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-

• *The authors are with the Department of Computer Science and Engineering, School of Computing and Informatics, and Information Assurance Center, Arizona State University, PO Box 878809, Tempe, AZ 85287-8804. E-mail: {yau, yin.yin}@asu.edu.*

Manuscript received 17 Nov. 2008; revised 4 Dec. 2008; accepted 5 Dec. 2008; published online 11 Dec. 2008.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2008-11-0103.
Digital Object Identifier no. 10.1109/TSC.2008.14.

TABLE 1
The Databases in the Motivating Example

Disease	Pattern
heart	p_1
heart	p_2
cancer	p_3
cancer	p_4

(a)

SSN	Pattern
ssn_1	p_1
ssn_2	p_1
ssn_3	p_2
ssn_4	p_3

(b)

SSN	Medicine	Reaction
ssn_1	med_1	$reac_1$
ssn_2	med_1	$reac_2$
ssn_3	med_3	$reac_3$
ssn_5	med_5	$reac_5$

(c)

Disease	Drug
$heart$	med_1
$cancer$	med_2
flu	med_3
$diabetes$	med_4

(d)

(a) Research $T1$. (b) DNA $T2$. (c) Hospital $T3$. (d) Pharmacy $T4$.

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 $T1(Disease, Pattern)$ storing diseases and corresponding DNA patterns, a DNA database $T2(SSN, Pattern)$ storing personal DNA patterns, a hospital database $T3(SSN, Medicine, Reaction)$ storing all patients' diagnosis histories, and a pharmacy database $T4(Disease, Drug)$ 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, $Tmp1$, $Tmp2$, and $Tmp3$, 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 $Q1$'s result, which is needed by $Q2$ as an input, our repository will randomize $Q1$'s result and make the randomized result still usable for $Q2$. 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 $Q2$ in this motivating example. To protect $Q1$'s result $\{p_1, p_2\}$ without disabling $Q2$, $\{p_1, p_2\}$ 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 $Q2$ by comparing $H(Tmp1.Pattern)$ and $H(T2.Pattern)$. This simple hash solution can avoid the need for our repository to know $Q1$'s results, but still keep the mapping relation between nonheart diseases and patients' SSNs. Since $H(p_3)$ does not appear in the $Q1$'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 $Q1$'s result, where the context-awareness implies that when a medical research institute shares its database Research $T1$ with our repository, it should know that its DNA pattern data will be used to match the DNA pattern data from $T2$. While the simple hash solution only randomizes the items in $Q1$'s result (i.e., p_1, p_2), our Context-Aware Data Sharing algorithm randomizes all patterns in $T1$, but ensures that only p_1 and p_2 can be used to evaluate $Q2$. 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 $T1$, $T2$, $T3$, and $T4$.

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 2
The Queries Required by the Motivating Example

```
Q1 -> Tmp1
SELECT T1.Pattern
FROM T1
WHERE T1.Pattern = "heart"
```

```
Q2 -> Tmp2
SELECT T2.SSN
FROM Tmp1, T2
WHERE Tmp1.Pattern = T2.Pattern
```

```
Q3 -> Tmp3
SELECT T4.Drug
FROM T4
WHERE T4.Disease = "heart"
```

```
Q4
SELECT T3.Reaction
FROM Tmp2, Tmp3, T3
WHERE T3.SSN = Tmp2.SSN AND
     T3.Medicine = Tmp3.Drug
```

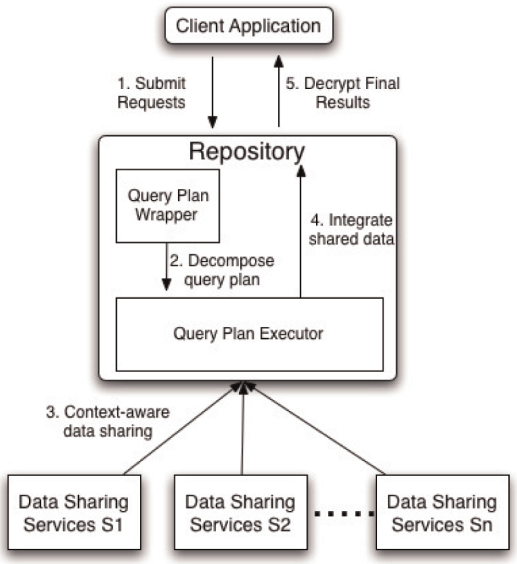


Fig. 1. Our privacy preserving repository for data integration across data sharing services.

