ABACUS: A Distributed Middleware for Privacy Preserving Data Sharing Across Private Data Warehouses*

Fatih Emekci, Divyakant Agrawal, and Amr El Abbadi

Department of Computer Science,
University of California at Santa Barbara
{fatih, agrawal, amr}@cs.ucsb.edu

Abstract. Recent trends in the global economy force competitive enterprises to collaborate with each other to analyze markets in a better way and make decisions based on that. Therefore, they might want to share their data with each other to run data mining algorithms over the union of their data to get more accurate and representative results. During this process they do not want to reveal their data to each other due to the legal issues and competition. However, current systems do not consider privacy preservation in data sharing across private data sources. To satisfy this requirement, we propose a distributed middleware, ABACUS, to perform intersection, join, and aggregation queries over multiple private data warehouses in a privacy preserving manner. Our analytical evaluations show that ABACUS is efficient and scalable.

1 Introduction

Recent trends in the global economy force competitive enterprises to collaborate with each other for the purpose of market analysis. One of the most important examples of such collaboration is data sharing to mine and understand the market trends to be used in decision making. However, although enterprises are willing to share information with each other, they do not want to reveal their data. Due to the legal issues and competition in the market, datasources want to preserve the privacy of their data while sharing them. For example, consider a scenario consisting of two hotels, H_1 and H_2, and two airlines, A_1 and A_2. Assume hotel H_1 wants to offer a new deal to each of its customers including hotel and flight expenses based on his/her flight history. Therefore, hotel H_1 needs to learn flight history of its customers from airlines A_1 and A_2. One method to learn flight history of customers is that airlines send all of their data to hotel H_1 so that hotel H_1 can extract desired information. However, these airlines also work with hotel H_2, which is a competitor of hotel H_1, and thus they may not want to send all of their data to hotel H_1. That is because hotel H_1 can discover the customers of hotel H_2 and try to attract them. Therefore, if airlines want to work with both hotels, they cannot send their data to any of these hotels. Similarly, hotel H_1 cannot send its data to airlines so that airlines can extract the information that hotel H_1 needs since airlines will discover

* This research was funded in parts by NSF grants IS-02-23022 and CNF-04-23336.

G. Alonso (Ed.): Middleware 2005, LNCS 3790, pp. 21–41, 2005.

each other's customers. In order to be able to collaborate, hotel H_1 should take its customers' information from airlines in a way that airlines A_1 and A_2 share their data with hotel H_1 by only revealing common customers (i.e., revealing $H_1 \cap A_1$ to H_1 and A_1 and $H_1 \cap A_2$ to H_1 and A_2). By using such a method hotel H_1 cannot discover new customers which may be customers of hotel H_2 and also airlines cannot discover new customers which may be customers of the other airline. In addition to this, hotel H_1 may want to know the total amount of its customers' travel expenses or total expenditure of a customer for its future business decisions and offers. Other enterprises may be willing to collaborate with hotel H_1, if they can preserve their privacy. The essential operations to perform these collaborations are privacy preserving intersection, join and aggregation queries. Unfortunately, we cannot use traditional query processing techniques since they do not consider privacy issues. Therefore, there is a need for privacy preserving query processing and data sharing across multiple private data warehouses.

Data integration and sharing has emerged as an important practical problem from a data management point of view [3,4,7,8,9]. Techniques used for this purpose commonly assume that the data sources are willing to allow access to all their data without privacy concerns during query processing. This assumption, however, is unrealistic in real life since most of the time data sources are competing enterprises. There have been several techniques in the areas of database and cryptography for privacy preserving data sharing. One of them is to use trusted third parties such that data sources hand over their data and a third party performs the computation on their behalf [1,10]. The level of trust may not be acceptable in these methods. Another approach is using secure multi-party computation where given m parties and their respective inputs $x_1, x_2, .., x_m$, a function $f(x_1, x_2, ..., x_m)$ is computed such that all parties can only learn $f(x_1, x_2, ..., x_m)$ but nothing else [6,7,11]. The computation and the communication costs make this method impractical for database operations working over a large number of elements.

