MVAL: Addressing the Insider Threat by Valuation-based Query Processing

Stefan Barthel
Institute of Technical and Business Information Systems
Otto-von-Guericke-University Magdeburg
Magdeburg, Germany
stefan.barthel@ovgu.de

Eike Schallehn
Institute of Technical and Business Information Systems
Otto-von-Guericke-University Magdeburg
Magdeburg, Germany
eike.schallehn@ovgu.de

ABSTRACT

The research presented in this paper is inspired by problems of conventional database security mechanisms to address the insider threat, i.e. authorized users abusing granted privileges for illegal or disadvantageous accesses. The basic idea is to restrict the data one user can access by a valuation of data, e.g. a monetary value of data items, and, based on that, introducing limits for accesses. The specific topic of the present paper is the conceptual background, how the process of querying valuated data leads to valuated query results. For this, by analyzing operations of the relational algebra and SQL, derivation functions are added.

1. INTRODUCTION

An acknowledged main threat to data security are fraudulent accesses by authorized users, often referred to as the insider threat [2]. To address this problem, in [1] we proposed a novel approach of detecting authorization misuse based on a valuation of data, i.e. of an assigned description of the worth of data management in a system, which could for instance be interpreted as monetary values. Accordingly, possible security leaks exist if users access more valuable data than they are allowed to within a query or cumulated over a given time period. E.g., a bank account manager accessing a single customer record does not represent a problem, while dumping all data in an unrestricted query should be prohibited. Here, common approaches like role-based security mechanisms typically fail.

According to our proposal, the data valuation is first of all based on the relation definitions, i.e. as part of the data dictionary information about the value of data items such as attribute values and, derived from that, entire tuples and relations. Then, a key question is how the valuation of a query result can be derived from the input valuations, because performing operations on the base data causes transformations that have an impact on the data's significance.

This problem is addressed in the research presented here by considering relational and SQL operations and describing possible valuation derivations for them.

25^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 28.05.2013 - 31.05.2013, Ilmenau, Germany.

2. PRINCIPLES OF DATA VALUATION

In [1] we outlined our approach of a leakage-resistant data valuation which computes a monetary value (*mval*) for each query. This is based on the following basic principles: Every attribute $A_i \in R$ of a base relation schema R is valuated by a certain monetary value ($mval(A_i) \in \mathbb{R}$). The attribute valuation for base tables are part of the data dictionary and can for instance be specified as an extension of the SQL DDL:

```
CREATE TABLE table_1
(
  attribute_1 INT PRIMARY KEY MVAL 0.1,
  attribute_2 UNIQUE COMMENT 'important' MVAL 10,
  attribute_3 DATE
);
```

With these attribute valuations, we derive a monetary value for one tuple $t \in r(R)$ given by Equation (1), as well as the total monetary value of the relation $r(R)$ given by Equation (2), if data is extracted by a query.

$$mval(t \in r(R)) = \sum_{A_i \in R} mval(A_i) \quad (1)$$

$$mval(r(R)) = \sum_{t \in r(R)} mval(t) = |r(R)| * mval(t \in r(R)) \quad (2)$$

To be able to consider the *mval* for a query as well as several queries of one user over a certain period of time, we log all *mvals* in an alert log and compare the current cumulated *mval* per user to two thresholds. If a user exceeds the first threshold – suspicious threshold – she will be categorized as suspect. After additionally exceeding the truncation threshold her query output will be limited by hiding tuples and presenting a user notification. We embedded our approach in an additional layer in the security defense-in-depth model for raw data, which we have enhanced by a backup entity (see Fig. 1). Furthermore, encryption has to be established to prevent data theft via unauthorized, physical reads as well as backup theft. In this paper we are going into detail about how to handle operations like joins, aggregate functions, stored procedures as well as common functions.

