PSpec-SQL: Enabling Fine-Grained Control for Distributed Data Analytics

Chen Luo, Fei He, Fei Peng, Dong Yan, Dan Zhang, Xin Zhou

Abstract—Business organizations regularly collect customer data to improve their services. Organizations may want to share data within themselves or even with third-parties to maximize data utility. Since business data contain lots of customer data, organizations must respect customers’ privacy expounded by privacy laws. In this paper, we present PSpec-SQL, a distributed data analytics system that automatically enforces privacy compliance for SQL queries. Our system provides a high-level language PSpec for the data owner to specify her data usage policy. As usual, the data analyst queries data to perform data analysis, but our system checks each query to ensure only policy-compliant queries are executed. We have implemented a prototype of PSpec-SQL on top of Spark-SQL, and carried out a case study on the TPC benchmarks. The results show the practicability of our system with negligible overhead over query processing.

Index Terms—Access Control, Data Analytics, Privacy Protection.

1 INTRODUCTION

Recent advances in cloud computing bring great opportunities for many small and medium enterprises to discover values from the business data with data analytics techniques [1]. Cloud providers like Amazon Web Service have already provided data analytics platforms as cloud services. Instead of building IT infrastructures on their own, the enterprises can directly leverage cloud infrastructures and outsource data analytics tasks to third party data analysts to perform data analysis.

However, when analyzing business data, one major concern of the enterprises is the privacy issue. As the business data contains lots of private information such as identities and personal preferences, misusing such data can cause privacy breaches and severely degrade the enterprise’s reputation. A traditional technique to protect customer’s privacy [2] is based on syntactic anonymization. Through anonymization, data related to private information are removed or generalized. But anonymization lacks flexibility since the data is anonymized beforehand and fails to adjust the data privacy and utility based on the actual needs of different data analytics tasks.

The privacy issue is even more intractable with the introduction of distributed computing into the working environment. From the data providers’ viewpoint, their private data are computed on remote, unknown, and potentially untrusted machines. From the data analysts’ viewpoint, the input data may originate from multiple sources. Traditional techniques for centralized systems are thus not sufficient. Privacy protection for distributed data analytics is demanded.

There exist some distributed big data analytics systems, e.g., Spark-SQL [3] and Hive [4]. These systems greatly hide the complexities of distributed systems, while providing simple SQL-like data models and query languages for efficient large-scale data analytics. Most existing distributed data analytics systems share a similar workflow of query execution. The submitted query is first parsed and analyzed in the master node, and then distributed to a set of worker nodes. These worker nodes concurrently execute to produce the final results.

Differential privacy [5], [6] is a privacy protection technique adopted in some distributed data analytics systems. A carefully designed curator resolves a pre-determined set of queries from third parties. Through noisy answers, one can argue that private information cannot be leaked through the curator. Designing differentially private curators however requires tedious analysis and mathematical insights. Adopting differential privacy in organizations with varying business data can be too challenging at present.

Thus, a more practical approach is to let the data owner specify data use restrictions to avoid the misuse of privacy-related data. Similar to differential privacy, a curator is used to mediate data and queries. If a query does not access the data according to these restrictions, the curator immediately rejects the query. One central requirement for the specification language is to get a good balance between data privacy and data utility, such that the customer’s privacy is well protected while the data is still usable.

Conventional access control languages, such as EPAL [7] and XACML [8], are not suitable for privacy protection in data analytics. A data usage specification may involve complicated trigger conditions and intricate handling, which can readily exceed the expressiveness of these languages. Consider the sales information of an online retail company. The company may be willing to share the amounts of all transactions last year. It may be hesitant to share the amounts of all transactions associated with an IP address last year. Instead of blindly rejecting access of sensitive information,
the company may release such sensitive information after desensitization. For instance, transaction amounts associated with truncated IP addresses can be shared, as well as average transaction amounts associated with full IP addresses.

Purpose-based access control techniques [9], [10], [11], [12], [13], which extend the traditional database systems to enforce privacy policies, are still not fully applicable to the distributed data analytics. These techniques allow the data owner to declaratively specify data access privileges in the database systems. However, their specifications directly operate on relational data, and need be written by database experts. To adhere to privacy laws and policies, privacy specifications are more expected to be handled by legal experts [14], who have little background in databases, and the specification language is thus necessarily at an abstract level to hide the details of data models. Such abstraction also benefits organizations by allowing specification reuse as personal-related data concepts are often shared in spite of various data sources. Furthermore, the semi-structured complex data types, e.g., struct, array and map, supported by distributed data analytics systems further complicate the policy enforcement process.

Moreover, all previous works in access control techniques [7], [8], [11], [12], [13] consider information being independent of each other, i.e., accessing multiple pieces of information together is equivalent to accessing each piece separately. However, this is not the case for data analytics because of the association among information. For example, access customer’s address and health condition separately may be acceptable, while access them together will cause severe privacy breach since it directly leaks health condition for each customer.

The data usage specification alone does not guarantee compliance unless it can be automatically enforced. As the first step towards enforcement, the gap between the abstract concepts in the specification and the underlying data (relational data in this paper) should be filled through a data labeling process. For the performance consideration, we consider column-level labeling so that queries can be statically checked with little performance overhead. This is important for today’s distributed big data systems, which are often targeting at large-scale and long running data analytics queries. Nevertheless, data labeling is non-trivial for two issues. Sometimes the proper meanings of some columns cannot be determined statically, but rather depend on how the table is accessed with other tables. For example, the Address table may contain both customer addresses and store addresses, and such ambiguity can only be resolved when the query is present. Moreover, the complex data types as mentioned above further complicate the data labeling process. But once the data is properly labeled, the privacy specification can be enforced against submitted queries with program analysis techniques.

Putting things together, we present PSpec-SQL, a distributed data analytics system which automatically enforces data-usage compliance. PSpec-SQL is built upon Spark-SQL[1] and provides a simple and high-level specification language PSpec for the data owner to specify data usage restrictions. At runtime, each submitted query is checked to ensure only specification-compliant queries are executed.

Contributions. Our main contributions in this paper are summarized as follows:

- We introduce PSpec, a high-level data usage specification language for distributed data analytics systems, with explicit support for data association and desensitization.
- To link the PSpec concepts with the underlying data, we provide comprehensive support for labeling relational data.
- We present the checking algorithm for SQL queries with information flow analysis techniques.
- Finally, to evaluate our system, we carried out a case study on three industry standard benchmarks[15]. The results show the usability and practicability of our system with negligible overhead over query execution.

A preliminary discussion of the PSpec language was included in a conference paper [16] by the same authors. Compared to [16], this paper makes the following new contributions. Firstly, [16] was focused on the PSpec language only, while this paper presents a PSpec-based distributed data analytics system that automatically enforces privacy compliance against SQL queries. Among the four contributions summarized in the above paragraph, the second and third ones are completely new compared to [16]. Secondly, even for the PSpec language, it is only sketched at a very high level in [16]. In contrast, this paper gives a full description of PSpec, including its formal syntax and semantics.

