_r$ falls into one of the four categories: FA_r, IL_r, EL_r, or RA_r.

Definition 1. *FA_r (Fully Accessible Attributes): when r has an empty WHERE clause, FA_r is the set of attributes that appear in its SELECT clause.*

$$FA_r = \{a \mid a \in SELECT_r \wedge WHERE_r = \emptyset\}$$

For instance, FA_{r3} = {disease, age, BP}. r_3 allows users to see all triples of (disease, age, BP) from the patients data set.

r_3 An example rule with FA attributes
SELECT disease, age, BP FROM patients

Definition 2. *IL_r (Implicitly Limited Attributes): when r has a non-empty WHERE clause, IL_r is the set of attributes from its SELECT clause not in the WHERE clause.*

$$IL_r = \{a \mid a \in SELECT_r \setminus WHERE_r \wedge WHERE_r \neq \emptyset\}$$

Definition 3. *EL_r (Explicitly Limited Attributes): when r has a non-empty WHERE clause, EL_r is the set of attributes from its SELECT clause that also belong to the WHERE clause.*

$$EL_r = \{a \mid a \in SELECT_r \cap WHERE_r \wedge WHERE_r \neq \emptyset\}$$

r_4 An example rule with IL and EL attributes
SELECT disease, BP FROM patients WHERE BP >= 121.1 AND BP < 125.2

As the name suggests, an EL attribute allows explicitly limited access. Thus, a rule with it would function as a by-value rule. Given an EL attribute, users can see its individual values, counts, and the relationship between it and other attributes (if any), as long as its values meet the restriction conditions. In the example of r_4, BP is an EL attribute, and users can see all individual values of blood pressure falling within 121.1 and 125.2. By issuing a series of queries while varying blood pressure values, users may learn that r_4 restricts the accessible range of blood pressure to [121.1, 125.2). They are also allowed to learn the relationship between BP and disease.

Contrary to this, there is no way for users to learn access control information about an IL attribute. In the same example, the diseases shown to users are limited by the restriction predicates of another attributes, in this case, BP. Users should not be allowed to conduct a series of queries while varying disease values. Thus, they would neither know nor be able to predict how many and what other diseases are in the current data set.

Definition 4. *RA_r (Restriction Attributes): when r has a non-empty WHERE clause, RA_r is the set of attributes from its WHERE clause not in the SELECT clause.*

$$RA_r = \{a \mid a \in WHERE_r \backslash SELECT_r \wedge WHERE_r \neq \emptyset\}$$

In r_5, RA_r={age}\{disease}={age}. Note that EL_r and RA_r are disjoint partitions of $WHERE_r$ (i.e. $RA_r \cup EL_r = WHERE_r$ and $RA_r \cap EL_r = \emptyset$). RA attributes are not shown to users, but restrict other attributes to be shown to users. A rule with RA attributes would function as a by-range rule, because users are not allowed to learn the relationship between RA and non-RA attributes. To prevent linkage attacks, if a rule has some RA attributes, a query should be allowed only when both have the same restriction predicates about RA attributes.

r_5 An example rule with RA attributes
SELECT disease FROM patients WHERE age >= 18

For instance, r_5 allows users to see {dis_2, dis_1, dis_2, dis_3, dis_4, dis_1,

dis_3, dis_4}, but not any correlation between age and disease. If we allow a query with range predicates of 18≤age<20, users get {dis_2}. Not only does this output reveal that the patient with dis_2 is either 18 or 19 years old, but it gives users a higher chance of correctly guessing the patient's age (i.e. 50%). Given a rule with some RA attributes, if a query and the rule have different restriction predicates about such attributes, the query should be disallowed to prevent linkage attacks. Thus, we either reject such query or rewrite it so as to be compliant with the rule.

4.3 Evaluation against a policy with a single rule

As we mentioned in Chapter 4.1, P consists of one or more rules. This subchapter considers a policy with a single rule, and the next one deals a policy with multiple rules. Assume that r is the only rule in P. Given q, our evaluation algorithm has three possible outcomes: accept, reject, and rewrite. (1) If q is compliant with r (i.e. what q asks about is allowed by r), it accepts q and forwards it to the target data warehouse system. How to deal with not accepted queries depends on data administrators. If they prefer rejection, it rejects q. Otherwise it rewrites q into a compliant query q′, and then forwards q′ to the target system. Note that q′ reveals more information about r and thus more about P. Thus, data administrators may prefer to reject not accepted queries. Algorithm 1 represents our evaluation algorithm when a policy has a single rule.

input: r and q

output: "accept" or "reject/rewrite"

```
1   if WHERE_r=∅ then
2       if (SELECT_q ∪ WHERE_q) ⊆ SELECT_r then
3           return "accept";
4       else /* rewrite or reject q */
5           remove attributes not in SELECT_r from q or reject;
6   else
7       if SELECT_q ⊄ SELECT_r then /* rewrite or reject q */
8           SELECT_q ← SELECT_q ∩ SELECT_r or reject;
9       if WHERE_r ⊄ WHERE_q then /* rewrite or reject q */
10          add attributes in WHERE_r \ WHERE_q and their respective conditions to the WHERE
            clause of q or reject;
11      if (WHERE_q \ WHERE_r) ⊄ IL_r then /* rewrite or reject q */
12          remove attributes in WHERE_q \ WHERE_r \ IL_r and their respective conditions from
            the WHERE clause of q or reject;
13      foreach attribute a in EL_r do
14          if IsSubRange(rg(a_q), rg(a_r)) is false then /* rewrite or reject q */
15              replace rg(a_q) with rg(a_q) ∩ rg(a_r) or reject;
16      foreach attribute a in RA_r do
17          if IsSameRange(rg(a_q), rg(a_r)) is false then /* rewrite or reject q */
18              replace rg(a_q) with rg(a_r) or reject;
19      return "accept";
```

Algorithm 1 Evaluation against a policy with a single rule

To explain the evaluation process, we use q_1.

q_1 An example user query
SELECT disease, age FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 125.2 AND zip = '52241'

1) When $WHERE_r = \emptyset$ (lines 2–5): In this case, all attributes in $SELECT_r$ are fully accessible. If $WHERE_q$ is a subset of $SELECT_r$, what is revealed by q is inferable from what users would learn by running r as query. Thus, q is compliant with r and we accept q. For instance, if r allows users to see blood pressures and ages of all patients, it also lets them see blood pressures in patients whose ages are between 18 and 50. On the other hand, if $WHERE_q \not\subset SELECT_r$, we reject/rewrite q. Since the answer by q would reveal some linkage information (i.e. what users are not allowed to know). Given a policy with a single rule r_3, Algorithm 1 rejects/rewrites q_1. Since zip of $WHERE_{q1}$ is not an element of $SELECT_{r3}$ (i.e. outside the view defined by r_3), q_1 is rejected or rewritten into q_2.

q_2 Rewritten version of q_1 so as to be compliant with r_3
SELECT disease, age FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 125.2

2) When $WHERE_r \neq \emptyset$ (lines 7–19): In this case, $SELECT_r = IL_r \cup EL_r$ and $WHERE_r = EL_r \cup RA_r$.

2-a) $\texttt{SELECT}_q \not\subset \texttt{SELECT}_r$ (line 8): For q to be compliant with r, $\texttt{SELECT}_q$ must be a subset of $\texttt{SELECT}_r$. Otherwise q is asking about information outside the view defined by r. In such case, Algorithm 1 either rejects q or rewrites it by removing attributes not in $\texttt{SELECT}_r$ from $\texttt{SELECT}_q$. If we consider a policy with a single rule r_6, q_1 is not compliant with this rule. r_6 allows users to see disease information, but q_1 is asking about age as well as disease. Thus, we either reject q_1 or rewrite it by removing age from $\texttt{SELECT}_{q1}$.

r_6
SELECT disease FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 125.2 AND zip = '52241'

2-b) $\texttt{WHERE}_r \not\subset \texttt{WHERE}_q$ (line 10): For q to be accepted, the view defined by it should be contained in the view defined by r. Thus, for each attribute in $\texttt{WHERE}_r$, q should have restriction conditions about it. If $\texttt{WHERE}_r \not\subset \texttt{WHERE}_q$, conditions of q are less restrictive than those of r. Some records to meet the conditions of q may not satisfy those of r (i.e. they may be outside the view defined by r). In such case, q should be rejected or rewritten to be more restrictive than r. Suppose a policy with a single rule r_7. Whereas $\texttt{WHERE}_{r7}$ has four elements (i.e. age, BP, zip, doctor), $\texttt{WHERE}_{q1}$ has three elements (i.e. age, BP, zip). It is obvious that q_1 is asking about a "broader" view than what r_7 allows. Thus, Algorithm 1 rejects q_1 or rewrites it. In this

case, the rewritten query is the same as r_7.