Most of the data stored in a database can be easily identified as directly interpretable. One example would be an

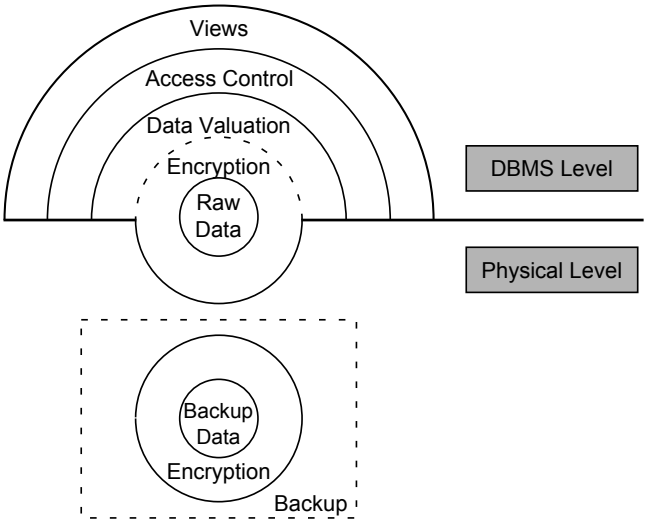

Figure 1: Security defense model on DBMS and physical level

employee-table, where each tuple has a value for attributes "first name", "surname" and "gender". In this case, it is also quite easy to calculate the monetary value for a query ($r(R_{emp})$) by simply summarizing all *mval* per attribute and multiply those with the number of involved rows (see Eq. (3)).

$$mval(r(R_{emp})) = \sum_{A_i \in R_{emp}} mval(A_i) * |r(R_{emp})| \qquad (3)$$

However, it becomes more challenging if an additional attribute "license plate number" is added, which does have some unset or unknown attribute values - in most cases *NULL* values. By knowing there is a *NULL* value for a certain record, this could be interpreted as either simply unknown whether there is any car or unset because this person has no car. So there is an uncertainty that could lead to an information gain which would be uncovered if no adequate valuation exists. Some other potentially implicit information gains are originated from joins and aggregate functions which we do mention in the regarding section.

Because the terms *information gain* and *information loss* are widely used and do not have a uniform definition, we do define them for further use. We call a situation where an attacker received new data (resp. information) *information gain* and the same situation in the view of the data owner an *information loss*.

Uncertainty Factor

Some operators used for query processing obviously reduce the information content of the result set (e.g. selection, aggregations, semi joins, joins with resulting NULL values), but there is still an uncertain, implicit information gain. Since, the information gain by uncertainty is blurry, meaning in some cases more indicative than in others, we have to distinguish uncertainty of one attribute value generated out of one source attribute value (e.g., generated NULL values) and attribute values which are derived from information of several source attribute values (e.g., aggregations). In case of one source attribute value, an information gain by uncertainty has to be less valuable than properly set attribute values. Therefore, the monetary value should be only a percentage of the respective monetary value of an attribute value. If several source attribute values are involved, we recommend to value the computed attribute value as a percentage of the monetary value of all participating source attribute values. In general, we suggest a maximum of 50% for both valuations. Furthermore, we need to consider the overall purpose of our leakage-resistant data valuation which shall prevent extractions of large amounts of data. Therefore, the percentage needs to be increased with the amount of data, but not in a way that an unset or unknown attribute value becomes equivalent valuable than a properly set one. For that reason, exponential growth is not a suitable option. Additionally, we have to focus a certain area of application, because a trillion attributes (10^{12}) are conceivable whereas a septillion attributes (10^{24}) are currently not realistic. From the overall view on our data valuation, we assume depending on the application, that the extraction of sensitive data becomes critical when 10^3 up to 10^9 attribute values will be extracted. Therefore, the growth of our uncertainty factor *UF* increases much more until 10^9 attribute values than afterwards, which predominantly points to a logarithmic growth. We also do not need to have a huge difference of the factor if theoretically much more attribute values shall be extracted (e.g., 10^{14} and more), because with respect to an extraction limiting approach, it is way too much data to return. This assumption does also refer to a logarithmic increase. We conclude that the most promising formula that was adapted to fit our needs is shown in Eq. (4).

$$UF = \frac{1}{30} log_{10}(|val_{A_i,\ldots,A_k}| + 1) \qquad (4)$$

3. DERIVING VALUATIONS FOR DATABASE OPERATIONS

In this chapter we will describe valuation derivation for main database operations by first discussing core relational operations. Furthermore, we address specifics of join operations and finally functions (aggregate, user-defined, stored procedures) which are defined in SQL.

3.1 Core Operations of Relational Algebra

The relational algebra [4] consists of six basic operators, where *selection*, *projection*, and *rename* are unary operations and *union*, *set difference*, and *Cartesian product* are operators that take two relations as input (binary operation). Due to the fact that applying *rename* to a relation or attribute will not change the monetary value, we will only consider the rest.