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 date 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 and construct a query plan graph (to be defined in Section 5) from users' integration requirements. Based on this 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 $\{p_1, p_2, \cdots, p_m\}$ 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 $P = \{p_1, p_2, \cdots, p_m\}$ 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 $\{G_1, G_2, \cdots, G_m\}$ 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

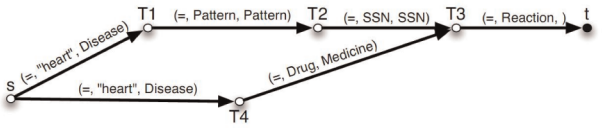


Fig. 2. The query plan graph of the motivating example.

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 $S_1, S_2, \cdots, S_n$, the query plan graph $G = \{V, E, C\}$ is a labeled directed acyclic graph. $V = \{v_1, v_2, \cdots, v_m, s, t\}$ 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. $E = \{e_1, e_2, \cdots, e_l\}$ is a set of edges, and each edge $e_{i,j} = (v_i, v_j)$ represents a data integration relation between data sharing services v_i and v_j. Finally, $C = \{c_1, c_2, \cdots, c_l\}$ is a set of labels attached with each $e_{i,j} \in E$, and each label $c_{i,j} = (op, attr1, attr2) \in C$ specifies that the data from the data sharing services 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 $Q2$, $(s, T4)$ represents the query $Q3$, 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 G 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

```
<?xml version="1.0" encoding="UTF-8"?>
<!--doc:Define the query plan as a collection of edges. -->
<!ELEMENT plan (edge+, node+)>

<!--doc:Specify edge. -->
<!ELEMENT edge (r, condition*)>
<!ATTLIST edge id ID #REQUIRED>
<!ATTLIST edge head IDREF #REQUIRED>
<!ATTLIST edge tail IDREF #REQUIRED>
<!--doc:The random number assigned to the edge.-->
<!ELEMENT r (#PCDATA)>

<!--doc:Specify condition.-->
<!ELEMENT condition (op?, attr1?, attr2?) >
<!ELEMENT op (#PCDATA)>
<!ELEMENT attr1 (#PCDATA)>
<!ELEMENT attr2 (#PCDATA)>

<!--doc:Specify node. -->
<!ELEMENT node (name)>
<!ATTLIST node id ID #REQUIRED>
<!ELEMENT name (#PCDATA)>
```

Fig. 3. The DTD schema of the query plan specification language ($QPSL$).

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 $G_i = (V_i, E_i, C_i, \overline{r}_i)$, $(1 \leq i \leq |V|)$

```
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 ← {v_i}, E_i, C_i, r̄_i ← ∅;
6  |  for j ← 1 to |V| do
7  |  |  if (v_j, v_i) ∈ E then
8  |  |  |  V_i ← V_i ∪ {v_j};
9  |  |  |  E_i ← E_i ∪ {(v_j, v_i)};
10 |  |  |  C_i ← C_i ∪ (v_j, v_i)'s label;
11 |  |  |  r̄_i ← r̄_i ∪ {r_{j,i}};
12 |  |  end
13 |  |  if (v_i, v_j) ∈ E then
14 |  |  |  V_i ← V_i ∪ {v_j};
15 |  |  |  E_i ← E_i ∪ {(v_i, v_j)};
16 |  |  |  C_i ← C_i ∪ (v_i, v_j)'s label;
17 |  |  |  r̄_i ← r̄_i ∪ {r_{i,j}};
18 |  |  end
19 |  end
20 |  G_i ← (V_i, E_i, C_i, r̄_i);
21 end
22 return G_1, G_2, ···, G_{|V|};
```

We denote the subgraph G_i of v_i as $(V_i, E_i, C_i, \overline{r}_i)$, 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 $\overline{r}_i$ 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 [27] and the privacy preserving e-mail routing application proposed in [11], where only the authorities can match the shared data.

As mentioned before, when data sharing and integration are across various organizations without a central authority, the restriction on matching capability is stronger in the following aspects:

Requirement 1. The repository can match two data sharing services' data only if these two services' data is required to be matched by the user's integration requests.

Requirement 2. When the user's integration requests require matching between v_i's data and tmp, where tmp is the result of matching between two other services v_j and v_k, the repository cannot match v_i's data with the data of v_j or v_k if the data of v_j or v_k is not in tmp.

To enforce the above restrictions, our *Context-Aware Data Sharing Algorithm* shares data in the following two steps:

1. For each edge (v_i, v_j) of G, v_i and v_j share their data with the random number r assigned in Algorithm 1 to ensure that our repository can match their data, but cannot use their shared information to match other nodes' data.
2. Node v_j computes the random factor R to further randomize its shared information to ensure that only v_j's data matching v_i's data can be used to match v_k's data, when (v_j, v_k) is one of v_j's out-edges in G.