In this paper, we address the problem of privacy preserving data sharing over multiple private data warehouses. We propose a distributed middleware, ABACUS, to perform intersection, join, and aggregation queries over multiple private data warehouses in a privacy preserving manner. Privacy preservation means that parties involved in the query would only be able to learn the query result but nothing else. In addition, we introduce new types of aggregation queries needed in this context and propose efficient techniques to process them. ABACUS operates as a proxy among private data warehouses and allows users to pose queries over multiple private data warehouses. Our analytical evaluations demonstrate that ABACUS provides an efficient and scalable scheme to perform intersection, join, and aggregation queries.

The rest of the paper is organized as follows. Section 2 formulates the problem and presents the architecture overview. Section 3 describes intersection and join query processing. Aggregation query processing is discussed in Section 4 and the analysis is presented in Section 5. The last section concludes the paper.

2 Problem Definition and Architecture Overview

Enterprises gather data from their multiple operational databases into a data warehouse, which is one the most popular ways of storing data to support decision-making in or-

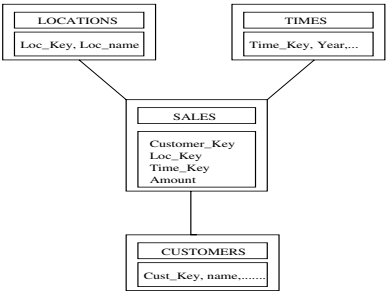

Fig. 1. An Example of Star Schema

ganizations. Data warehouse systems or OLAP (Online Analytical Processing) systems are different than OLTP (On-Line Transaction Processing) systems which are designed for fast updates. Thus, large enterprises have both OLAP and OLTP systems to support both an on-line community who expect fast response time for executing transactions and off-line users who expect to analyze the data in a reasonable amount of time. Most enterprises create a large data warehouse, and periodically extract data from OLTP systems into data warehouse to be able to analyze data without interfering with online users. Data Warehouses usually use star schema for fast execution of queries over aggregated data. Star schema has dimension tables and a fact table containing a foreign key for each of the dimension tables. Furthermore, it is usually not normalized for efficient query response time since fewer joins, a bottleneck in query processing, are performed. Figure 1 shows an instance of a star schema with the fact table, *Sales*, and the dimension tables, *Customers*, *Times*, and *Locations*. Current commercial data warehouses support efficient methods to examine data. However, they do not support privacy preserving data sharing across multiple private data warehouses, which is useful for analyzing the market instead of a single company's data.

The problem of query processing across multiple private data warehouses is defined as follows:

> Let D_1, D_2, ..., D_m be the data warehouses (defined with a star schema) of a set of m data sources $P = \{P_1, ..., P_m\}$ and q be a query spanning D_1 through D_m. The problem is to compute the answer of q without revealing any additional information to any of the data sources.

Agrawal et al. [2] solved the problem of privacy preserving query processing across private databases by restricting it to two data sources with some relaxation in an honest-but-curious environment [6] for intersection and equijoin operations. The honest-but-curious environment means that parties follow the protocols correctly but keep all messages sent and received during the course of the query processing. The relaxation reveals the sizes of the tables or lists in the database to the other party. However, the proposed technique has two shortcomings: 1) Encryption is a computation intensive operation which is not suitable for database operations where large numbers of items need to be processed. 2) It does not support aggregation queries, which are among the most important queries.

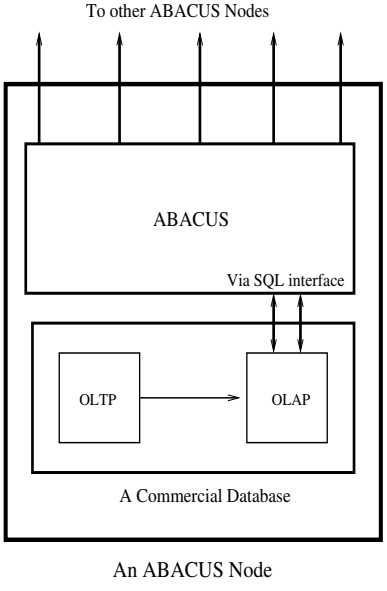

Fig. 2. The architecture of a ABACUS node

In this paper, we propose ABACUS for privacy preserving data sharing across multiple private data warehouses. ABACUS eliminates the need for third parties by taking advantage of the star schema and executes intersection, join and aggregation queries in a privacy preserving manner. In addition, we introduce new types of aggregation operators which are useful in the context of data warehouse and solve them efficiently. ABACUS also operates in an honest-but-curios environment and it reveals the size of tables and lists similar to [2].