r_7
SELECT disease, age FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 125.2 AND zip = '52241'
AND doctor = 'doc2'

2-c) ($WHERE_q \backslash WHERE_r$) $\not\subset IL_r$ (line 12): For the view defined by q to be contained in the view defined by r, the "extra" attributes in $WHERE_q \backslash WHERE_r$ must appear in IL_r (i.e. $SELECT_r \backslash WHERE_r$). Given a policy with a single rule r_8, q_1 is incompliant with r_8. BP in $WHERE_{q1}$ appears neither in $WHERE_{r8}$ nor in $SELECT_{r8}$. Algorithm 1 either rejects q_1 or rewrites it by removing BP-related restriction conditions from $WHERE_{q1}$. In this case, the rewritten query is the same as r_8.

r_8
SELECT disease, age FROM patients
WHERE age >= 18 AND zip = '52241'

Contrary to this, r_9 has BP in its SELECT clause, and thus a policy with a single rule r_9 would accept q_1. Further, because BP appears in $SELECT_{r9}$, users are allowed to place any BP-related constraints on the WHERE clauses of their queries.

r_9
SELECT disease, age, BP FROM patients
WHERE age >= 18 AND zip = ‘52241’

Given r and q, the steps above are to filter out attributes (accordingly their respective constraints) not allowed by r from q. Then, for each remaining attribute in $WHERE_q$, Algorithm 1 checks if the amount of data its restriction conditions specify is allowed by r. Given each remaining attribute a in $WHERE_q$, let d_{aq} be the amount of data a-related constraints of q specify. Similarly, let d_{ar} be the amount of data a-related constraints of r specify. For q to be accepted by r, d_{aq} should be either contained in or equal to d_{ar}. Whether d_{aq} should be the same as d_{ar} or not depends on a’s category (i.e. a belongs to EL_r or RA_r). This containment check was designed to facilitate “early reject”. By rejecting queries at the early stage before sending them to the target query engines, we can save valuable time and resource of the target systems. If all remaining attributes in $WHERE_q$ satisfy their respective constraint containments, q is accepted. Otherwise q is rejected or rewritten. As we mentioned in Chapter 4.1, we classify attributes in $WHERE_r$ as continuous or discrete. We explain how to check constraint containment for continuous attributes. The check on discrete attributes can be done in a similar way to that on continuous attributes.

To check constraint containment, we define two functions:

- `IsSubRange(`I_1`, `I_2`)`, where I_1 and I_2 are non-empty intervals. This function checks if I_1 is strictly contained in I_2. If so, it returns true. Otherwise it returns false.
- `IsSameRange(`I_1`, `I_2`)`, where I_1 and I_2 are non-empty intervals. This function checks if I_1 and I_2 are exactly the same. If so, it returns true. Otherwise it returns false.

2-d) For each `a` in EL_r (lines 14–15): Users should see values of `a` as long as $rg(a_q) \subseteq rg(a_r)$. Otherwise, they may learn information not allowed (e.g. attribute values outside the view defined by `r`). If $rg(a_q) \not\subset rg(a_r)$, we reject `q` or rewrites it by replacing $rg(a_q)$ with $rg(a_q) \cap rg(a_r)$. Thus, users can see individual values of attributes in EL_r as long as they are within the view defined by `r`.

r_{10}
SELECT disease, BP FROM patients
WHERE age >= 18 AND BP >= 100 AND BP < 140 AND doctor = ‘doc2’

r_{10} allows users to see (`disease`, `BP`) pair of doc_2’s adult patients as long as their blood pressures are between 100 and 140. For three reasons (lines 7, 9, 11), q_1 is incompliant with r_{10}. Thus, evaluation of q_1 against r_{10} can show how we rewrite a query through multiple steps. First, even though r_{10} allows users to see information about

disease and blood pressure, q_1 is asking about patients' ages (line 7). Thus, age should be removed from $SELECT_q$ (line 8). Secondly, q_1 is asking about a broader view than what r_{10} allows (line 9). Thus, we add "doctor='doc2'" to the WHERE clause of q_1 (line 10). Lastly, zip in $WHERE_q \backslash WHERE_r$ is not in $SELECT_r$ (line 11), and we remove zip-related constraints from the WHERE clause of q_1 (line 12). q_1 violates no constraint containment, because $BP \in EL_r$ and $rg(BP_q) \subset rg(BP_r)$. q_3 represents the rewritten version of q_1 to be compliant with r_{10}.

q_3 Rewritten version of q_1 against r_{10}

```
SELECT disease FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 125.2 AND doctor = 'doc2'
```

2-e) For each a in RA_r (lines 17–18): Users should see values of a as long as $rg(a_q)=rg(a_r)$. As we mentioned in Chapter 4.2, a rule with RA attributes would function as a by-range rule. To prevent linkage attacks, q should be allowed only when it and r have the same restriction predicates about RA attributes. If $rg(a_q) \neq rg(a_r)$, we reject q or rewrite it by adjusting $rg(a_q)$ to $rg(a_r)$. Considering q_1 and r_{11}, the former violates the latter. BP is an RA attribute, but $rg(BP_{q1}) \neq rg(BP_{r11})$. Thus, we reject q_1 or rewrite it by replacing $rg(BP_{q1})$ with $rg(BP_{r11})$. In this case, the rewritten query is the same as r_{11}.

r_{11}
SELECT disease, age FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 128 AND zip = ‘52241’

On the other hand, r_{12} accepts q_1. The only difference between r_{11} and r_{12} is that BP appears in the SELECT clause of r_{12}. In this case, BP is an EL attribute, not an RA attribute. Thus, q_1 is accepted as long as $rg(BP_{q1}) \subset rg(BP_{r12})$.

r_{12}
SELECT disease, age, BP FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 128 AND zip = ‘52241’

4.4 Evaluation against a policy with multiple rules

In this subchapter, we consider a policy P with multiple rules. Algorithm 2 describes how we evaluate q against P with n rules, $\{r_i | 1 \leq i \leq n\}$. We accept q as long as any rule in P accepts it. In other words, we consider a policy a union of separate rules. Also, if q is compliant with any rule in P, we consider it to satisfy P. Some DBAs may want to accept q when it simultaneously satisfies a certain set of rules. In such case, they can use our policy integration algorithm, which will be introduced in Chapter 4.6.

input: **{**r_i| 1≤*i*≤n} and q

output: "accept" or "reject/rewrite"

```
1    foreach r_i do
2        if r_i accepts q then
3            return "accept";
4    if rewriting is not set then
5        return "reject";

6    idx = num_select = num_where = overlap_where = 0;
7    foreach r_i do
8        tmpnum_select = |SELECT_ri ∩ SELECT_q|;
9        tmpnum_where = tmpoverlap_where = 0;
10       if tmpnum_select > num_select then
11           idx = i; num_select = tmpnum_select; num_where = |WHERE_ri ∩ WHERE_q|;
12           overlap_where = CalcOverlap(WHERE_ri, WHERE_q);
13       else if tmpnum_select == num_select then
14           tmpnum_where = |WHERE_ri ∩ WHERE_q|;
15           if tmpnum_where > num_where then
16               idx = i; num_where = tmpnum_where;
17               overlap_where = CalcOverlap(WHERE_ri, WHERE_q);
18           else if tmpnum_where == num_where then
19               tmpoverlap_where = CalcOverlap(WHERE_ri, WHERE_q);
20               if tmpoverlap_where > overelap_where then
21                   idx = i; overlap_where = tmpoverlap_where;

22   if idx > 0 then
23       rewrite q to be compliant with r_idx by Algorithm 1;
24       return "rewrite";
25   else return "reject";
```

Algorithm 2 Evaluation against a policy with multiple rules

Algorithm 2 consists of two phases. The only difference between them is rewriting. In the first phase, it assumes no rewriting. Whenever it meets any rule in P with which q is compliant, it returns “accept” and ends. If no rule in P accepts q, executing the second phase depends on data administrators. If they prefer rejection, Algorithm 2 returns “reject” and ends. Otherwise, it tries to rewrite q so as to be compliant with some r_i in P. The key process for rewriting is to find some rule with which q is the most compliant. Note that rewriting is not always possible. If what q is asking about does not overlap with any view defined by r_i, q should be rejected. When there are two or more rules which allow part of what q asks about, Algorithm 2 compares such rules and picks one with which q is the most compliant. Intuitively speaking, the most compliant rule is a rule that will answer as much information as possible. If r_c in P is the rule with which q is the most compliant, there must be the largest overlap between the view defined by r_c and what q asks about. After choosing such r_c, Algorithm 2 applies Algorithm 1 to r_c and q. In other words, it rewrites q so as to be compliant with r_c.

