A Privacy Preserving Repository for Data Integration across Data Sharing Services

Stephen S. Yau, Fellow, IEEE, and Yin Yin

Abstract—Current data sharing and integration among various organizations require a central and trusted authority to collect data from all data sources and then integrate the collected data. This process tends to complicate the update of data and to compromise data sources’ privacy. In this paper, a repository for integrating data from various data sharing services without central authorities is presented. With our repository, data sharing services can update and control the access and limit the usage of their shared data, instead of submitting data to authorities, and, hence, our repository will promote data sharing and integration. The major differences between our repository and existing central authorities are: 1) Our repository collects data from data sharing services based on users’ integration requirements rather than all the data from the data sharing services as existing central authorities. 2) While existing central authorities have full control of the collected data, the capability of our repository is restricted to computing the integration results required by users and cannot get other information about the data or use it for other purposes. 3) The data collected by our repository cannot be used to generate other results except that of the specified data integration request, and, hence, the compromise of our repository can only reveal the results of the specified data integration request, while the compromise of central authorities will reveal all data.

Index Terms—Privacy concerns of service-oriented solutions, privacy management in data collection, services composition.

1 INTRODUCTION

MUCH effort has been devoted to facilitating data sharing and integration among various organizations. However, the development of such systems is hindered by the lack of robust and flexible techniques to protect the privacy of the shared data. Existing data sharing and integration systems are usually implemented as centralized data warehouses collecting and storing data from various data sources. Typically, data sources and data warehouses expect to sign business agreements in which the scope of the shared data and corresponding privacy policies are specified. For example, all shared data will be kept confidential and will not be disclosed to other unrelated third parties or be used for other purposes. While this solution works well for a single organization or a federation of organizations, where trust relations have been well established, serious problems will arise when some data warehouses cannot be trusted by data sources. In such cases, data sources will refuse to share their data because they have no control of its usages and disclosures once the data is shared. In fact, data warehouses indeed can reveal or abuse the shared data. Furthermore, even if data warehouses adhere to the agreement, there is no guarantee that they have sufficient capability to protect the data.

The most significant problem of existing data sharing and integration solutions is that they give data warehouses too much power, which may not be needed for data sharing. For instance, a hospital may be asked to share its patients’ social security numbers (SSNs) because they are used to locate patients’ records from various hospitals. Unfortunately, SSNs can also be used for other purposes, such as checking patients’ credit histories. But, when SSNs are only used as keys to link records from various hospitals, the SSNs can be replaced by their hash values without affecting their functionality as keys.

This example suggests that it is more convenient and secure to share and integrate data by developing a data sharing service for each data source to share data and a repository to collect data from data sharing services, where data sharing services control their own data and only share data according to integration requirements, instead of sharing all data to the repository. Unlike existing business process languages, such as WS-BPEL [1], which focus on the protection of the access to services and the integrity and confidentiality of service messages [19], we assume that our repository can access all shared data, which is well protected. The security requirement of data sharing is to ensure that data sharing services share only the information of the data needed by the repository to satisfy users’ specific integration requirements, and the repository cannot use the shared information to generate other results except those required by users. In this paper, the attributes of data and how it will be integrated are considered as the context of the data. With our query plan language, a data sharing service for data integration, which is represented by a node in the query plan graph, has a context consisting of only its adjacent nodes and edges in the query plan graph.

In this paper, we will present a privacy preserving repository to accept integration requirements from users, help data sharing services share data and safeguard their privacy, collect and integrate the required data from data sharing services, and return the integration results to users. Our repository will focus on the matching operations and has the following major benefits:

1. The data sharing services can update and control the access and usages of their shared data. That is, data-
sharing services can update their data whenever necessary and determine who and how their shared data can be used.

2. The data is shared based on the need-to-share principle, which means that the released information of the data is sufficient to support users’ integration requirements, but contains no more information of the data.

3. The repository’s capability is limited to collecting data from data sharing services and integrating the data to satisfy users’ integration requirements. Except the information needed to be revealed for data integration, the repository will not have extra information about the data and cannot use it for other purposes.

2 A Motivating Example

Let us consider a healthcare information system which collaborates with multiple organizations through sharing and integrating data. The organizations may include medical research institutes, DNA databases, hospitals, and pharmacies for the purpose of studying the reactions of popular heart medicines sold in pharmacies. For the sake of simplicity, we assume that the system only communicates with one medical research database $T_1(Disease, Pattern)$ storing diseases and corresponding DNA patterns, a DNA database $T_2(SSN, Pattern)$ storing personal DNA patterns, a hospital database $T_3(SSN, Medicine, Reaction)$ storing all patients’ diagnosis histories, and a pharmacy database $T_4(Disease, Drug)$ storing popular drugs for each disease. The databases’ schemas and data are listed in Table 1.

This example may be expressed in terms of four SQL queries shown in Table 2, where Q1, Q2, and Q3 generate three temporary tables, $Tmp_1, Tmp_2,$ and $Tmp_3,$ respectively, and the last query, Q4, outputs the final results. With the existing central warehouse solution, all data shown in Table 1 is collected by a central authority which can execute all queries. However, our repository is allowed to collect only the needed information about data for integration. On the other hand, because the repository needs some extra information to execute queries, such as $Q_1$’s result, which is needed by $Q_2$ as an input, our repository will randomize $Q_1$’s result and make the randomized result still usable for $Q_2$. Although existing privacy-preserving query processing approaches, such as [6], [7], [10], [16], [22], [25], [28], can evaluate a query on randomized data, none of them can handle a series of queries, where some queries need other queries’ results as inputs, such as $Q_2$ in this motivating example. To protect $Q_1$’s result \{ \{p_1, p_2\} \} without disabling $Q_2$, \{ \{p_1, p_2\} \} is replaced by \{ \{H(p_1), H(p_2)\} \}, where $H$ is a hash function. Because the hashed DNA patterns will usually remain unique, the repository can evaluate $Q_2$ by comparing $H(Tmp_1.Pattern)$ and $H(T2.Pattern)$. This simple hash solution can avoid the need for our repository to know $Q_1$’s results, but still keep the mapping relation between nonheart diseases and patients’ SSNs. Since $H(p_3)$ does not appear in the $Q_1$’s hashed result \{ \{H(p_1), H(p_2)\} \}, our repository can find that the patient with $ssn_4$ is not a heart disease patient.