ABACUS is a distributed middleware operating on top of any commercial database as shown in Figure 2. It provides a user interface where users can pose queries over multiple private data warehouses. ABACUS executes queries by running ABACUS nodes operating on different data warehouses. Each ABACUS node interacts with its data warehouse via SQL interface supplied by the underlying commercial database. Then, it contacts other ABACUS nodes and shares its data with them to process queries in a privacy preserving manner using the protocols proposed in this paper.

ABACUS does not aim to solve the problem of revealing additional information to a datasource which poses multiple queries and combines their results in order to obtain additional information about the data. In addition, it does not solve the problem of data discovery and schema mediation. Solutions to these problems are discussed briefly in [2] and they could be used in ABACUS.

3 Intersection and Join Query Processing

Intersection and join queries are the two important types of queries supported by current commercial databases without privacy concerns. However, in the context of data

sharing across multiple private data warehouses we need privacy preserving intersection and join queries. Therefore, we will define the problems of privacy preserving intersection, and join queries and also show how to solve them efficiently in Section 3.1 and Section 3.2 respectively.

3.1 Aggregated Intersection

Intersection queries constitute the first step for collaboration over common data items. For example, a company may need to know other companies' opinions about its customers. For this kind of collaboration, two companies need to find the common customers as a first step, i.e., intersection. The intersection of two customer lists can be found easily unless they do not hesitate to reveal their customers to each other. However, most of the time companies may not want to reveal their customer lists but only common customers to each other due to legal issues or competition. To support such a type of collaboration, a method for privacy preserving intersection where parties can only learn items in the intersection but nothing else is needed. Therefore, we first define the problem of finding the intersection of lists in the context of data warehouse while preserving the privacy called *aggregated intersection query processing* as follows:

> Let L_1, L_2,..., L_m be the lists containing secret data stored by a set of datasources $P = \{P_1, P_2, ..., P_m\}$ respectively. For each data source P_i, the problem is to find all other data sources, P_j, with $e \in L_j$ for each item $e \in L_i$ in a privacy preserving manner, i.e., if P_j does not have e in L_j, then P_j will not know $e \in L_i$.

Example 1. We illustrate the aggregated intersection problem with an example. Consider three datasources P_1, P_2 and P_3 involved in executing aggregated intersection query with customer tables T_1, T_2 and T_3 respectively as shown in Figure 3. At the end of aggregated query processing, datasources will only learn the common customers they share with other data sources but nothing else. In this example, all data sources will know 6565 exists in tables T_1, T_2, and T_3. P_1 and P_2 will also know that 8080 is common in their tables. Similarly, P_1 and P_3 will know that 7070 is common in tables T_1 and T_3. However, P_2 should not be able to know that P_1 and P_3 have 7070 in their tables. Similarly, P_3 must not know that P_1 and P_2 have 8080 in their tables.

Our solution to the above problem is based on using one-way secure cryptographic hash functions. These hash functions are widely used in many real life applications such as password protection, message authentication, and digital signatures. The examples of such hash functions include *SHA-1, MD4*, and *MD5* [12]. A simple solution to the aggregated intersection problem could use one-way hash functions and compare hashed values of items to determine whether they are the same or not. Basically, data source P_i computes the hashed list of list L_i by computing the hash value of each item in L_i. Then, it sends the hashed list to data source P_j so that it can compare the incoming hashed list with its own hashed list to find the common items in L_i and L_j. According to this scheme, in Example 1, data source P_1 uses a hash function H and sends the list of hashed values, $\{H(6565), H(7070), H(8080)\}$, to P_2 and P_3. Then, P_2 compares the hashed list with its own hashed list, $\{H(6565), H(8080)\}$, and determines that 6565 and 8080 are common. Since the hash function is a one-way hash function, P_2 will not

Costumers

SSN	Name	Surname	Address	Phone
6565	Jack	Brial	6616 K Rd. Xyz 93090 ZT	890–908–4545
7070	...	...	...	...
8080	...	...	...	...