How to find the most compliant rule is as follows. To measure the amount of compliance between q and each r_i in P, we compute the overlap between what q asks about and the view defined by r_i. For this, we defined three metrics: num_{select}, num_{where}, and $overlap_{where}$. num_{select} is the number of attributes common in $SELECT_q$ and $SELECT_{ri}$. Similarly, num_{where} is the number of attributes common in $WHERE_q$ and $WHERE_{ri}$. $overlap_{where}$ is the

result by CalcOverlap($WHERE_q$, $WHERE_{ri}$). This function measures overlap between constraints on attributes common in $WHERE_q$ and $WHERE_{ri}$. For each $a \in WHERE_q \cap WHERE_{ri}$, let d_{aq} be the amount of data a-related constraints of q specify. Similarly, let d_{ari} be the amount of data a-related constraints of r_i specify. Also, let q' and $d_{aq'}$ be the rewritten version of q against r_i and the amount of data a-related constraints of q' specify. Each a in $WHERE_{ri}$ is classified as continues or discrete. Also, it belongs to either EL_{ri} or RA_{ri}. For a continuous attribute a, $rg(a_{q'})$ is either $rg(a_q) \cap rg(a_{ri})(a \in EL_{ri})$ or $rg(a_{ri})(a \in RA_{ri})$. For a discrete attribute a, $s(a_{q'})$ is either $s(a_q) \cap s(a_{ri})$ $(a \in EL_{ri})$ or $s(a_{ri})(a \in RA_{ri})$. Considering a continuous attribute a which belongs to EL_{ri}, $rg(a_q)$, $rg(a_{ri})$, $rg(a_{q'})$ correspond to d_{aq}, d_{ari}, $d_{aq'}$, respectively. This function intends to measure how much intervals (or how many values) from a query or a rule remain after rewriting. The larger amount of data a rule allows a query, the larger intervals (or the more values) would remain after rewriting. For each $a \in WHERE_q \cap WHERE_{ri}$, this function takes the minimum of two overlaps: the overlap between d_{aq} and $d_{q'}$ and that between d_{ari} and $d_{q'}$. For a continuous attribute a, this function takes $rg(a_{q'}) \div \max\{rg(a_q), rg(a_{ri})\}$. Similarly, for a discrete attribute a, it takes $|s(a_{q'})| \div \max\{|s(a_q)|, s(a_{ri})|\}$. Then, it returns the sum of such minimums. In order to choose the rule with which q is the most compliant, Algorithm 2 uses num_{select}, num_{where}, and $overlap_{where}$ in this order. If r_c is the rule with the biggest num_{select}, it considers r_c the most compliant rule. If two or more rules have the same biggest num_{select}, it compares their respective num_{where}. If two or more rules still have the same biggest num_{where}, it compares their respective

$overlap_{where}$. Finally, it chooses any rule with the biggest $overlap_{where}$.

q_4 An example user query
SELECT disease, age, BP FROM patients
WHERE age >= 18 AND BP >= 121.1 AND BP < 125.2 AND zip = '52241'

Table 4.1 A policy with multiple rules: r_{13} and r_{14}

r_{13}	r_{14}
SELECT disease, BP	SELECT disease, BP
FROM patients	FROM patients
WHERE age >=18	WHERE BP >= 120
AND doctor = 'doc2'	AND BP < 140

Considering an evaluation of q_4 against P with r_{13} and r_{14}, q_4 is compliant with none of rules in P. If DBA prefers rewriting, we have to decide with which q_4 is more compliant. For r_{13}, num_{select} is |{disease, age}| and num_{where} is |{age}|. For r_{14}, num_{select} is |{disease, BP}| and num_{where} is |{BP}|. Because num_{select} and num_{where} of two rules are same, we have to compare their respective $overlap_{where}$. $WHERE_{q4} \cap WHERE_{r13}$ is {age} and d_{age_q4} and d_{age_r13} exactly remain after rewriting (i.e. $rg(age_{rewritten}) = rg(age_{q4}) = rg(age_{r13})$). For r_{14}, $WHERE_{q4} \cap WHERE_{r13}$ is {BP}, and part of d_{BP_r14} remains after rewriting (i.e. $rg(BP_{rewritten}) = rg(BP_{q4}) \subset rg(BP_{r14})$). We consider that q_4 is more compliant with r_{13} than r_{14}, and q_4 is rewritten to be compliant with r_{13} by Algorithm 1. The final rewritten query is shown below.

q_5 Rewritten version of q_4 against a policy with r_{13} and r_{14}
SELECT disease, age FROM patients
WHERE age >=18 AND doctor = ‘doc2’

4.5 Evaluation of a query with sub-queries

This subchapter considers an evaluation of a query with sub-queries against a policy P. Algorithm 1 evaluates a query against a policy with a single rule. Algorithm 2 is the extended version of Algorithm 1. Based on the latter, the former considers a policy with multiple rules. Algorithm 3 below is the complete version of our query evaluation algorithm. Based on Algorithm 2, it supports an evaluation of a query with sub-queries. Given a query with sub-queries, the remaining part except sub-queries is called an outer query. In our algorithm, the outer query and sub-queries are separately evaluated. Moreover, each sub-query is evaluated as if it were a separate query. If DBA prefers rejection, the query is accepted when its outer query and all sub-queries are accepted. When DBA prefers rewriting, some rejected queries can be rewritten. If the outer query and all sub-queries are either accepted or rewritten, the original query can be rewritten. After evaluating all queries separately, Algorithm 3 assembles their respective evaluation results. The conjunction of queries within a specific view is still within that view. Thus, the assembled query is guaranteed to be within the view defined by P.

input: {r_i|1≤i≤n} and q with sub-queries q_{sj} (1≤*j*≤m)

output: "accept" or "reject/rewrite"

res = "reject";

if q has sub-queries **then**

foreach q_{sj} **do**

apply Algorithm 2 to q_{sj} and {r_i} and save output to res;

if res is "reject" **then** return "reject";

remove q_{sj}, col_{sj}, and op_{sj} from q and let q_o denote the remaining query;

apply Algorithm2 to q_o and {r_i} and save output to res;

if res is "reject" **then**

return "reject";

else if q_o and all q_{sj}'s are accepted **then**

return "accept";

else

if q_o is rewritten **then** let $q_{o'}$ denote the rewritten version of q_o;

else $q_{o'} = q_o$;

foreach q_{sj} **do**

if q_{sj} is rewritten **then** let $q_{sj'}$ denote the rewritten version of q_{sj};

else $q_{sj'} = q_{sj}$;

foreach $q_{sj'}$ **do**

$q_{o'}$ = Assemble($q_{o'}$, col_{sj}, op_{sj}, $q_{sj'}$);

return "rewrite";

else

apply Algorithm 2 to q and {r_i} and save output to res;

return res;

Algorithm 3 Evaluation of a query with sub-queries

If we examine how Algorithm 3 works, it first splits q into two: an outer query q_o and a collection of sub-queries $\{q_{sj} | 1 \leq j \leq m\}$. Each q_{sj} is connected to q_o by a column col_{sj} and an operator op_{sj}. Strictly speaking, q_o is the remaining part after removing each q_{sj}, col_{sj}, and op_{sj} from q. Algorithm 3 evaluates each q_{sj} as if it were a separate query. If every q_{sj} is accepted, it proceeds to evaluate q_o. If q_o is also accepted, it assembles the evaluation results of q_o and q_{sj}. The assembled query is sent to the target data warehouse system. When q is rejected (i.e. any of q_o or q_{sj} is not accepted), Algorithm 3 tries rewriting if DBA prefers it. If q_o and each q_{sj} are accepted or rewritten, it assembles their respective evaluation results. To combine results, we define the following function:

- Assemble($q_{o'}$, col_{sj}, p_{sj}, $q_{sj'}$), where $q_{o'}$ and $q_{sj'}$ denote accepted (or rewritten) versions of q_o and q_{sj}. If q_o is accepted, $q_{o'}$ is q_o. Otherwise, it is a rewritten version of q_o against some rule in P. Similarly, we get $q_{sj'}$ from q_{sj}. While iterating through j ($1 \leq j \leq m$), this function appends $q_{sj'}$ to $q_{o'}$ using col_{sj} and op_{sj}.

