PSpec-SQL: Enforcing Privacy Policies in Big Data Analytics

Chen Luo*, Fei He*, Fei Peng*, Dong Yan†, Dan Zhang† and Xin Zhou†
*School of Software, Tsinghua University,
luoc13@mails.tsinghua.edu.cn, hefei@tsinghua.edu.cn, pengf12@mails.tsinghua.edu.cn
†Intel Labs China, {dong.yan, dan.d.zhang, xin.zhou}@intel.com

***Abstract*—Organizations often share business data with third-parties to perform data analytics. However, the business data may contain a lot of customers' private information. Protecting such private information from being misused has become the major concern of these organizations. In this paper, we present PSpec-SQL, a privacy-integrated data analytics system that automatically enforces privacy compliances against the submitted queries. With our system, the data owner encodes her data usage policy into a privacy specification. As usual, the data analyst queries data to perform data analysis, but our system checks each query to ensure only the privacy-compliant queries are executed.**

We have implemented a prototype of PSpec-SQL on top of Spark-SQL, and carried out a case study on TPC-DS benchmark. The results show the practicability of our system with negligible overhead over query processing.

I. Introduction

It is often highly desirable for organizations to share their data with third-parties to perform data analytics. For example, a retail company may pay data analysts to discover the sales trend to make future business decisions, and a hospital may share some medical records to researchers to promote the study of certain disease. However, these data often contain a lot of customers' private information, such as the financial and health information. Improper use of these information can cause severe privacy breaches, which in turn will significantly degrade the organization's reputation and even incur charges or penalties from the government. Protecting customers' privacy is definitely a major concern of these organizations.

A natural idea for privacy protection is to anonymize the data before sharing them with third-parties. However, as the amount of the business data can be very large, this ad-hoc solution is often highly tedious and cost-ineffective. Moreover, ad-hoc anonymization also lacks flexibility. Since anonymization is usually performed beforehand, it is hard to dynamically adjust the privacy and utility based on the actual needs of different analytic tasks.

Thus, from the data owner's point of view, a more desirable and practical approach is to share one copy of data together with a privacy specification. A privacy specification depicts the data usage restrictions with respect to particular analysts and analytic tasks. For example, with traditional access control techniques, the data owner may create several views and grant access privileges based on user roles. However, to adhere to the applicable laws and privacy policies, these specifications are expected to be handled by legal teams [1], who have little technical background. The concepts and operations used in the specification are thus necessarily to be at an abstract level, such that the legal specialists can work on them.

Moreover, in data analytics, information may be correlated and dynamic, which is hard to express with traditional access control techniques and further renders them non-applicable. First, sometimes the access of different information separately is acceptable, but accessing them together may cause problems because of the *association* among them. For example, the access of the customer's address and health condition separately may be acceptable, while accessing them together will cause a severe privacy breach since it directly leaks the health condition for each customer. Second, instead of blindly forbidding the access of sensitive information, proper desensitization is useful to maintain utility. The full IP address may locate individuals, while the *truncated* IP may be acceptable and still usable for data analytics. Data *association* and *desensitization* are important to balance data privacy and utility, which makes them indispensable for practical privacy specifications.

The privacy specification alone does not achieve privacy protection unless it can be automatically enforced. As the first step towards enforcement, the gap between the abstract concepts in the specification and the underling data (relational data in this paper) should be filled through a process called *data labeling*. Although happening at a column level, data labeling is a non-trivial process for two issues. First, the proper meanings of some columns may not be determined statically, but rather depend on how the table is joined with other tables. For example, an *Address* table may contain both customers' addresses and stores' addresses, and such ambiguity can only be resolved when the query is given. Moreover, modern SQL-like data analytics systems also support complex data types, including map, array and struct, which further complicate the data labeling process. But once the data is properly labeled, the privacy specification can be enforced against the submitted queries with program analysis techniques.

Putting things together, we present PSpec-SQL, a privacy-integrated data analytics system which automatically enforces privacy compliances for relational data. PSpec-SQL is built upon Spark-SQL [1], and allows the data owner to specify data usage restrictions with PSpec, a simple and high-level privacy specification language. At runtime, each submitted query is checked with information flow analysis to ensure only the privacy-compliant queries are executed. Note that our system is complementary to the existing privacy-preserving data analysis techniques, e.g., differential privacy [2], and we also show how

[1] https://spark.apache.org/sql/

to integrate PSpec-SQL with differential privacy as a means to implement desensitize operations.

Our Contributions. In this paper, we present the design and implementation of PSpec-SQL, which mainly consists of the following parts:

- We present PSpec, a high-level privacy specification language suitable for data analytics.
- To link the PSpec concepts with the underling data, we provide comprehensive support for labeling relational data.
- We present the privacy checking algorithm for SQL queries with information flow analysis techniques.
- Finally, to evaluate our system, we carried out a case study on TPC-DS benchmark [3]. The results show the usability and practicability of our system with negligible overhead over query execution.

The rest of the paper is organized as follows. Section II presents an overview of our system, and Sections III to V introduce the details of the PSpec language, data labeling and privacy checking algorithm respectively. Section VI discusses how to integrate our system with differential privacy to implement desensitize operations. Section VII reports our implementation and evaluation. Section VIII further discusses several other issues and Section IX briefly reviews the related works. Finally, Section X concludes this paper.

II. System Overview

In this section, we discuss a motiving example and present an overview of our system.

A. Motivating Example

As a motivating and running example, consider a retail company that owns customer and sales data. The data schema is shown in Figure 1, where primary keys are noted with parentheses and foreign key references are represented by arrows. Each table is briefly explained as follows.

- *Customer*: stores customers' personal information, and customers' addresses is stored in the *Address* table.
- *Store_Sales*: stores sales' information, and references the *Customer*, *Store* and *Item* tables.
- *Item*: stores items' information.
- *Store*: contains stores' information, and stores' addresses are stored in the *Address* table.
- *Address*: stores the addresses for customers and stores.

In order to perform data analysis, the company shares the business data with third-party companies under certain agreements. However, one major concern of the retail company is that the customer's information (the *Customer* and *Address* tables) may be improperly used by the data analyst. For example, the malicious data analyst may query all customers' names with living addresses and sell them to other companies, which directly violates the company's privacy policy and causes severe privacy breaches.

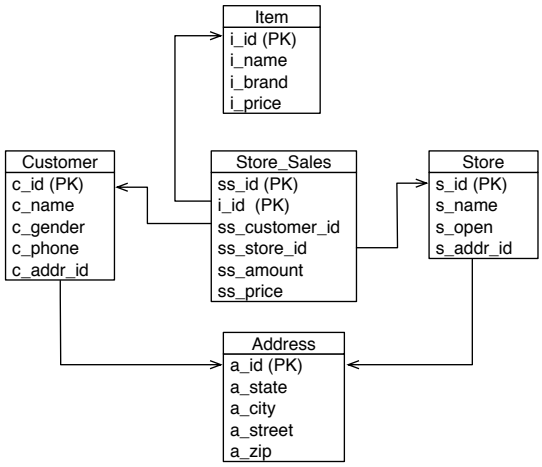


Fig. 1: Example Schema

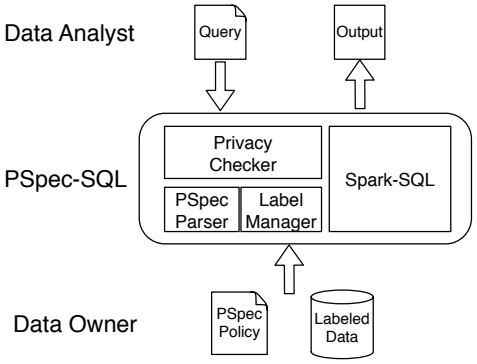


Fig. 2: System Overview

B. PSpec-SQL Overview

The scenario considered by our system mainly involves three logical participants:

- The data owner, e.g., the retail company, who shares the data to third-parties under certain agreements.
- The data analyst, who performs data analysis over the shared data.
- PSpec-SQL, which manages the data and allows the analyst to submit queries to perform data analysis.