Projection

The projection $\pi_{attr\_list}(r(R))$ is a unary operation and eliminates all attributes (columns) of an input relation $r(R)$ except those mentioned in the attribute list. For computation of the monetary value of such a projection, only *mval* for chosen attributes of the input relation are considered while taking into account that a projection may eliminate

duplicates (shown in Eq. (5)).

$$mval(\pi_{A_j,..,A_k}(r(R))) = \sum_{i=j}^{k} mval(A_i) * |\pi_{A_j,..,A_k}(r(R))| \tag{5}$$

Selection

According to the relational algebra, a *selection* of a certain relation $\sigma_{pred}r(R)$ reduces tuples to a subset which satisfy specified predicates. Because the *selection* reduces the number of tuples, the calculation of the monetary value does not have to consider those filtered tuples and only the number of present tuples are relevant (shown in Eq. (6)).

$$mval(\sigma_{pred}(r(R))) = mval(t \in r(R)) * |\sigma_{pred}(r(R))| \tag{6}$$

Set Union

A relation of all distinct elements (resp. tuples) of any two relations is called the *union* (denoted by $\cup$) of those relations. For performing set union, the two involved relations must be union-compatible – they must have the same set of attributes. In symbols, the union is represented as $R_1 \cup R_2 = \{x : x \in R_1 \vee x \in R_2\}$. However, if two relations contain identical tuples, within a resulting relation these tuples do only exist once, meaning duplicates are eliminated. Accordingly, the *mval* of a union of two relations is computed by adding *mval* of both relations, subtracted with *mval* of duplicates (shown in Eq. (7)).

$$\begin{aligned} mval(R_1 \cup R_2) = mval(r(R_1)) + \\ mval(r(R_2)) - \sum_i mval(t_i \in r(R_1 \cap R_2) \end{aligned} \tag{7}$$

Set Difference

The difference of relations R_1 and R_2 is the relation that contains all the tuples that are in R_1, but do not belong to R_2. The set difference is denoted by $R_1 - R_2$ or $R_1 \backslash R_2$ and defined by $R_1 \backslash R_2 = \{x : x \in R_1 \wedge x \notin R_2\}$. Also, the set difference is union-compatible, meaning the relations must have the same number of attributes and the domain of each attribute is the same in both R_1 and R_2. The *mval* of a set difference of two relations is computed by subtracting the *mval* of tuples that have both relations in common from the monetary value of R_1 given by Equation (8).

$$mval(R_1 \backslash R_2) = mval(r(R_1) - \sum_i mval(t_i \in r(R_1 \cap R_2) \tag{8}$$

Cartesian Product

The Cartesian product, also known as cross product, is an operator which works on two relations, just as set union and set difference. However, the Cartesian product is the costliest operator to evaluate [9], because it combines the tuples of one relation with all the tuples of the other relation – it pairs rows from both tables. Therefore, if the input relations R_1 and R_2 have n and m rows, respectively, the result set will contain $n * m$ rows and consist of columns of R_1 and the columns of R_2. Because, the number of tuples of the outgoing relations are known, the monetary value is a summation of all attribute valuations multiplied by number of rows of both relations given by Equation (9). We are fully aware that by a user mistake, e.g. using cross join instead of natural join, thresholds will be exceeded and the user will be classified as potentially suspicious. However, we recommend a multiplication of the monetary value of both source relations instead of a summation due to the fact that the calculation of the monetary value needs to be consistent also by combining different operators. For that reason, by following our recommendation, we ensure that an inner join is valuated with the same monetary value as the respective combination of a cross join (Cartesian product) and selection on the join condition.

$$\begin{aligned} mval(r(R_1 \times R_2)) = \\ mval(t \in r(R_1)) * |r(R_1)| + mval(t \in r(R_2)) * |r(R_2)| \end{aligned} \tag{9}$$

3.2 Join Operations

In the context of relational databases, a join is a binary operation of two tables (resp. data sources). The result set of a join is an association of tuples from one table with tuples from another table by concatenating concerned attributes. Joining is an important operation and most often performance critical to certain queries that target tables whose relationships to each other cannot be followed directly. Because the type of join affects the number of resulting tuples and their attributes, the monetary value of each join needs to be calculated independently.