q_6 An example user query with a sub-query
SELECT BP FROM patients WHERE age = (SELECT MIN(age) FROM patients WHERE doctor = ‘doc1’)

r_{15}
SELECT age, BP, doctor FROM patients WHERE zip = ‘52241’ AND (doctor = ‘doc1’ OR doctor = ‘doc2’)

q_6 is asking about the blood pressure of doc_1's youngest patient. Consider an evaluation of q_6 against a policy with a single rule r_{15}. Because q_6 has a sub-query, `Algorithm 3` splits it into q_o and q_{s1}. Here, q_o is "`SELECT` BP `FROM` patients" and q_{s1} is "`SELECT MIN`(age) `FROM` patients `WHERE` doctor='doc1'". In this case, q_{s1} is connected to q_o by col_{s1} (i.e. `age`) and operator op_{s1} (i.e. '='). `Algorithm 3` evaluates q_{s1}, and then q_o. Neither is compliant with r_{15}, but both are rewritable. If rewriting is set, `Algorithm 3` appends $q_{s1'}$ to $q_{o'}$ using col_{s1} and op_{s1}. The assembled query is shown below.

q_7 Rewritten version of q_6 against a policy with r_{15}
`SELECT` age, BP, doctor `FROM` patients
`WHERE` zip = '52241' `AND` (doctor = 'doc1' `OR` doctor = 'doc2')

4.6 Policy integration

Given a policy `P`, we consider it as a union of separate rules. Thus, in our evaluation, `P` accepts `q` as long as any rule in `P` accepts it. However, there are some cases when `P` should accept `q` as long as a certain set of rules accept it. For instance, a hospital with its own privacy policy may need to comply with HIPAA (federal-level) and state-level privacy policies. In such case, as long as policies are written in SQL, our algorithm can be used to find an intersection of all policies. Starting with one policy, rules from the other policy may be evaluated against the first policy by our evaluation algorithm. The evaluation result is the new policy that is compliant with both policies. By repeating this process, our algorithm can be used for integrating any number of policies.

This process can be slow when there are many policies to consider, but policy integration rarely happens. Thus, the overhead caused from policy integration would not be a matter of concern in most systems.

4.7 Satisfying FGAC properties

In Chapter 2.3, we laid out desiderata for FGAC frameworks. We return to discuss how our evaluation algorithm achieves them.

1) `Aggregation:` Ours correctly evaluates aggregate functions such as summation and maximum. An intuitive approach to correctly handling some issues was presented in Chapter 3.2. As it supports the union of multiple rules in a policy, DBA may use one rule to define a limited view (e.g. doctors can see the records of patients whose ages are over eighteen) and another rule to define an aggregated view (e.g. doctors can see the average blood pressure of all patients).

2) `Anti-inference:` Our algorithm is robust against inference attacks. As introduced in Chapter 3.2, every query accepted (or rewritten) by ours reveals a subset of information allowed by a given policy `P`. If users infer any relationship between attributes, this must be within what they are allowed to learn by `P`. This property comes from the soundness and compliance of our evaluation algorithm discussed next.

3) Soundness: The soundness of rejecting an incompliant query is trivial. Our evaluation algorithm rewrites a query by adding more constraints to it until it is at least as restrictive as one of the rules in P. In other words, the result returned by the rewritten query is what could be inferred from the result returned by the chosen rule.

4) Compliance: The compliance of Algorithm 1 is straightforward. Given a query q and a rule r, at each step, the information asked by q but not in the view defined by r is removed. In terms of attributes, users can see the attributes allowed by r. In terms of rows, they can see the rows to meet the conditions in WHERE clause of r, i.e. the subset of rows in the view defined by r. Not only does it remove attributes and rows outside of the view defined by r, but protects data by ensuring that all attributes in WHERE_r appear in the WHERE clause of the final query and adjusting constraints in the WHERE clause of q so as to be allowed by r. Algorithm 2 achieves compliance because it chooses some rule r in P and applies Algorithm 1 to r and q. The accepted/rewritten query is compliant with r, and in turn with P.

5) Maximality: Algorithm 1 satisfies maximality (i.e. any less constraint leads to noncompliance). Algorithm 2 does not guarantee maximality, instead uses heuristics to identify the rule with maximal overlap.

6) *k*-anonymity: Ours supports *k*-anonymity as described in Chapter 3.3. Any rule `r` can be modified to be *k*-anonymous by nesting itself as a sub-query with "`COUNT(*)` > *k*" condition. Note that this notion of *k*-anonymity is not defined at the policy-level, but on the view defined by `r`. Each view defined by a *k*-anonymous rule is guaranteed to return at least *k* records, but there may be another rule that defines an overlapping view with less than *k* records.

7) Efficiency: Our efficiency is discussed in Chapter 5.

8) Convenience: For DBA convenience, we use rules written in SQL.

Chapter 5. Performance Evaluation

5.1 Overview of prototype implementation

This dissertation aims at a safe and efficient security framework that can work with data warehouse systems. As the first step toward achieving this goal, we designed query evaluation algorithms, and `Algorithm 3` is the complete version. As the second step, based on `Algorithm 3`, we built two prototypes that enforce privacy-policies for user queries. We wanted our design to work with popular data warehouse systems, and typical examples include database and Hadoop-based query engine. Thus, we made our first prototype work with databases, and the second one work with Hadoop-based query engines.

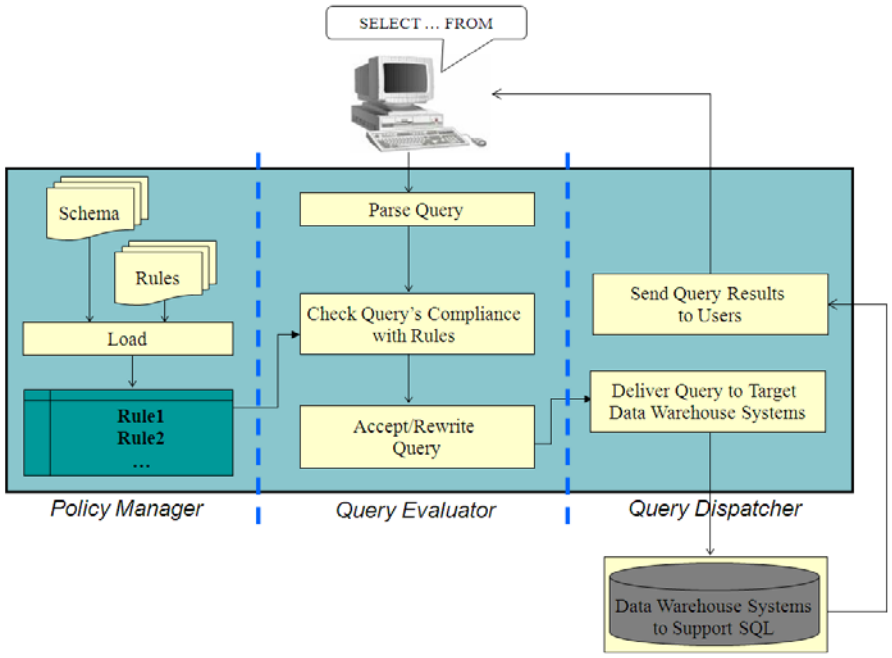


Figure 5.1 Overall architecture

Fig. 5.1 depicts the overall architecture of our prototypes, which consists of three components: `Policy Manager`, `Query Evaluator` and `Dispatcher`. `Policy Manager` is for policy integration explained in Chapter 4.6. DBA may want a query to be accepted as long as it is compliant with a set of rules, not a single rule. In that case, `Policy Manager` updates the policy by adding new rules gotten from intersections of the existing rules. `Query Evaluator`, as the name suggests, evaluates a query against the policy. The query may be rejected. If rewriting is set, it tries to rewrite the query to be compliant with some rule in the policy. Then, it forwards the final query (i.e. accepted or rewritten one) to `Query Dispatcher`, which accesses data sources using JDBC.