The rest of the paper is organized as follows. Section 2 presents an overview of our system. Sections 3, 4 and 5 introduce the details of the PSpec language, data labeling, and the privacy checking algorithm respectively. Section 6 reports our implementation and evaluation. Section 7 further discusses some possible extensions of our system and Section 8 briefly reviews the related works. Finally, Section 9 concludes this paper.

2 System Overview

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

2.1 Motivating Example

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

- Customer: stores customers’ personal information, and customer addresses are stored in the Address table.
- Store_Sales: stores sales information, and refers to the Customer, Store, and Item tables.
- Item: stores items information.
- Store: contains stores information, and store addresses are stored in the Address table.

1. https://spark.apache.org/sql/
With the business data at hand, the company lets data analysts to query the data to perform data analysis. However, one major concern of the retail company is that the customer data (the Customer and Address tables) may be improperly used by the data analyst. For example, the malicious data analyst may query all customer names with living addresses and sell them to other companies, which directly violates related privacy laws and causes severe privacy breaches. Thus, to ensure the data are properly used, the legal team in the company enacts some privacy-related data use policies, which are expected to be automatically enforced by data analytics systems.

### 2.2 PSpec-SQL Overview

The scenario considered in this paper mainly involves the following logical participants:

- **Customer**: stores the addresses for customers and stores.
- **Address**: stores the addresses for customers and stores.
- **Data Owner**: e.g., the retail company, who owns the business data.
- **Data Analyst**: who queries the business data to perform data analysis.
- **Data Analytics System**: which manages the business data and allows the analyst to submit queries.

Fig. 2 depicts an architecture overview of our data analytics system, i.e., PSpec-SQL. PSpec-SQL is built upon Spark-SQL, and contains several new modules, including the PSpec parser, label manager and policy checker. The PSpec parser parses the PSpec policy defined by the data owner, and the label manager manages a set of labels which connect the abstract data concepts in PSpec with the relational data. With the PSpec policy and the labeled data set, the policy checker checks each submitted query during query processing phase to ensure the specification compliance of submitted queries.

We further elaborate the work flow of our system with the motivating example. As mentioned before, to ensure the data are properly used, the legal team of the retail company may have enacted several data use policies according to the related policy and laws. Typical policies include “the direct output of the customer name is forbidden”, “only the aggregated sales price is allowed when outputted with personal information together” and “the use of the customer living state, city, and street together is forbidden”. The data owner may also adjust the data use 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, as shown in Section 3.