To further protect the privacy of such information, we will develop a Context-Aware Data Sharing algorithm (Algorithm 2, Section 7) to randomize $Q_1$’s result, where the context-awareness implies that when a medical research institute shares its database $Research T_1$ with our repository, it should know that its DNA pattern data will be used to match the DNA pattern data from $T_2$. While the simple hash solution only randomizes the items in $Q_1$’s result (i.e., \{ \{p_1, p_2\} \}), our Context-Aware Data Sharing algorithm randomizes all patterns in $T_1$, but ensures that only $p_1$ and $p_2$ can be used to evaluate $Q_2$. Hence, the mapping between nonheart diseases and SSNs are well protected.

In this paper, we will use the above example to show how our repository for studying the reactions of popular heart medicines sold in pharmacies cannot reveal any additional information about the data of databases $T_1, T_2, T_3,$ and $T_4$.

3 Preliminaries

3.1 System Architecture and Assumptions

In existing data integration systems, it is assumed that there is a central and trusted authority collecting all data from data

<table>
<thead>
<tr>
<th>TABLE 1</th>
<th>The Databases in the Motivating Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Disease</strong></td>
<td><strong>Pattern</strong></td>
</tr>
<tr>
<td>heart</td>
<td>$p_1$</td>
</tr>
<tr>
<td>heart</td>
<td>$p_2$</td>
</tr>
<tr>
<td>cancer</td>
<td>$p_3$</td>
</tr>
<tr>
<td>cancer</td>
<td>$p_4$</td>
</tr>
<tr>
<td><strong>SSN</strong></td>
<td><strong>Pattern</strong></td>
</tr>
<tr>
<td>$ssn_1$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>$ssn_2$</td>
<td>$p_1$</td>
</tr>
<tr>
<td>$ssn_3$</td>
<td>$p_2$</td>
</tr>
<tr>
<td>$ssn_4$</td>
<td>$p_3$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TABLE 2</th>
<th>The Queries Required by the Motivating Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1: $T_1.Pattern$</td>
<td>FROM T1 WHERE $T_1.Pattern = &quot;heart&quot;</td>
</tr>
<tr>
<td>Q2: $T_2.Pattern$</td>
<td>FROM Tmp1, T2 WHERE $Tmp_1.Pattern = T2.Pattern</td>
</tr>
<tr>
<td>Q3: $T_4.Drug$</td>
<td>FROM T4 WHERE $T_4.Disease = &quot;heart&quot;</td>
</tr>
</tbody>
</table>
sharing services and computing integration results for users based on the collected data. Such an assumption is often not valid for data sharing services across various organizations.

In our system, as shown in Fig. 1, our repository collects only the required data for users’ integration requests. We assume that our repository will correctly construct the query plans for users’ integration requirements, decompose query plans, discover and fetch data from distributed data sharing services, integrate all data together, and, finally, return the final results to users. Furthermore, we assume that our repository is granted the access to the shared data by all data sharing services, and all shared data is well protected. Because the data sharing services use our context-aware data sharing algorithm, our repository cannot learn extra information from the inferential relations of the information it obtains during the integration process.

Our repository consists of two components: the query plan wrapper and the query plan executor. The query plan wrapper is responsible for analyzing integration requirements and constructing query plans for the query plan executor. Since the wrapper development and optimization have been extensively studied [5], [8], [18], [19], [21], [30], we assume that the query plan wrapper can select data sharing services or its precedent queries’ outputs as inputs. The above assumption, we will focus on how to decompose the query plan graph into a set of small subgraphs for each data sharing service to guide data sharing services to prepare shared data.

The query plan executor is responsible for executing query plans to fetch data from data sharing services and producing the final results. In this paper, we will develop a secure query plan executor which can execute query plans without additional information about the data of data sharing services.

3.2 Privacy Preserving Query Plan with Repository

To formulate the privacy preserving data integration across data sharing services, we need to define the query plan:

**Definition 1 (Query Plan).** A query plan $P$ is a partially ordered set of queries $\{p_1, p_2, \ldots, p_m\}$ with two properties:

- Each $p_i$ can be evaluated only after all of its precedent queries have been evaluated.
- Each $p_i$ can use the data directly from data sharing services or its precedent queries’ outputs as inputs.

The final result of $P$ is the outputs of $p_i$ with no successive queries, and all other queries’ outputs are intermediate results.

The above definition indicates that a query plan $P$ has a much richer structure than a single query or a set of independent queries. First, there is a partial order relation among queries in $P$. Second, only the outputs of queries in $P$ without successive queries constitute the final result and all other intermediate results should be protected. Consequently, we have the following definition:

**Definition 2 (Privacy Preserving Repository).** For a query plan $P = \{p_1, p_2, \ldots, p_m\}$ and a repository $REP$, $REP$ is a privacy preserving repository for data integration if $REP$ executes $P$ in a privacy preserving manner as follows: 1) $REP$ only has $P$’s final result encrypted with user’s public key and has no information on $P$’s intermediate results; and 2) $REP$ cannot use the data shared for $P$ to evaluate any other queries.