5.2 Evaluation using popular databases

5.2.1 Experimental setup

The experiments in Chapter 5.2 are to compare the performance overhead caused from privacy-policy enforcement by DBMS security features and ours. We conducted experiments using a desktop with dual-core Intel Xeon CPU 3.06GHz, 6GB RAM, and 250GB HDD, which ran Red Hat Enterprise Linux 5.5 and Oracle 11g Release 1 (11.1.0.6.0). The performance was measured on the same machine running Oracle. For performance comparison, we randomly generated a data set, rules and queries. DBGEN is the database population program used in the TPC-H [49], a decision support benchmark. Using DBGEN, we built 10GB database population. We also generated ten policy

and query sets. Each policy set consists of six policy files with a single rule, and each query set consists of six query files with 100 queries. All rules and queries have the same number of attributes in their `SELECT` and `WHERE` clauses. Given a rule, if a single attribute appears in its `SELECT` clause, exactly one attribute appears in its `WHERE` clause. The reason each set has six files is that the number of attributes in `SELECT` clause (accordingly `WHERE` clause) of rules or queries in each file varies from three to eight. For instance, queries in the first file of a query set have three attributes in their `SELECT` and `WHERE` clauses. Similarly, queries in the second file of the query set have four attributes in their `SELECT` and `WHERE` clauses. We limited the number of attributes in `SELECT` and `WHERE` clauses of queries and rules to less than ten. Queries that inquire values of ten or more attributes or have constraints about such large number of attributes are not real. In the following experiments, our prototype tried to rewrite queries rather than reject them. Thus, time taken by ours includes time taken for rewriting. In the following figures, X-axis labels take the form of `xXy`, where `x` and `y` denote the number of attributes in `SELECT` and `WHERE` clauses of rules/queries, respectively. Y-axis value is time taken in milliseconds.

5.2.2 Experimental results

We compared the performance of our prototype and security feature provided by leading commercial DBMS vendor. Oracle VPD, introduced in Oracle8i, enables security policies (rules from the standpoint of our prototype) to be directly attached to a database object (a table, view, or synonym). It adds a

dynamic `WHERE` clause to a query issued against these objects, so the policies are automatically applied whenever they are accessed. Although aiming at the same function (i.e. privacy-policy enforcement), ours and Oracle VPD have different implementation details. First, there is a difference in rule selection. Ours enforces privacy-policy with OR syntax. A query may be compliant with many rules in a policy. In such case, it chooses the most compliant rule and evaluates the query against the chosen rule. On the other hand, Oracle VPD enforces security with AND syntax. When multiple policies are attached to a database object, all of them are applied to any access to the object. To work around this OR vs. AND discrepancy, we generated policies with a single rule. Secondly, there is a difference in rule targets. Our rules can reference multiple tables or views. On the contrary, a policy of Oracle VPD can be attached to a single database object. For performance comparison, our rule and a policy of Oracle VPD should be semantically equivalent. Whenever our rule references multiple tables/views, we formulated a single database object, i.e. a join view of them. Then, we translated our rule into a policy of Oracle VPD, which is attached to the join view and semantically equivalent to our rule. In the following experiments, we assumed that security policies are already added to Oracle VPD and needed views (e.g. join views above) are created before.

As shown in Fig. 5.2 and Fig. 5.3, whether we enforced security using ours or Oracle VPD, the average time taken for a query was barely different in two cases. In other words, our prototype is comparable in overhead to the leading commercial DBMS. Fig. 5.4 supports this observation showing that 99.99%

of time was taken in DBMS when we enforced security using our prototype.

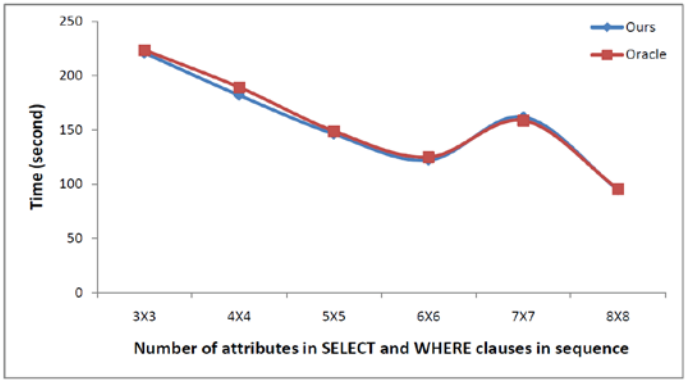


Figure 5.2 Total runtime per query

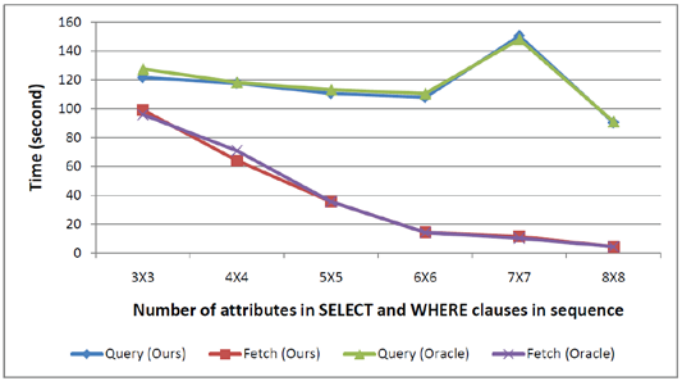


Figure 5.3 Querying vs. fetching time

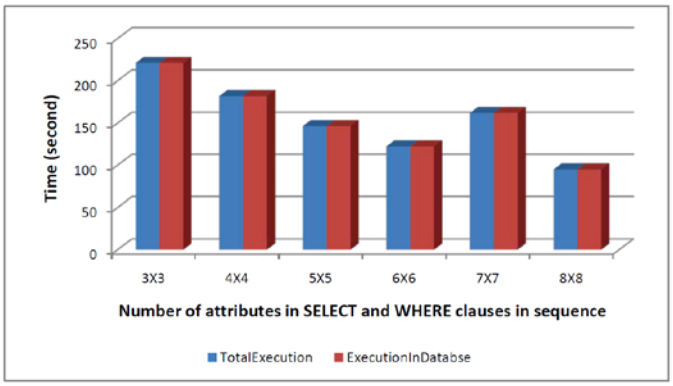


Figure 5.4 Total and DB execution time per query

5.2.3 Complexity analysis

Given a policy P and a query q, let m be the number of rules in P and n be the number of distinct attributes appearing in r ($r \in P$) or q, i.e. $n = |SELECT_r \cup WHERE_r \cup SELECT_q \cup WHERE_q|$.

1) Evaluation against a single rule: If there is enough memory to hold a boolean array of size n, the time to check $S_1 \subseteq S_2$ is O(cn), where c is a constant overhead of accessing the memory. Also, if there is enough memory to hold another boolean array of size n, the set assignments such as line 8 take O(cn). Each $a \in WHERE_q \cap WHERE_r$ is either continuous or discrete. If it is continuous, IsSubRange(I_1,I_2) and IsSameRange(I_1,I_2) need comparisons about left and right bounds. We assume that a single comparison takes constant time d. Otherwise, two functions take O(cn) for set containment check. The worst case scenario is that q hits all if statements below line 6. Thus, the worst-case time complexity of Algorithm 1 (t_s) is as follows:

$t_s \le \max\{cn \text{ (line 2)} + 2cn \text{ (line 5)},$
$\quad 6cn \text{ (lines 7–12)} + dn \text{ (lines 13–15)} + dn \text{ (lines 16–18)}\}$
$= \max\{3cn, 6cn+2dn\} = O(n)$

The complexity above assumed that every $a \in WHERE_q \cap WHERE_r$ is continuous. If we assume every a is discrete, lines 13-18 take 2*cn. Thus, t_s takes O(n), whether each a is continuous or discrete.

2) Evaluation against multiple rules: As we introduced in Chapter 4.4, evaluation against multiple rules consists of two phases. Let t_i be the complexity of i^{th} phase. t_1 is $m*O(n)=O(mn)$, since the first phase runs `Algorithm 1` for `q` and each `r`. As we mentioned in Chapter 4.4, `CalcOverlap` measures how much intervals (or how many values) from `q` or `r` remain after rewriting. Similar to the evaluation against a single rule, if `a` ($\in WHERE_q \cap WHERE_r$) is continuous, this function needs set containment checks and boundary comparisons. Otherwise, it needs set containment checks. Thus, the worst-case complexity of `Algorithm 2` (t_m) is as follows:

$t_m \leq O(mn)$ (lines 1–3) + $m \times O(n)$ (lines 7–21) + $O(n)$ (lines 22–24) = $O(mn)$

This complexity is linear with respect to `n` and `m`. As shown in Fig. 5.5, time taken at the second phase (i.e. rewriting) is longer than time taken in the first phase (i.e. decision).

