Limiting Answers to Queries to Enhance Security of Mobile Database

Dongyi Chen and Subhasish Mazumdar

Abstract—**While mobile computing has seen tremendous growth and popularity, it has also introduced vulnerabilities in information systems. When a mobile personal computing device is stolen or misplaced, a great amount of data obtained from database servers can be compromised; hence, it is useful to limit the amount of sensitive data on mobile clients. In a number of applications, it is necessary to limit the amount of answers in response to a user query in order to enhance the security of a database; for example, an army base can answer queries asking for the phone numbers of its residents and yet, it should not reveal the whole book. Since databases are large and dynamic in content and structure, and the results of queries are unpredictable, it is not feasible to manually specify exactly which tuple should be suppressed for which user. In this paper, our approach is based on declarative specifications: the Database Administrator specifies the secrecies, i.e., the queries whose answers need to be limited, and the user privileges, i.e., the number of tuples that can be revealed when a user query intersects with a secrecy. The output of every query that intersects with one of the secrecies will be limited in the number of tuples revealed.**

Index Terms—**Mobile database, query filtering, security.**

I. Introduction

While mobile computing has led to tremendous growth towards the availability of data to users any time and anywhere, it has also led to certain weaknesses. Mobility poses new challenges to the mobile database management [1], [2]. For example, when a mobile device is lost, a great amount of data obtained from database servers can be compromised. This problem of security is different from the traditional one.

Previous related work have focused on authorization, e.g., context-sensitive authorization systems and the protection of context information used in authorization rules or facts [3]-[6]. Another approach has aimed at developing mobile secure policy or code management system on the administrator domain [7]-[9]. It has been shown that restricted policies can be developed to ensure individual privacy by publishing data without revealing confidential information [10]-[13].

None of the above methods consider limiting the amount of sensitive data at the query level. In a number of applications, it is necessary to limit the number of answers in the response to a user query in order to enhance the security of a mobile database. A well-known example is that of an army base which can answer queries asking for the phone numbers of its residents and yet, it should not reveal the whole phone book. A more interesting example is that of a soldier whose query for available resources should be answered mindful of the fact that the soldier may be captured by enemy forces and the information compromised. It is useful to note that authorization is not the answer as the queries are valid and should be answered. Similarly, cryptography is not adequate either; consider the fact that most mobile devices do not have the level of protection that desktops do and also they are kept switched on for ease of access enabling access to data already on the machine. In such cases, it is useful to limit the amount of sensitive data on mobile clients.

However, databases are typically very large, differ markedly in design, and cater to a large number of users, which makes it infeasible to manually specify exactly when and to whom an answer tuple should be suppressed [14]. Our approach is based on declarative specifications: the Database Administrator (DBA) specifies the secrecies, i.e., the queries whose answers need to be limited, and the user privileges, i.e., number of tuples that can be revealed when a user query intersects with one of the secrecies.

In the next section, we describe our query filtering system. In the next section, we outline performance issues. Subsequently, we discuss future work and finally present concluding remarks.

II. Query Filtering System

Often, a user of an information system keeps accessing certain pieces of sensitive information. For example, a stock broker might just be interested in his/her customers' information in a stock-trading system. As mentioned earlier, the query-based filtering system is designed to help the DBA to limit the sensitive data disclosed. In this system, the DBA specifies the secrecies and the user privileges. The following three tables are to be configured by DBA:

1) USERS (userid, loginname) is for storing users information of the system.
2) SECRECIES(SecrecyID, Secrecy) is for storing query statements, called secrecies, which represent the information that the DBA wants to limit. Each secrecy has a corresponding filter, which we explain later.
3) ACCESSNUMBER(UserID,SecrecyID, AccessNumber) is for storing the user privilege information by specifying the maximum quantity of query result for a user to execute a query that intersects with a secrecy.

These tables are invisible to the users; only the DBA can access them. The first time a query is executed that intersects with a secrecy, a filter will be created for that user according to the query results. Every later query that intersects with the same secrecy will consult this filter and may also modify its contents.

Manuscript received August 18, 2012; revised October 9, 2012.

The authors are with the New Mexico Institute of Mining and Technology, Socorro, NM 87801 USA (e-mail: dchen@ cs.nmt.edu, mazemdar@ cs.nmt.edu).