T 1

SSN	Name	Surname	Address	Phone
6565	Jack	Brial	6616 K Rd. Xyz 93090 ZT	890–908–4545
8080				

T 2

SSN	Name	Surname	Address	Phone	E–mail
6565	Jack	Brial	6616 K Rd. Xyz 93090 ZT	890–908–4545	bj@utz.edu
7070					

T 3

Fig. 3. Illustration of Aggregated Intersection

be able to know 7070 is in L_1. This basic solution, however, suffers from the following two problems: 1) If the domain size is small, then item x whose hash value is $H(x)$ could be computed by exhaustively searching the whole domain. 2) Hash collisions might produce inaccurate results.

In the context of data warehouses, data sources usually have more information about the secret items. For example, all data sources in Example 1 keep *name, last name, phone* and *address* information as well as *SSN* (Social Security Number) of a customer in their customers tables. If all of these information is used in hashing, then the domain will become large. For instance, instead of hashing SSN, a concatenation of SSN, name, last name, phone and address could be used in hashing, i.e., $H(6565|Jack|Brial|6616KRD\ XyzZT93090|8909084545)$ could be used instead of $H(6565)$ for a customer with an SSN 6565. This method allows us to enlarge the domain size and makes exhaustive search impossible. The aggregated intersection problem is to find the common secret items in the dimensions tables in the context of data warehouse. ABACUS uses the common attributes in all of the tables to hash secret items i.e., the values of common attributes are used instead of a value of a primary key. For example, the attributes *SSN, Name, Surname, Phone,* and *Address* are common in T_1, T_2 and T_3 in Example 1. If 5 attributes each of which is 10 characters long are used in hashing, the domain size would be $28^{50} \approx 2^{250}$ which makes exhaustive search impossible.

As mentioned before, hash collisions might result in sharing a secret item which is not in the intersection. H maps values to $\mid DomH \mid$ which is assumed to be arbitrarily large compare to the intersection size. Let $N = \mid DomH \mid$; in the random oracle model, the probability that n hash values have at least one collision equals [2]: $Pr[collision] = 1 - exp(\frac{-n(n-1)}{2N})$. For 1024 bit hash values and $n = 1$ million, this probability is 10^{-295}

[2]. Thus, the solution to expand the domain size minimizes the probability of data vulnerability by exhaustive search and also helps in reducing the probability of hash collisions, and therefore, errors in the queries are significantly reduced.

3.2 Aggregated Join Queries

One of the most important query operators supported by current commercial database systems is the join operator. Privacy preserving join operations have not been previously considered in database research or in current database management systems. However, they might be needed in data sharing across private data sources. For example, a company (e.g. a hotel) might want to know the transaction details of its customers in other companies (e.g. airlines) in the market to classify them according to their transactions. For instance, a hotel can identify the customers that travel frequently and offer special promotions to them. To be able to do this, it needs to join its customers table with other companies' sales tables. Since other companies may benefit from this process, they might be willing to share transaction details. However, during this process companies are not willing to reveal any information about a customer who is not a customer of the other company as well as his/her existence. Traditional join query processing techniques cannot be used to process these queries since they do not consider privacy issues. In order to satisfy these requirements, we propose a new join operator, the *aggregated join query operator*, to be used for privacy preserving data sharing across private data warehouses. We first formally define the *aggregated join query processing* problem and then propose a solution.