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

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

```sql
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 use policies as usual.

### 3 Specification Language

In this section, we present our specification language PSpec. As mentioned, since PSpec targets at legal experts, who may have little background in databases, it is thus designed as a high-level language based on abstract concepts. Moreover, PSpec is designed for specifying data use restrictions in the distributed data analytics systems. Most of these systems provide a SQL-like query language, which allows the user to apply a sequence of transformations to produce the final results. The PSpec rules thus specify restrictions on these data accesses and transformations in order for privacy protection.
In the remainder of this section, we first present the language syntax as well as the design choices, then introduce the formal semantics.

3.1 PSpec Syntax

Since PSpec is based on abstract concepts, we separate PSpec into the vocabulary part and policy part as in EPAL [2]. Briefly, the vocabulary part defines abstract concepts while the policy part defines a set of the PSpec rules.

3.1.1 Vocabulary

In the vocabulary part, the data owner defines a set of user categories and data categories. A user category represents a role of the analyst. A data category represents a privacy-related data concept. Both user and data categories are hierarchical. In a category hierarchy, a parent category is the generalization of its child categories; a child category is a specialization of its parent category. When a parent category is referred, all its descendants are also referred.

Fig. 3 shows category hierarchies for the retail company. The top-level user category is Analyst, which represents all data analysts. 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 association, such as the combination of Birth, Zip, and Gender [2]. Finally, Sensitive Attribute carries sensitive personal information, such as customer sales records.

The data owner should also identify and specify all supported desensitization operations for data categories. Informally, desensitize operations can be used to make sensitive data categories ready for access. For example, the desensitize operation for Sale_Price can be an aggregate operation, such as Min, Max, and Avg etc., while for Zip it can be the truncate operation. Techniques from data anonymization [2] and differential privacy [3, 4] can also be employed to realize the desensitization operation. A data category automatically inherits its ancestors’ desensitize operations.

3.1.2 Policy

Before introducing details, we discuss data access issues pertaining to data analytics. In data analytics, accessing multiple pieces of information together may not be equivalent to accessing each piece separately because of association. For example, accessing a customer’s address and salary separately may be acceptable. Yet accessing them together causes a severe privacy breach. Instead of forbidding accessing any sensitive information, desensitization is useful to balance between privacy and utility in data analytics. For instance, full IP addresses may locate individuals, and hence are considered harmful; truncated IP addresses may be acceptable and useful to data analysis.

The data owner defines a set of PSpec rules to regulate data access. Informally, each rule states under what restrictions can a user category access certain data categories together. In order to describe intricate data usage specifications, PSpec provides explicit support for data association and desensitization. We require a user category to satisfy all the applicable rules. The reason is that usually only a part of the business data is related to customers’ privacy. This allows the data owner to focus on privacy-related data.

The rule grammar of PSpec is shown in Fig. 4. Each PSpec rule contains a scope part and a restriction part (separated by =>). The scope part is defined by User-Ref and Data-Assoc (short for Data Association). User-Ref refers a user category (<user>) to specify applicable user categories. When a category is referred, all its descendants are also referred except those excluded explicitly. Data-Assoc specifies applicable association of data categories by a sequence of Data-Refs. Each Data-Ref refers a data category (<data>) with an action. An action specifies how a data category is accessed. Two types of actions are distinguished in our system since they have different privacy implications. The projection action means that the data category is output by a query, and the condition action denotes that the data category is used in the control flow of a query. The actions also form a hierarchy where access is the parent of projection and condition (see Fig. 5). Note that from the traditional information flow point of view, even the condition action may cause information leakage. We intentionally separate projection from condition since the former has a stronger implication and let the user to balance the utility and privacy based on their needs. For Data-Assoc, data categories referred in each Data-Ref must be disjoint, i.e., no referred data category overlaps with each other.

A rule can forbid the query or specify desensitize requirements by restrictions. A restriction is a sequence of Desensitize with the same length as the data association of the rule. Each Desensitize specifies a set of admissible desensitize operations (<operation>), and requires the corresponding data category in the data association be desensitized with one of them (∅ denotes no desensitization required). The corresponding data category must support the specified desensitize operations as defined in the vocabulary. A query must satisfy one of these restrictions in an applicable rule. 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 example rules in Section 2.2. For simplicity, we assume all rules are applicable to Analyst. First, the company forbids the analyst access any customer’s name, which can be formalized as follows:

\[ r_1: \text{Analyst}, [\text{access Name}] \Rightarrow \text{forbid} \]

Second, the company requires the sale price be aggregated when output with personal information together.

\[ r_2: \text{Analyst}, [\text{projection All exclude Sensitive Attribute, projection Sale_Price} \Rightarrow \{(\text{avg, max, min, sum})\} \]

In \( r_2 \), we define a data association of length two. The first data access refers to the projection on all but sensitive data categories. The second data access refers to the projection on sales price. Rule \( r_2 \) further requires Sale_Price be aggregated by one of the avg, max, min, or sum operations. Note that \( r_2 \) refers to data association and cannot be specified by EPAL [7] and XACML [8].
3.2 Formal Semantics

After introducing the syntax of PSpec, now we discuss its formal semantics. Recall that PSpec specifies restrictions on data access, thus the semantics are defined against queries. In the following, we first introduce some notions, and then discuss the semantics of the query and the PSpec policy respectively. All notations are summarized in Table 1.