Figure 2 shows an architecture overview of PSpec-SQL. PSpec-SQL is built on Spark-SQL, and the privacy related modules mainly consist of the PSpec parser, label manager and privacy checker. The PSpec parser parses and manages the PSpec policy specified by the data owner, and the label manager manages a set of labels which link the abstract data concepts in PSpec with the underlying relational data together. With the PSpec policy and the labeled data set, the privacy checker checks each submitted query during query processing to ensure the privacy-compliances of the queries.

We further elaborate the workflow of our system with the motivating example. To protect customers' privacy, the legal team of the retail company may have enacted several data usage policies according to the privacy policy and related

privacy laws. Examples include "the output of *customer name* is forbidden", "only the aggregated *sale price* is allowed when outputting with *personal information* together" and "the use of customers' living *state*, *city* and *street* together is forbidden". The data owner may also want to adjust the privacy requirements based on the needs of the data analysis tasks. With our PSpec language, these policies can be easily encoded into the enforceable PSpec policies.

Since PSpec only contains abstract data concepts, such as *customer name* and *sale price*, the company needs to label the relational data in Figure 1 with the corresponding data concepts. For example, the *name* column in the *Customer* table is labeled with *customer name*, while the *ss_price* column in the *Store_Sales* table is labeled with *sale price*. Note that data labeling only happens at column level, which will not incur too much human effort.

Finally, with the PSpec policy and the labeled data set, the privacy checker performs information flow analysis on each submitted query to detect possible violations. For example, the following query is stopped since it outputs *state*, *city* and *street* together, which directly violates the third rule above.

```
SELECT state, city, street, avg(ss_price)
FROM Customer JOIN Store_Sales ON c_id = ss_customer_id
              JOIN Address ON c_addr_id = a_id
GROUP BY state, city, street
```

Note that our system requires no extra effort from the data analyst. To perform data analysis, the data analyst only needs to write queries conforming to the data usage policies as usual.

Threat Model. We assume both the data owner and our system are trusted, while the data analyst is not. The only channel for the data analyst to access data is to submit SQL queries to our system. Some of the submitted queries may violate the data owner's privacy policy and cause privacy breaches. The primary goal of our system is to ensure the data owner's policy is properly enforced and the customer's privacy is well protected.

III. Privacy Language

In this section, we present our privacy specification language PSpec. In practice, since the privacy-related data usage restrictions are mainly handled by the legal teams, who may have little background in computer science, PSpec is designed as a high-level language based on abstract concepts. In the remainder of this section, we first present the language syntax, then introduce the formal semantics.

A. PSpec Syntax

Since PSpec is based on abstract concepts, we separate PSpec into the *vocabulary* part and *policy* part as in EPAL [4]. Briefly, the vocabulary part defines abstract concepts while the policy part defines a set of PSpec rules.

1) Vocabulary: In the vocabulary part, the data owner can define a set of user categories and data categories, which can be referred in the PSpec rules. User categories represent different roles for data analysts, while data categories represent privacy-related data concepts, and both of them are organized as hierarchies respectively. In a category hierarchy, the parent category is the generalization of its child categories, while the child category is the specialization of its parent category.

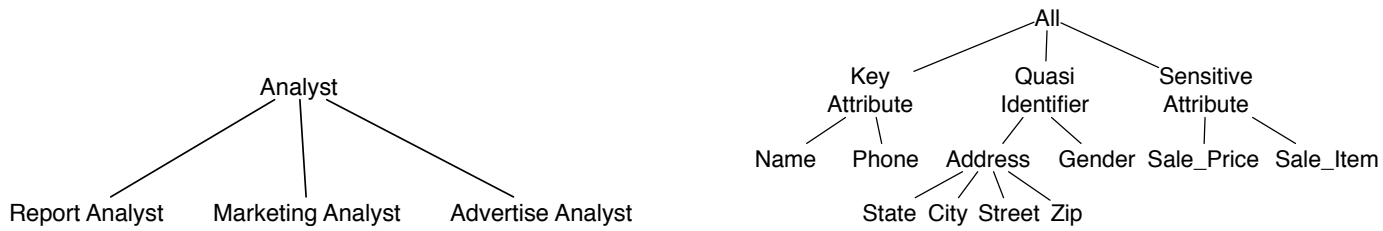
Example hierarchies for the retail company are shown in Figure 3. The top-level user category is *Analyst*, which represents all data analysts. The *Analyst* is divided into *Report Analyst*, *Marketing Analyst* and *Advertise Analyst* based on the tasks assigned to them. The data categories are classified as *Key Attribute*, *Quasi Identifier* and *Sensitive Attribute*. *Key Attribute* denotes the attributes that can uniquely identify an individual, such as *Name* and *Phone* etc. *Quasi Identifier* represents the attributes that may locate individuals by combining them together, and a typical combination is *Birth*, *Zip* and *Gender* [5]. Finally, *Sensitive Attribute* carries sensitive personal information, and contains customers' sales records in the retail company.

For clarity, the data owner should also identify and specify all supported desensitize operations for data categories, and a data category automatically inherits its ancestors' desensitize operations. Informally, a desensitize operation can be used to make some sensitive data category acceptable to access. For example, the desensitize operations for *Sale_Price* can be aggregate operations, including *Min*, *Max* and *Avg* etc., while for *Zip* it can be the *truncate* operation.

2) Policy: The PSpec policy is composed of a set of rules, each of which informally states the restrictions to be satisfied when a *user category* accesses certain *data categories*. We further require a user category should satisfy all the applicable rules. The underling reason is that usually only a part of the business data relate to customers' privacy, thus the restrictive rules allow data owners to only focus on the privacy-related part. Moreover, the semantics also simplify reasoning the effects of a policy since any single rule must be enforced by the entire policy.

The grammar of the PSpec rules is shown in Figure 4, and each rule contains a *scope* part and a *restriction* part (separated by $=>$). The *scope* part is defined by the *User-Ref* and *Data-Assoc* (short for *Data Association*) elements. The *User-Ref* element defines the applicable user categories by referring a user category (<user>). By default, the PSpec rule is also applicable to all descendants of the referred category, and this can be refined by explicitly excluding certain child categories. The *Data-Assoc* element defines the applicable association of data categories by referring a vector of the *Data-Ref* elements. Each *Data-Ref* element refers a data category (<data>) with an *action* element, and the applicable data categories can also be refined by excluding certain descendant categories. The *action* element specifies how the data category is accessed. Currently, we distinguish two types of actions, i.e., *projection*, which means the data categories are outputted by a query, and *condition*, where the data categories are used in the control flow of a query, since they have different privacy implications. The actions also form a hierarchy where *access* is the parent of *projection* and *condition*. For a data association, we require the data categories referred in each *Data-Ref* element must be disjoint, i.e., no referred data category is the parent of another.