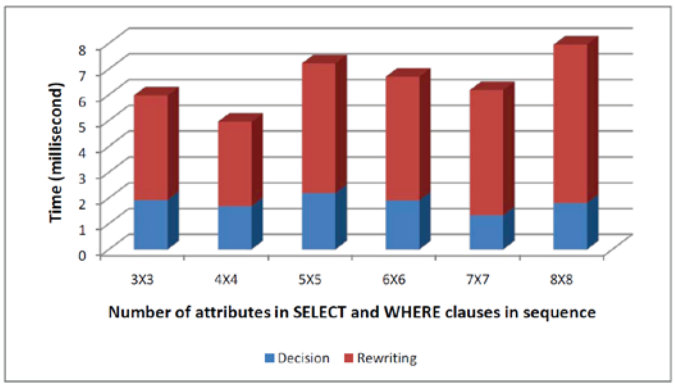


Figure 5.5 Decision and rewriting time per query (using Oracle)

5.3 Evaluation using Hadoop-based query engines

5.3.1 Experimental setup

This chapter discusses the performance overhead caused from privacy-policy enforcement when our prototype works with Hadoop-based query engines. Experiments were conducted on a 24-node cluster at the University of San Francisco. Each node had four 2.0GHz Dual-Core AMD Opteron Processor 270 with 4GB memory and 600GB HDD, and ran 64-bit Fedora 14. We used Hadoop version 0.20.2, Hive version 0.7.0 and Cloudbase version 1.3.1. We adopted the test data set from the experiments by Pavlo et al. [50]. The data set had 750 million rows, totaling 97.4GB. It was loaded into `uservisits` table, which has nine columns. We implemented our own rule and query generator. It is similar to QGEN generator distributed by TPC, i.e. generating random SQLs over `uservisits` table. We generated multiple policy and query sets. A policy set (respectively query set) consists of multiple policy files (respectively query files). The rules or queries in the same file have the same number of attributes in `SELECT` and `WHERE` clauses. The number of attributes in each file of a policy or query set varied from one to nine, so that each set had nine files.

5.3.2 Experimental results

We conducted two experiments to measure the performance overhead incurred by access control policy enforcement. In experiments, we assumed rewriting.

When no rule in a policy accepted a query, our prototype tried to rewrite it. We conducted query evaluation on a single node and then delivered the final query (i.e. accepted or rewritten one) to target MapReduce system.

The first experiment was designed to illustrate how the number of attributes in `SELECT` and `WHERE` clauses affects the overhead. For the first experiment, we generated ten policy and query sets. Each policy file contained ten rules, and each query file had 100 queries. The time taken was averaged over three runs. Fig. 5.6 shows that rewriting takes longer than decision. It also shows that time taken for rewriting increases according to the number of attributes. However, it still took less than 0.8ms (i.e. 0.1ms for decision and 0.67ms for rewriting in 9X9 case). Compared to this, computation on Hadoop usually took several seconds. Thus, we can say that our prototype incurred negligible overhead.

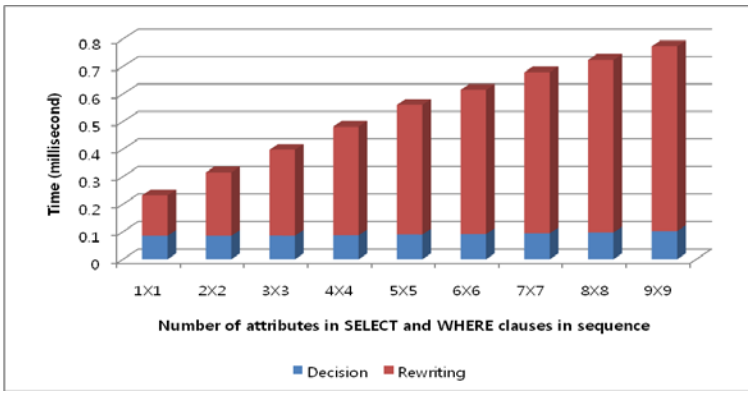


Figure 5.6 Decision and rewriting time per query (using Hadoop)

According to evaluation results, queries can be classified into three categories.

In other words, they are accepted, rewritten or rejected. Queries of the first category require decision time, which is negligible compared to rewriting time as shown in Fig. 5.6. Fig. 5.7 shows time taken by rewritten/rejected queries.

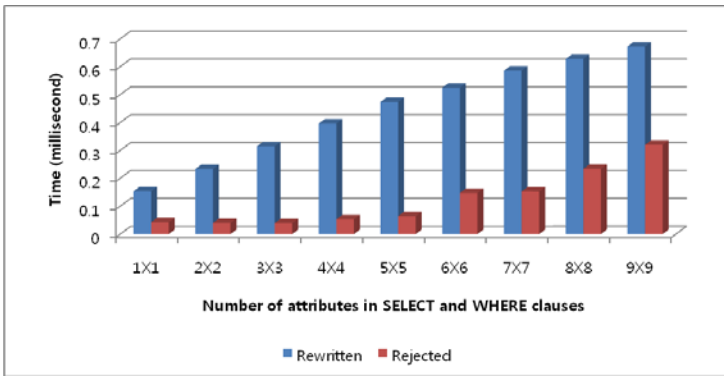


Figure 5.7 Time taken for evaluation

The second experiment is to show how the number of rules in a policy affects the overhead. For the second experiment, we generated 20 new policy sets, but used the same ten query sets used in the first experiment. Each file in the same policy set had the same number of rules, and the number of rules in each file varied from 5 to 100, resulting in 20 sets. Fig. 5.8 shows the time taken when the number of attributes in `SELECT` and `WHERE` clauses of rules was five. Time taken for rewriting was scaled according to the number of rules, and reached 3.457ms when a policy contained 100 rules. This overhead is trivial. Moreover, a privacy-policy with such many rules is not real. This result shows that our prototype scales well with the number of rules per policy.

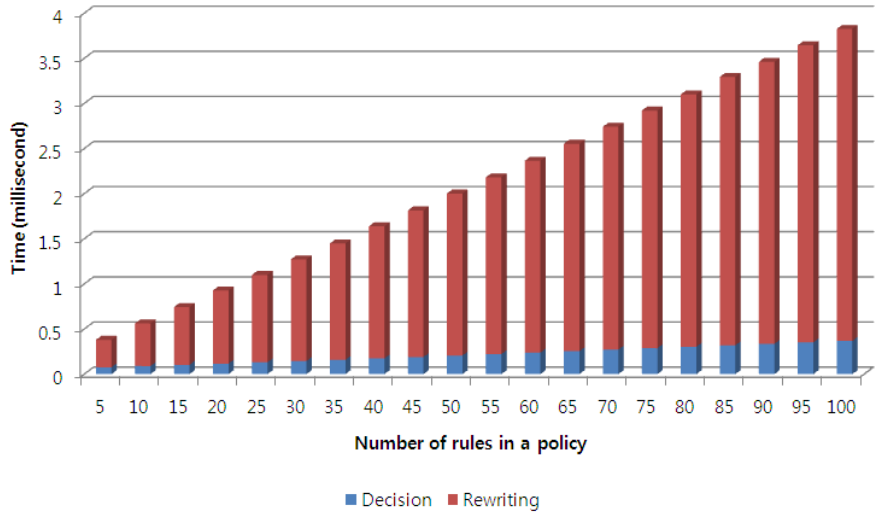


Figure 5.8 Time taken according to the number of rules in a policy

Fig. 5.9 illustrates the ratio of time taken for query evaluation to total runtime. From query sets used in the first experiment, we randomly chose 30 queries. Our prototype evaluated them, and then delivered the final queries to Hive. This figure shows that the overhead incurred by our prototype is very small, and consistently under 0.8%.

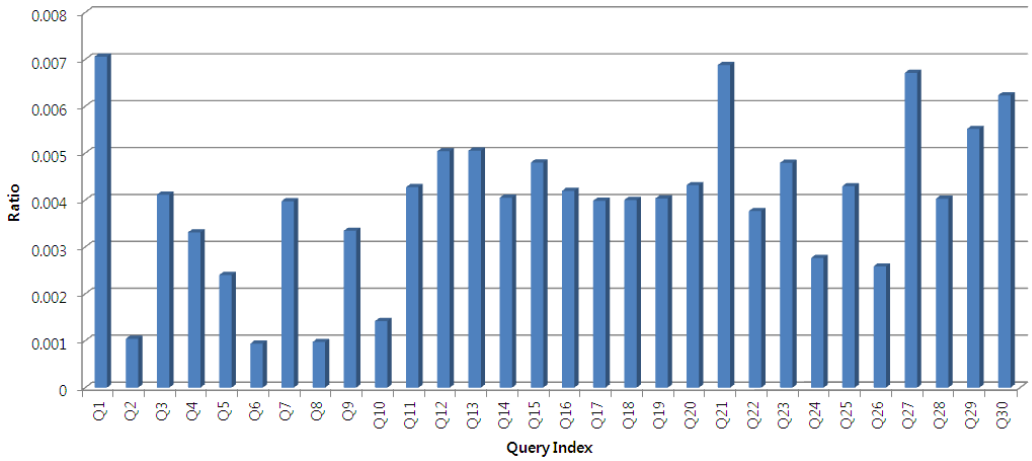


Figure 5.9 Ratio of evaluation time to total runtime

Chapter 6. Conclusion

This dissertation presents a practical FGAC framework for secure data sharing in typical data warehouse systems. We wanted to overcome limitations seen in prior works (i.e. vulnerability to inference attacks and incorrect aggregate answers) while enforcing privacy-policies with minimal overhead. To achieve these goals, we first categorized four levels of information revelation (i.e. by-value, by-value set, by-range, and by-range set). Based on these privacy levels, we proposed a query evaluation algorithm, which provides controlled access to confidential data by query rewriting. FGAC mechanisms should support desired properties such as correctness of evaluation, support for privacy levels, efficiency of query evaluation, and convenience by SQL specification, and the proposed algorithm satisfies them. We implemented the algorithm using two typical data warehouse systems: DBMS and Hadoop-based query engine. Our prototypes provide access control on top of the query engine layer, which has great advantages over leaving controlled access to ad-hoc implementation at the application level. Our algorithm can easily work with any data warehouse systems supporting SQL and JDBC. Moreover, experimental results showed that the overhead caused from our privacy-policy enforcement is negligible. By using schema-level evaluation, we are able to reject queries before actually running them and save processing time.