Let $\mathcal{U}_{user}$, $\mathcal{U}_{data}$, and $\mathcal{U}_{act}$ be the universal set of all user categories, data categories and actions, respectively. We use $u \in \mathcal{U}_{user}$, $d \in \mathcal{U}_{data}$, and $a \in \mathcal{U}_{act}$ to denote a user category, data category, and action, respectively. Especially, we denote the pair $(d, a)$ a data access. Let $\mathcal{U}_{da} = \mathcal{U}_{data} \times \mathcal{U}_{act}$ be the set of all data accesses with respect to $\mathcal{U}_{data}$ and $\mathcal{U}_{act}$. An associated data access $\tau$ of size $n$ is an $n$-tuple $\tau = (\tau_1, \tau_2, \ldots, \tau_n)$, and $\tau_i \in \mathcal{U}_{da}$ for $1 \leq i \leq n$. Denote $\mathcal{U}_{da}^n$ the set of all associated data accesses of length $n$. Given two data access tuples $\tau$ and $\tau'$, we say $\tau$ is included in $\tau'$ (written $\tau \leq \tau'$) if all data accesses contained in $\tau$ are also contained in $\tau'$. Informally, if we treat both $\tau$ and $\tau'$ as a set, $\tau \leq \tau'$ just means $\tau$ is a subset of $\tau'$.

Let $\mathcal{U}_{op}$ be the universal set of all available desensitize operations. We use $\Delta \subseteq \mathcal{U}_{op}$ to denote a set of desensitize operations, and $\delta \in \mathcal{U}_{op}$ a desensitize operation. Specifically, denote $\delta_0 \in \mathcal{U}_{op}$ the empty desensitize operation, representing the corresponding data category can be directly accessed.

From the information flow point of view [17, 18], the semantics of a query $q$ is defined upon a set of data leakages. A data leakage is a pair $(\langle d, a \rangle, \delta)$, where $\langle d, a \rangle$ is a data access and $\delta$ a desensitize operation. A data leakage $(\langle d, a \rangle, \delta)$ means that the data $d$ is leaked through action channel $a$ after the desensitize operation $\delta$. Let $|q|$ be the number of queries.

Finally, the company forbids the access of state, city, and street together.

**r₃:** Analyst, [access State, access City, access Street] => forbid

Note that $r₃$ only forbids access to them together. One can access any one or two of them freely. Again, $r₃$ cannot be specified by the existing access control languages [7, 8].

The reader may have already noticed that PSpec does not model access purpose directly, since it is difficult for data analytics systems to automatically verify the claimed purpose. For simplicity, we assume each data analyst is assigned a proper user category by the system administrator based on her tasks and purposes. However, it is straightforward to extend PSpec with other language elements like purpose and obligation. This issue is further discussed in Section 4. Also note that a data category is considered desensitized as long as one of the specified desensitize operations is performed. We do not support a sequence of desensitize operations since it complicates both the data owner for writing the 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.
data leakages in \( q \). For ease of discussion, we extract the data accesses and desensitize operations from the data leakages of \( q \), each as a \(|q|\)-tuple.

**Definition 3.1 (Query).** A query \( q \) is a triple \( q = (q.\text{user}, q.\text{asso}, q.\text{des}) \), where

- \( q.\text{user} \in U_{\text{user}} \) is the user who submits \( q \),
- \( q.\text{asso} \in \bigcup_{\text{data}}^{|q|} \) is a tuple of data accesses, and
- \( q.\text{des} = (\delta_1, \delta_2, \ldots, \delta_{|q|}) \) is the desensitize operations in \( q \).

To avoid semantic inconsistency, we assume any data category occurring in a query is a leaf node in the data hierarchy. The reason is that a data category is restricted often implies its ancestors are also restricted. For example, \( \text{Zip} \) should be truncated implies \( \text{Address} \) should also be truncated. However, this could lead to semantic inconsistency since the truncate operation may not be supported by \( \text{Address} \). To handle this, we require a query only access leaf data categories, but this will not be a limitation in practice since one can simply label one piece of data with multiple data categories (see Section 4).

For example, consider \( \text{Report Analyst} \) submits a query \( q_1 \), which projects \( \text{State} \), \( \text{City} \), and aggregated \( \text{Sale Price} \). Obviously, \(|q_1| = 3\), and

\[
q_1.\text{user} = \text{Report Analyst}, \\
q_1.\text{asso} = \langle (\text{State}, \text{projection}), (\text{City}, \text{projection}), (\text{Sale Price}, \text{projection}) \rangle, \\
q_1.\text{des} = (\delta_8, \delta_9, \text{avg}).
\]

Note that \( \delta_8 \) is used if one wants to access data without any desensitization.

Since a PSpec policy contains a set of PSpec rules, we first introduce the semantics of a PSpec rule. Note that the associated data accesses referred by a rule \( r \) have the same size, denoted as \(|r|\). Given a rule \( r \), we use \( r.\text{User} \subseteq U_{\text{user}} \) and \( r.\text{asso} \subseteq \bigcup_{\text{data}}^{|r|} \) to denote the set of user categories and the set of associated data accesses applicable to \( r \), respectively. In the grammar of Fig. 4, \( r.\text{User} \) is specified by User-Ref. It corresponds to the descendants of the specified user category minus the descendants of any of the excluded user categories. \( r.\text{asso} \) is specified by Data-Assoc in the grammar. Let \( r.\text{asso} \) be the set of data accesses specified by the \( i \)-th Data-Ref of Data-Assoc, which is obtained similarly as \( r.\text{User} \). \( r.\text{asso} \) is the Cartesian product of \( r.\text{asso} \) for \( 1 \leq i \leq n \).

Given an associated data access \( \tau = (\langle d_1, a_1 \rangle, \langle d_2, a_2 \rangle, \ldots, \langle d_n, a_n \rangle) \) of size \( n \), a restriction \( \bar{\xi} \) is an \( n \)-tuple \( (\Delta_1, \Delta_2, \ldots, \Delta_n) \), where \( \Delta_i \subseteq U_{\text{op}} \) specifies the set of admissible desensitize operations. An operation in \( \Delta_i \) must be taken to protect \( d_i \) against \( a_i \). Given a rule \( r \), we use \( r.\text{Res} \) to denote the set of restrictions specified in \( r \). Each restriction \( \xi \in r.\text{Res} \) is specified by Restriction, in the grammar.

**Definition 3.2 (Rule).** A rule \( r \) is a triple \( r = (r.\text{User}, r.\text{asso}, r.\text{Res}) \), where

- \( r.\text{User} \subseteq U_{\text{user}} \) is the set of user categories,
- \( r.\text{asso} \subseteq \bigcup_{\text{data}}^{|r|} \) is the set of associated data accesses, and
- \( r.\text{Res} = \{\xi_1, \xi_2, \ldots, \xi_m\} \) is the set of restrictions. Especially, \( r.\text{Res} = \emptyset \) if \( r \) is a forbidden rule.

For example, consider the following rule:

\[
r_4: \text{Report Analyst}, \text{[projection Address, projection Sale_Price]} \Rightarrow (\{1\}, \{\text{sum, avg, min, max}\})
\]

Obviously, \(|r_4| = 2\), and

\[
r_4.\text{User} = \{\text{Report Analyst}\}
\]

\[
r_4.\text{asso} = \{(\langle \text{Address, projection} \rangle, \langle \text{Sale_Price, projection} \rangle), \langle \text{State, projection} \rangle, \langle \text{Sale_Price, projection} \rangle, \langle \text{Street, projection} \rangle, \langle \text{Sale_Price, projection} \rangle, \langle \text{Zip, projection} \rangle, \langle \text{Sale_Price, projection} \rangle\}
\]

\[
r_4.\text{Res} = \{\langle \text{\emptyset, \{sum, avg, min, max\}\rangle\}
\]

Note that for ease of discussion, \( U_{\text{op}} \) is used if there is no desensitize requirement, i.e., Restriction, in the grammar contains no desensitize operation.

**Definition 3.3.** A rule \( r \) is applicable to a query \( q \) (or equivalently, \( q \) triggers \( r \)) if

- \( q.\text{user} \in r.\text{User} \), and
- there exists at least one associated data access \( \tau \) in \( r.\text{asso} \), such that \( \tau \leq q.\text{asso} \).