In the *restriction* part, a rule can directly *forbid* the query or specify several restrictions. A restriction is a vector of the *Desensitize* elements with the same length of the data association defined in the rule. Each *Desensitize* element



(a) Example User Hierarchy

(b) Example Data Hierarchy

Fig. 3: Example Category Hierarchies

```
Rule       ::= User-Ref, Data-Assoc =>
               ('forbid' | Restriction_1,..., Restriction_m)
User-Ref   ::= <user> ('exclude' <user>+)?
Data-Assoc ::= '[' Data-Ref_1,···,Data-Ref_n ']'
Data-Ref   ::= Action <data> ('exclude' <data>+)?
Action     ::= 'access' | 'projection' | 'condition'
Restriction::= '[' Desensitize_1,···,Desensitize_n ']'
Desensitize::= '{'(<operation>_1,···,<operation>_k)?'}'
```

Fig. 4: PSpec Rule Grammar

specifies a set of desensitize operations (<operation>), and requires the corresponding data category in the data association be desensitized with one of them ($\emptyset$ denotes no desensitization required). Note that the specified desensitize operations must be supported by the corresponding data category as defined in the vocabulary. We require a query satisfy one of these restrictions. Thus, the restriction part is essentially a disjunction of restrictions, while each restriction is a conjunction of restricted data categories, which makes PSpec expressive enough to specify complex policies.

Consider the retail company again. The example rules mentioned in Section II-B can be written as follows:

```
r1: Analyst,[projection Name]=>forbid
r2: Analyst,[projection All exclude Sensitive Attribute,
     projection Sale_Price]=>[{},{avg,max,min,sum}]
r3: Analyst,[access State,access City,access Street]=>forbid
```

For simplicity, we assume all rules are applicable to *Analyst*. The rule r_1 directly forbids the projection of *Name*. The rule r_2 requires when *Sale_Price* is projected together with personal identifiable information (*All* excludes *Sensitive Attribute*), it must be aggregated with avg, max, min or sum. The last rule r_3 forbids the access of *State*, *City* and *Street* together, but one can access any one or two of them freely.

The reader may have already noticed that a data category is considered desensitized as long as one of the specified desensitize operations is performed. We do not support the sequence of desensitize operations since it complicates both the data owner for writing PSpec rules and the data analyst for writing proper queries. For this case, the data owner can simply define a new operation that combines all necessary operations inside. Also note that PSpec does not model *access purpose* directly, since it is difficult for data analytic systems to automatically verify the claimed purpose. Thus, for simplicity, we assume each data analyst is assigned a proper user category by the system administrator based on her tasks.

B. Formal Semantics

Now we present the formal semantics of PSpec. Recall that PSpec specifies restrictions on data usage, thus the semantics are defined against queries. As a convention, we use lower case letters to denote elements, and upper case letters to denote sets.

Since we are interested in how data categories are used by data analysts, a query q is thus modeled as $q = (u_q, D_q)$, where u_q denotes the user category submitting q, and D_q is a set of triples (a, d, op). From information flow point of view [6], [7], (a, d, op) denotes the data category d is leaked through the channel a (projection or condition) after the desensitize operation op.

For clarity, we assume any data category occurring in a query is a *leaf node* in the data hierarchy, i.e., the data should only be labeled with leaf data categories. The reason is that when a data category is restricted, it often implies that its ancestors are also restricted. For example, *Zip* should be *truncated* implies *Address* should also be *truncated* since *Address* contains *Zip*. However, this may lead to semantic inconsistency since the *truncate* operation may not be supported by *Address*. But this requirement will not be a limitation in practice since one can simply label one column with multiple leaf categories when necessary as in Section IV.

Before discussing the semantics of the PSpec rules, we first define some notations. Given a rule r, we use $r.u$ to denote the user category referred by r, and $exclude(r.u)$ to denote the set of excluded user categories. We use $r.d$ and $r.a$ to denote the vectors of data categories and actions referred by r respectively, and $exclude(r.d[i])$ to denote the i-th set of excluded data categories. For a non-forbidden rule r, we use $r.res_i$ to denote the i-th restriction in r, and $r.res_i[j]$ to denote the j-th desensitize operation set in $r.res_i$.

For example, consider the rule r_2 in the previous subsection. Elements in r_2 can be denoted as follows: $r_2.u$ = *Analyst*, $r_2.d$ = [*All*, *Sale_Prices*], $r_2.a$=[projection, projection] and $exclude(r_2.d[1])$={*Sensitive Attribute*}. The only restriction in r_2 is $r_2.res_1$ = [{}, {avg,max,min,sum}].

For ease of discussion, we expand the hierarchical elements referred by the PSpec rules. We use $h^{\downarrow}$ to denote the set of descendants of h, including h itself. Given a rule r, we denote the applicable user categories in r as

$$r.U = r.u^{\downarrow} - \bigcup_{u' \in exclude(r.u)} u'^{\downarrow}$$

Similarly, we denote the vectors of the applicable data categories and actions in r as $r.D$ and $r.A$, where for each i:

$$r.D[i] = r.d[i]^{\downarrow} - \bigcup_{d' \in exclude(r.d[i])} d'^{\downarrow}$$
$$r.A[i] = r.a[i]^{\downarrow}$$

After expansion, the semantics of the PSpec rules can be defined on set operations.

Given a query $q = (u_q, D_q)$, we say a rule r is *applicable* to q (or equivalently, q *triggers* r) iff the following holds:

- $u_q \in r.U$, and
- $\forall i.(r.D[i], r.A[i]) \cap D_q \neq \emptyset$, where $(r.D[i], r.A[i]) \cap D_q = \{(a, d, op) | d \in r.D[i] \wedge a \in r.A[i] \wedge (a, d, op) \in D_q\}$.

Given a vector of desensitize operations v and a restriction res with the same length n, we say v *satisfies* res iff for any $i \in [1, n]$, either $res[i] = \emptyset$ or $v[i] \in res[i]$.

With the definitions above, we say a query $q = (u_q, D_q)$ *satisfies* a rule r if either r is not applicable to q, or the following holds:

- r is not forbidden, and
- for any desensitize operation vector $v \in V$, there exists a restriction res in r such that v satisfies res, where $V = \{v | \forall i.(v[i] \in \{op | \exists (a, d, op) \in (r.D[i], r.A[i]) \cap D_q\})\}$.

Intuitively, a query satisfies an applicable rule only if all data associations accessed by the query are properly desensitized. Consider a query q submitted by *Report Analyst* and projecting *State*, *City* and *Sale_Price* directly without any desensitization, which can be written as:

$$q = (ReportAnalyst, \{(projection, State, null), (projection, City, null), (projection, Sale\_Price, null)\}).$$

For the example rules in the previous subsection, obviously q only triggers r_2. Since q accesses two data associations specified in r_2, which are *State* with *Sale_Price* and *City* with *Sale_Price*, q should desensitize both of them as required by r_2. However, the only desensitize operation vector $v \in V$ for both data associations is $[null, null]$, which does not satisfy $r_2.res_1$ since it requires *Sale_Price* should be aggregated. Thus, q satisfies r_1 and r_3 while not r_2.

Finally, we say a query q *satisfies* a PSpec policy p if q satisfies all rules in p.

IV. Data Labeling

After introducing the privacy specification language PSpec, we shall then discuss how to enforce the PSpec policy automatically. Since the PSpec rules are defined upon abstract data categories, the data owner should first label her data with proper data categories. Currently, we support data labeling on column level so that the submitted queries can be statically checked. Thus, the data owner should label table columns with proper data categories based on column semantics, and labeling one column with multiple data categories is also supported if the column corresponds to multiple data categories. However, recall from Section III-B, columns can only be labeled with leaf data categories.