Inner Join

An inner join produces a result table containing composite rows of involved tables that match some pre-defined, or explicitly specified, join condition. This join condition can be any simple or compound search condition, but does not have to contain a subquery reference. The valuation of an inner join is computed by the sum of the monetary values of all attributes of a composite row multiplied by the number of rows within the result set. Because the join attribute A_{join} of two joined tables has to be counted only once, we need to subtract it (shown in Eq. (10)).

$$\begin{aligned} mval(r(R_1 \bowtie R_2) = |r(R_1 \bowtie R_2)| * \\ (mval(t \in r(R_1)) + (mval(t \in r(R_2)) - mval(A_{join})) \end{aligned} \tag{10}$$

Outer Join

An outer join does not require matching records for each tuple of concerned tables. The joined result table retains all rows from at least one of the tables mentioned in the *FROM* clause, as long as those rows are consistent with the search condition. Outer joins are subdivided further into left, right, and full outer joins. The result set of a left outer join (or left join) includes all rows of the first mentioned table (left of the join keyword) merged with attribute values of the right table where the join attribute matches. In case there is no match, attributes of the right table are set to *NULL*. The right outer join (or right join) will return rows that have data in the right table, even if there's no matching rows in the left table enhanced by atteributes (with NULL values) of the left table. A full outer join is used to retain the non-matching information of all affected tables by including non-matching rows in the result set. To cumulate the monetary value for a query that contains a left or right outer join, we only need to compute the monetary value of an inner join of both

tables and add the *mval* of an antijoin $r(R_1 \triangleright R_2) \subseteq r(R_1)$ which includes only tuples of R_1 that do not have a join partner in R_2(shown in Eq. (11)). For the monetary value of a full outer join, we additionally would consider an antijoin $r(R_2 \triangleright R_1) \subseteq r(R_2)$ which includes tuples of R_2 that do not have a join partner given by Equation (12)).

$$mval(r(R_1 \rtimes R_2)) = mval(r(R_1 \bowtie R_2)) + mval(r(R_1 \triangleright R_2)) \quad (11)$$

$$mval(r(R_1 \ltimes R_2)) = mval(r(R_1 \bowtie R_2)) + mval(r(R_1 \triangleright R_2)) + mval(r(R_2 \triangleright R_1)) \quad (12)$$

Semi Join

A semi join is similar to the inner join, but with the addition that only attributes of one relation are represented in the result set. Semi joins are subdivided further into left and right semi joins. The left semi join operator returns each row from the first input relation (left of the join keyword) when there is a matching row in the second input relation (right of the join keyword). The right semi join is computed vice versa. The monetary value for a query that uses semi joins can be easily cumulated by multiplying the sum of monetary values for included attributes with number of matching rows of the outgoing relation (shown in Eq. (13)).

$$mval(r(R_1 \ltimes R_2)) = \sum_{A_i \in R_1} mval(A_i) * |r(R_1 \ltimes R_2)| \quad (13)$$

Nevertheless, we do have an information gain by knowing join attributes of R_1 have some join partners within R_2 which are not considered. But adding our uncertainty factor *UF* in this equation would lead to inconsistency by cumulating the *mval* of a semi join compared to the *mval* of a combination of a natural join and a projection. In future work, we will solve this issue by presenting a calculation that is based on a combination of projections and joins to cover such an implicit information gain.

3.3 Aggregate Functions

In computer science, an aggregate function is a function where the values of multiple rows are grouped together as input on certain criteria to form a single value of more significant meaning. The SQL aggregate functions are useful when mathematical operations must be performed on all or on a group of values. For that reason, they are frequently used with the *GROUP BY* clause within a *SELECT* statement. According to the SQL standard, the following aggregate function are implemented in most DBMS and the ones used most often: *COUNT*, *AVG*, *SUM*, MAX, and MIN.

All aggregate functions are deterministic, i.e. they return the same value any time they are called by using the same set of input values. SQL aggregate functions return a single value, calculated from values within one column of a arbitrary relation [10]. However, it should be noted that except for *COUNT*, these functions return a *NULL* value when no rows are selected. For example, the function *SUM* performed on no rows returns *NULL*, not zero as one might expect. Furthermore, except for *COUNT*, aggregate functions ignore *NULL* values at all during computation. All aggregate function are defined in SQL:2011 standard or ISO/IEC 9075:2011 (under the general title "Information technology - Database languages - SQL") which is the seventh revision of the ISO (1987) and ANSI (1986) standard for the SQL database query language.