The aggregated join query processing problem is formally defined as follows:

> Assume data source P_1 has a dimension table $P_1.T_d$ and data sources P_2, P_3, .., P_m have fact tables $P_2.T_f, P_3.T_f, ..., P_m.T_f$ respectively with common attribute A. Then, the goal is to compute $P_1.T_d \bowtie P_2.T_f \cup P_1.T_d \bowtie P_3.T_f \cup .... \cup P_1.T_d \bowtie P_m.T_f$ such that none of the data sources learn any extra information other than the query result. Query poser P_1 will learn only the tuples t such that $t \in P_i.T_f$ for which $t.A \in P_1.T_d.A$. In other words, P_i shares a list, L_v , of tuples in $P_i.T_f$ for each value $v \in P_i.T_f.A$ with P_1 if $\exists t \in P_1.T_d$ such that $t.A = v$, and nothing else where $i = 2, 3..., m$.

We illustrate the problem with an example. Assume the three data warehouses in Example 1 want to execute an aggregated join query. And assume P_1 poses the aggregated join query to find the aggregated join of its dimension table, *Customers Table*, with the fact tables, *Sales Table*, of the other data sources as shown in Figure 4. The problem is to provide an answer to this query without revealing any additional information. For this example, P_2 will return the tuples with SSN 6565 and 8080 in its Sales table without knowing P_1 has 7070 in its customers table. Similarly, P_3 will return all tuples with SSN 6565 and 7070 without knowing P_1 has a customer with an SSN 8080. In addition, P_1 will not be revealed the transaction details of other customers which are not in its Customers table, e.g. a customer with an SSN 9090.

ABACUS executes the aggregated join query, $\bigcup_{i=2}^{m} P_1.T_d \bowtie P_i.T_f$, in two phases: *Intersection Phase* and *Join Computation Phase*. In the intersection phase, P_1 and P_j compute the intersection of their dimension tables, $P_1.T_d \cap P_j.T_d$ with the method discussed in Section 3.1 (i.e., P_1 sends a hashed list of its customers so that P_j can know

SSN	Date	Amount
6565	7/21/2004	10
6565	9/27/2004	48
8080	1/1/2004	23

Sales Table at P_2

SSN	Date	Amount
6565	7/9/2004	23
6565	9/7/2004	84
7070	2/2/2004	79
9090	2/2/2004	92

Sales Table at P_3

SSN	Name	Surname	Address	Phone
6565	Jack	Brial	6616 K Rd. Xyz 93090 ZT	890–908–4545
7070				
8080				

Customers Table at P_1

Fig. 4. Illustration of Aggregated Join

common customers). Then, P_j sends all tuples $t \in P_j.T_f$ where $t.A \in \Pi_A(P_1.T_d \cap P_j.T_d)$ to P_1.

During the query processing, no extra useful information gets revealed. In the intersection phase, all data sources compute the intersection of the dimension tables and in the join computation phase, all data sources other than the query poser send the related tuples from their fact tables. As a result, no site gains extra useful information other than the intersection and the join results.

4 Aggregate Query Processing

The traditional aggregation operation is generally used to compute the aggregate of a list of values such as SUM, AVERAGE or MIN/MAX. One kind of privacy preserving aggregation can be thought of as computing the aggregation of values in the union of lists coming from different data sources such that each data source will only know the final aggregate but nothing else. To execute these queries, each data source can compute its local aggregate and the final aggregate can be computed in such a way that none of the data sources will know the local aggregate of other data sources (*Secure multiparty computation* or the technique described in this section can be used to compute the final aggregate value for SUM and AVERAGE). However, data sources may not be willing to execute aggregation operations over their whole data or may want to know more than the sum of the values in several lists. Therefore, there is a need for new types of aggregation queries. In this section, we will introduce *Row-Based Aggregation* and *Column-Based Aggregation* queries. We formally define *Row-Based Aggregation* queries and show how to process them in Section 4.1. Then, we will present *Column-Based Aggregation* queries and techniques to execute them efficiently in Section 4.2.