Besides labeling columns, the data owner should also provide implementations of the desensitize operations specified in the vocabulary as user-defined functions (UDFs) and register them into our system. The data owner may optionally specify multiple implementations for a same desensitize operation for different columns. For example, consider the *truncate* operation for the *IP* category. Suppose *IP* is stored as dot separated strings in the column *IP1*, while integer in the column *IP2*. Thus, the data owner can specify different UDFs *truncate1* and *truncate2* for the same *truncate* operation to keep the implementation simple.

As we can see now, data labeling is a straightforward process except two issues, which will be discussed below.

A. Conditional Labeling

One problem with data labeling is that the proper data categories for some columns cannot be determined statically, but depend on what table the current table is joined with. An example is the tables in Figure 1. The data stored in the *Address* table could be either the customer's address or store's address since the *Customer* table and the *Store* table both have a foreign key referring to *Address*.

The phenomenon is common for the fact and dimension tables in data warehouse systems. To handle this, we support conditional labeling, i.e., a column is labeled with certain data category based on the *join condition*. A join condition specifies the table is joined with what table on what column(s), and currently only equi-join is supported. For the previous example, the data owner can specify the *zip* column is labeled with the *Zip* category only if the table is joined with the *Customer* table on *a_id=c_addr_id*. The actual data categories for these conditionally labeled columns are determined during runtime, which will be discussed in the next section.

B. Complex Data Types

In the previous discussion, we implicitly assume the column is the basic unit for data labeling. However, to handle semi-structured data, big data systems such as Spark-SQL also support complex data types including array, struct and map, which can be nested arbitrarily.

When labeling columns with complex data types, we treat each sub type separately to be more accurate. The basic idea is as follows:

- array: array items can be labeled separately based on array indices.
- struct: for struct, each field is labeled separately since fields usually have different semantics.
- map: for map, currently we support labeling values for some predefined keys.

Note that labeling on these columns should also be recursive since the types can be arbitrarily nested.

Interestingly, even some columns with primitive types may have composite semantics. For example, the *birth* column

with the *Date* type may contain all information of *birth_year*, *birth_month* and *birth_day*, and they might be extracted with the UDFs *getYear*, *getMonth* and *getDay* respectively. To enable fine-grained control on these columns, we treat them similarly as complex types. Specifically, the data owner can label each *extract operation* (UDFs for extracting information) on these columns separately. For example, the extract operation *getYear* on the *birth* column can be labeled as *birth_year*, while *getMonth* on the *birth* column as *birth_month*.

At last, we discuss a bit more on the difference between desensitize operations and extract operations. The incentives of both operations are quite similar, i.e., to obtain partial information from the input. However, the key difference is that desensitization is irreversible, while it is not the case for extraction. For example, one can easily restore full birth information by calling *getYear*, *getMonth* and *getDay* on the *birth* column in a same query. Thus, these operations should be treated as extract operations rather than desensitize operations.

V. Privacy Checking

With the PSpec policy and the labeled data set, we now discuss the privacy checking algorithm for SQL queries. The basic idea is to track how each data category flows (through what desensitize operation) in a query. To achieve this, we first construct attribute lineage trees to represent the attribute flow, which is then used to extract to the data category flow. Finally, the data category flow is checked against the PSpec policy according to the semantics in Section III-B. Here we assume readers are familiar with query processing in relational database systems, and the reference material can be found in [8]. For ease of discussion, we assume the submitted queries have been parsed and optimized as optimized query plans by the query processor.

A. Lineage Tree Construction

In the following, we use the term *stored attributes* to refer to the attributes that originate from tables. Since data categories and desensitize operations are essentially stored attributes and UDFs respectively, we first construct *attribute lineage trees* to track how each attribute flows in a query. The leaf nodes of a lineage tree must be stored attributes, while the non-leaf nodes are transformations applied to the children.

Consider the following query that tries to find out average sale price for female customers in each state and city:

```
SELECT concat(a_state, a_city), AVG(ss_price)
FROM Store_Sales JOIN Customer ON ss_customer_id = c_id
                 JOIN Address  ON c_addr_id = a_id
WHERE c_gender = 'F'
GROUP BY a_state, a_city
```

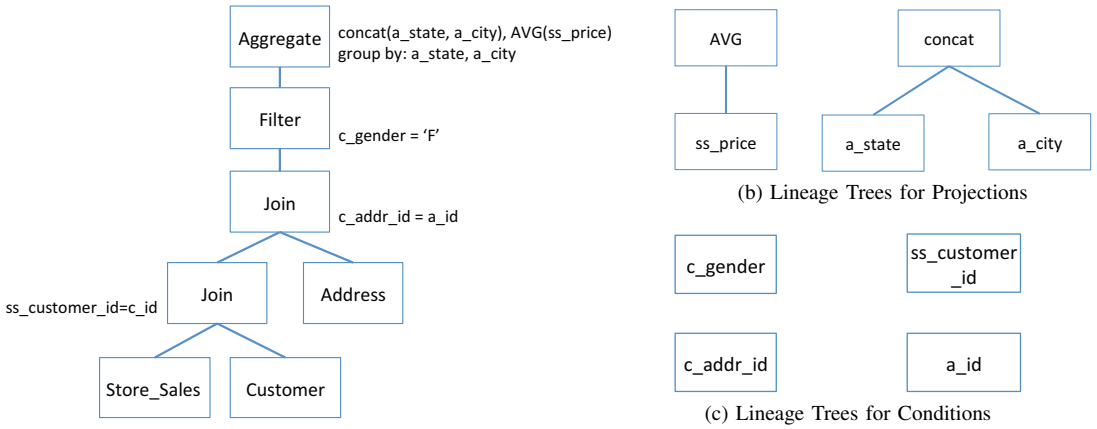
The logical plan is shown in Figure 5a, and the lineage trees for attributes used in projection and condition are shown in Figure 5b and Figure 5c respectively. Since stored attributes are directly used in conditions, the lineage trees in Figure 5c only contain leaf nodes. With lineage trees, we can clearly see how stored attributes are used in the query. For example, *ss_price* is projected after the function AVG.

Now we discuss how to construct attribute lineage trees from a query plan. Recall that our system only allows the data analyst to query the data for data analysis rather than updating or deleting records, thus we only consider the select queries here. The basic algorithm framework is shown in Figure 6. The function PROPAGATE takes as input a query plan, and returns a pair of sets which contain the lineage trees for attributes used in projection and condition respectively.

Before the algorithm starts, we attach each plan operator with a map $projections$ that maps each attribute outputted by the plan operator to a lineage tree, and use a global set $conditions$ to track the lineage trees for all attributes used in conditions. Then PROPAGATE performs post-order traversal over the query plan, and for each operator it calls TRANSFORM and COLLECT to update $plan.projections$ and $conditions$ respectively. Briefly, TRANSFORM calculates the map $projections$ based on the attribute transformations in current operator, and COLLECT returns the lineage trees for the attributes used in the predicates in current operator.