Our algorithm may be used for data management products to facilitate HIPAA

and HITECH compliance and protect the privacy of patients and security of medical data. In the long run, this not only reduces the risk of personal distress and economic consequences, but also benefits medical researchers and IT professionals. While current work uses healthcare as the domain of interest, the proposed algorithm is applicable in other areas where both privacy and collaboration are important. It can be used to augment traditional role-based access control in mobile applications [51] or exchange accounting and financial information with stakeholders. In a global economy where service suppliers and providers work together across state lines and legal jurisdictions, an approach like ours that can integrate multiple sets of access control policies and efficiently evaluate queries across them must be useful.

Bibliography

[1] C. J. Truffer, S. Keehan, S. Smith, J. Cylus, A. Sisko, J. A. Poisal, J. Lizonitz, and M. K. Clemens, "Health Spending Projections Through 2019: The Recession's Impact Continues," *Health Affairs*, vol. 29, no. 3, pp. 522-529, 2010.

[2] PogoWasRight.org. (2007) Medical Privacy at Risk: A Call for Effective Legislative Action. [Online]. http://www.phiprivacy.net/MedicalPrivacy/MedicalPrivacyBreachesStudy_2007.pdf

[3] J. Richardson, P. Schwarz, and L. Cabrera, "CACL: Efficient Fine-Grained Protection for Objects," in *Proceedings of the 7th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92)*, 1992, pp. 263-275.

[4] Oracle, "Oracle Virtual Private Database," White Paper 2005.

[5] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy, "Extending Query Rewriting Techniques for Fine-Grained Access Control," in *Proceedings of the 2004 ACM SIGMOD international conference on Management of data (SIGMOD'04)*, 2004, pp. 551-562.

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI'04)*, 2004, pp. 137-150.

[7] M. Olson, "HADOOP: Scalable, Flexible Data Storage and Analysis," *IQT Quarterly*, vol. 1, no. 3, pp. 14-18, 2010.

[8] T. White, *Hadoop: The Definitive Guide*, 3rd ed.: O'Reilly Media, 2012.

[9] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu. and R. Murthy, "Hive - A Petabyte Scale Data Warehouse Using Hadoop," in *Proceedings of the 26th IEEE International Conference on Data Engineering (ICDE 2010)*, 2010, pp. 996-1005.

[10] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-

So-Foreign Language for Data Processing," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*, 2008, pp. 1099-1110.

[11] Apache HBase. [Online]. http://hbase.apache.org/

[12] Hypertable. [Online]. http://hypertable.org/

[13] CloudBase. [Online]. http://cloudbase.sourceforge.net/

[14] jaql. [Online]. http://code.google.com/p/jsql/wiki/

[15] A. Becherer, "Hadoop Security Design Just Add Kerberos? Really?," in *Black Hat USA 2010*, 2010.

[16] B. Brenner. (2010) Cloud security still a struggle for many companies. [Online]. http://www.csoonline.com/article/620713/cloud-security-still-a-struggle-for-many-companies-survey-finds

[17] S. Byrne. (2010) Emerging technology trends increase risks of protecting corporate information. [Online]. http://www.ey.com/US/en/Newsroo-m/News-releases/Emerging-technology-trends-increase-risks-of-protecting-corporate-information

[18] F. Y. Rashid. (2010) IBM Looks To Cloud Security With New Services. [Online]. http://www.techweekeurope.co.uk/news/ibm-looks-to-cloud-security-with-new-services-10843

[19] I. Roy, S. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and Privacy for MapReduce," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation (NSDI'10)*, 2010, pp. 297-312.

[20] V. Khadilkar, A. Gupta, M. Kantarcioglu, L. Khan, and B. Thuraisingham, "Secure Data Storage and Retrieval in the Cloud," in *Proceedings of the 6th International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2010)*, 2010.

[21] T. H. Hinke, H. S. Delugach, and R. P. Wolf, "Protecting databases from inference attacks," *Computers & Security*, vol. 16, no. 8, pp. 687–708, 1997.

[22] L. Sweeney, "k-ANONYMITY: A MODEL FOR PROTECTING PRIVACY," *International Journal of Uncertainty, Fuzziness, Knowledge-Based Systems*, vol. 10, no. 5, pp. 557–570, 2002.

[23] R. Shokri, C. Troncoso, C. Diaz, J. Freudiger, and J. P. Hubau, "Unraveling an Old Cloak: k-anonymity for Location Privacy," in *Proceedings of the 2010 Workshop on Privacy in the Electronic Society (WPES 2010)*, 2010, pp. 115-118.

[24] T. Hashem and L. Kulik, "Don't trust anyone": Privacy protection for location-based services," *Pervasive and Mobile Computing*, vol. 7, no. 1, pp. 44-59, 2011.

[25] C. Dwork, "An Ad Omnia Approach to Defining and Achieving Private Data Analysis," in *Proceedings of the 1st ACM SIGKDD International workshop on Privacy, Securiy, and Trust in KDD (PinKDD'07)*, vol. 1, 2007, pp. 1-13.

[26] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-Diversity: Privacy Beyond k-Anonymity," *ACM Transactions on Knowledge Discovery and Data*, vol. 1, no. 1, p. Article No. 3, 2007.

[27] N. Li, T. Li, and S. Venkatasubramanian, "t-Closeness: Privacy Beyond k-Anonymity and l-Diversity," in *Proceedings of the IEEE 23rd International Conference on Data Engineering (ICDE 2007)*, 2007, pp. 106-115.

[28] R. J. Bayardo and R. Agrawal, "Data Privacy Through Optimal k-Anonymization," in *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, 2005, pp. 217-228.

[29] D. Rebollo-Monedero, J. Forne, and J. Domingo-Ferrer, "From t-Closeness-Like Privacy to Postrandomization via Information Theory," *IEEE Transactions on Knowledge and Data Engineering*, vol. 22, no. 11, pp. 1623-1636, 2010.

[30] J. Brickell and V. Shmatikov, "The Cost of Privacy: Destruction of Data-Mining Utility in Anonymized Data Publishing," in *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2008)*, 2008, pp. 70-78.

[31] T. Li and N. Li, "On the Tradeoff Between Privacy and Utility in Data Publishing,"

in *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD 2009)*, 2009, pp. 517-526.

[32] M. Sramka, R. Safavi-Naini, J. Denzinger, and M. Askari, "A Practice-oriented Framework for Measuring Privacy and Utility in Data Sanitization Systems," in *Proceedings of the 2010 EDBT/ICDT Workshops (EDBT/ICDT 2010)*, 2010, p. Article No. 27.

[33] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic Databases," in *Proceedings of the 28th International Conference on Very Large Databases (VLDB'02)*, 2002, pp. 143-154.

[34] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi, "Extending Relational Database Systems to Automatically Enforce Privacy Policies," in *Proceedings of the 21st International Conference on Data Engineering (ICDE 2005)*, 2005, pp. 1013-1022.

[35] K. LeFevre, Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt, "Limiting Disclosure in Hippocratic Databases," in *Proceedings of the 30th International Conference on Very Large Databases (VLDB'04)*, 2004, pp. 108-119.

[36] B. Lampson, "Protection," *ACM Operating Systems Review*, pp. 8-24, 1974.

[37] M. A. Harrison, M. L. Ruzzo, and J. D. Ullman, "Protection in Operating Systems," *Communications of the ACM*, pp. 461-471, 1976.