4.1 Row-Based Aggregation

Enterprises may want to know the total expenditure of a customer in the market. For example, hotels and airlines may want to classify their customers based on their travel

expenses. Without privacy concerns it is easy to perform this classification task. One of the enterprises may collect data from all enterprises and perform the computation. However, they may not be willing to reveal their value during this operation. For example, an airline company may not be willing to reveal an expenditure of a customer to other airlines since other airlines may try to attract this customer. For instance, if a customer's expenditure in company C_1 is 80, and another company C_2 knows that his/her expenditure in C_1 is 80, then C_2 can offer a new deal to this customer and try to attract him/her using this information. Although enterprises may not be willing to reveal their earnings from a customer, they may want to know the total expenditure of the customer without revealing their values. For example, these hotels and airlines may be willing to know the total expenditure of a customer in these hotels and airlines (i.e., total expenditure in the market) without revealing their earnings from this customer so that competing hotels and airlines protect their private information from each other. Since the traditional aggregation operation is not strong enough to support these needs, ABACUS proposes a new type of aggregation queries, *Row-Based Aggregation* queries, and a new technique to execute them in a privacy preserving manner in the context of data warehouses.

For the sake of this discussion, we will first define the row-based aggregation on a table with two attributes namely *Key* and *Value*. Then, we will discuss how this can be generalized to support queries in data warehouses. The *Row-Based Aggregation* query processing problem is defined as follows:

> Let T_1, T_2,..., T_m be the tables stored by a set of source peers $P = \{P_1, P_2, ..., P_m\}$ ($m \geq 3$) respectively containing a *Key* and a *Value* attributes. Each data source, P_i, would like to learn the aggregate for each $Key \in T_i$, $\sum_{j=1}^{m} Value$ $\exists [Key, Value] \in T_j$. Then, the problem is to obtain the answer of the query without revealing any additional information.

The above problem formulation is for SUM queries. We solve the above problem with some relaxation. The relaxation is that a data source with *Key* in its database can learn who else has the same *Key* (Note that this information is the same as the result of aggregated intersection). However, it is impossible to learn the *Value* associated with that *Key* at the other data warehouses. Extending our solution to support AVERAGE queries is straightforward and discussed briefly at the end of this section.

Example 2. Let us illustrate the problem with an example. Consider four companies, P_1, P_2, P_3 and P_4, that want to classify their customers according to their total expenditures from these companies. They have tables T_1, T_2, T_3 and T_4 each of which with two attributes *customer SSN* and the *amount of expenditure* as $[Key, Value]$ pairs. The contents of the tables are as follows:

$T_1 = \{[6565, 10], [7070, 20], [8080, 30]\}$
$T_2 = \{[6565, 50], [8080, 30]\}$
$T_3 = \{[6565, 10], [7070, 20], [8080, 30]\}$
$T_4 = \{[6565, 10], [7070, 20]\}$

To classify customers, one should know their total expenditures in the market. In other words, a row-based aggregation is needed for this process so that at the end of query pro-

cessing P_1, P_2, P_3 and P_4 will get the following lists respectively as an answer without knowing any additional information: $\{[6565, 80], [7070, 60], [8080, 90]\}$, $\{[6565, 80], [8080, 90]\}$, $\{[6565, 80], [7070, 60], [8080, 90]\}$, $\{[6565, 80], [7070, 60]\}$. The first item in the above first list, $[6565, 80]$ means that the customer with SSN 6565 has a total expenditure of 80 in companies P_1, P_2, P_3 and P_4. ABACUS can process row-based aggregation queries in a privacy preserving manner while revealing some information which is typically acceptable in an honest-but-curious environment. For example, each company will learn who else has a customer with the same SSN as in its list. For instance, P_1 will know that 6565 exists in all data sources, while 7070 also exists in P_3 and P_4, and 8080 exists in P_2 and P_3. During this query processing, none of the data sources will be able to learn the value of a specific key of the other data sources. For example, P_1 will not learn that 6565 has an expenditure of 50 in P_2 , but will learn that 80 is the total expenditure of 6565 in all of the companies. Note that, if only two data sources have the same key, they may not share their values with each other by rejecting aggregation on that key (because they can learn each other's value for that key). ABACUS allows users to configure their privacy policies for this kind of policy related issues and handle them efficiently. We will discuss these issues later in this section.