The remaining problem is to define TRANSFORM and COLLECT for each plan operator. By default, TRANSFORM simply returns the map $projections$ of the child operator if the operator does not perform any attribute transformation, while COLLECT simply returns $\emptyset$ for operators that do not contain any predicate. Example operators include *Limit*, *Distinct* etc. While TRANSFORM and COLLECT for the following operators will be discussed briefly:

- *Table*: TRANSFORM returns a map that maps each stored attribute to a single node lineage tree whose root is the stored attribute. COLLECT works as default.
- *Project*: TRANSFORM returns a map that maps each attribute defined by the projection list to a lineage tree with the corresponding transformations added onto the lineage trees for the attributes defining the new attribute. COLLECT returns the lineage trees for the attributes used in the predicates in the *CaseWhen* and *If* statements (if any). When resolving predicates, COLLECT should also add the corresponding transformations on top of the lineage trees.
- *Filter*: TRANSFORM works as default. COLLECT returns the lineage trees by resolving the predicates in the filter condition.
- *Join*: TRANSFORM returns the union of the map $projections$ of the left and right operators. COLLECT returns the lineage trees by resolving the predicates in the join condition.
- *Aggregate*: is similar to *Project*, and aggregate functions including COUNT, SUM, AVG, MIN and MAX are simply treated as transformations.
- *Sort*: TRANSFORM works as default. COLLECT returns lineage trees for the attributes in the sort expression.
- Binary operators: for the binary operators *Union/Intersect/Except*, the output attributes actually combine the corresponding attributes in the left and right operators. Thus, TRANSFORM returns a map that maps each attribute to a lineage tree that combines the lineage trees for the corresponding attributes in the left and right child operators with a dummy transformation. COLLECT works as default.



(a) Example Logical Plan

(b) Lineage Trees for Projections

(c) Lineage Trees for Conditions

Fig. 5: Example Query

1: $conditions \leftarrow \emptyset$
2: **function** PROPAGATE($plan$)
3: **for** $child$ in $plan.children$ **do**
4: PROPAGATE($child$)
5: $plan.projections \leftarrow$ TRANSFORM($plan$)
6: $conditions \leftarrow conditions \cup$ COLLECT($plan$)
7: **return** $(plan.projections.values, conditions)$

Fig. 6: Algorithm for Computing Lineage Trees

Note that for the *CaseWhen* and *If* statements in the *Project* and *Aggregate* operators, the lineage trees for the attributes used in the predicates should be added into $conditions$ only if the defined attributes are projected or used in the subsequent operators. However, to simplify the discussion, we assume the logical plan has been optimized and unused attributes have been removed. Otherwise, one needs to track $conditions$ for each attribute separately as for $projections$.

For example, consider the logical plan in Figure 5a. For the table operators *Store_Sales*, *Customer* and *Address*, $conditions$ is always $\emptyset$, while $projections$ of each operator is initialized as discussed above. For *Join* on *Store_Sales* and *Customer*, $projections$ is the union of $projections$ of *Store_Sales* and *Customer*, while the lineage trees for *ss_customer_id* and *c_id* are added into $conditions$. Another *Join* is processed similarly. Then for *Filter*, $projections$ is inherited from its child, while the lineage tree for *c_gender* is added into $conditions$. Finally for *Aggregate*, the aggregate expression defines two attributes, which are *concat* on *a_state* and *a_city*, and *AVG* on *ss_price*. Thus, the map $projections$ maps the defined attributes to lineage trees in Figure 5b, and the set $conditions$ contains lineage trees in Figure 5c.

B. Flow Extraction

Lineage trees are only the intermediates structures representing the attribute flow, we should then extract the data category flow for privacy checking. To achieve this, we need to determine the data category for each stored attribute and map transformations to the corresponding desensitize operations. It is straightforward to determine the data categories for the statically labeled attributes, while the difficulty here is to determine the data categories for the attributes with conditional labels and complex data types as discussed in Section IV.

Resolve Conditional Labels. To resolve the data categories for conditionally labeled attributes, we need to determine which join condition is satisfied by the query. However, we cannot simply check the predicates in the *Join* operator, since the query may filter join results in the *Filter* operator or even the *CaseWhen* and *If* statements. To handle this, we track all equality predicates in the query and build an undirected equality graph with nodes being stored attributes or constants and edges indicating two nodes are equal. Then a join condition is satisfied by a query if the join columns are reachable from one to another in the equality graph.

When tracking the equality predicates, two issues need special care here. First, since a table may be referenced multiple times in a query, we should differentiate the stored attributes from multiple references of a same table. Second, when resolving the equality predicates, some approximation is necessary to guarantee safety. The principle is that for any equality predicate, we consider the stored attributes or constants appearing in the left and right part equal pair-wisely. For example, consider $a+b=c$, we need to safely assume both $a=c$ and $b=c$ hold since otherwise the analyst may let b evaluate to 0 at runtime to bypass the mechanism. However, the approximation rarely causes false positives for normal queries.

We further elaborate the process with an example. Consider the following query that finds addresses for both male customers and stores:

```
(SELECT a_state, a_city, a_street
 FROM Address JOIN Customer ON a_id = c_addr_id
 WHERE c_gender = 'M')
INTERSECT
(SELECT a_state, a_city, a_street
 FROM Address JOIN Store
 WHERE a_id = s_addr_id)
```

Suppose the columns in the *Address* table are labeled with the *State*, *City* and *Street* categories only when joined with the *Customer* table on *a_id* = *c_addr_id*. We assume stored attributes are equipped with unique ids based on table references. Thus, the stored attributes from the *Address* table

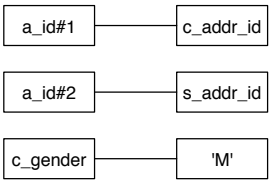


Fig. 7: Example Equality Graph

are differentiated by {*a_id*#1, *a_state*#1, *a_city*#1, *a_street*#1} and {*a_id*#2, *a_state*#2, *a_city*#2, *a_street*#2} (#N indicates attribute id). After analyzing the equality predicates in the query, the equality graph is shown in Figure 7. Finally after performing reachability analysis on the equality graph, we realize that the join condition for *a_state*#1, *a_city*#1 and *a_street*#1 is satisfied since *a_id#1 = c_addr_id*, while for *a_state*#2, *a_city*#2 and *a_street*#2 is not.

Resolve Complex Data Types. The basic principle for resolving the data categories for the attributes with complex data types is that if a type is directly accessed with no subtype selectors, then the data categories for all its subtypes are considered accessed. For example, consider an attribute A of struct type with fields a, b and c. For $A.a$, only the data category for the field a is accessed. However, if A is directly accessed, then the data categories for a, b and c are all considered accessed. The attributes with composite semantics are treated similarly, i.e., we identify the extract operations performed on these attributes and the effective data categories are determined based on the extract operations. And to guarantee safety, we consider an extract operation is effective only if it is the direct parent of the stored attribute in the lineage tree.

Resolve Desensitize Operations. After the data categories are determined for each stored attribute, we should further identify the desensitize operation performed for each data category. To do so, we first extract the transformation path for each data category by traversing lineage trees from leaf to root, then identify which desensitize operation exists in the path. Only the first desensitize operation is effective since we assume a data category can only be desensitized once. For example, consider the lineage trees in Figure 5. Since *AVG* is a desensitize operation for *Sale_Price*, we know that *Sale_Price* is projected after the *AVG* operation. However, *State* and *City* are directly projected since *concat* is not a desensitize operation for them.

C. Policy Checking

Recall from Section III-B that a query q is modeled as a pair (u_q, D_q), where u_q is the user category that submits q and D_q is a set of triples (a, d, op). In our system, u_q is obtained by assigning each data analyst with a proper user category by the system administrator. And after flow extraction, we have already known how each data category d flows to the channel a through what desensitize operation op, which exactly constitutes the set D_q.

With (u_q, D_q), we can check whether the query satisfies the policy according to the semantics discussed in Section III-B. If any violation is detected, the query is stopped and the violated rule is returned for the data analyst to revise the query. Otherwise, the query is proceeded to further executions.

One advantage of the privacy checking is that it is totally based on information flow analysis on the query plan without touching the underlying data being queried. Thus, it is suitable for big data systems as it only incurs negligible overhead over query execution. However, one possible limitation is that currently we check each query separately, since in practice most queries are independent and it is usually difficult to link results from multiple queries together. This issue will be further discussed in Section VIII.