4 OVERVIEW OF OUR APPROACH

As discussed in Section 1, our goal is to develop a repository to facilitate the data integration across data sharing services. In this section, we will present the process of the data integration via our privacy preserving repository $REP$. The process can be summarized as follows:

- **Step 1.** The user sends his/her public key $pk$ and the requirements about data integration to our repository $REP$.

- **Step 2.** The query plan wrapper of $REP$ analyzes the user’s integration requirements and converts them to a query plan graph $G$, and then decomposes $G$ to a set of subgraphs $\{G_1, G_2, \ldots, G_m\}$ using the Decompose Algorithm (Algorithm 1, Section 6) and sends the subgraphs to the query plan executor. Every subgraph $G_i$ represents the context of one data sharing service for conducting context-aware data sharing.

- **Step 3.** For every $G_i$, the query plan executor looks for the corresponding data sharing service $S_i$ and sends $G_i$ to $S_i$, which prepares the data using the Context-Aware Data Sharing Algorithm (Algorithm 2, Section 7) and returns all randomized data to the query plan executor.

- **Step 4.** The query plan executor executes the Integrate Algorithm (Algorithm 3, Section 8) on all returned data to execute the $G$ and outputs the results $FinalRes$ of user’s request, which is encrypted with the user’s public key $pk$.

- **Step 5.** $REP$ sends $FinalRes$ to the user who then decrypts it with his/her secret key $sk$.

5 A QUERY PLAN LANGUAGE

In this section, we will present an XML-like language, called QPSL, representing the data integration process as a query plan graph $G$. This language will help the repository figure out what data should be retrieved from data sharing services.
services and how to integrate the data together, and will help data sharing services share their data without revealing more information than the evaluation of G needs.

5.1 The Query Plan Graph

For a data integration requirement involving n data sharing services \( S_1, S_2, \ldots, S_n \), the query plan graph \( G = \{V, E, C\} \) is a labeled directed acyclic graph. \( V = \{v_1, v_2, \ldots, v_m, s, t\} \) is a set of nodes with each \( v_i \) representing a data sharing service, \( s \) representing the source node for collecting the inputs from users, and \( t \) representing the sink node for receiving all final result of the query plan. \( E = \{e_1, e_2, \ldots, e_l\} \) is a set of edges, and each edge \( e_{ij} = (v_i, v_j) \) represents a data integration relation between data sharing services \( v_i \) and \( v_j \). Finally, \( C = \{c_1, c_2, \ldots, c_l\} \) is a set of labels attached with each \( e_{ij} \in E \), and each label \( c_{ij} = (op, attr1, attr2) \in C \) specifies that the data from the data sharing service \( v_i \) and \( v_j \) is integrated by the data integration operator \( op \) between \( v_i \)'s attribute \( attr1 \) and \( v_j \)'s attribute \( attr2 \). Generally, the operator \( op \) can be any binary comparison operator chosen from \( \{=, \neq, >, <\} \) or any aggregate operator chosen from \( \{SUM, AVG, MAX, MIN\} \). However, in this paper, we focus on the operator = and discuss \( SUM \) and \( AVG \) in Section 11.

The motivating example in Section 2 can be represented by the query plan graph shown in Fig. 2, where each edge represents a query in Table 2. The edge \( (s, T1) \) represents the query \( Q1 \), \( (T1, T2) \) represents the query \( Q2 \), and \( (T2, T3) \) and \( (T4, T3) \) together represent the query \( Q4 \). The sink node \( t \)'s in-edge \( (T3, t) \) does not represent any query, but the final result of the data integration.

Besides representing queries, the query plan graph shown in Fig. 2 also represents queries’ partial order relation defined in Definition 1 by edges’ direction.

5.2 QPSL Schema

In this section, we will present a Query Plan Specification Language (QPSL) to represent a query plan graph as a XML document. With the DTD schema [1] depicted in Fig. 3, QPSL will represent \( G \) as a set of edge and node elements.

Each edge element has three attributes \( id, head, \) and \( tail \), where \( id \) is the edge’s unique identity, \( head \) is the edge’s head node, and \( tail \) is the edge’s tail node. In addition, each edge element also has two subelements \( r \) and \( condition \), where \( r \) is a random number used by data sharing services and our repository for privacy preserving data integration, and \( condition \) represents the label attached with the edge.

Each node element has an attribute \( id \) representing the data sharing service’s unique identity and an element \( name \) representing the service’s name.

6 QUERY PLAN DECOMPOSITION

Because each data sharing service only needs to know its related data integration operations, but the query plan graph \( G \) contains the information about all data sharing services, the query plan wrapper should decompose \( G \) and send only the query plan subgraphs to their corresponding data sharing services. It reduces the system communication overhead as well as every data sharing service’s computation overhead on parsing the query plan.

From a given query plan graph \( G = (V, E, C) \) with \( m \) nodes, the Decompose Algorithm (Algorithm 1) will construct a subgraph \( G_i \) for each node \( v_i \) by extracting \( v_i \)'s adjacent nodes and corresponding edges and the labels attached to these edges from \( G \). Furthermore, Algorithm 1 assigns a random number to each edge. Although each edge will appear in the subgraphs for both its head and tail nodes, Algorithm 1 assigns the same random number to the edge in both subgraphs it appeared.