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 $G_i = (V_i, E_i, C_i, \overline{r}_i)$, 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

Initialize $I_i, O_i, AttrIn_i \leftarrow \emptyset$
foreach *node* v_j *satisfying* $(v_j, v_i) \in E_i$ **do**
  $(op, first, second) \leftarrow (v_j, v_i)$'s label
  (v_j, v_i)'s random number $r_{j,i} \leftarrow \overline{r}_i$
  foreach *data record* rec *of the service* S_i **do**
  $I_{j,i} \leftarrow I_{j,i} \cup H_1(r_{j,i}, rec.second)$
  $I_i \leftarrow I_i \cup I_{j,i}$
  $AttrIn_i \leftarrow AttrIn_i \cup second$
end
foreach *node* v_j *satisfying* $(v_i, v_j) \in E_i$ **do**
  $(op, first, second) \leftarrow (v_i, v_j)$'s label
  (v_i, v_j)'s random number $r_{i,j} \leftarrow \overline{r}_i$
  foreach *data record* rec *of the database* D_i **do**
    $R \leftarrow 0$
    foreach *attribute* $a \in AttrIn_i$ **do**
    $R \leftarrow R \oplus H_2(r_{i,j}, rec.a)$
    if v_j *is the sink node* t **then**
      $O_{i,j,1} \leftarrow \emptyset$
      $O_{i,j,2} \leftarrow E_{pk}(rec.first) \oplus R$
    end
    else
      $O_{i,j,1} \leftarrow H_1(r_{i,j}, rec.first) \oplus R$
      $O_{i,j,2} \leftarrow H_2(r_{i,j}, rec.first) \oplus R$
    end
    $O_{i,j} \leftarrow O_{i,j} \cup (O_{i,j,1}, O_{i,j,2})$
  end
  $O_i \leftarrow O_i \cup O_{i,j}$
end
return $RD_i \leftarrow I_i \cup O_i$

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 i_1, i_2, and o_1 according to v's in-edges and out-edges. Specifically, v computes $I_1 = H_1(r_1, v_{1,1})$ and $I_2 = H_1(r_2, v_{1,2})$ for the two in-edges and $O_1 = H_1(r_3, v_{1,3})$ for the out-edge, where r_1, r_2, and r_3 are three random numbers assigned to edges (i_1, v), (i_2, v), and (v, o_1) in Algorithm 1. Similarly, i_1 and i_2 will

TABLE 3
The Five Sets of Randomized Data According to Fig. 2 and Algorithm 2

O_s: Disease
$H_1(r_{s,1}, heart)$
$H_2(r_{s,1}, heart)$
$H_1(r_{s,4}, heart)$
$H_2(r_{s,4}, heart)$

(a)

I_1: Disease	O_1: Pattern
$H_1(r_{s,1}, heart)$	$H_1(r_{1,2}, p_1) \oplus H_2(r_{s,1}, heart)$
	$H_2(r_{1,2}, p_1) \oplus H_2(r_{s,1}, heart)$
$H_1(r_{s,1}, heart)$	$H_1(r_{1,2}, p_2) \oplus H_2(r_{s,1}, heart)$
	$H_2(r_{1,2}, p_2) \oplus H_2(r_{s,1}, heart)$
$H_1(r_{s,1}, cancer)$	$H_1(r_{1,2}, p_3) \oplus H_2(r_{s,1}, cancer)$
	$H_2(r_{1,2}, p_3) \oplus H_2(r_{s,1}, cancer)$
$H_1(r_{s,1}, cancer)$	$H_1(r_{1,2}, p_4) \oplus H_2(r_{s,1}, cancer)$
	$H_2(r_{1,2}, p_4) \oplus H_2(r_{s,1}, cancer)$

(b)

I_2: Pattern	O_2: SSN
$H_1(r_{1,2}, p_1)$	$H_1(r_{2,3}, ssn_1) \oplus H_2(r_{1,2}, p_1)$
	$H_2(r_{2,3}, ssn_1) \oplus H_2(r_{1,2}, p_1)$
$H_1(r_{1,2}, p_1))$	$H_1(r_{2,3}, ssn_2) \oplus H_2(r_{1,2}, p_1)$
	$H_2(r_{2,3}, ssn_2) \oplus H_2(r_{1,2}, p_1)$
$H_1(r_{1,2}, p_2)$	$H_1(r_{2,3}, ssn_3) \oplus H_2(r_{1,2}, p_2)$
	$H_2(r_{2,3}, ssn_3) \oplus H_2(r_{1,2}, p_2)$
$H_1(r_{1,2}, p_3)$	$H_1(r_{2,3}, ssn_4) \oplus H_2(r_{1,2}, p_3)$
	$H_2(r_{2,3}, ssn_4) \oplus H_2(r_{1,2}, p_3)$

(c)