VI. Integrate with Differential Privacy

In our system, we assume the desensitize operation is irreversible and any properly desensitized information is safe to access. Some common desensitize operations are generalization and suppression for categorical values, or aggregation for numerical values. However, direct aggregation may not be safe since individuals' information might be reversed with some background knowledge or by differencing attacks. In this section, we discuss how to extend our system with differential privacy to implement safe aggregation-based desensitize operations. Due to space limitation, we assume some basic familiarities with different privacy (reference materials can be found in [2]), and mainly discuss the specific problems pertaining to our system.

Briefly, differential privacy requires the aggregations on two datasets differing on only one record should be approximately the same, and thus ensures the participation of any individual cannot be observed from the results. The typical work-flow of a differentially-private data analytics system is as follows. Each analyst is assigned a privacy budget ϵ. When submitting a query q, the analyst also specifies the desired accuracy ϵ_q. Then, the system checks whether there is enough budget left for answering q. If so, the system subtracts ϵ_q from ϵ and returns the perturbed query result. Otherwise, the system denies the query q. Implementing differential privacy with SQL queries is relatively straightforward, but for our system, the following problems need to be addressed.

Additional Inputs from Data Owners. Since differential privacy perturbs a query based on the underling dataset, we thus require some additional inputs from data owners regarding certain characteristics of the dataset. These inputs include the *ranges* of the aggregated columns, and the *multiplicity* of the columns used in join conditions, which refers to the maximum number of records associated with the same column value in a table. For example, the multiplicity of a primary key must be one. Note that these values may not necessarily be computed from the dataset, but can be estimated with certain domain knowledge. We also do not require the data owner provide these values for all columns. Instead, for each aggregate operation in a query, we first attempt to compute the required noise based on the available information, and the operation is considered a desensitize operation only when this process succeeds.

Integration with PSpec. Since our ultimate goal is to integrate differential privacy with PSpec, we may not need to perturb all aggregate operations in a query as long as the query

satisfies the PSpec policy. For example, consider the following rule r and query q.

```
r: Analyst,[access Zip,projection Sale_Price]
                              =>[{truncate},{}],[{},{avg,sum}]
q: {(projection,Zip,truncate),(projection,Sale_Price,avg)}
```

Intuitively, the rule r requires when a query accesses *Zip* and projects $Sale\_Price$ together, either *Zip* should be truncated or *Sale_Price* should be aggregated. While for the query q, since both *Zip* is truncated and *Sale_Price* is aggregated with *avg*, we can let q satisfy the first restriction in r such that *avg* is considered as a normal aggregate operation and no privacy budget is consumed.

Thus, given a PSpec policy and a query, it is ideal to check whether the query satisfies the policy with minimal budget consumption. However, the complexity for the problem is exponential w.r.t. both the number of rules and data associations. To handle this, we employ a greedy search strategy to compute an approximate solution. Briefly, for each rule and data association accessed by the query, we first compute the budget consumption for each satisfied restriction based on the previous choices, then choose the one with minimal consumption. Since no backtracking is performed, the greedy search incurs no extra complexity for privacy checking.

VII. Implementation and Evaluation

We have implemented PSpec-SQL on top of Spark-SQL 1.3.0 with support for Hive tables [2]. The PSpec parser and label manager are both implemented as standalone modules, while the privacy checker is implemented inside the query processing component of Spark-SQL, i.e., the spark-catalyst module. For PSpec, we also developed a graphical authoring tool to facilitate data owners write the PSpec policy. For differential privacy, we use Laplace mechanism [2] to perturb the query result. The source code of our prototype implementation is available on github [3]. Note that the privacy checker can also be implemented as a separate proxy over existing database systems, and the main change is to construct the lineage trees from SQL parse trees instead of query plans.

To show the practicability of our system, we carried out a case study on TPC-DS benchmark [3]. We further qualitatively compared PSpec-SQL with other systems.

A. Case Study on TPC-DS Benchmark

TPC-DS benchmark is a general benchmark for decision support systems, and models a national-wide retail company. It includes a database schema (7 fact tables and 17 dimension tables) and 99 queries. In the case study, we mainly focus on the privacy aspect of the TPC-DS benchmark and possible violations in the TPC-DS queries.

1) PSpec Vocabulary and Policy: Some TPC-DS tables may contain customers' private information, such as *Customer*, *Customer_demographics* etc. To ensure these privacy-related data are properly used, we have made a sample PSpec policy and the process is illustrated as follows.

[2]http://hive.apache.org/

[3]https://github.com/thufv/privacy

TABLE I: Supported Desensitize Operations

Data Category	Supported Desensitize Operations
All	count
Zip	truncate
Income	range
Vehicle	isZero
Price	sum, avg, min, max

TABLE II: PSpec rules for TPC-DS

ID	Rule
1	Analyst,[access KA]=>forbid
2	Analyst,[projection Street]=>forbid
3	Analyst,[projection Name]=>forbid
4	Analyst,[projection Birth,projection Address, projection Gender]=>forbid
5	Analyst,[projection Birth,projection Address, projection Marital]=>forbid
6	Analyst,[projection Birth,projection Address, projection Education]=>forbid
7	Analyst,[access F_Name,access L_Name]=>forbid
8	Analyst,[access S_Num,access S_Name, access Suite]=>forbid
9	Analyst,[access B_Day,access B_Month, access B_Year]=>forbid
10	Analyst,[access Zip]=>[{truncate}]
11	Analyst,[access QI,access Vehicle]=>[{},{isZero}]
12	Analyst,[access QI,access Income]=>[{},{range}]
13	Analyst,[access QI,access Price] =>[{},{sum,count,avg,min,max}]

Vocabulary. For simplicity, we assume only one user category named *Analyst*, who is responsible for analyzing the sales trend. The data hierarchy is shown in Figure 8. Most data categories are self-explanatory, and we use some abbreviations when the meanings are clear. Similar to previous work [5], [9], the privacy-related information is classified as *KA* (Key Attribute), *QI* (Quasi Identifier) and *SA* (Sensitive Attribute), which are further divided into finer-grained categories corresponding to columns in the TPC-DS tables.

The supported desensitize operations for some data categories are shown in Table I. Note that we have not identified too many desensitize operations since the schema of TPC-DS is well designed and most columns have already corresponded to fine-grained meanings. Thus, the PSpec rules can directly refer these fine-grained categories without desensitize operations. However, desensitize operations will be more useful when columns have coarse-grained semantics.

Policy. We then write some PSpec rules to ensure the privacy related data is properly used while keeping the data still usable for data analysis. The list of rules is shown in Table II, which specifies several data usage restrictions based on the task assigned for *Analyst*.

The general goal of the rules is to prevent *Analyst* accessing and projecting private information for individuals, which may cause privacy breach and is not required by her tasks. The first rule forbids the access of *KA* since *KA* can directly locate individuals. The rules 2 to 6 forbid the projection of certain data categories, since these data categories have the possibility to identify individuals when linking with some public datasets. And in the rules 4 to 6 we have identified several possible quasi-identifiers as in [5]. The rules 7 to 9 further forbid the access of certain combinations of data categories since

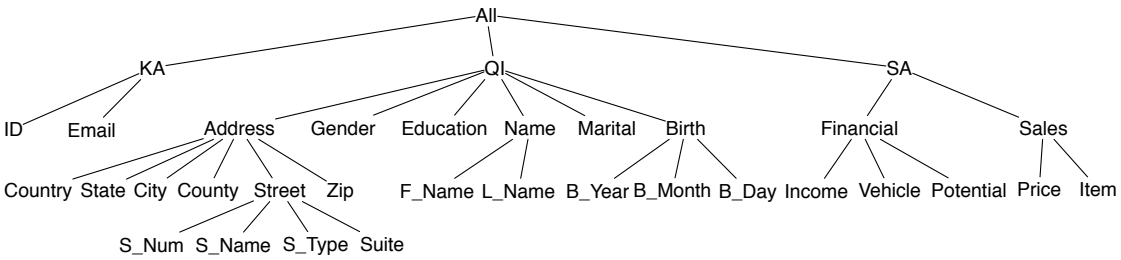


Fig. 8: Data Hierarchy for TPC-DS

these combinations have a great chance to identify individuals. Finally, the rules 10 to 13 require some data categories should be desensitized, for example, only the aggregated sales price or the truncated zip code can be used.