The first condition in above definition performs user scope check, and the second condition performs the scope check of associated data access. The rule is applicable to the query only if both checks are passed. Moreover, if there exists an \( \tau \) in \( r.\text{asso} \) such that the second condition holds, we further define \( q.\text{des}|_{\tau} \) as the projection of \( q.\text{des} \) to \( \tau \), obtained by removing \( q.\text{des}|_{\tau} \) \( \{ \) \( (i \)-th element of \( q.\text{des} \) \( ) \) \( \} \) if \( q.\text{asso}|_{\tau} \{ \) \( (i \)-th element of \( q.\text{asso} \) \( ) \) \( \} \) does not occur in \( \tau \) for \( 1 \leq i \leq |q| \). The size of \( q.\text{des}|_{\tau} \) equals to \(|\tau| \). Note that there may be more than one \( \tau \) satisfying the second condition. With different \( \tau \), the set \( q.\text{des}|_{\tau} \) may be different. We thus define

\[
q.\text{des}|_{\tau} = \{q.\text{des}|_{\tau}| \tau \in r.\text{asso} \text{ and } \tau \leq q.\text{asso}\}
\]

**Definition 3.4.** A \( n \)-tuple of desensitize operations \( \delta = (\delta_1, \delta_2, \ldots, \delta_n) \) satisfies a restriction \( \xi = (\Delta_1, \Delta_2, \ldots, \Delta_n) \) if \( \delta_i \in \Delta_i \) for \( 1 \leq i \leq n \).

**Definition 3.5 (Rule Satisfaction).** A query \( q \) satisfies a rule \( r \) if and only if \( r.\text{Res} \) is satisfied by \( q.\text{des}|_{\tau} \) for some \( \tau \) which occurs in \( r.\text{asso} \). If \( \tau \) exists at least one restriction \( \xi \in r.\text{Res} \), such that \( \delta \) satisfies \( \xi \).

Intuitively, a query satisfies a triggered rule if all data categories accessed by the query are properly desensitized. Consider the rule \( r_4 \) and the query \( q_1 \) in this subsection. \( r_4 \) is applicable to \( q_1 \) since \( q_1.\text{user} \in r_4.\text{User} \) and two former associated data accesses in \( r_4.\text{asso} \) are included in \( q_1.\text{asso} \). Hence, \( q_1.\text{des}|_{r_4} = \{\{\delta_8, \text{avg}\}\} \). Since \( \delta_8, \text{avg} \) satisfies the restriction in \( r_4.\text{Res} \), \( r_4.\text{satisfied by} q_1 \) (Definition 3.5).

**Definition 3.6 (Policy Satisfaction).** A query \( q \) satisfies a PSpec policy iff \( q \) satisfies all rules in the policy.
4 Data Labeling

After introducing the 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 to enable policy enforcement. Currently, we support data labeling on column level so that queries can be statically checked with little performance overhead. Thus, the data owner should label data columns with proper data categories based on column semantics. Labeling one column with multiple data categories is also supported if the column corresponds to multiple data categories. However, recall from Section 3.2, 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 provide multiple implementations for the 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 column IP1, while integer in column IP2. Thus, the data owner can provide UDFs truncate1 and truncate2 for the truncate operation for columns IP1 and IP2 respectively to keep UDFs simple.

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

4.1 Conditional Labeling

One problem with data labeling is that the proper data categories for some columns cannot be determined statically, but rather depend on what table the current table is joined with. Consider the tables in Fig. 1. The data stored in the Address table could be either the customer address or store address since the Customer table and the Store table both have a foreign key referring to Address. Thus, the actual data category for the Address table depend on what table it is joined with, which can only be determined when the query is present.

This 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 a join condition. A join condition specifies the table is joined with what table on what column(s). For the previous example, the data owner can use conditional labeling to label the zip column as follows:

LABEL Address.zip WITH Zip WHEN JOIN Address.a_id = Customer.c_addr_id

This states that the zip column of the Address table is labeled with the Zip category only if this table is joined with the Customer table on a_id=c_addr_id in a query. In other cases, for example, the Address table is accessed without joining with the Customer table, this zip column will not be labeled with the Zip category. The actual data categories for these conditionally labeled columns are determined during the query checking phase, which will be discussed in Section 5.2.

4.2 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, distributed data analytics 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. 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 predefined keys.

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

As a concrete example, consider the following column c_customer with struct type which stores customer information:

c_customer: struct{
  c_name: string,
  c_gender: string,
  c_address: struct{
    c_state: string,
    c_city: string,
    c_zip: string
  },
  c_contact: map[string, string]
}

c_customer is a struct column which contains primitive columns (c_name, c_gender), a nested struct column c_address for storing the customer address information, and a map column c_contact for storing the contact information with pre-defined keys such as ‘phone’ and ‘fax’. To label column c_customer, the user can label each sub column separately. For example, c_customer.c_name can be labeled with category Name, while c_customer.c_address.c_state, c_customer.c_address.c_city, and c_customer.c_address.c_zip, can be labeled with categories State, City, and Zip respectively. Similarly, c_customer.c_contact[‘phone’] can be labeled with category Phone.

Interestingly, even some columns with primitive types may have composite semantics. For example, a birth column with the Date type may contain all information of categories birth_year, birth_month, and birth_day, which further can be extracted with UDFs getYear, getMonth, and getDay respectively. To enable fine-grained control on composite 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 the birth_year category, while getMonth on the birth column as the birth_month category.

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.

5 Policy Enforcement

With the PSpec policy and the labeled data set, we now discuss the policy enforcement process against data analytics queries written in SQL-like languages. Most existing distributed data analytics systems share a similar workflow of query execution. The submitted query is first parsed and analyzed in the master node, e.g., the Driver in Spark-SQL, which is then compiled into a distributed job. Finally, a set of worker nodes concurrently executes the compiled distributed job on partitioned data to transform data and produce final results.

Since we check queries statically, the queries are checked by our policy checker before being compiled into distributed jobs in the master node. The basic idea is to analyze 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 the data category flow. Finally, the data category flow is checked against the PSpec policy according to the semantics in Section 5.2. Here we assume readers are familiar with query processing in relational database systems, and the reference material can be found in [19]. For ease of discussion, we assume queries have been parsed and optimized as optimized query plans by the query processor.

5.1 Lineage Tree Construction

In the following, we use the term stored attributes to refer to the attributes that originate from tables, i.e., columns. Since data categories and desensitize operations are essentially stored attributes and UDFs respectively, we first construct attribute lineage trees to track how each stored 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 the average sales price for female customers in each state and city:

```sql
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
```

Fig. 6a shows the logical plan for this query generated by Spark-SQL. From the logical plan, we construct the corresponding lineage trees for attributes used in projection and condition as shown in Fig. 6b and Fig. 6c respectively. Since stored attributes are directly used in conditions, the lineage trees in Fig. 6c 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 AVG function.

Now we discuss how to construct attribute lineage trees from a query plan. Recall that in our system the source data is managed by data owners while analysts are only allowed to access the data for analysis, thus we only consider select queries here. The basic algorithm framework is shown in Fig. 7. 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. Function PROPAGATE performs post-order traversal over the query plan. For each plan operator, it calls function TRANSFORM and function COLLECT to update plan.projections and conditions respectively. Briefly, TRANSFORM calculates 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 functions TRANSFORM and COLLECT for each plan operator. By default, TRANSFORM simply returns map projections of the child operator if the current operator does not perform any attribute transformation, while COLLECT simply returns ∅ for operators that do not contain any predicate. Example operators include Limit and Distinct etc. While TRANSFORM and COLLECT for the following operators need special care and will be explained 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 on top of the lineage trees for the attributes used to define the new attribute. COLLECT returns the lineage trees for the attributes used in the predicates in the If and CaseWhen statements (if any). Both statements are supported by SQL. Especially, an If statement is of the form: if $c$ then $v_1$ else $v_2$, where $c$ is a predicate, $v_1$ and $v_2$ are two values; a CaseWhen statement is more general and needs to evaluate a set of predicates. When resolving predicates, COLLECT should also add the corresponding transformations on top of the existing 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 map projections of the left and right operators. COLLECT returns the lineage trees by resolving the predicates in the join condition.
- **Aggregate**: Aggregate is processed similarly as Project, and aggregate functions like COUNT, SUM, AVG, MIN and MAX are simply treated as transformations.
- **Sort**: TRANSFORM works as default. COLLECT returns the lineage trees for the attributes used in the sort expression.
- **Binary operators**: For 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
1:  \texttt{conditions} \leftarrow \emptyset \\
2:  \texttt{function PROPEGATE(plan)} \\
3:      \texttt{for child in plan.children do} \\
4:          \texttt{PROPEGATE(child)} \\
5:      \texttt{plan.projections} \leftarrow \texttt{TRANSFORM(plan)} \\
6:      \texttt{conditions} \leftarrow \texttt{conditions} \cup \texttt{COLLECT(plan)} \\
7:  \texttt{return (plan.projections.values,conditions)} \\

Fig. 7. Algorithm for Computing Lineage Trees

the left and right operators with a dummy transformation. \texttt{COLLECT} works as default.

For example, consider the logical plan in Fig. 6a. For table operators \texttt{Store_Sales}, \texttt{Customer}, and \texttt{Address}, \texttt{conditions} is always \emptyset, while \texttt{projections} of each operator is initialized as discussed above. For \texttt{Join} on \texttt{Store_Sales} and \texttt{Customer}, \texttt{projections} is the union of \texttt{projections} of \texttt{Store_Sales} and \texttt{Customer}, while the lineage trees for attributes \texttt{ss_customer_id} and \texttt{c_id} are added into \texttt{conditions}. Another \texttt{Join} is processed similarly. Then for \texttt{Filter}, \texttt{projections} is inherited from its child, while the lineage tree for attribute \texttt{c_gender} is added into \texttt{conditions}. Finally for \texttt{Aggregate}, the aggregate expression defines two attributes, which are \texttt{concat} on \texttt{a_state} and \texttt{a_city}, and \texttt{AVG} on \texttt{ss_price}. Thus, \texttt{projections} maps the defined attributes to the lineage trees in Fig. 6b and \texttt{conditions} contains the lineage trees in Fig. 6c.

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

5.2 Flow Extraction

Since lineage trees are only intermediate structures representing attribute flows, we should then extract data category flows for policy checking. To achieve this, we need to determine the data category for each stored attribute and map UDFs to corresponding desensitize operations. It is straightforward to determine the data categories for statically labeled attributes, while the difficulty here is to determine the data categories for attributes with conditional labels and complex data types as discussed in Section 4.

**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 \texttt{Join} operator, since the query may filter join results in the \texttt{Filter} operator or even the \texttt{CaseWhen} and \texttt{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 in a query, 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, approximation is necessary to ensure 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. Consider predicate \( a + b = c \), 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:

\[
\begin{align*}
&\text{(SELECT a_state, a_city, a_street} \\
&\text{FROM Address JOIN Customer ON a_id = c_addr_id} \\
&\text{WHERE c_gender = 'M')} \\
&\text{INTERSECT} \\
&\text{(SELECT a_state, a_city, a_street} \\
&\text{FROM Address JOIN Store} \\
&\text{WHERE a_id = s_addr_id)}
\end{align*}
\]

Suppose the columns in the \texttt{Address} table are labeled with the \texttt{State}, \texttt{City}, and \texttt{Street} categories only when the \texttt{Address} table is joined with the \texttt{Customer} table on \texttt{a_id} = \texttt{c_addr_id}. We assume stored attributes are equipped with
We have already known how each data category is modeled as a triple $(a, q, \text{asso}, q, \text{des})$, which exactly constitutes $q, \text{asso}$, and $q, \text{des}$.

With $(q, \text{user}, q, \text{asso}, q, \text{des})$, we can check whether the query satisfies the policy according to the semantics discussed in Section 3.2. 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 subsequent executions. Thus, our system guarantees only queries satisfying the PSpec policy are executed.

One advantage of our query checking algorithm is that it is totally based on information flow analysis on the query plan without touching the underlying data being queried. We discuss the complexity of our policy checking algorithms in the following. The complexities of lineage tree construction and flow extraction are omitted here, since they are both straightforward one-pass algorithms. While for query checking, in the worst case, it may take $O(d \cdot n^d)$ time to check the rule satisfaction for a query (Definition 3.3), where $n$ is the number of data categories in the vocabulary, and $d$ the length of associated data accesses in the rule. In practice, the query checking algorithm is quite efficient, since the rules rarely refer to long associated data accesses, which makes $d$ a small constant. From our evaluation, checking complex queries can always be finished within milliseconds, which shows it is suitable for big data systems since it only incurs negligible overhead over query execution. Our algorithm do not handle multiple correlated queries, since in practice most queries are independent and linking results from multiple queries together will further exponentially increase the complexity of our algorithm.

6 IMPLEMENTATION AND EVALUATION

We have implemented PSpec-SQL on top of Spark-SQL 1.5.0 with support for Hive tables. The PSpec parser and label manager are both implemented as standalone modules, while the policy 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. The source code of our prototype implementation is available on GitHub. Note that the policy checker can also be implemented as a separate proxy over existing database systems. The main change is to construct the lineage trees from SQL parse trees instead of query plans.

To evaluate our prototype implementation, we carried out a case study on several industry standard benchmarks obtained from the Transaction Processing Performance Council (TPC) with the focus on usability, performance, and practicability. Each benchmark consists of several database tables and a number of SQL-based queries. All benchmarks contain some customers’ privacy information. In the case study, we focus on protecting these customers’ information from being improperly used via our PSpec-SQL system.

In practice, a data owner may possess several databases, and he may want to formulate a common privacy policy over all his possessed databases. To reflect this requirement,
we conducted the case study in the following steps. First, we
sketched a vocabulary among all benchmarks, and drafted
a common PSpec policy. Second, we labeled the relevant
columns of tables in these benchmarks with the proper
data categories. Finally, we checked the compliance of all
queries included in these benchmarks with our PSpec policy.
The violated queries were detected and reported by our
PSpec-SQL system. The detailed process is discussed below
following a description of TPC benchmarks used in our
evaluation.

6.1 TPC Benchmarks
The Transaction Processing Performance Council (TPC) is a
worldwide consortium whose goal is to define and dissemi-
nate industry standard benchmarks for databases and trans-
actional systems [20]. Many top hardware and software ven-
dors, including Microsoft, Intel, IBM, Dell, Hewlett Packard
(HP), Cisco etc, are members of TPC. The industry members
participate in the development of TPC benchmarks.

There are currently 11 industry benchmarks actively
maintained by TPC. Many of these benchmarks are de-
signed to measure performance (like speed, reliability etc)
of transaction processing. Since our focus is the privacy pro-
tection of data analytics, we require the benchmark contain
SQL analytics queries on some customers’ information. As
a result, three TPC benchmarks are selected, namely TPC-
DS [15], TPC-H [20], and TPCx-BB [21].

TPC-DS benchmark [15] is a general benchmark for
decision support systems. TPC-DS models a national-wide
retail company, which sells goods through stores, catalogs,
and the Internet. It includes a database schema (7 fact tables