**Algorithm 1: Decompose Algorithm.**

Input: The query plan graph \( G = (V, E, C) \)
Output: A set of subgraphs \( G_i = (V_i, E_i, C_i, \bar{r}_i) \), \( (1 \leq i \leq |V|) \)

1. foreach edge \((v_j, v_i)\) in the graph \( G \) do
2. assign a random number \( r_{j,i} \) to \((v_j, v_i)\);
3. end
4. for \( i = 1 \) to \( |V| \) do
5. initialize \( V_i \rightarrow \{v_i\} \), \( E_i, C_i, \bar{r}_i \rightarrow \emptyset \);
6. for \( j = 1 \) to \( |V| \) do
7. if \((v_j, v_i) \in E \) then
8. \( V_i \leftarrow V_i \cup \{v_j\} \);
9. \( E_i \leftarrow E_i \cup \{(v_j, v_i)\} \);
10. \( C_i \leftarrow C_i \cup \{(v_j, v_i)\}'s \) label;
11. \( \bar{r}_i \leftarrow \bar{r}_i \cup \{r_{j,i}\} \);
12. end
13. end
14. end
15. \( G_i \leftarrow (V_i, E_i, C_i, \bar{r}_i) \);
16. end
17. return \( G_1, G_2, \ldots, G_{|V|} \);
We denote the subgraph $G_i$ of $v_i$ as $(V_i, E_i, C_i, r_i)$, where $V_i$ consists of all $v_i$’s adjacent nodes, $E_i$ all the adjacent edges, $C_i$ all the labels attached with $E_i$, and $r_i$ contains all random numbers assigned to $E_i$. Hence, $G_i$ represents all data integration operations of the data sharing service represented by $v_i$.

7 Context-Aware Data Sharing

In Algorithm 1, the query plan graph $G$ is decomposed to a set of subgraphs. For each data sharing service, its subgraph $G_i$ consists of the information about other data sharing services whose data will be integrated with its own data and how the data will be integrated together. Hence, we call the subgraph of $v_i$ the context of the data sharing service of $v_i$ in current $G$. Because data sharing services are aware of its context in the whole data integration process, they can determine which information should be shared and how to limit the usage of the shared data.

In this section, we will present a Context-Aware Data Sharing Algorithm to help data sharing services share information with the repository. We will focus on the matching operations to determine whether two records are matched according to the equality test between their attribute values.

Basically, the matching between two data records can always be replaced by the matching between their hash values. Hash functions’ low conflict probability ensures the correctness of the hash-based matching and hash functions’ one-way property enables a third party to match two data records without revealing their values. Thus, the hash function is a simple solution for privacy preserving data matching. However, with two records’ hash values, the two records can always be matched by anyone, which makes the hash-based matching inappropriate in certain cases, such as the privacy preserving data storage application proposed in [11], where only the authorities can satisfy both Requirement 1 and Requirement 2.

Our Context-Aware Data Sharing Algorithm is given in Algorithm 2, and the shared data of our motivating example is listed in Table 3 as an example of the algorithm’s output.

Algorithm 2: Context-aware Data Sharing Algorithm

```
Input: Service $S_i$, a subgraph $G_i = (V_i, E_i, C_i, r_i)$, two hash functions $H_1, H_2$, and the encryption algorithm $E$ with the public key $pk$;
Output: A set of randomized data $RD_i$
1 Initialize $I_i, O_i, AttrIn_i ← ∅$
2 foreach node $v_j$ satisfying $(v_j, v_i) ∈ E_i$ do
   3 $(op, first, second) ← (v_j, v_i)$’s label
   4 $(v_j, v_i)$’s random number $r_{j,i} ← r_i$
   5 foreach data record rec of the service $S_i$ do
      6 $I_{j,i} ← I_{j,i} ∪ H_1(r_{j,i}, rec.second)$
      7 $I_i ← I_i ∪ I_{j,i}$
      8 $AttrIn_i ← AttrIn_i ∪ second$
9 end
10 foreach node $v_j$ satisfying $(v_i, v_j) ∈ E_i$ do
11  $(op, first, second) ← (v_i, v_j)$’s label
12  $(v_i, v_j)$’s random number $r_{i,j} ← r_i$
13 foreach data record rec of the database $D_i$ do
14   $R ← 0$
15   foreach attribute $a$ ∈ AttrIn_i do
16      $R ← R ⊕ H_2(r_{i,j}, rec.a)$
17   if $v_j$ is the sink node then
18      $O_{i,j,1} ← ∅$
19      $O_{i,j,2} ← E_{pk}(rec.first) ⊕ R$
20   end
21   else
22      $O_{i,j,1} ← H_1(r_{i,j}, rec.first) ⊕ R$
23      $O_{i,j,2} ← H_2(r_{i,j}, rec.first) ⊕ R$
24   end
25   $O_{i,j} ← O_{i,j} ∪ (O_{i,j,1}, O_{i,j,2})$
26 end
27 $O_i ← O_i ∪ O_{i,j}$
28 return $RD_i ← I_i ∪ O_i$
```

In the following sections, we will show that Algorithm 2 satisfies both Requirement 1 and Requirement 2.

7.1 Requirement 1

If the user’s integration request requires matching between two services’ data, there should be an edge between these two services in the corresponding $G$. Hence, to show that Algorithm 2 satisfies Requirement 1, we only need to prove that the information shared by each edge’s head node and tail node can only be used by our repository to match their own data according to the edge label.

To show how Algorithm 2 satisfies Requirement 1, consider the simple query plan subgraph for node $v$ shown in Fig. 4, where $v$ shares the information about rec to our repository to match rec with the data of $v_1$ and $v_2$, according to $v$’s in-edges and out-edges. Specifically, $v$ computes $I_1 = H_1(r_{1,v_1})$ and $I_2 = H_1(r_{2,v_1})$ for the two in-edges and $O_1 = H_1(r_{v_1}, v_{1,2})$ for the out-edge, where $r_{1,v_1}$ and $r_{2,v_1}$ are three random numbers assigned to edges $(i_1, v)$, $(i_2, v)$, and $(v_1, o)$ in Algorithm 1. Similarly, $i_1$ and $i_2$ will...
share information according to their out-edges, and $o_1$ will share information according to its in-edge. All shared information is depicted in Fig. 4. Because Algorithm 1 assigns random numbers $r_1$, $r_2$, and $r_3$ independently and only the repository knows them, only the repository can use the shared information to match $v'$s data with $i_1$, $i_2$, and $o_1$'s data. Furthermore, because each edge's head node and tail node use the same random number to share information, our repository can only use $I_1$ and $I_2$ to match $i_1$ and $i_2$'s data and use $O_1$ to match $o_1$'s data.