When a user sends a query, the system computes whether or not it matches any secrecy. If there is no matching secrecy, the system returns the whole query result. However, if there is a matching secrecy, the system will consult the filter table related to that secrecy. If the number of tuples is less than the user privilege, then the difference denotes the number of brand-new tuples that can be returned to the user and stored in the filter table. Any tuple in the result that is already in the related filter table can also be returned. If the secrecy related filter table does not exist, which means that the user is executing a query intersecting with this secrecy for the first time, the system will create a related filter table automatically and use it as described above. When the user executes a similar query that matches the secrecy again, the system will filter the result with the filtering data. Furthermore, the data in the filter table has a lifetime and a function outdated() returns true on tuples that are too old. While filtering, we replace some "outdated" filtering data in the related filtering table with some new data from the current query. For example, in our prototype, if a tuple has not been accessed for one month, is taken to be "outdated data".

A. Algorithm

The system is built in the server side to protect data from being intercepted during the transmission. When a query statement was sent to the server, the system would work correspondingly. Fig 1 illustrates the algorithm of the main function.

In this algorithm, the parameter input is the query sent by the user. First, in line 1, the current user's id is retrieved and stored in uid. Next, the inputted query statement will be parsed into a standard format in line 2, as discussed in section B. Thirdly, the most similar secrecy (MSS) will be identified and stored in s in line 3, as discussed in section C. If no MSS is found, then the full results of the input would be returned, which is handled in lines 4-7. Fourthly, in line 8, the maximum number of tuples that can be accessed by the user to the query is retrieved and stored in max. Fifthly, in line 9, the name of related filter table (RFT) is formed. If the RFT did not exist, the system would create an RFT as well as the corresponding triggers, as shown in lines 10-16 and discussed in section D and E. Sixthly, three accessorial temporary tables will be created in line 17 and discussed in section F. Seventhly, the filtering process will be executed in line 18-45, as discussed in section G. Finally, the results will be outputted in line 46, as discussed in section H.

B. Parser

Because one query statement can be represented in various formats, which makes it very difficult to evaluate the similarity between two queries, we designed a parser to parse every query statement into a standard format. The format follows these rules:

1) every letter is uppercase;
2) all the aliases are replaced by real table names;
3) all the attributes are represented in the form "table. attribute".

For example, the query statement
select a.×,b.name
from hrd a, project b
where a. proid=b. proid
and budget>=50000
order by a. empid
would be parsed into
select hrd.empid, hrd. proid, project. name from hrd, project
where hrd. proid=project. proid
and project. budget>=50000
order by hrd. empid.

```
Main(input)
  --get current userID
1  uid ← getUserID();
      --Parse the inputted query into a standard format
2  Parse(input)
      --Find out the Most Similar Secrecy(MSS) in table SECRECIES
3  s ← MSS(input)
4  If s = 0 then
        --if there is no MSS, then return all the results of the inputted query
5      Cursor c for input;
6      Return c;
7  End if
      --get maximum tuple number the user can access
8    Select AccessNumber into max from AccessNumber where
UserID=uid and secrecyid=s;
      -- name of the Related Filter Table (RFT)
9  rft ← 'FILTER_' + uid + '_' + s;
      --get related filter table
10  Select count(*) into i from USER_TAB_COLS where
tablen_name=rtf;
11 If i = 0  --no related filter table: create it
12     (uid,s);
13    For each table tb in secrecy whose secrecyID is s
        --  to synchronize data in tb with those in tb's RFT
14        Trigger(tb); -- create trigger
15    End loop
16 End if
      --Create three temporary tables: T1, T2 and T3
17 CreateTemporaryTables();
  --Filter the result
18 Cursor c for input;
19 For each tuple t in c
20   Insert(t,T1);
21 End loop;
22 For each tuple t in T1
23   m ← 0;          --there is no corresponding tuple in rft
    --try to identify the corresponding tuple of t from rtf,
24   match(t,r);     -- and cache it into r
25   if r is not null then
26      Insert(t,T2);
27      Update r set last_access_date=sysdate;
28   else
29      Select count(*) into j from rft;
30      If max>j then
31         Insert(t,T2);
32         Insert(t,rft);
33      Else
34         Insert(t,T3);
35      End if;
36   End if;
37 End for
38 For each tuple t in T3
39   o ← the most "outdated" tuple from RFT;
40   if o is not null
41      delete(o,rft);
42      insert(t,T2);
43      insert(t,rft);
44   end if
45 End for
46 Output(T2)
End Main;
```

Fig. 1. Algorithm of the main filter function.

C. MSS Function

This function is for finding the most similar secrecy (MSS) of the query. The system will go through the SECRECIES table, and find out the most similar secrecy to the inputted query by the rule "with the maximum matching table/condition number, and the minimum unmatched table/condition number".

Matching table is the table name that exists in both the inputted query statement and the secrecy. In table I, the table name HRD, which is shown in bold and italics, is considered as a matching table because it occurs in both the inputted query statement and the secrecy.