and 17 dimension tables), which is organized as a snowflake
schema, and 99 queries. For customer-related information,
TPC-DS stores the customer data into the following tables:
Customer, Customer_Address, Customer_Demographics. It also
stores the items bought by customers through Sales tables.

TPC-H benchmark [20] is another benchmark for deci-
sion support systems that allow concurrent data modific-
ations. It models a wholesale supplier that sells or distribu-
tes products worldwide. TPC-H contains 8 base tables and 21
complex SQL queries. It stores the customers’ data into the
Customer table, and the customers’ orders into the Orders
table.

Finally, TPCx-BB benchmark [21] is designed for bench-
marking big data analytics systems such as Spark-SQL [3]
and Hive [4]. It models a similar national-wide retail com-
pany to that of TPC-DS benchmark. However, it empha-
sizes the usage of User-Defined Functions (UDFs) to per-
form more complex data analysis tasks. It further stores
the customer’s click streams into the Web_Clickstreams and
Web_Page tables to keep track of the online activities of
customers. For the query-side, TPCx-BB contains 30 queries
with the emphasis of complex data analysis and machine
learning tasks.

6.2 PSpec Policy
Note that some tables of these benchmarks store and use
customers’ personal information. To ensure these privacy-
related data are properly used, we made a sample PSpec
policy and the process is illustrated as follows.

<table>
<thead>
<tr>
<th>Data Category</th>
<th>Supported Desensitize Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>All</td>
<td>count</td>
</tr>
<tr>
<td>Zip</td>
<td>truncate</td>
</tr>
<tr>
<td>Phone</td>
<td>substr</td>
</tr>
<tr>
<td>URL</td>
<td>substr</td>
</tr>
<tr>
<td>Income</td>
<td>range</td>
</tr>
<tr>
<td>Vehicle</td>
<td>isZero</td>
</tr>
<tr>
<td>Price</td>
<td>sum, avg, min, max</td>
</tr>
</tbody>
</table>

6.2.1 Vocabulary
For simplicity, we assume there is only one user category,
named Analyst. Based on the customer data stored in the
three benchmarks, we sketched a hierarchy of data cate-
gories, shown in Fig. 9. Most data categories in Fig. 9
are self-explanatory, and abbreviations are used when their
meanings are clear. Similar to previous work [2], [22], the
customer-related information is classified as KA (Key At-
ttribute), QI (Quasi Identifier), and SA (Sensitive Attribute),
which are further divided into finer-grained categories.

The supported desensitize operations for some data cate-
gories are shown in Table 2. Note that we have not identified
too many desensitize operations since the schemes of these
benchmarks are well designed and most of their columns
already have fine-grained meanings. Thus, the PSpec rules
can directly refer these fine-grained categories without de-
sensitize operations. In fact, desensitize operations will be
more required when columns have coarse-grained semantics.

6.2.2 PSpec Rules
After identifying user and data categories, we wrote some
PSpec rules to ensure that the privacy-related data is prop-
erly used. These rules are listed in Table 3 where each
rule specifies a data usage restriction to the customers’
information.

The first three rules forbid the direct access of KAs, since
KAs can directly locate individuals. Among the three KAs,
the ID and Email attributes are completely forbidden to be
accessed, and the Phone attribute requires a desensitize op-
eration. Among the many QIs, the Street and Name attributes
can also be used to identify an individual, and are thus
forbidden to be accessed (in rules 4 to 5). Rules 6 to 8 forbid
the associated access of several combinations of QIs since
each of these combinations has also a great chance to locate
individuals. Rule 9 requires a desensitize operation before
the access of Zip. In 2, a sophisticated scene was discussed,
where linking QIs with some public datasets can also locate
individuals. Rules 10 to 13 were used to avoid such kind of
privacy breach. Finally, rules 14 to 18 regulate that some
sensitive attributes must be well desensitized, when they are
accessed in combination with any quasi identifier.

The above PSpec rules were mainly specified by a junior
undergraduate student (who have received one-hour training
in PSpec, and is familiar with the TPC benchmarks). This
student succeeded in writing most of these PSpec rules in
three-hours. This illustrates the usability of the PSpec
language.
TABLE 3

<table>
<thead>
<tr>
<th>Rule</th>
<th>PSpec rules for TPC benchmarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Analyst, [access ID] =&gt; forbid</td>
</tr>
<tr>
<td>2</td>
<td>Analyst, [access Email] =&gt; forbid</td>
</tr>
<tr>
<td>3</td>
<td>Analyst, [access Phone] =&gt; [substr]</td>
</tr>
<tr>
<td>4</td>
<td>Analyst, [projection Street] =&gt; forbid</td>
</tr>
<tr>
<td>5</td>
<td>Analyst, [projection Name] =&gt; forbid</td>
</tr>
<tr>
<td>6</td>
<td>Analyst, [access套, access S_Name, access Suite] =&gt; forbid</td>
</tr>
<tr>
<td>7</td>
<td>Analyst, [access ID, access S_Name] =&gt; forbid</td>
</tr>
<tr>
<td>8</td>
<td>Analyst, [access B_Day, access B_Month, access Suite] =&gt; forbid</td>
</tr>
<tr>
<td>9</td>
<td>Analyst, [access Zip] =&gt; {[truncate]}</td>
</tr>
<tr>
<td>10</td>
<td>Analyst, [projection Birth, projection Address, projection Gender] =&gt; forbid</td>
</tr>
<tr>
<td>11</td>
<td>Analyst, [projection Birth, projection Address, projection Nationality] =&gt; forbid</td>
</tr>
<tr>
<td>12</td>
<td>Analyst, [projection Birth, projection Address, projection Education] =&gt; forbid</td>
</tr>
<tr>
<td>13</td>
<td>Analyst, [projection Birth, projection Address, projection Financial] =&gt; forbid</td>
</tr>
<tr>
<td>14</td>
<td>Analyst, [access QI, access Vehicle] =&gt; {[isZero]}</td>
</tr>
<tr>
<td>15</td>
<td>Analyst, [access QI, access Income] =&gt; {[range]}</td>
</tr>
<tr>
<td>16</td>
<td>Analyst, [access QI, access Price] =&gt; {[sum, count, avg, min, max]}</td>
</tr>
<tr>
<td>17</td>
<td>Analyst, [access QI, projection View_Item] =&gt; {[count]}</td>
</tr>
<tr>
<td>18</td>
<td>Analyst, [access QI, projection URL] =&gt; {[count, substr]}</td>
</tr>
</tbody>
</table>

It should also be noted that Table 3 may not be a complete set of rules for protecting privacy of the benchmarks. Actually, this case study is not for privacy specification, but rather for evaluating the PSpec language. User privacy may subject to different regulations in different organizations and scenarios. In practice, the data owner should adjust the PSpec rules to protect user privacy based on the actual policy requirements and regulations.

6.2.3 Data Labeling

With the data categories, we then labeled the table columns in these three benchmarks with proper data categories. We also implemented desensitize operations, including truncate, range, and isZero, as UDFs and registered them into our system.

In general, the data labeling will not incur too much effort. Many tables of these benchmarks can be skipped in this process, since they are irrelevant to the customers’ information. Even for tables that contain customers’ information, many of their columns are with primitive types (for example, the Customer table of TPC-DS), and thus can be directly labeled based on their semantics.

However, due to our case study spanning several databases, there are several complex scenes that need more careful handling. First, the same information may be stored at different granularity in different databases. For example, TPC-H stores the customer address using a single column named Address, while TPC-DS divides the customer address into several parts and stores them in State, City and Street columns, respectively. To address this, we simply labeled the Address column of TPC-H with a tuple (State, City, Street) of data categories, reflecting the composite semantics of this column. Second, the information stored in a column may be multi-fold. For example, the Name column of the Nation table in TPC-H stores the nationality information for both suppliers and customers, but we may only concern the customers’ nationalities. To resolve this, we used conditional labeling to label this column only when the Nation table is joined with the Customer table. Another example is the _current_price column of the Item table in TPC-DS. 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.

6.3 Query Checking

There are in total 150 queries among all three benchmarks. These queries range from reporting queries, ad hoc queries, iterative queries to data mining queries. For example, TPC-DS contains 99 queries, each of which corresponds to a particular business question. 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.

Some queries contain some syntax that is not supported by Spark-SQL, such as window functions, the WITH statement and sub queries in the WHERE statement, we transform these queries into their equivalences that can be processed by Spark-SQL (the window function is an exception, which was simply removed). The transformed results can be found in our GitHub repository. Moreover, some queries in the TPCx-BB benchmarks integrate Python codes to perform machine learning-related analysis. These Python codes were simply treated as uninterpreted UDFs.

All queries were submitted to our PSpec-SQL system to check their compliance with the sample PSpec policy. We
measured the time cost of policy checking on each query. All queries were efficiently checked. The shortest takes only 1ms and the longest takes 96ms with the average time being 8ms. Our query checking algorithm is totally based on the information flow analysis without touching the underlying data, it only incurs negligible overhead and is thus suitable for online data analysis.

Finally, our PSpec-SQL system reported 25 policy violations, among which 21 were caused by queries of TPC-DS. Note that TPC-DS is a comprehensive benchmark with various kinds of data analytics queries. Its queries 23, 24, 30, 34, 38, 46, 68, 73, 74, 81, and 84 attempt to directly project the customer’s first name and last name, thus violate rules 5 and 7 in Table 3. Moreover, its queries 30 and 81 even attempt to project the customer’s birth day, email, or living street in combination with the customer name, which can cause severe privacy violations. Other violations, including queries 13, 15, 45, 48, 64, and 85 of TPC-DS, directly access Price, Income, or Zip without desensitization as required.