I_4: Disease	O_4: Drug
$H_1(r_{s,4}, heart)$	$H_1(r_{4,3}, med_1) \oplus H_2(r_{s,4}, heart)$
	$H_2(r_{4,3}, med_1) \oplus H_2(r_{s,4}, heart)$
$H_1(r_{s,4}, cancer)$	$H_1(r_{4,3}, med_2) \oplus H_2(r_{s,4}, cancer)$
	$H_2(r_{4,3}, med_2) \oplus H_2(r_{s,4}, cancer)$
$H_1(r_{s,4}, flu)$	$H_1(r_{4,3}, med_3) \oplus H_2(r_{s,4}, flu)$
	$H_2(r_{4,3}, med_3) \oplus H_2(r_{s,4}, flue)$
$H_1(r_{s,4}, diabetes)$	$H_1(r_{4,3}, med_4) \oplus H_2(r_{s,4}, diabetes)$
	$H_2(r_{4,3}, med_4) \oplus H_2(r_{s,4}, diabetes)$

(d)

$I_{2,3}$: SSN	$I_{4,3}$: Medicine	O_3: Reaction
$H_1(r_{2,3}, ssn_1)$	$H_1(r_{4,3}, med_1)$	$E_{pk}(r_{3,t}, reac_1) \oplus H_2(r_{2,3}, ssn_1) \oplus H_2(r_{4,3}, med_1)$
$H_1(r_{2,3}, ssn_2)$	$H_1(r_{4,3}, med_1)$	$E_{pk}(r_{3,t}, reac_2) \oplus H_2(r_{2,3}, ssn_2) \oplus H_2(r_{4,3}, med_1)$
$H_1(r_{2,3}, ssn_3)$	$H_1(r_{4,3}, med_3)$	$E_{pk}(r_{3,t}, reac_3) \oplus H_2(r_{2,3}, ssn_3) \oplus H_2(r_{4,3}, med_3)$
$H_1(r_{2,3}, ssn_5)$	$H_1(r_{4,3}, med_5)$	$E_{pk}(r_{3,t}, reac_5) \oplus H_2(r_{2,3}, ssn_5) \oplus H_2(r_{4,3}, med_5)$

(e)

(a) RD_s. *(b)* RD_1. *(c)* RD_2. *(d)* RD_4. *(e)* RD_3.

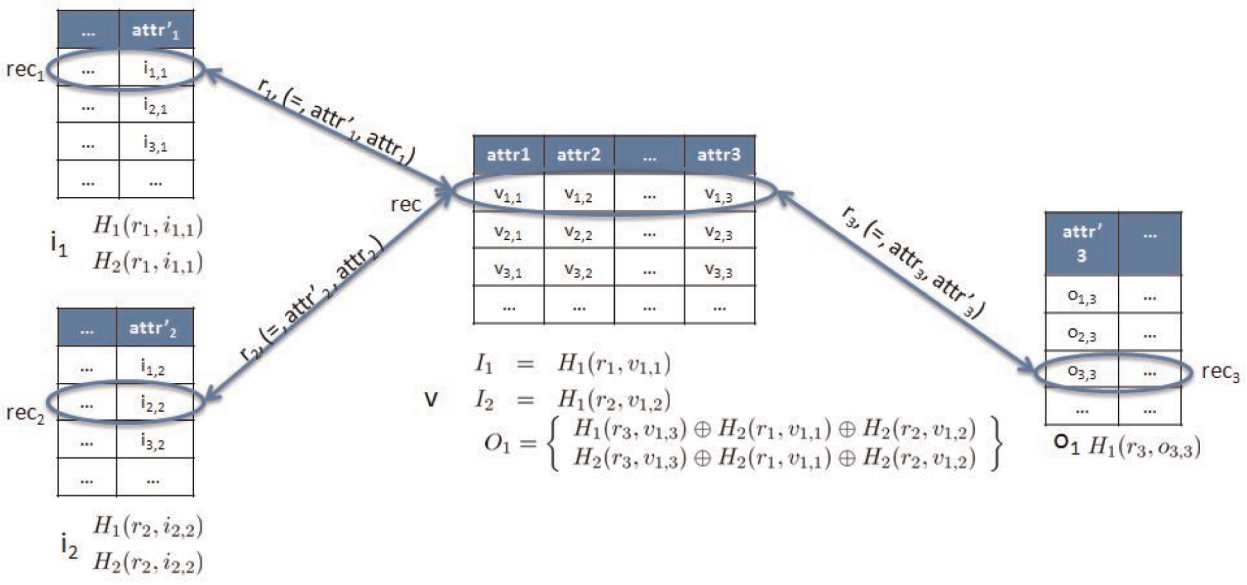


Fig. 4. A subgraph generated by Algorithm 1.