To be able to compute the monetary value of a derived, aggregated attribute, we need to consider two more factors. First of all, we divided aggregate function into two groups: *informative* and *conservative*.

1. *Informative* are those aggregate functions where the aggregated value of a certain aggregate function leads to an information gain of the entire input of all attribute values. This means that every single attribute value participates in the computation of the aggregated attribute value. Representatives for *informative* aggregate functions are *COUNT*, *AVG* and *SUM*.
2. *Conservative*, on the contrary, are those functions where the aggregated value is represented by only one attribute value, but in consideration of all other attribute values. So if the aggregated value are again separated from the input set, all other attribute values will remain. *Conservative* aggregate functions are *MAX* and *MIN*.

The second factor that needs to be considered is the number of attributes that are used to compute the aggregated values. In case of a *conservative* aggregate function, it is simple, because only one attribute value is part of the output. For that reason we recommend to leave the *mval* of the source attribute unchanged (shown in Eq. (14)).

$$mval(A_i) = mval(MAX(A_i)) = mval(MIN(A_i)) \quad (14)$$

For the *informative* aggregate functions the computation is more challenging due to several participating attribute values. Because several input attribute values are concerned, we recommend the usage of our *uncertainty factor* which we already mentioned in a prior section. With the uncertainty factor it is possible to integrate the number of attribute values in a way that a higher number of concerned attributes leads to an increase in percentage terms of the monetary value of the aggregated attribute value given by Equation (15).

$$mval(COUNT(A_i)) = mval(SUM(A_i)) = mval(AVG(A_i)) = \frac{1}{30} log_{10}(|A_i| + 1) * mval(A_i) \quad (15)$$

3.4 Scalar Functions

Besides the SQL aggregate functions, which return a single value, calculated from values in a column, there are also scalar functions defined in SQL, that return a single value based on the input value. The possibly most commonly used and well known scalar functions are:

- UCASE() - Converts a field to upper case
- LCASE() - Converts a field to lower case
- LEN() - Returns the length of a text field
- ROUND() - Rounds a number to a specified degree
- FORMAT() - Formats how a field is to be displayed

Returned values of this scalar functions are always derived from one source attribute value, and some of them do not even change the main content of the attribute value. Therefore, we recommend that the monetary value of the source attribute stays untouched.

3.5 User-Defined Functions

User-defined functions (*UDF*) are subroutines made up of one or several SQL or programming extension statements that can be used to encapsulate code for reuse. Most database management systems (DBMS) allow users to create their own user-defined functions and do not limit them to the built-in functions of their SQL programming language (e.g., TSQL, PL/SQL, etc.). User-defined functions in most systems are created by using the *CREATE FUNCTION* statement and other users than the owner must be granted appropriate permissions on a function before they can use it. Furthermore, *UDFs* can be either deterministic or nondeterministic. A deterministic function always returns the same results if the input is the equal and a nondeterministic function returns different results every time it is called.

On the basis of the multiple possibilities offered by most DBMS, it is impossible to estimate all feasible results of a *UDF*. Also, due to several features like shrinking, concatenating, and encrypting of return values, a valuation of a single or an array of output values is practically impossible. For this reason we decided not to calculate the monetary value depending on the output of a *UDF*, much more we do consider the attribute values that are passed to an *UDF* (shown in Eq. (16)). This assumption is also the most reliable, because it does not matter what happens inside an *UDF* – like a black box – the information loss after inserting cannot get worse.

$$mval(UDF_{output}(A_a, .., A_g)) = mval(UDF_{input}(A_k, .., A_p)) = \sum_{i=k}^{p} mval(A_i) \quad (16)$$

3.6 Stored Procedures

Stored procedures (*SP*) are stored similar to user-defined functions (*UDF*) within a database system. The major difference is that stored procedures have to be called and the return values of *UDFs* are used in other SQL statements in the same way pre-installed functions are used (e.g., *LEN*, *ROUND*, etc.). A stored procedure, which is depending on the DBMS also called *proc*, *sproc*, *StoredProc* or *SP*, is a group of SQL statements compiled into a single execution plan [13] and mostly developed for applications that need to access easily a relational database system. Furthermore, *SPs* combine and provide logic also for extensive or complex processing that requires execution of several SQL statement, which had to be implemented in an application before. Also a nesting of *SPs* is feasible by executing one stored procedure from within another. A typical use for *SPs* refers to data validation (integrated into the database) or access control mechanisms [13].