The above results can be stated in the following theorem, and the proof is given in the Appendix:

**Theorem 1.** Let $rec_1$ and $rec_2$ be $v_1$ and $v_2$'s records, respectively. Let $r$ be the random number assigned to edge $(v_1, v_2)$. Our repository matching $rec_1$'s attribute $attr_1$ with $rec_2$'s attribute $attr_2$ using Algorithm 2 has the following properties:

![Fig. 4. A subgraph generated by Algorithm 1.](image-url)
1. **Correctness.** If $\text{rec}_1, \text{attr}_1$ matches $\text{rec}_2, \text{attr}_2$, their hash values $H_1(r, \text{rec}_1, \text{attr}_1)$ and $H_1(r, \text{rec}_2, \text{attr}_2)$ also match.

2. **Robustness.** If the hash values $H_1(r, \text{rec}_1, \text{attr}_1)$ matches $H_1(r, \text{rec}_2, \text{attr}_2)$, $\text{rec}_1, \text{attr}_1$ and $\text{rec}_2, \text{attr}_2$ will match with large probability.

3. **Independence.** The hash values $H_1(r, \text{rec}_1, \text{attr}_1)$ and $H_1(r, \text{rec}_2, \text{attr}_2)$ are independent from the information shared for other edges, and, hence, they can only be used for the matching between $\text{rec}_1, \text{attr}_1$ and $\text{rec}_2, \text{attr}_2$.

### 7.2 Requirement 2

For a record $\text{rec}$ of $v$, $\text{rec}$ is said to pass the evaluation of $v$’s in-edges if and only if there is a record $\text{rec'} \in i$ matching $\text{rec}$ successfully for any $v$’s in-edge $(i, v)$. Hence, to show that Algorithm 2 satisfies Requirement 2, we only need to prove that, for an edge $(v_i, v_j)$, the information shared by $v_i$ about its record $\text{rec}$ can be used by our repository to match $v_j$’s data only when $\text{rec}$ passes the evaluation of $v_i$’s in-edges.

We still use the subgraph shown in Fig. 4 to show how Algorithm 2 satisfies Requirement 2. First, $v$ collects all of its attributes specified in the labels of in-edges as $\text{AttrIn} = \{\text{attr}_1, \text{attr}_2\}$ and then computes the random factor $R$ as $H_2(r_1, v_{i1}) \oplus H_2(r_2, v_{i2})$ for record $\text{rec}$, where $H_2(r_1, v_{i1})$ is shared by $v_i$ for $\text{rec}_1$ and $H_2(r_2, v_{i2})$ is shared by $v_j$ for $\text{rec}_2$. Then, $v$ randomizes its shared information for the out-edge $(v, a)$ with the random factor $R$ as $O_1 = \{H_1(r_3, v_{i3}) \oplus R, H_2(r_3, v_{i3}) \oplus R\}$ if $v_i$ is not the sink node $t$; otherwise, $O_1 = \{E_{ab}(r_3, v_{i3}) \oplus R\}$. Hence, the repository has to first remove the random factor $R$ from $O_1$ before it can use the information $O_1$ to evaluate $v$’s out-edge, which in turn requires that $\text{rec}$ pass the evaluation of both $(v_i, v)$ and $(v_j, v)$. The above results can be stated in the following theorem, and the proof is given in the Appendix:

**Theorem 2.** Let $(v_1, v_2)$ match $v_1$’s attribute $\text{attr}_1$ with $v_2$’s attribute $\text{attr}_2$. Let $\text{rec}_1$ be a record of $v_1$. With the random factor $R_1$ computed by Algorithm 2 and the random number $r$ assigned for $(v_1, v_2)$ by Algorithm 1, $v_1$ shares $(O_1, O_2)$ as the information about $\text{rec}_1$, where $O_1 = H_1(r, \text{rec}_1, \text{attr}_1) \oplus R_1$ and $O_2 = H_2(r, \text{rec}_1, \text{attr}_1) \oplus R_1$. Our repository can remove the random factor $R_1$ from $(O_1, O_2)$ if and only if the following two conditions are satisfied:

- **Cond. 1.** All $(v_1, v_2)$’s precedent edges have been evaluated.
- **Cond. 2.** The record $\text{rec}_1$ should pass the evaluation of $v_1$’s in-edges.

### 8 Data Integration

When our repository receives the shared information from all data sharing services, the repository should follow the query plan graph $G$ and integrate the received information together to compute the integration results for the user. In this section, we will present the integration process as the Integration Algorithm (Algorithm 3).

**Algorithm 3: Integrate Algorithm.**

**Input:** The query plan graph $G = (V, E, C)$ and the information shared by services \{RD$_9$, RD$_1$, RD$_2$, $\cdots$, RD$_n$\};

**Output:** The final results $\text{FinalRes}$

1. Initialize $\text{FinalRes}$, $\text{Walked} \leftarrow \emptyset$;
2. UnWalked $\leftarrow E$
3. **foreach** node $v_i$ satisfying $(s, v_i) \in E$ **do**
   4. $\text{attr}_1, \text{attr}_2 \leftarrow$ the attributes in the label of $(s, v_i)$;
   5. $\text{attrOut} \leftarrow v_i$’s attributes specified in its out-edges;
   6. $\text{Match(RD}_i, \text{RD}_1, \text{attr}_1, \text{attr}_2, \text{attrOut})$;
   7. $\text{UnWalked} \leftarrow \text{UnWalked}/(s, v_i)$;
   8. $\text{Walked} \leftarrow \text{Walked} \cup (s, v_i)$;
9. **while** UnWalked is not empty **do**
   10. $(v_i, v_j) \leftarrow R \text{UnWalked};$
   11. if all $v_i$’s in-edges $\in \text{Walked}$ **then**
      12. $\text{attr}_1, \text{attr}_2 \leftarrow$ the attributes in the label of $(v_i, v_j)$;
      13. $\text{attrOut} \leftarrow v_j$’s attributes specified in its out-edges;
      14. $\text{Match(RD}_i, \text{RD}_j, \text{attr}_1, \text{attr}_2, \text{attrOut})$;
      15. $\text{UnWalked} \leftarrow \text{UnWalked}/(v_i, v_j)$;
      16. $\text{Walked} \leftarrow \text{Walked} \cup (v_i, v_j)$;
18. **end**