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:*

1. ***Correctness.*** *If $rec_1.attr_1$ matches $rec_2.attr_2$, their hash values $H_1(r, rec_1.attr_1)$ and $H_1(r, rec_2.attr_2)$ also match.*
2. ***Robustness.*** *If the hash values $H_1(r, rec_1.attr_1)$ matches $H_1(r, rec_2.attr_2)$, $rec_1.attr_1$ and $rec_2.attr_2$ will match with large probability.*
3. ***Independence.*** *The hash values $H_1(r, rec_1.attr_1)$ and $H_1(r, rec_2.attr_2)$ are independent from the information shared for other edges, and, hence, they can only be used for the matching between $rec_1.attr_1$ and $rec_2.attr_2$.*

7.2 Requirement 2

For a record rec of v, rec is said to pass the evaluation of v's in-edges if and only if there is a record $rec' \in i$ matching 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 rec can be used by our repository to match v_j's data only when 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 $AttrIn = \{attr_1, attr_2\}$ and then computes the random factor R as $H_2(r_1, v_{1,1}) \oplus H_2(r_2, v_{1,2})$ for record rec, where $H_2(r_1, v_{1,1})$ is shared by i_1 for rec_1 and $H_2(r_2, v_{1,2})$ is shared by i_2 for rec_2. Then, v randomizes its shared information for the out-edge (v, o_1) with the random factor R as $O_1 = \{H_1(r_3, v_{1,3}) \oplus R, H_2(r_3, v_{1,3}) \oplus R\}$ if o_1 is not the sink node t; otherwise, $O_1 = \{E_{pk}(r_3, v_{1,3}) \oplus R\}$. 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 rec pass the evaluation of both (i_1, v) and (i_2, 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 $attr_1$ with v_2's attribute $attr_2$. Let 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 rec_1, where $O_1 = H_1(r, rec_1.attr_1) \oplus R_1$ and $O_2 = H_2(r, rec_1.attr_1) \oplus R_1$. 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 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 $\{RD_s, RD_1, RD_2, \cdots, RD_n\}$;
Output: The final results $FinalRes$

```
Initialize FinalRes, Walked ← ∅, UnWalked ← E
foreach node v_i satisfying (s, v_i) ∈ E do
    attr_1, attr_2 ← the attributes in the label of (s, v_i);
    attrOut ← v_i's attributes specified in its out-edges;
    Match(RD_s, RD_i, attr1, attr2, attrOut);
    UnWalked ← UnWalked/(s, v_i);
    Walked ← Walked ∪ (s, v_i);
end
while UnWalked is not empty do
    (v_i, v_j) ←_R UnWalked;
    if all v_i's in-edges ∈ Walked then
        attr_1, attr_2 ← the attributes in the label of (v_i, v_j);
        attrOut ← v_j's attributes specified in its out-edges;
        Match(RD_i, RD_j, attr1, attr2, attrOut);
        UnWalked ← UnWalked/(v_i, v_j);
        Walked ← Walked ∪ (v_i, v_j);
    end
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 *Match* function, which is explained here with the matching of RD_1 and 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 $AttrOut$. In our example, the edge is to match $RD_2.pattern$ with $RD_1.pattern$, and RD_2 has only one attribute in its out-edge. Hence, we have $AttrOut = RD_2.ssn$.
- **Match.** In this step, REP scans and matches tail nodes' records with the records from head nodes. In our example, REP matches RD_2's records with RD_1's records according to their attribute pattern. Let RD_1's records be rec_i and RD_2's records be rec'_i, where $1 \leq i \leq 4$. Recall that RD_2's record rec'_i passes the evaluation of the edge from RD_1 to RD_2 only if there is a record $rec_j \in RD_1$ that $rec_j.pattern = rec'_i.pattern$. According to Table 3, the first three records of RD_2 pass the evaluation, and the last one fails.
- **Remove random factors.** Assume that the edge's tail node's record rec passes the evaluation of the edge. To use rec to evaluate the tail node's out-edges, REP

```
<?xml version="1.0" encoding="UTF-8"?>
<!--doc:Define randomizedData as a collection of recodes.-->
<!ELEMENT randomizedData (recode*)>
<!ATTLIST randomizedData id ID #REQUIRED>
<!ATTLIST randomizedData name CDATA #REQUIRED>

<!--doc:Specify recode. -->
<!ELEMENT recode (attrIn*, attrOut*)>

<!--doc:Specify attrIn and attrOut. -->
<!ELEMENT attrIn (name, r, comparison)>
<!ELEMENT attrOut (name, r, comparison,
            random_factor, decryption)>