TABLE I: THE MATCHING TABLES

Input query statement	Secrecy
select … from HRD,PROJECT where…	select … from HRD,EMP where …

Unmatched table is a table name that exists in the secrecy, but not in the inputted query statement. For example, in table II, EMP, shown in bold and italics, is an unmatched table.

TABLE II: THE UNMATCHED TABLES

Input query statement	Secrecy
select … from HRD,PROJECT where…	select … from HRD,EMP where …

Matching condition is a predicate in the inputted query statement, which also occurs in the secrecy. In the standard format, some semantic condition matching can be determined through literal string matching. Two strings are regarded as matched either if they are identical or become identical using the commutativity of equality. Table III shows an example of a matching condition. "HRD.PROID=PROJECT.PROID" is regarded as the same as "PROJECT.PROID= HRD.PROID" because if the two sides of the equality are switched, then the two strings become identical.

TABLE III: THE MATCHING CONDITIONS

Input query statement	Secrecy
select … from … where HRD.PROID=PROJECT.PROID AND PROJECT.BUDGET < =50000	select …from … where PROJECT.PROID =HRD.PROID AND PROJECT.BUDGET < 50000

Unmatched condition is a predicate that is in the secrecy but cannot be found in the inputted query statement. Two strings are regarded as matched only if they are literally the same as each other. Table IV shows an example of an unmatched condition: "PROJECT.BUDGET<50000" does not match "PROJECT.BUDGET<=50000".

TABLE IV: THE UNMATCHED CONDITION

Input query statement	Secrecy
select …from … where HRD.PROID=PROJECT.PROID AND PROJECT.BUDGET < =50000	select …from … where PROJECT.PROID =HRD.PROID AND PROJECT.BUDGET < 50000

D. Create RTF Function

When the related filter table (RFT) is not found, the system will create an RFT by using the name as the format: "FILTER_USERID_SECRECYID", Where USERID is the unique ID of a user, and the SECRECYID is the unique ID of MSS in table SECRECIES. The filter table includes all the attributes of the tables in the secrecy, and each attribute is in the format "table_attribute". Besides that, the RFT has an additional attribute "LAST_ACCESS_DATE" to indicate the date when last update has been applied to that tuple; this is useful for the outdated function.

E. Trigger

For every table which has an RFT, synchronization should be maintained between the data in it and its RFT. The synchronization means that once a tuple in T1 is updated or deleted, the same operation will be executed on the corresponding tuple in all T1's RFTs. This is implemented using triggers in Oracle [15]. The problem then is how to identify all the RFTs of a table. This is addressed by a two step process: first, arbitrarily pick up one of the attributes of the table and parse it into the form "table_attribute". Because all the filter tables include all the attributes of the related business tables, any attribute can work. Second, search the system table USER_TAB_COLS, which records all the tables and their attributes, according to the attributes achieved from the first step for all the RFT.

F. Temporary Tables

The system creates three temporary tables based on the input query and named after the filter table name. For example, if the filter table is FILTER_1_2, then the three temporary tables will be FILTER_1_2_tmp denoted as T1, FILTER_1_2_output denoted as T2 and FILTER_1_2_wait denoted as T3.

1) FILTER_1_2_tmp is for storing the original query results of the input query before filtering.
2) FILTER_1_2_ output is for storing the output result after filtering.
3) FILTER_1_2_wait is for storing some intermediate results which might be outputted but need further estimation.

These three tables have the same attributes, which include all the attributes of the tables in the input query. For example, even if the input query is

select name from hrd,project where …,

the attributes of the three temporary tables will also be "HRD_EMPID, HRD_PROID, PROJECT_PROID, PROJECT_NAME, PROJECT_BUDGET" to make sure that the attributes include all the primary keys of the table HRD and PROJECT; this simplifies the procedure of finding the corresponding tuples among the RFTs and these temporary tables. The way to find out the corresponding tuples is similar to that mentioned in the trigger part, but uses the exact attribute names.

G. Filtering

First, the full results of the original inputted query are retrieved and cached in T1. Secondly, for each tuple in T1, if there exists a corresponding tuple in RTF or the number of tuples in RTF is less than max, the maximum number of tuples that the user can access, it will be cached into T2. Otherwise, it will be cached into T3. Thirdly, if there exists some "outdated" tuples in RTF, which means they are useless and can be replaced by useful data, they will be replaced by tuples arbitrarily chosen from T3; those chosen tuples will also be cached into T2 and stored into RTF.

H. Output

The system gets the attributes and order by clause from the inputted query, then parses the attributes to the format "table_attribute". After that, the system gets the results from T2 projected by the parsed attributes and processed using the order by clause, if any, and output the results.