Intuitively, this algorithm starts from the source node $s$ and navigates all edges to match the information shared by each edge’s head and tail nodes in the partial order specified by $G$. The algorithm will arrive at the sink node $t$ and output the final result of the whole query plan. The most important part of this algorithm is the $\text{Match}$ function, which is explained here with the matching of $\text{RD}_1$ and $\text{RD}_2$ in Table 3 as an example:

- **Initialize.** Before the repository REP evaluates an edge, it first retrieves the edge’s label information from $G$ to find out which attributes are to be matched. Meanwhile, REP collects all attributes shared by the edge tail node for its own out-edges as $\text{AttrOut}$. In our example, the edge is to match $\text{RD}_2, \text{pattern}$ with $\text{RD}_1, \text{pattern}$, and $\text{RD}_2$ has only one attribute in its out-edge. Hence, we have $\text{AttrOut} = \text{RD}_2, \text{ssn}$.
- **Match.** In this step, REP scans and matches tail nodes’ records with the records from head nodes. In our example, REP matches $\text{RD}_2$’s records with $\text{RD}_1$’s records according to their attribute pattern. Let $\text{RD}_1$’s records be $\text{rec}_1$ and $\text{RD}_2$’s records be $\text{rec}_2$, where $1 \leq i \leq 4$. Recall that $\text{RD}_2$’s record $\text{rec}_2$ passes the evaluation of the edge from $\text{RD}_1$ to $\text{RD}_2$ only if there is a record $\text{rec}_j \in \text{RD}_1$ that $\text{rec}_j, \text{pattern} = \text{rec}_2, \text{pattern}$. According to Table 3, the first three records of $\text{RD}_2$ pass the evaluation, and the last one fails.
- **Remove random factors.** Assume that the edge’s tail node’s record $\text{rec}$ passes the evaluation of the edge. To use $\text{rec}$ to evaluate the tail node’s out-edges, $\text{REP}$
first needs to remove random factors from the shared information about rec for all attr ∈ AttrOut. In our example, only the first three records of RD2 passed the evaluation and there is only one attribute ssn ∈ AttrOut. Therefore, the random factors can only be removed from the first three records according to Theorem 1. The last record’s random factor cannot be removed and be further used for consequent matching.

- **Collect outputs.** If the edge’s tail node is the sink node t, REP collects the outputs of the whole query plan in this step. For example, if RD2 is the sink node t, REP will output the first three records’ encrypted ssn into FinalRes.

9. PERFORMANCE EVALUATION

In this section, we analyze the performance of our algorithms and conduct extensive experimental evaluation. To evaluate the performance of our algorithms, we construct different size data sets from the real adult income database (available at http://archive.ics.uci.edu/ml/datasets/Adult), which contains a table with roughly 100,000 records.

We conducted our experiments on a 3GHz Pentium 4 Processor running Windows XP with 2GB RAM. We chose C++ to implement the experiments in Microsoft Visual Studio 2005 with the Xerces 2.8.0 library (available at http://xerces.apache.org) for the creating and parsing of XML documents and the Crypto++ 5.5.2 library (available at http://www.cryptopp.com) for the implementation of hash functions and the public encryption scheme [13]. We also used MySQL to store data and handle data queries.

9.1 Context-Aware Data Sharing

As discussed in Section 7, the context-aware data sharing limits the usage of shared data within the specified context, i.e., the data sharing service node’s in-edges and out-edges, and only reveals the information essential for the evaluation of these edges. Suppose a data sharing service has n1 in-edges, n2 out-edges for all its subsequent operations, and n3 out-edges directed to the sink node t for the final result. Assume that the times for a hashing operation and an encryption operation are t_h and t_e, respectively. According to Algorithm 2, the time for context-aware data sharing of this data sharing service is O((n1 + n2)t_h N) + O(n3t_e N), where N is the number of records shared by the data service.

Fig. 6 shows our experimental result, where the data sharing service has one in-edge from the source node s and one out-edge for the final result, and N ranges from 100 to 30,000. In this context, the expected time is O(t_h N + t_e N), which is proportional to N. This expected time is verified by the experimental result. When N is 30,000, the data sharing service can complete data sharing within about 300 seconds. Note that only data sharing services which have out-edges to the sink node t need to encrypt their data. Most data sharing services, whose data are only used for further...
integration, only need hashing operations, which are much faster than encryption operations.

9.2 Integration

The data integration is executed based on the edges in $G$. For an edge with the head node $v_1$ and the tail node $v_2$, the integration based on that edge will match $v_2$’s records with $v_1$’s records. Suppose $v_1$ has $N_1$ records and $v_2$ has $N_2$ records in which $p$ percent will match $v_1$’s records successfully. According to Algorithm 3 and the discussion in Section 8, the integration time for the edge $(v_1, v_2)$ is $O(N_1 N_2/p)$.

Fig. 7 shows our experimental result, where the data service has an in-edge from the source node $s$ and an out-edge for the final result, and $N$ ranges from 100 to 30,000. When $N$ is 30,000, REP can complete data integration within around 600 seconds with selectivity 0.3. When the selectivity drops to 0.03, the time reduces to around 50 seconds.