2) Data Labeling: With the PSpec policy, we then label the TPC-DS tables with proper data categories. We have also implemented desensitize operations including *truncate*, *range* and *isZero* as UDFs and registered them into our system.

We first identify all privacy-related tables. Tables including *Customer*, *Customer_Address*, *Customer_Demographics*, *Household_Demographics* and *Income_Band* directly relate to customers' private information. The *Item* table and the sale/return tables correspond to the *Sales* category.

As the TPC-DS tables only contain primitive types, most columns can be labeled directly with the corresponding data categories. An exception is the *Item* table, and its columns should be labeled with the *Price* and *Item* categories only when it is joined with the sale/return tables, since we only care about items that customers have bought or returned. As we can see, although the complete schema may contain many tables, the number of privacy-related tables is relatively small and data labeling phase will not incur too much effort.

3) Query Checking: TPC-DS contains 99 queries, each of which corresponds to a particular business question. The queries range from reporting queries, ad hoc queries, iterative queries to data mining queries. As we have counted, the simplest query contains 18 lines of code (formatted) and refers only 1 table, while the most complex query contains 456 lines of code and refers 14 tables. Since some TPC-DS queries contain several syntax which is not supported by Spark-SQL, including window function, WITH statement and sub query in WHERE statement, we transform these queries into equivalences while can be processed by Spark-SQL (window function is an exception, which is simply removed). The transformed results can be found in our github repository.

With the PSpec policy and labeled data set described before, we submitted the transformed queries to our system to detect possible violations. First, we measured the time cost by privacy checking for each query. All privacy checking finishes within milliseconds, the shortest takes only *1 ms* and the longest takes *96 ms* with the average time being *7 ms*. Thus, as mentioned, our system only incurs negligible overhead and is suitable for online data analysis.

Second, we collected and inspected the privacy violations, *21 queries* in total, reported by our system. Due to the space limitation, we will not list the details of all violated queries, while some typical examples will be discussed here. Some typical violations, including the queries 23, 24, 30, 34, 38, 46, 68, 73, 74, 81 and 84, project customer's first name and last name, which directly violates the rules 3 and 7. Even worse, the queries 30 and 81 even project customer's birth day, email or living street with customer's name together, which causes severe privacy violations. Other violations, including the queries 13, 15, 45, 48, 64 and 85, directly access *Price*, *Income* or *Zip* without desensitization as required.

Our system also reports 4 false positives, which all relate to desensitize operations. The queries 8 and 19 desensitize *Zip* with *substr* rather than the specified *truncate* operation, which is not recognized by our system. Similarly, the queries 34 and 74 directly test whether *Vehicle* is greater than zero in condition, which is equivalent to the *isZero* operation. However, the false positives are also reasonable, since *truncate* and *isZero* are not built-in UDFs and these false positives can be easily eliminated by simply revising the queries.

B. Qualitative Analysis

In this section, we qualitatively compare PSpec-SQL with the existing systems, i.e., PINQ [10], Airavat [11] and Sen et al. [1] from the following aspects.

Target Scenario. Both PINQ and Airavat consider a general data analysis scenario where sensitive data are analyzed by data analysts in a privacy preserving manner, and the primary goal is to prevent malicious data analysts. Sen et al. targets at auditing privacy compliances of data usages within organizations, and thus the major challenge is to automatically infer data flow from big data systems. While in PSpec-SQL, the data owner shares data with third-parties with certain agreements and usage restrictions, and our primary goal is to enforce these restrictions automatically against the queries.

Data Model. PINQ provides a declarative query language based on LINQ with certain restrictions, e.g., no output of raw records and a restricted form of `Join` operator etc. Airavat supports the MapReduce programming model, but only a set of trusted reducers are available for analysts. Sen et al. do not target at any specific data model, but rather provide an ad-hoc solution for checking data usage flow from MapReduce-like big data systems. In contrast, PSpec-SQL provides full support for the relational data model and SQL language, which have been widely used in practice, and requires no extra effort from data analysts.

Specification Language. Both PINQ and Airavat contain no specification language but only the parameters related to

differential privacy, e.g., the privacy budget. Sen et al. provides LEGALEASE, a privacy language for encoding privacy policies relating to data usages. But LEGALEASE is designed for handling general data usage privacy policies, i.e., store, use and share, and is not totally suitable for online data analytics, e.g., no flexible support for specifying desensitizations. While PSpec-SQL offers PSpec, a high-level privacy language designed for data analytics with explicit support for data association and desensitization.

Privacy Guarantee. The privacy guarantees of PINQ and Airavat are offered by differential privacy. However, they still suffer from some sophisticated side-channel attacks [12], e.g., state attacks and timing attacks. Sen et al. provide weak guarantee, and malicious developers can bypass the mechanism easily by misleading the label inference process. In contrast, the privacy guarantee of PSpec-SQL relies on the PSpec policy specified by the data owner, and this issue will be further discussed in the next section.

VIII. DISCUSSION

In this section we discuss some other issues related to our system.

PSpec Elements. *Purpose*, *condition* and *obligation* are important concepts in privacy policies [13]. In PSpec, data desensitization can be viewed as a special *condition* that must be satisfied by the queries. For ease of discussion and policy enforcement, we have intended to leave out *purpose* and *obligation* elements. However, it is straightforward to extend the syntax and semantics of PSpec with these elements.

Privacy Guarantee. As mentioned before, the privacy guarantee of our system depends on the PSpec policy specified by the data owner, since any query satisfying the policy is considered safe to execute. For example, the following rule requires only aggregations can be accessed, which provides as strong guarantees as PINQ [10] does:

```
Analyst, [projection All]=>[{min,max,avg,sum,count}]
```

Furthermore, our system provides more flexibility for data owners to adjust the privacy requirements for different scenarios and tasks, e.g., the privacy requirement for sales records is usually weaker than that for medical records.

Possible Limitations. Currently, our system may suffer from two possible limitations. First, each submitted query is checked independently, which may allow data analysts bypass certain rules by linking multiple data categories in separate queries together. However, it is difficult to do so in practice since one must ensure the results returned by multiple queries have similar order and size. As a remedy solution, one might consider checking the union of multiple queries submitted by the analyst as a whole. But this strategy is too restrictive since most queries are usually irrelevant and the analyst could not revise the queries submitted before. Thus, as a future work, we plan to develop certain heuristics or patterns to identify correlated queries to better handle this problem.

Another possible limitation comes from the fact that the privacy protection of our system directly relies on the PSpec policy specified by the data owner. Thus, it may require certain expertise for data owners to write proper policies to prevent against malicious analysts. While in our scenario, this problem can be alleviated by the fact that the data are only shared with certain certified third-parties, not the general public, and the rare malicious analysts could be held accountable by the legal agreements between the organizations. Moreover, we also plan to develop some templates or guidelines to guide data owners to write policies.