<!ELEMENT name (#PCDATA)>
<!ELEMENT r (#PCDATA)>
<!ELEMENT comparison (#PCDATA)>
<!ELEMENT random_factor (#PCDATA)>
<!ELEMENT decryption (a, b)>
<!ELEMENT a (#PCDATA)>
<!ELEMENT b (#PCDATA)>
```

Fig. 5. The DTD schema of the shared data.

first needs to remove random factors from the shared information about rec for all $attr \in AttrOut$. In our example, only the first three records of RD_2 passed the evaluation and there is only one attribute $ssn \in AttrOut$. 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 RD_2 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 30,000 records.

We constructed nine different size data sets by extracting the adult income database's first 100, 500, 1,000, 5,000, 10,000, 15,000, 20,000, 25,000 and 30,000 records. For each data set, we developed a data sharing service to share the data set. All shared data will be represented, stored, and exchanged as XML documents using the DTD schema depicted in Fig. 5, where r is the random number assigned by Algorithm 1 for edges, $comparison$ is the operation specified by the edge's label, $random\_factor$ is the random factor computed in Algorithm 2, and a and b are two parameters of the public encryption scheme [13]. All XML documents will be parsed with XQuery [3].

Because our experiment focuses on the performance evaluation, we only need to develop a simple query plan $s \xrightarrow{=,"l",education} d \xrightarrow{=,native-country,null} t$, which has an in-edge from the source node to the data sharing service node d matching the user's input l with the data sharing service's attribute $education$, and one out-edge from the data service node d to t outputting the attribute $native-country$. When the user changes his input l, this query plan studies the native countries of adults who have the education level from $11th$, $HS-grad$ to $Bachelor$. While $HS-grad$ and $Bachelor$ represent the highest selectivity 0.3 and the lowest selectivity 0.03, respectively, $11th$ has the average selectivity 0.16 in all education levels.

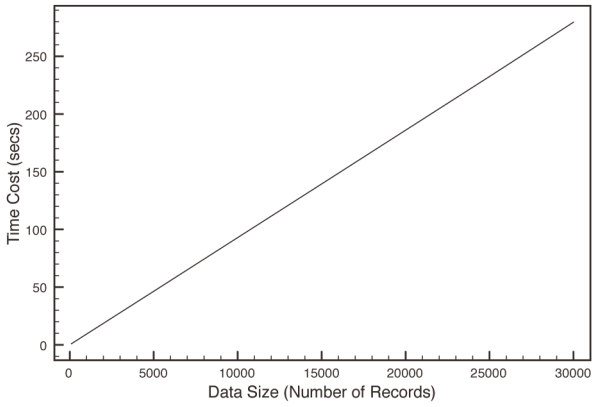


Fig. 6. The performance of Context-Aware Data Sharing Algorithm.

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 n_1 in-edges, n_2 out-edges for all its subsequent operations, and n_3 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 $O((n_1+n_2)t_hN)+O(n_3t_eN)$, 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 $O(t_hN+t_eN)$, 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

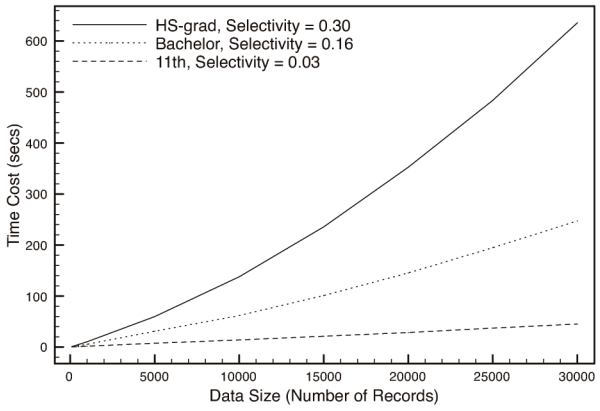


Fig. 7. The performance of Integration algorithm.

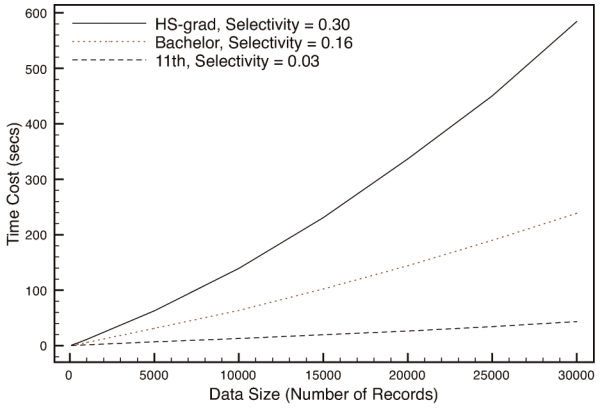


Fig. 8. The performance of decryption.

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 $O(Nt_d)$.

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 have 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 conflict 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 $H_1(r, rec_1.attr_1)$ shared by v_1 to check whether $rec_1.attr_1 = rec_3.attr_3$ with large probability. Suppose the random number assigned for v_3 is r' and the hash function H_1's confliction 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 $1/2^{|r|}$. 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 $p' + 1/2^{|r|}$, 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 $(i_1, i_2) \prec \cdots \prec (i_m, v_1) \prec (v_1, v_2)$, 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 $attr_{i1}$ with i_2's attribute $attr_{i2}$, and i_2's record rec_{i2} is randomized by the random factor $R_{i2} = H_2(r_{i1}, rec_{i2}.attr_{i2}) \oplus R'_{i2}$, where r_{i1} is the random number assigned to (i_1, i_2) and R'_{i2} is the remaining part of the random factor computed according to i_2's other in-edges. Hence, to remove R_{i2}, our repository must compute $H_2(r_{i1}, rec_{i2}.attr_{i2})$ first. However, without evaluating (i_1, i_2), the best information that our repository possesses is $H_2(r_{i1}, rec_{i1}.attr_{i1}) \oplus R_{i1}$ from node i_1, where R_{i1} is the random factor of i_1 for its record rec_{i1}, which does not reveal any information related to $H_2(r_{i1}, rec_{i2}.attr_{i2})$ even when $rec_{i1}.attr_{i1} = rec_{i2}.attr_{i2}$, because i_1's random factor R_{i1} 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_{i2} is unknown. Recursively, our repository cannot evaluate (v_1, v_2) without evaluating (i_m, v_1) first.

If $Cond1$ is satisfied, but $Cond2$ is not satisfied, i.e., for one in-edge (v_j, v_1) of v_1, there is no record $rec_j \in v_j$ satisfying that $rec_j.attr'_j = rec_1.attr_j$. In this case, our repository cannot learn $H_2(r_j, rec_1.attr_j)$ from the information shared by v_j and, therefore, cannot learn and remove v_1's random factor R_1 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_1's in-edges as $\{(v_j, v_1), 3 \leq j \leq m\}$, where (v_j, v_1) matches v_1's attribute $attr_j$ with v_j's attribute $attr'_j$. First, when $Cond1$ is satisfied, for each edge (v_j, v_1), 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_1) itself are v_1's precedent edges, the number of v_j'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_1). That is, our repository can compute $(O_{1,j}, O_{2,j}) = (H_1(r_j, rec_j.attr'_j),$

$H_2(r_j, rec_j.attr'_j))$ if the record $rec_j \in v_j$ satisfies $Cond2$, where r_j is the random number assigned to (v_j, v_1). Note that v_1's random factor is $R_1 = \bigoplus_{j=3}^{m} H_2(r_j, rec_1.attr_j)$, where $r_3, \cdots, r_m$ are the random numbers assigned to v_1's in-edges $(v_3, v_1), \cdots, (v_m, v_1)$. If v_1's record rec_1 satisfies $Cond2$, for each in-edge (v_j, v_1), there should be a record $rec_j \in v_j$ satisfying $rec_1.attr_j = rec_j.attr'_j$. Furthermore, from the information shared by v_j for its record rec_j, our repository learns $(O_{1,j}, O_{2,j})$, where $H_2(r_j, rec_1.attr_j) = O_{2,j}$. Hence, our repository can compute v_1's random factor as $R_1 = \bigoplus_{j=3}^{m} O_{2,j}$, and remove it from (O_1, O_2). □

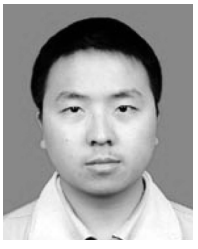
ACKNOWLEDGMENTS

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

REFERENCES

[1] Web Services Business Process Execution Language Version 2.0, OASIS Standard, http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, 2007.
[2] DTD Schema, http://www.w3.org/TR/REC-xml/#dt-doctype, 2008.
[3] XQuery, http://www.w3.org/TR/xquery, 2007.
[4] M. Abdalla, M. Bellare, D. Catalano, E. Kiltz, T. Kohno, T. Lange, J. Malone-Lee, G. Neven, P. Paillier, and H. Shi, "Searchable Encryption Revisited: Consistency Properties, Relation to Anonymous IBE, and Extensions," *Advances in Cryptology (CRYPTO '05)*, vol. 3621, pp. 205-222, 2005.
[5] S. Adali, K.S. Candan, Y. Papakonstantinou, and V.S. Subrahmanian, "Query Caching and Optimization in Distributed Mediator Systems," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '96)*, pp. 137-148, 1996.
[6] R. Agrawal, A.V. Evfimievski, and R. Srikant, "Information Sharing across Private Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data (SIGMOD '03)*, pp. 86-97, 2003.
[7] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, "Order-Preserving Encryption for Numeric Data," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '04)*, pp. 563-574, 2004.
[8] Y. Arens, C.A. Knoblock, and W. Shen, "Query Reformulation for Dynamic Information Integration," *J. Intelligent Information Systems*, vol. 6, no. 2, pp. 99-130, 1996.
[9] J. Baek, R. Safavi-Naini, and W. Susilo, "On the Integration of Public Key Data Encryption and Public Key Encryption with Keyword Search," *Proc. Ninth Information Security Conf. (ISC '06)*, vol. 4176, pp. 217-232, 2006.