Because stored procedures have such a complex structure, nesting is also legitimate and *SPs* are "only" a group of SQL statements, we recommend to value each single statement within a *SP* and sum up all partial results (shown in Eq. (17). With this assumption we do follow the principal that single SQL statements are moved into stored procedures to provide a simple access for applications which only need to call the procedures.

$$mval(SP(r(R_j), .., r(R_k))) = \sum_{i=j}^{k} mval(r(R_i)) \quad (17)$$

Furthermore, by summing all partial result, we make sure that the worst case of information loss is considered, entirely in line with our general idea of a leakage resistant data valuation that should prevent a massive data extraction. However, since *SPs* represent a completed unit, by reaching the truncate threshold the whole *SP* will be blocked and rolled back. For that reason, we recommend smaller *SPs* resp. split existing *SPs* in DBS with an enabled leakage resistant data valuation.

4. RELATED WORK

Conventional database management systems mostly use access control models to face unauthorized access on data. However, these are insufficient when an authorized individual extracts data regardless whether she is the owner or has stolen that account. Several methods were conceived to eliminate those weaknesses. We refer to Park and Giordano [14], who give an overview of requirements needed to address the insider threat.

Authorization views partially achieve those crucial goals of an extended access control and have been proposed several times. For example, Rizvi et al. [15] as well as Rosenthal et al. [16] use authorization-transparent views. In detail, incoming user queries are only admitted, if they can be answered using information contained in authorization views. Contrary to this, we do not prohibit a query in its entirety. Another approach based on views was introduced by Motro [12]. Motro handles only conjunctive queries and answers a query only with a part of the result set, but without any indication why it is partial. We do handle information enhancing (e.g., joins), as well as coarsening operations (e.g., aggregation) and we do display a user notification. All authorization view approaches require an explicit definition of a view for each possible access need, which also imposes the burden of knowing and directly querying these views. In contrast, the monetary values of attributes are set while defining the tables and the user can query the tables or views she is used to. Moreover, the equivalence test of general relational queries is undecidable and equivalence for conjunctive queries is known to be NP complete [3]. Therefore, the leakage-resistant data valuation is more applicable, because it does not have to face those challenges.

However, none of these methods does consider the sensitivity level of data that is extracted by an authorized user. In the field of *privacy-preserving data publishing* (*PPDP*), on the contrary, several methods are provided for publishing useful information while preserving data privacy. In detail, multiple security-related measures (e.g., k-anonymity [17], l-Diversity [11]) have been proposed, which aggregate information within a data extract in a way that they can not lead to an identification of a single individual. We refer to Fung et al. [5], who give a detailed overview of recent developments in methods and tools of *PPDP*. However, these mechanisms are mainly used for privacy-preserving tasks and are not in use when an insider accesses data. They are not applicable for our scenario, because they do not consider a line by line extraction over time as well as the information loss by aggregating attributes.

To the best of our knowledge, there is only the approach of Harel et al. ([6], [7], [8]) that is comparable to our data valuation to prevent suspicious, authorized data extractions. Harel et al. introduce the *Misuseability Weight (M-score)* that desribes the sensitivity level of the data exposed to

the user. Hence, Harel et al. focus on the protection of the quality of information, whereas our approach predominantly preserves the extraction of a collection of data (quantity of information). Harel et al. also do not consider extractions over time, logging of malicious requester and the backup process. In addition, mapping attributes to a certain monetary value is much more applicable and intuitive, than mapping to a artificial M-score.

Our extended authorization control does not limit the system to a simple query-authorization control without any protection against the insider threat, rather we allow a query to be executed whenever the information carried by the query is legitimate according to the specified authorizations and thresholds.

5. CONCLUSIONS AND FUTURE WORK

In this paper we described conceptual background details for a novel approach for database security. The key contribution is to derive valuations for query results by considering the most important operations of the relational algebra as well as SQL and providing specific *mval* functions for each of them. While some of these rules are straight forward, e.g. for core operations like selection and projection, other operations like specific join operations require some more thorough considerations. Further operations, e.g. grouping and aggregation or user-defined function, would actually require application specific valuations. To minimize the overhead for using valuation-based security, we discuss and recommend some reasonable valuation functions for these cases, too.