9.3 Decryption

The estimation for the time for the decryption is relatively easy. It depends on the number of records in the final result and the decryption time of the encryption scheme. Suppose the final result contains $N$ records and the time for one decryption operation is $t_d$. The whole decryption time will be $O(N t_d)$.

In our experiment, although we chose different attributes with various selectivities for data integration, Fig. 8 shows that the time is proportional to the data size. For instance, with the attribute $HS$-grad and a data set with 30,000 records, there are around 9,000 records in the final result which can be decrypted by users within 600 seconds.

10 RELATED WORK

10.1 Searchable Encryption Schemes

In [11], [27], a symmetric searchable encryption scheme and an asymmetric searchable encryption scheme are proposed to store users’ data in a third party. These schemes conceal users’ data from the third party and enable the third party to match data with users’ searching requests and return the matched data to users. To satisfy these two seemingly contradictory requirements, both [11], [27] introduce additional private information (i.e., a symmetric key in [27] and an asymmetric key in [11]) to manipulate the original data or its hash values. Following [11], [27], many improved approaches have been proposed [1], [9], [12], [14], [26]. However, all of these approaches only focus on how to control the third party’s search capability between two parties. Because some integration applications in our repository require data from more than two data sharing services, our repository may need to integrate multiple data sets provided by various data sharing services.

10.2 Privacy Preserving Query Processing

Much research has been done on the design of efficient privacy preserving query processing techniques [6], [7], [10], [15], [16], [22], [25], [28]. The basic idea of these approaches is to execute queries on cryptographically or noncryptography manipulated data. Although the assumptions and goals of these approaches vary greatly, all of them suffer from two shortcomings: 1) Existing techniques only include the evaluation for one query and do not consider the role of the query’s output in the whole application. 2) They also do not consider the inferential relations among different queries in one application. These shortcomings make them unsuitable to be used in a complex data integration application that needs to process a set of queries in a given partial order.

10.3 Secure Multiparty Computation

Besides existing privacy preserving query processing techniques, a technique named secure multiparty computation [17], [24] can handle any data integration requirements. Generally speaking, any data integration application can be modeled as a multiparty function that accepts inputs from data sharing services and only releases the final result to the user. However, it needs to represent functions as garbled circuits, which typically require huge numbers of gates and, hence, introduce excessive overhead.

11 CONCLUSION

In this paper, we have presented a privacy preserving repository to integrate data from various data sharing services. In contrast to existing data sharing techniques, our
repository only collects the minimum amount of information from data sharing services based on users’ integration requests, and data sharing services can restrict our repository to use their shared information only for users’ integration requests, but not other purposes.

Although in this paper we have only focused on matching operations, our repository can be easily extended to support SUM and AVG aggregate operations with additive homomorphic encryption schemes, like the Paillier encryption scheme [23]. The experimental results show that our algorithms possess linear complexity and can be completed within reasonable time even when the data set has 30,000 records.

Future research along this topic includes how to extend the expressiveness of our specification language, enable our repository to support more types of data integration operations, and improve of our repository’s performance for much larger scale of data size. A possible approach for performance improvement is to enable the precomputation of data, which allows the data sharing services to obtain some preliminary information about their data for accelerating data sharing.

In this paper, we assume that our repository can access all shared data and focus on how data sharing services share data for specific data integration requests to prevent our repository from using the shared data for other purposes. Future research is needed to investigate the behavior of our repository when there are conflicts among data sharing services’ policies on the shared data. A possible solution to this problem is to use the policy reconciliation technique in [29].

APPENDIX A

PROOF OF THEOREM 1

Proof. The proof of correctness is straightforward because the hash function $H_1$ is deterministic, which always generates the same output for the same input.

The robustness comes from the collision resistance property of hash functions. That is, for a hash function, the probability that two different inputs have the same hash values does not exceed the hash function’s collision probability $p'$, which is usually negligible.

To show the independence, consider a third node $v_3$ with record $rec_3$ and attribute $attr_3$. The repository cannot use $H_1(r, rec_3.attr_1)$ shared by $v_1$ to check whether $rec_3.attr_1 = rec_3.attr_3$ with large probability. Suppose the random number assigned for $v_3$ is $r'$ and the hash function $H_1$’s collision probability is $p'$. Then, if $rec_1.attr_1 \neq rec_3.attr_3$, the probability that their hash values match does not exceed $p'$ because of hash function’s collision-resistance property. In another case, if $rec_1.attr_1 = rec_3.attr_3$, because $r$ and $r'$ are two independent random numbers, there are two possible causes for the event $H_1(r, rec_1.attr_1) = H_2(r', rec_3.attr_3)$. First, the event may occur if $r = r'$ whose probability does not exceed $1/2^{30}$. Second, the event may occur when the hash value collides whose probability does not exceed $p'$. Thus, the overall probability of the event does not exceed $p' + 1/2^{30}$, which is still negligible.

APPENDIX B

PROOF OF THEOREM 2

Proof. First, we prove that the two conditions $Cond1$ and $Cond2$ are necessary for removing $R_1$.