[10] M. Bellare, A. Boldyreva, and A. O'Neill, "Deterministic and Efficiently Searchable Encryption," *Advances in Cryptology (CRYPTO '07)*, vol. 4622, pp. 535-552, 2007.
[11] D. Boneh, G.D. Crescenzo, R. Ostrovsky, and G. Persiano, "Public Key Encryption with Keyword Search," *Advances in Cryptology (EUROCRYPT '04)*, vol. 3027, pp. 506-522, 2004.
[12] D. Boneh and B. Waters, "Conjunctive, Subset, and Range Queries on Encrypted Data," *Proc. Theory of Cryptography Conf. (TCC '07)*, vol. 4392, pp. 535-554, 2007.
[13] E. Bresson, D. Catalano, and D. Pointcheval, "A Simple Public-Key Cryptosystem with a Double Trapdoor Decryption Mechanism and Its Applications," *Advances in Cryptology (ASIACRYPT '03)*, vol. 2894, pp. 37-54, 2003.
[14] R. Curtmola, J.A. Garay, S. Kamara, and R. Ostrovsky, "Searchable Symmetric Encryption: Improved Definitions and Efficient Constructions," *Proc. ACM Conf. Computer and Comm. Security (CCS '06)*, pp. 79-88, 2006.
[15] F. Emekci, D. Agrawal, A.E. Abbadi, and A. Gulbeden, "Privacy Preserving Query Processing Using Third Parties," *Proc. 22nd Int'l Conf. Data Eng. (ICDE '06)*, p. 27, 2006.
[16] T. Ge and S.B. Zdonik, "Answering Aggregation Queries in a Secure System Model," *Proc. 33rd Int'l Conf. Very Large Data Bases (VLDB '07)*, pp. 519-530, 2007.
[17] O. Goldreich, *Foundations of Cryptography Volume II Basic Applications*. Cambridge Univ. Press, 2001.
[18] Z.G. Ives, D. Florescu, M. Friedman, A.Y. Levy, and D.S. Weld, "An Adaptive Query Execution System for Data Integration," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '99)*, pp. 299-310, 1999.
[19] K.P. Fischer, U. Bleimann, W. Fuhrmann, and S.M. Furnell, "Security Policy Enforcement in BPEL-Defined Collaborative Business Processes," *Proc. 23rd Int'l Conf. Data Eng. Workshop*, pp. 685-694, 2007.
[20] N. Kushmerick, D.S. Weld, and R.B. Doorenbos, "Wrapper Induction for Information Extraction," *Proc. Int'l Joint Conf. Artificial Intelligence*, pp. 729-737, 1997.
[21] A.Y. Levy, A. Rajaraman, and J.J. Ordille, "Querying Heterogeneous Information Sources Using Source Descriptions," *Proc. 22th Int'l Conf. Very Large Data Bases (VLDB '96)*, pp. 251-262, 1996.
[22] Y. Lindell and B. Pinkas, "Privacy Preserving Data Mining," *J. Cryptology*, vol. 15, no. 3, pp. 177-206, 2002.
[23] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," *Advances in Cryptology (EUROCRYPT '99)*, vol. 1592, pp. 223-238, 1999.
[24] B. Pinkas, "Cryptographic Techniques for Privacy-Preserving Data Mining," *SIGKDD Explorations*, vol. 4, no. 2, pp. 12-19, 2002.
[25] M. Scannapieco, I. Figotin, E. Bertino, and A.K. Elmagarmid, "Privacy Preserving Schema and Data Matching," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '07)*, pp. 653-664, 2007.
[26] E. Shi, J. Bethencourt, H.T.-H. Chan, D.X. Song, and A. Perrig, "Multidimensional Range Query over Encrypted Data," *Proc. IEEE Symp. Security and Privacy (S&P '07)*, pp. 350-364, 2007.
[27] D.X. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data," *Proc. IEEE Symp. Security and Privacy (S&P '00)*, pp. 44-55, 2000.
[28] L. Xiong, S. Chitti, and L. Liu, "Preserving Data Privacy for Outsourcing Data Aggregation Services," *ACM Trans. Internet Technology*, vol. 7, no. 3, pp. 17-45, 2007.
[29] S.S. Yau and Z. Chen, "Security Policy Integration and Conflict Reconciliation for Collaborations among Organizations in Ubiquitous Computing Environments," *Proc. Fifth Int'l Conf. Ubiquitous Intelligence and Computing*, pp. 3-19, 2008.
[30] B. Yu, G. Li, K.R. Sollins, and A.K.H. Tung, "Effective Keyword-Based Selection of Relational Databases," *Proc. ACM Int'l Conf. Management of Data (SIGMOD '07)*, pp. 139-150, 2007.

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 in 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.