For queries of TPC-H and TPCx-BB, we found only 4 violations. The reason is that queries in these two benchmarks mainly focus on analyzing the general trend of sales and products, without accessing the customer information in detail. But still, queries 10 and 18 of TPC-H, queries 6 and 13 of TPCx-BB directly output customer names, and thus violate rules 5 and 7 in Table 3. Moreover, the query 10 of TPC-H also attempts to output the customer’s address and phone information, thus violates rules 3 and 4, too. The query 6 of TPCx-BB also attempts to output the customer’s country and email information, thus violates rules 2 and 4, too.

Among the 25 reported violations, 4 are false positives. These false positives all belong to TPC-DS and are all caused by unsupported desensitize operations. For example, queries 8 and 19 of TPC-DS desensitize Zip with substr rather than the specified truncate operation. Since substr is not a supported desensitize operation of Zip (see Table 2), these queries were recognized as violations by our system. Similarly, queries 34 and 74 of TPC-DS directly check whether Vehicle is greater than zero in their conditions, but not using the specified isZero operation. All these false positives can be easily eliminated by simply revising the queries to use the appropriate UDFs.

7 Discussion

In this section, we briefly discuss some possible extensions of our system.

**PSpec Elements.** Purpose, condition, and obligation are important concepts in privacy policies [23]. In PSpec, data desensitization can be viewed as a special condition that must be satisfied by queries in order to access data. For ease of discussion and policy enforcement, we have intended to leave out the purpose and obligation elements. However, it is straightforward to extend the syntax and semantics of PSpec with purposes and obligations to express data can only be accessed for certain purposes or what should be performed after a query is executed.

**Record-level Control.** As records of a same table may have different requirements, an interesting extension of our system is to support record-level control. Here we briefly discuss the basic idea, while postpone its implementation as a future work. Following the idea of predicated grants [24], we can extend the PSpec rules with an extra predicate element, which defines a predicate over data categories to specify accessible records. An example predicate could be “only customers with ages greater than 18 can be accessed”. During the privacy checking phase, we first need to transform the defined predicates into concrete predicates over database columns, which can then be easily enforced by rewriting the filter condition of a query to filter out inaccessible records.

**Automatic Data Labeling.** To simplify users’ effort, another possible extension of our system is to automatically label the relational data based on their semantics. By treating the data hierarchy as a special schema, many previous works on schema mapping [25], [26] can be referred for implementing this automatic labeling approach. To improve precision, it is also worth considering making this approach semi-automatic, i.e., let the user guide the labeling process but with minimal extra effort.

8 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 the security community to check whether programs leak sensitive information [17], [18]. This technique has been widely used in the security community, ranging from certificating secure programs [17], preventing security attacks [27], and detecting privacy leakages in mobile applications [28]. Compared with these works, we use information flow analysis for a different purpose, i.e., checking privacy compliance of SQL queries.

**Access Control Language.** In the past decades, many policy specification languages have been proposed to formalize text-based privacy and security policies. P3P [29] is a privacy language for formalizing website privacy policies with the focus on how personal information is collected, shared and retained. EPAL [17] targets at formalizing and enforcing privacy policies within enterprises. XACML [8] is an OASIS standard access control language, which supports user-defined attributes and various policy/rule combining algorithms for different scenarios. Some of the previous works have motivated the design of PSpec, but PSpec differs from them in several aspects. First, PSpec is designed as a specific privacy language suitable for data analytics systems. Second, most of the previous works consider data elements independent of each other, which does not hold in data analytics. PSpec provides explicitly support for associated data access to handle this.

**Database Access Control.** Another category of the related works is database access control, as our work naturally applies to traditional database systems. The sequence of works [9], [30], [10] proposed the idea of Hippocratic database. Query auditing techniques [31], [32] audit submitted queries to detect whether sensitive information is leaked. The works [11], [12] enforce purpose-based policies within database systems. Colombo and Ferrari [13] further consider action-aware policies, i.e., actions performed by queries on certain categories of data. DataLawyer [33]
enforces general data use policies encoded into SQL-like queries. Recent works [24, 25] further enforce purpose-based policy within NoSQL data stores, e.g., MongoDB. However, we differ from these works in that we provide a high-level specification language to hide the details of data models, while these works directly operate on the concrete data models, e.g., relational data or NoSQL data. Lacking the high-level specification language may cause barriers for legal experts, who often have no background in database systems.

**Privacy-Preserving Systems.** The last line of the related works falls into privacy-preserving systems. PINQ [56] is a privacy-preserving data analysis platform built on top of LINQ and differential privacy. Airaval [57] extends MapReduce [58] and integrates mandatory access control and differential privacy to provide security and privacy guarantees. GUPT [59] enforces differential privacy for arbitrary computations with the sample and aggregate framework [40]. The major difference is that our system aims at enforcing privacy policies by specifying data usage restrictions, while these systems mainly rely on differential privacy. As we have already discussed in Section 1, data usage specification is (at least currently) more practical than differential privacy for business data sharing.

Recently, Sen et al. [14] presented a system for auditing privacy compliance for big data systems within organizations. The proposed system is composed of two modules: LEGALEASE for specifying privacy policies and GUROK serving as a data inventory. Our work differs from [14] in that we mainly target at checking SQL queries in an online manner and provide comprehensive support for relational data. The system in [14] is designed for handling general privacy-related data use policies, i.e., store, use, and share, and can only operate offline.

### 9 Conclusion

In this paper, we present PSpec-SQL, a distributed data analytics system that automatically enforces privacy compliance for SQL queries. Our system offers a high-level specification language PSpec for data owners to specify their data usage policies, and provides comprehensive support for labeling relational data. Finally, our system checks submitted queries at runtime for compliance to ensure only policy-compliant queries are executed.

### 10 Acknowledgment

This work was supported in part by the National Natural Science Foundation of China under Grant No. 61672310 and Grant No. 61527812, the National Science and Technology Major Project under Grant No. 2016ZX01038101.

### References


Chen Luo received the B.S. degree from Tongji University in 2013, and the M.S. from Tsinghua University in 2016. He is currently a PhD student in University of California Irvine, US. His research interests include formal methods and database systems.

Fei He received the B.S. degree in computer science and technology from National University of Defense Technology in 2002, and the Ph.D. degree in computer science and technology from Tsinghua University in 2008. He is currently an Associate Professor in the School of Software at Tsinghua University, Beijing, China. His research interests include formal verification and program analysis.

Chen Luo received the B.S. degree from Tongji University in 2013, and the M.S. from Tsinghua University in 2016. He is currently a PhD student in University of California Irvine, US. His research interests include formal methods and database systems.

Fei He received the B.S. degree in computer science and technology from National University of Defense Technology in 2002, and the Ph.D. degree in computer science and technology from Tsinghua University in 2008. He is currently an Associate Professor in the School of Software at Tsinghua University, Beijing, China. His research interests include formal verification and program analysis.

Fei Peng received the B.S. degree from Tsinghua University in 2016. He is currently a Master student in Columbia University. His research interests include formal methods and program verification.

Dong Yan received the Ph.D. degree in computer science and technology from Tsinghua University in 2015. He worked at Intel Labs China as a data analysis platform researcher from 2015 to 2016. He is currently a Postdoc in Tsinghua University. His research interests include distributed system, big data and large-scale machine learning.

Dan Zhang received the Ph.D. degree in computer science and technology from Xi’ an Jiaotong University in 2006. He worked at Intel Labs China as a research scientist from 2006 to 2016. He is currently Architect in UISEE corporation. His research interests include autonomous driving, big data, data privacy preservation, robotics, compiler, etc.

Fei He received the B.S. degree in computer science and technology from Tsinghua University in 2002, and the Ph.D. degree in computer science and technology from Tsinghua University in 2008. He is currently an Associate Professor in the School of Software at Tsinghua University, Beijing, China. His research interests include formal verification and program analysis.

Xin Zhou received the Master degree in Computer Science from Fudan University in 2002. He worked at Intel Lab China as a senior researcher until 2016. He is currently the product general manager of UISEE corporation. His research interests include machine learning, big data and privacy.