[38] R. Murthy and E. Sedlar, "Flexible and Efficient Access Control in Oracle," in *Proceedings of the 2007 ACM SIGMOD international conference on Management of data (SIGMOD '07)*, 2007, pp. 973-980.

[39] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational data bases," in *Proceedings of the 9th annual ACM symposium on Theory of computing (STOC '77)*, 1977, pp. 77-90.

[40] H. R. Chinaei, "An Ordered Bag Semantics for SQL," University of Waterloo, Masters's Thesis 2007.

[41] T. S. Jayram, P. G. Kolaitis, and E. Vee, "The Containment Problem for Real

Conjunctive Queries with Inequalities," in *Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (PODS'06)*, 2006, pp. 80-89.

[42] A. Y. Halevy, "Answering queries using views: A survey," *The VLDB Journal*, vol. 10, no. 4, pp. 270-294, 2001.

[43] H. J. Moon, C. A. Curino, and C. Zaniolo, "Scalable Architecture and Query Optimization for Transaction-time DBs with Evolving Schemas," in *Proceedings of the 2010 ACM SIGMOD international conference on Management of data (SIGMOD '10)*, 2010.

[44] J. Y. Kao, "Computing query answers with consistent support," in *Proceedings of the 3rd SIGMOD PhD Workshop on Innovative Database Research (IDAR 2009)*, 2009, pp. 207-218.

[45] Q. Wang, T. Yu, N. Li, J. Lobo, E. Bertino, K. Irwin, and J. W. Byun, "On the Correctness Criteria of Fine-Grained Access Control in Relational Databases," in *Proceedings of the 33rd International Conference on Very Large Databases (VLDB '07)*, 2007, pp. 555-566.

[46] C. Dwork and A. Smith, "Differential Privacy for Statistics: What we Know and What we Want to Learn," *Journal of Privacy and Confidentiality*, vol. 1, no. 2, pp. 135–154, 2009.

[47] A. Machanavajjhala, A. Korolova, and A. D. Sarma, "Personalized Social Recommendations - Accurate or Private?," *Proceedings of the 2011 VLDB Endowment (PVLDB 2011)*, vol. 4, no. 7, pp. 440-450, 2011.

[48] C. Băsescu, C. Leordeanu, A. Costan, A. Carpen-Amarie, and G. Antoniu, "Managing Data Access on Clouds: A Generic Framework for Enforcing Security Policies," in *Proceedings of the 25th IEEE International Conference on Advanced Information Networking and Applications (AINA-2011)*, 2011, pp. 459-466.

[49] TPC. (2011) TPC BenchmarkTM H Standard Specification Revision 2.14.0. [Online]. http://www.tpc.org/tpch/spec/tpch2.14.0.pdf

[50] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A Comparison of Approaches to Large-Scale Data Analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09)*, 2009, pp. 165-178.

[51] F. Currim, E. Jung, X. Xiao, and I. Jo, "Privacy Policy Enforcement For Health Information Data Access," in *Proceedings of the 1st ACM International Workshop on Medical-grade Wireless Networks (WiMD 2009)*, 2009, pp. 39-44.

초 록

정보 접근의 증가로 인해 공유 데이터 저장소에 저장된 민감한 데이터에 대한 보안이 중요한 문제로 대두되고 있다. 예를 들어, 미국의 건강 정보 기술 법안(Information Technology for Economic and Clinical Health Act, HITECH)은 환자의 병력 데이터에 대해 보안 조치를 취하지 않은 경우, 무조건 해당 기관을 처벌하도록 규정하고 있다. 따라서 공유 저장소의 민감한 데이터에 대한 세분화된 접근 제어 방법(Fine-Grained Access Control, FGAC)이 시급하다. 그러나 현재까지의 보안 전략들은 너무나 많은 정보를 노출하거나 집합 연산에 대해 부정확한 결과를 생성해내는 문제점을 가지고 있다. 본 논문은 SQL을 사용하는 데이터 웨어하우스 시스템을 대상으로, 해당 시스템에 저장된 민감한 데이터에 대해 보안을 유지할 수 있을 뿐 아니라 정확한 집합 연산 결과를 제공할 수 있는 일반화된 접근 방법을 제안하는 것을 목표로 한다. 본 논문은 보안 전략이 만족시켜야만 하는 특성들을 제시하고, 정보 공개의 수준을 정의한 후, 보안 방침에 따라 사용자 질의를 평가하는 알고리즘을 제시한다. 여기에서 보안 방침은 하나 이상의 규칙으로 구성되며 각 보안 규칙은 SQL로 작성된다고 가정한다. 사용자 질의가 들어오면 보안

방침을 구성하는 규칙에 대해 하나하나 순서대로 해당 규칙이 사용자 질의를 허용하는지를 평가한다. 사용자 질의를 허용하는 보안 규칙이 발견되면, 사용자 질의는 원래 형태 그대로 허용된다. 그러나 보안 방침 내에 사용자 질의를 허용하는 규칙이 하나도 존재하지 않는다면, 데이터 관리자의 설정에 따라 사용자 질의가 거부되거나 혹은 보안 규칙에 맞도록 변경된 후에 허용된다. 각 보안 규칙에 포함된 속성들은 네 가지 범주로 분류되며, 각 범주는 정보 공개의 수준을 나타낸다. 보안 규칙의 각 속성들이 허용하는 정보 공개의 수준에 따라 보안 규칙이 사용자 질의를 허용하는지 결정함으로써 추론 공격(Inference Attack)과 같은 보안 공격에 대응하여 사용자 정보를 보호할 수 있게 된다. 특정 보안 규칙이 사용자 질의를 허용한다면 사용자 질의에 포함된 속성들 모두가 해당 보안 규칙에 의해 허용되어야 한다. 사용자 질의에 포함된 속성들이 보안 규칙에 의해 허용되느냐의 여부는 해당 속성들이 속한 범주에 따라 결정된다. 사용자 질의를 허용하는 보안 규칙이 하나도 없는 경우 (즉, 사용자 질의에 포함된 속성 모두를 허용하는 보안 규칙이 존재하지 않는 경우), 질의는 거부되거나 재 작성된다. 제안된 질의 평가 알고리즘은 질의 재 작성을 위한 첫 번째 단계로서, 보안 방침 내에서 해당 질의가 요청하는 정보를 가장 많이 허용하는 보안 규칙을 선택한 후, 해당 질의에서 선택된 규칙에 위배되는 부

분들을 제거하는 방식으로 질의 재 작성을 수행한다. 이러한 사용자 질의 평가 알고리즘은 프로토타입 시스템으로 구현되었다. 질의 평가 알고리즘을 필요로 하는 데이터 웨어하우스 시스템으로는 데이터베이스와 하둡(Hadoop) 기반의 질의 엔진이 대표적이다. 전통적으로 데이터베이스는 대용량 정보의 저장과 저장된 정보에 대한 효율적인 처리 기능을 제공해 왔다. 그러나 저장되고 분석되는 데이터 용량이 기하급수적으로 증가함에 따라 기존의 데이터베이스는 저장과 처리 용량에서 한계에 도달하게 되었다. 대용량 데이터를 효율적으로 처리할 수 있는 강력한 해결책으로서 클라우드 컴퓨팅이 대두되고 있으며, 사용자들에게 친숙한 인터페이스를 제공하고 질의를 작성하는 부담을 덜어주기 위해 SQL을 기본 언어로 사용하는 하둡 기반의 데이터 웨어하우스 시스템들이 다수 등장하고 있다. 따라서 이 두 가지 대표적인 데이터 웨어하우스 시스템을 대상으로 제안된 질의 평가 알고리즘을 통해 보안 방침을 집행할 수 있는 프로토타입 시스템을 구현했다. 구현된 프토로타입 시스템에 대한 실험 결과에 따르면 제안된 보안 전략은 부하가 매우 적을 뿐 아니라 확장성도 높은 것으로 나타났다.

주요어: 개인 정보, 세분화된 접근 제어, 정책 집행, 질의 평가, 데이터베이스, 하둡 기반의 데이터 웨어하우스 시스템
학 번: 2004-23601

감사의 글

많이 부족한 저를 졸업시키느라 정말 고생하신 염헌영 교수님께 우선 감사의 말씀을 드립니다. 그리고 부족한 논문을 심사해주신 신현식 교수님, 전화숙 교수님, 엄현상 교수님, 이경오 교수님께도 감사 드립니다.

지겹게 학교만 다니는 딸을 그 동안 참아준 아빠, 엄마도 고맙고 지금까지보다는 더 나은 딸이 되도록 노력하겠습니다. 못되게 굴 때도 있었는데 많이 참아주고 배려해 준 임영이 언니에게도 이 글을 빌어 고맙다는 말을 전하고 싶습니다.