A simple technique to compute the sum of values (i.e., $V_1 + V_2 + V_3 + V_4$) for a specific key *Key* in four key-value pairs $[Key, V_1]$, $[Key, V_2]$, $[Key, V_3]$, and $[Key, V_4]$ residing at four different parties P_1, P_2, P_3 and P_4 respectively without revealing V_1, V_2, V_3 and V_4 could be circulating a token with a label *H(Key)*. Using secure one-way hash function can prevent others from learning *Key* if they do not have *Key*. The process consists of two circulations. During the first circulation, every party, P_i, would add its value, V_i, and a random number, r_i, and pass the token to the next party. Therefore, P_1 creates a token with a label *H(Key)* and adds $V_1 + r_1$, then passes it to P_2. The other parties follow the same protocol and pass the token to the next one. At the end of first circulation, P_1will get $V_1 + r_1 + V_2 + r_2 + V_3 + r_3 + V_4 + r_4$ for a token with a label *H(Key)*. There is no way to determine the value of a specific party during the course of the first circulation because of the random numbers added. In the second circulation, all parties subtract the random numbers they added during the first circulation. Therefore, at the end of second circulation, P_1 would have a token with a label *H(Key)* and the sum of the values for that *Key*, $V_1 + V_2 + V_3 + V_4$. Although it seems secure, this basic technique has two problems. Since this process is needed for every item in the list, using the same random number for every item in the list may result in information leakage such as the difference between two values. To prevent this information leakage, parties should use a different random number for each item in their lists. Therefore, every data source should maintain a list of random numbers it used during this process which is not scalable for large lists. Another problem is that any two of the data sources could collude and learn the value of another data source. For example, P_2 and P_4 could learn the value of P_3, V_3. In the first circulation, P_4 would pass the token with a label *H(Key)* and $V_1 + r_1 + V_2 + r_2 + V_3 + r_3 + V_4 + r_4$ to P_1 and in the second circulation, P_1 would pass $V_1 + V_2 + r_2 + V_3 + r_3 + V_4 + r_4$ to P_2 . Since P_2 and P_4 know $V_1 + r_1 + V_2 + r_2 + V_3 + r_3 + V_4 + r_4$ and $V_1 + V_2 + r_2 + V_3 + r_3 + V_4 + r_4$ they could figure out r_1, and thus V_1(Note that P_1 passed $V_1 + r_1$ to P_2 in the first circulation). Therefore, P_2 and P_4

could collude and reveal the value of P_1 without revealing their values to each other or to the other parties.

In order to compute aggregation securely, ABACUS uses Shamir's secret sharing technique, which allows one to compute any linear combination of secret values. ABACUS uses this property to perform SUM and AVERAGE queries thus computing aggregation without revealing individual values.

Shamir's Secret Sharing

Shamir's secret sharing method [13] allows a dealer D to distribute a secret value v_s among n peers $\{P_1, P_2, ..., P_n\}$, such that knowledge of any k ($k \leq n$) peers is required to reconstruct the secret. Since, even complete knowledge of $k-1$ peers cannot reveal any information about the secret, Shamir's method is information theoretically secure. Dealer D chooses a random polynomial $q(x)$ of degree $k-1$ where the constant term is the secret value, v_s, and a publicly known set of n random points. The dealer computes the share of each peer as $q(x_i)$ and sends it to peer P_i. The method is summarized in Algorithm 1.

Algorithmus 1. Shamir's Secret Sharing Algorithm

Input:
v_s: Secret value;
D: Dealer of secret v_s;
P: set of peers P_1, ..., P_n to distribute secret;
Output:
$share_1$, ..., $share_n$: Shares of secret, v_s, for each peer P_i;
Procedure:
D creates a random polynomial $q(x) = a_{k-1}x^{k-1} + ... + a_1x^1 + a_0$ with degree $k-1$ and a constant term $a_0 = v_s$.
D chooses publicly known n random points, x_1, ...x_n, such that $x_i \neq 0$.
D computes share, $share_i$, of each peer, P_i, where $share_i = q(x_i)$ and sends it to P_i.

In order to construct the secret value v_s, any set of k peers will need to share the information they have received. After finding the polynomial $q(x)$, the secret value $v_s = q(0)$ can be reconstructed. $q(x)$ can be found using Lagrange interpolation such that $p(x_i) = share_i$ where $i = 1, ..., k$. The key observation is that at least k points and the respective shares are required to determine a unique polynomial $q(x)$ of degree $k-1$.