III. Performance Issues

Some simulation experiments aiming at studying the performance of the system have been done. A runtime environment was built, on which several Information Systems with 5 GB data on average were simulated. First, our system was embedded into those Information Systems easily. Except several tables being added, no other change was done to the original database. The interface connecting the database and the client was changed. We observe that our approach can be integrated with most information systems.

Secondly, we simulated some attack scenarios in which the account information of a user was stolen by a malicious person who attempted to access sensitive information by executing queries. Protected by our system, however, no sensitive information was compromised, except those already cached in the device. However, if the malicious person kept attacking the system for a long time, i.e., beyond the designed lifetime, then some other sensitive information could be compromised. Note that the lifetime implies that data becomes too stale to matter.

As designed, two parameters, the maximum accessing number and the computed parameter *outdated*, in this system were decisive for the security level of this system. It is very clear that the maximum accessing number is important because it determines the quantity of data cached in a user device. So, the smaller the maximum accessing number, the more secure the system, while providing less information to the user. That is a tradeoff, which should be considered by the DBA. The *outdated* function defines how long a piece of data will be regarded as useful data since it was visited last. On the other hand, the greater the lifetime, the more secure the system because the attacker has to wait longer to learn new data. We assigned the lifetime to be 30 days in our system, which might be changed in different situations.

Thirdly, as is expected, the query speed was reduced by adding seconds on average to the query execution time.

IV. Future Works

First, the parser needs enhancement to deal with semantic query. In a query statement, a condition can be represented in different format. For example, condition "$A >= B$" carries the same meaning of condition "$A > B$ or $A = B$". Second, we will do some optimization work on this system to improve its execution performance which is discussed in the last section. Thirdly, a data dictionary might be generated to facilitate the queries.

There are strategies for compensating for the loss in speed. For example, if the results of a query are a subset of one of the previous queries, the results can be retrieved directly from the filter rather than the data tables which usually contain much more data than the filters. Especially in a distributed environment, the time spent on transmitting data among different physical data resources can be saved, which will increase the query speed greatly.

References

[1] T. Imielinksi and B. R. Badrinath, "Wireless mobile computing: challenges in data management," *Communications of ACM*, vol. 37, no. 10, 1994.

[2] S. K. Madria, M. Mohania, *et al.*, "Mobile data and transaction management," *Information Sciences*, vol. 141, no. 3-4, pp. 279-309. 2002.

[3] J. Al-Muhtadi, A. Ranganathan, R. Campbell, and D. Mickunas, "Cerberus: A context-aware security scheme for smart spaces," *The First IEEE International Conference on Pervasive Computing and Communications IEEE Computer Society*, pp. 489-496, March 2003.

[4] M. J. Covington, W. Long, S. Srinivasan, A. K. Dey, M. Ahamad, and G. D. Abowd, "Securing context-aware applications using environment roles," *The Sixth ACM Symposium on Access Control Models and Technologies, ACM Press*, pp. 10-20, 2001.

[5] G. Myles, A. Friday, and N. Davies, "Preserving privacy in environments with location-based applications," *IEEE Pervasive Computing*, vol. 2, no. 1, pp. 56-64, 2003.

[6] K. Minami and D. Kotz. "Secure context-sensitive authorization," *Pervasive and Mobile Computing*, vol. 1, no. 1, pp. 123-156, March 2005.

[7] G. Edjlali, A. Acharya, and V. Chaudhary, "History-based access Control for mobile code," *Proceeding of ACM Computer and Communications Security Conference*, 1998.

[8] D. Evans and A. Twyman, "Flexible policy-directed code safety," *IEEE Secur Privacy*, May 1999.

[9] Y. Song, D. Brett, and Fleisch, "Rico: A security proxy for mobile code," *Computers and Security*, vol. 23, no. 4, pp. 338-351, June 2004.

[10] L. Sweeney, "k-Anonymity: A model for protecting privacy," *International Journal on Uncertainty, Fuzziness and Knowledge-based Systems*, vol. 10, no. 5, pp. 557-570, 2002.

[11] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-Diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data*, vol. 1, no. 1, March 2007.

[12] N. Li, T. Li and S. Venkatasubramanian, "t-Closeness: Privacy beyond k-anonymity and l-diversity," *ICDE*, 2007.

[13] D. J. Martin, D. Kifer. A. Machanavajjhala, J. Gehrke, and J. Y. Halpern, "Worst-case background knowledge for privacy preserving data publishing," *ICDE*, 2007.

[14] J. Melton and A. R. Simon, "Privileges, users, and security," *SQL: 1999*. 2002.

[15] Oracle Database PL/SQL User's Guide and Reference 10g Release 2 (10.2). B14261-01, 2005.