IX. RELATED WORKS

In this section, we discuss several related works, which fall into the following categories.

Information Flow Analysis. Information flow analysis on programs is originated from security community to check whether programs leak sensitive information [6], which informally requires the sensitive information in a program should not affect the output unless declassified [7]. The technique has been widely used in security community, ranging from certificating secure programs [6], preventing security attacks [14] and detecting privacy leakages in mobile applications [15]. Compared with these works, we use informational flow analysis for a different purpose, i.e., checking privacy compliances of SQL queries.

Privacy Language. Several privacy languages have been proposed to formalize text-based policies, including P3P [16], EPAL [4], XACML [17] and P-RBAC models [18]. Some of these works have motivated the design of PSpec, but the main difference is that these works serve for general purposes, while PSpec is designed as a specific language suitable for big data analytics systems with explicit support for data association and desensitization.

Differential Privacy. After differential privacy [2] was first proposed, numerous algorithms and techniques have been proposed to improve its performance and utility [19], [20], [21], [22]. However, our work differs from these in that we employ a policy-based approach and treat differential privacy as a means to implement desensitize operations, rather than improve differential privacy itself, as these works do.

Privacy Preserving Systems. Many privacy preserving systems have been developed in order to protect individuals' privacy. PINQ [10] is a privacy-preserving data analysis platform built on top of LINQ and differential privacy. Airavat [11] is an extension of MapReduce [23] and integrates mandatory access control and differential privacy to provide security and privacy guarantees. GUPT [24] enforces differential privacy for arbitrary computations with the sample and aggregate framework [20]. Sen et al. [1] presented a system for auditing privacy compliances in big data systems within organizations. The major differences between our system and these systems have been discussed in Section VII-B.

Database Privacy. The last line of the related works is the database privacy, since our approach naturally applies to database systems. The sequence of works by Agrawal [25], [26], [27] proposed the idea of Hippocratic database. Query auditing techniques [28], [29] audit the submitted queries to detect whether sensitive information is leaked. The works [30], [31] enforce purpose-based access control within database systems. Colombo and Ferrari [32] further consider action-aware policies, i.e., actions performed by queries on certain

categories of data. DataLawyaer [33] enforces general data use policies encoded into SQL-like queries. However, we differ from these works in that we provide a high-level specification language to hide the details of the underling data model, while these works directly operate on the relational data model, which may cause barriers for non-expert legal teams.

X. Conclusion

In this paper, we present PSpec-SQL, a privacy-integrated data analytics system automatically enforcing privacy compliances against SQL queries. Our system provides a high-level specification language PSpec for data owners to specify their data usage restrictions, and ensures only the privacy-compliant queries are executed at runtime. PSpec-SQL could also be easily integrated with the existing techniques, e.g., differential privacy, as desensitize operations.

In the future, we plan to extend our system in the following ways. First, we plan to extend our policy checking algorithm to handle history queries submitted by the same analyst. Second, we intend to develop certain policy analysis algorithms or guidelines to facilitate data owners writing policies. Finally, we also plan to perform more real-world case studies to further evaluate our system.

References

[1] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing, "Bootstrapping privacy compliance in big data systems," in *Proceedings of the 35th IEEE Symposium on Security & Privacy (Oakland)*, 2014.

[2] C. Dwork, "Differential privacy," in *Automata, languages and programming*. Springer, 2006, pp. 1–12.

[3] R. O. Nambiar and M. Poess, "The making of tpc-ds," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 1049–1058.

[4] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, "Enterprise privacy authorization language (epal 1.2)," *Submission to W3C*, 2003.

[5] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, 2002.

[6] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.

[7] A. Sabelfeld and D. Sands, "Dimensions and principles of declassification," in *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*. IEEE, 2005, pp. 255–269.

[8] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database system concepts*. McGraw-Hill New York, 1997, vol. 4.

[9] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 3, 2007.

[10] F. D. McSherry, "Privacy integrated queries: an extensible platform for privacy-preserving data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 19–30.

[11] I. Roy, S. T. Setty, A. Kilzer, V. Shmatikov, and E. Witchel, "Airavat: Security and privacy for mapreduce." in *NSDI*, vol. 10, 2010, pp. 297–312.

[12] A. Haeberlen, B. C. Pierce, and A. Narayan, "Differential privacy under fire." in *USENIX Security Symposium*, 2011.

[13] S. Fischer-Hübner, *IT-security and privacy: design and use of privacy-enhancing security mechanisms*. Springer-Verlag, 2001.

[14] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.

[15] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information flow tracking system for real-time privacy monitoring on smartphones," *Communications of the ACM*, vol. 57, no. 3, pp. 99–106, 2014.

[16] L. Cranor, M. Langheinrich, M. Marchiori, M. Presler-Marshall, and J. Reagle, "The platform for privacy preferences 1.0 (p3p1. 0) specification," *W3C recommendation*, vol. 16, 2002.

[17] T. Moses *et al.*, "Extensible access control markup language (xacml) version 2.0," *Oasis Standard*, vol. 200502, 2005.

[18] Q. Ni, E. Bertino, J. Lobo, C. Brodie, C.-M. Karat, J. Karat, and A. Trombeta, "Privacy-aware role-based access control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 13, no. 3, p. 24, 2010.

[19] C. Dwork, "Differential privacy: A survey of results," in *Theory and Applications of Models of Computation*. Springer, 2008, pp. 1–19.

[20] K. Nissim, S. Raskhodnikova, and A. Smith, "Smooth sensitivity and sampling in private data analysis," in *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. ACM, 2007, pp. 75–84.

[21] C. Li, M. Hay, V. Rastogi, G. Miklau, and A. McGregor, "Optimizing linear counting queries under differential privacy," in *Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2010, pp. 123–134.

[22] R. Chen, N. Mohammed, B. C. Fung, B. C. Desai, and L. Xiong, "Publishing set-valued data via differential privacy," *Proceedings of the VLDB Endowment*, vol. 4, no. 11, pp. 1087–1098, 2011.

[23] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[24] P. Mohan, A. Thakurta, E. Shi, D. Song, and D. Culler, "Gupt: privacy preserving data analysis made easy," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 349–360.

[25] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Hippocratic databases," in *Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 143–154.

[26] K. LeFevre, R. Agrawal, V. Ercegovac, R. Ramakrishnan, Y. Xu, and D. DeWitt, "Limiting disclosure in hippocratic databases," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. VLDB Endowment, 2004, pp. 108–119.

[27] R. Agrawal, P. Bird, T. Grandison, J. Kiernan, S. Logan, and W. Rjaibi, "Extending relational database systems to automatically enforce privacy policies," in *ICDE*, vol. 5, 2005, pp. 1013–1022.

[28] R. Motwani, S. U. Nabar, and D. Thomas, "Auditing sql queries," in *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*. IEEE, 2008, pp. 287–296.

[29] R. Kaushik and R. Ramamurthy, "Efficient auditing for complex sql queries," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. ACM, 2011, pp. 697–708.

[30] J.-W. Byun and N. Li, "Purpose based access control for privacy protection in relational database systems," *The VLDB Journal*, vol. 17, no. 4, pp. 603–619, 2008.

[31] P. Colombo and E. Ferrari, "Enforcement of purpose based access control within relational database management systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 26, no. 11, pp. 2703–2716, 2014.

[32] ——, "Efficient enforcement of action-aware purpose-based access control within relational database management systems," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 27, no. 8, pp. 2134–2147, Aug 2015.

[33] P. Upadhyaya, M. Balazinska, and D. Suciu, "Automatic enforcement of data use policies with datalawyer," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 213–225.