If $Cond1$ is not satisfied, suppose the edge $(i_1, i_2)$ is one of $(v_1, v_2)$’s precedent edges and has not been evaluated. Because $(i_1, i_2) \prec (v_1, v_2)$, there should be a series of edges that satisfies $(i_1, i_2) \prec \cdots \prec (i_n, v_1) \prec (i_j, v_2)$, where $(i_j, i_{j+1})$ is the direct precedent of $(i_{j+1}, i_{j+2})$. We assume that $(i_1, i_2)$ matches $i_1$’s attribute $attr_1$ with $i_2$’s attribute $attr_{i_2}$, and $i_2$’s record $rec_{i_2}$ is randomized by the random factor $R_{i_2} = H_2(r_{i_1}, rec_{i_2}.attr_{i_2}) \oplus R_{i_1}'$, where $r_{i_1}$ is the random number assigned to $(i_1, i_2)$ and $R_{i_1}'$ is the remaining part of the random factor computed according to $i_2$’s other in-edges. Hence, to remove $R_{i_2}$, our repository must compute $H_2(r_{i_1}, rec_{i_2}.attr_{i_2})$ first. However, without evaluating $(i_1, i_2)$, the best information that our repository possesses is $H_2(r_{i_1}, rec_{i_2}.attr_{i_2})$ first. In this case, our repository cannot learn $R_{i_1}$ when its record $rec_{i_1}$ does not reveal any information related to $H_2(r_{i_1}, rec_{i_2}.attr_{i_2})$ even when $rec_{i_1}.attr_{i_1} = rec_{i_2}.attr_{i_2}$, because $i_1$’s random factor $R_{i_1}$ is unknown. Consequently, our repository cannot evaluate $(i_2, i_3)$ without evaluating $(i_1, i_2)$ first because $i_2$’s random factor $R_{i_2}$ is unknown. Recursively, our repository cannot evaluate $(v_1, v_2)$ without evaluating $(i_0, v_1)$ first.

If $Cond1$ is satisfied, but $Cond2$ is not satisfied, i.e., for one in-edge $(v_j, v_i)$ of $v_1$, there is no record $rec_j \in v_j$ satisfying that $rec_j.attr_j = rec_i.attr_i$. In this case, our repository cannot learn $H_2(r_j, rec_i.attr_i)$ from the information shared by $v_j$ and, therefore, cannot learn and remove $v_i$’s random factor $R_i$ from $(O_1, O_2)$, where $r_j$ is the random number assigned for $(v_j, v_i)$. Thus, both $Cond1$ and $Cond2$ are necessary for removing $R_1$.

Now, we will prove that the conditions $Cond1$ and $Cond2$ are sufficient for removing $R_1$ through mathematical induction on the number of $(v_1, v_2)$’s precedent edges.

When the edge $(v_1, v_2)$ has no precedent edges, $Cond1$ and $Cond2$ are obviously satisfied. In this case, according to Algorithm 2, the set $AttrIn$ is empty and the random factor $R_1 = 0$. As a result, our repository can remove $R_1$ from $(O_1, O_2)$ trivially.

Assume that $Cond1$ and $Cond2$ are sufficient for removing $R_1$ when the number of the edge $(v_1, v_2)$’s precedent edges does not exceed $n$.

When $(v_1, v_2)$ has $n$ precedent edges and $n > 1$, we denote $v_i$’s in-edges as $\{(v_j, v_i), 3 \leq j \leq m\}$, where $(v_j, v_i)$ matches $v_i$’s attribute $attr_j$ with $v_j$’s attribute $attr_j'$, first, when $Cond1$ is satisfied, for each edge $(v_j, v_i)$, its precedent edges should have been evaluated. Second, all records of $v_j$ satisfying $Cond2$ should have passed the evaluation of $v_j$’s in-edges. Furthermore, because all $v_j$’s precedent edges and $(v_j, v_i)$ itself are $v_i$’s precedent edges, the number of $v_i$’s precedent edges cannot exceed $n - 1$. According to the induction assumption, our repository can remove $v_j$’s random factors from the information shared by $v_j$ for $(v_j, v_i)$. That is, our repository can compute $(O_{1,j}, O_{2,j}) = (H_1(r_j, rec_j.attr_{j'}),$
$H_2(r_j, rec_i, attr_j)$ if the record $rec_i \in v_j$ satisfies $Cond_2$, where $r_j$ is the random number assigned to $(v_j, v_1)$. Note that $v_j$’s random factor is $R_1 = \bigoplus_{i=3}^{m} H_2(r_j, rec_i, attr_j)$, where $r_3, \ldots, r_m$ are the random numbers assigned to $v_j$’s in-edges $(v_3, v_1), \ldots, (v_m, v_1)$. If $v_j$’s record $rec_1$ satisfies $Cond_2$, for each in-edge $(v_j, v_1)$, there should be a record $rec_i \in v_j$ satisfying $rec_1, attr_1 = rec_i, attr_i$. Furthermore, from the information shared by $v_j$ for its record $rec_i$, our repository learns $(O_{1j}, O_{2j})$, where $H_2(r_j, rec_1, attr_j) = O_{2j}$. Hence, our repository can compute $v_j$’s random factor as $R_1 = \bigoplus_{i=3}^{m} O_{2j}$, and remove it from $(O_1, O_2)$.

ACKNOWLEDGMENTS

This work was supported by the US National Science Foundation under grant number ITR-CYBERTRUST 0430565.

REFERENCES


Stephen S. Yau received the PhD degree in electrical engineering from the University of Illinois, Urbana. He is currently the director of the Information Assurance Center and a professor in the Department of Computer Science and Engineering, School of Computing and Informatics at Arizona State University, Tempe. He served as the Chair of the department from 1994 to 2001. He was previously with the University of Florida, Gainesville, and Northwestern University, Evanston, Illinois. He served as the president of the IEEE Computer Society and as the editor-in-chief of Computer magazine. His current research is distributed and service-oriented computing, adaptive middleware, software engineering and trustworthy computing, and data privacy. He is a fellow of the IEEE and the American Association for the Advancement of Science.

Yin Yin received the BS degree in mathematics from Wuhan University, China, and the MS degree in computer science from the Chinese Academy of Science. He is a PhD student in the Department of Computer Science and Engineering at Arizona State University, Tempe. His research interests include privacy protection, trustworthy computing, and cryptography.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/publications/dlib.