As the results presented here merely are of conceptual nature, our current and future research includes considering implementation alternatives, e.g. integrated with a given DBMS or as part of a middleware or driver as well as evaluating the overhead and the effectiveness of the approach. We will also come up with a detailed recommendation of how to set monetary values appropriate to different environments and situations. Furthermore, we plan to investigate further possible use cases for data valuation, such as billing of data-providing services on a fine-grained level and controlling benefit/cost trade-offs for data security and safety.

6. ACKNOWLEDGMENTS

This research has been funded in part by the German Federal Ministry of Education and Science (BMBF) through the Research Program under Contract FKZ: 13N10818.

7. REFERENCES

[1] S. Barthel and E. Schallehn. The Monetary Value of Information: A Leakage-Resistant Data Valuation. In *BTW Workshops*, BTW'2013, pages 131–138. Köln Verlag, 2013.

[2] E. Bertino and R. Sandhu. Database Security - Concepts, Approaches, and Challenges. *IEEE Dependable and Secure Comp.*, 2(1):2 –19, Mar. 2005.

[3] A. K. Chandra and P. M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proc. of the 9th Annual ACM Symposium on Theory of Computing*, STOC'77, pages 77–90. ACM, 1977.

[4] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *ACM Communication*, 13(6):377–387, June 1970.

[5] B. C. M. Fung, K. Wang, R. Chen, and P. S. Yu. Privacy-Preserving Data Publishing: A Survey of Recent Developments. *ACM Comput. Surv.*, 42(4):14:1–14:53, June 2010.

[6] A. Harel, A. Shabtai, L. Rokach, and Y. Elovici. M-score: Estimating the Potential Damage of Data Leakage Incident by Assigning Misuseability Weight. In *Proc. of the 2010 ACM Workshop on Insider Threats*, Insider Threats'10, pages 13–20. ACM, 2010.

[7] A. Harel, A. Shabtai, L. Rokach, and Y. Elovici. Eliciting Domain Expert Misuseability Conceptions. In *Proc. of the 6th Int'l Conference on Knowledge Capture*, K-CAP'11, pages 193–194. ACM, 2011.

[8] A. Harel, A. Shabtai, L. Rokach, and Y. Elovici. M-Score: A Misuseability Weight Measure. *IEEE Trans. Dependable Secur. Comput.*, 9(3):414–428, May 2012.

[9] T. Helleseth and T. Klove. The Number of Cross-Join Pairs in Maximum Length Linear Sequences. *IEEE Transactions on Information Theory*, 37(6):1731–1733, Nov. 1991.

[10] P. A. Laplante. *Dictionary of Computer Science, Engineering and Technology*. CRC Pressl, London,England, 1st edition, 2000.

[11] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam. L-Diversity: Privacy Beyond k-Anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1):1–50, Mar. 2007.

[12] A. Motro. An Access Authorization Model for Relational Databases Based on Algebraic Manipulation of View Definitions. In *Proc. of the 5th Int'l Conference on Data Engineering*, pages 339–347. IEEE Computer Society, 1989.

[13] J. Natarajan, S. Shaw, R. Bruchez, and M. Coles. *Pro T-SQL 2012 Programmer's Guide*. Apress, Berlin-Heidelberg, Germany, 3rd edition, 2012.

[14] J. S. Park and J. Giordano. Access Control Requirements for Preventing Insider Threats. In *Proc. of the 4th IEEE Int'l Conference on Intelligence and Security Informatics*, ISI'06, pages 529–534. Springer, 2006.

[15] S. Rizvi, A. Mendelzon, S. Sudarshan, and P. Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proc. of the 2004 ACM SIGMOD Int'l Conference on Management of Data*, SIGMOD'04, pages 551–562. ACM, 2004.

[16] A. Rosenthal and E. Sciore. View Security as the Basis for Data Warehouse Security. In *CAiSE Workshop on Design and Management of Data Warehouses*, DMDW'2000, pages 5–6. CEUR-WS, 2000.

[17] L. Sweeney. K-Anonymity: A Model For Protecting Privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 10(5):557–570, Oct. 2002.