Row-Based Aggregation in ABACUS

ABACUS executes row-based aggregation queries in three phases: *Distribution phase*, *Intermediate-Computation phase*, and *Final-Computation phase*.

Distribution Phase

After the query is posed, m data sources decide on the degree of the polynomial that is going to be used in Shamir's secret sharing (the degree of the polynomial should be greater than or equal to $m-1$). They also choose $n \geq m$ random values $X = \{x_1, ..., x_n\}$. Without loss of generality, we will use a polynomial of degree of $m-1$ and $n = m$ in our setting. Each data source P_i has a list of *Key-Value* pairs, $L_i = \{[K_1, V_1], ..., [K_{|L_i|}, V_{|L_i|}]\}$; P_i creates m shares from L_i, $share(L_i, P_1)$, ... ,$share(L_i, P_m)$, one for each of the data sources P_1 through P_m respectively (including itself).

P_i creates the shares by applying a one-way hash function and Shamir's secret sharing algorithm to each of the elements in L_i. For every element $[Key, Value]$ in L_i, P_i computes the share of data source P_j, $sh([Key, Value], P_j) = [H(Key), q(x_j)]$, using a hash function H and Algorithm 1 with $q(x)$ and X (the constant term in $q(x)$ will be replaced by the secret value, $Value$, to compute $q(x)$ in Shamir's secret sharing). Therefore, the list of shares of data source P_j from L_i is $share(L_i, P_j) = \{sh([K_1, V_1], P_j), ..., sh([K_{|L_i|}, V_{|L_i|}], P_j)\}$. Then, P_i sends $share(L_i, P_j)$ to the data source P_j. Note that P_i keeps its share, $share(L_i, P_i)$, for itself and since using the same $q(x)$ would results in information leakage, a random polynomial is used for each of the item in the list. Therefore, random polynomials q_1 through $q_{|L_i|}$ are used for the items 1 through $\mid L_i \mid$ in L_i.

In Example 2, assume P_1 with a list, $L_1 = \{[6565, 10], [7070, 20], [8080, 30]\}$ and four data sources decided on four random points $X = \{27, 65, 90, 123\}$. Since there are four data sources, a polynomial $q(x)$ of a degree three would be used with a hash function H while calculating the share of each data source. As a first step, P_1 chooses three random polynomials for each item in its list: $q_1(x) = 2x^3 - 2x^2 + 10$, $q_2(x) = x^3 - 5x^2 + 20$, $q_3(x) = x^3 - 13x^2 + 30$. Observe that the constant term of polynomial q_i is value of the ith item in L_1 and q_i is used for the ith item in L_1. Then, the shares of key-value pairs in L_1 for data source P_2 are calculated as follows:

$sh([6565, 10], P_2) = [H(6565), q_1(x_2)] = [H(6565), q_1(65)]$
$sh([7070, 20], P_2) = [H(7070), q_2(x_2)] = [H(7070), q_2(65)]$
$sh([8080, 30], P_2) = [H(8080), q_3(x_2)] = [H(6565), q_3(65)].$

Therefore, the share list for P_2, $share(L_1, P_2)$, is: $share(L_1, P_2) = \{[H(6565), q_1(65)], [H(7070), q_2(65)], [H(8080), q_3(65)]$. Similarly, other data sources' share lists are computed and are sent to them. P_1 would keep $share(L_1, P_1)$ for itself and sends $share(L_1, P_2)$, $share(L_1, P_3)$, and $share(L_1, P_4)$ to P_2, P_3, and P_4 respectively.

Distribution phase at data source P_i is summarized in Algorithm 2.

Algorithmus 2. Distribution Phase

Input:
X: Random Values $X = \{x_1, .., x_m\}$;
H: Secure one-way hash function
L_i: Secret list of *Key-Value* pairs at data source P_i;
Output:
$share(L_i, P_1), ..., share(L_i, P_m)$: Shares of secret list, L_i, for each data source P_j;
Procedure:
for Each secret *Key-Value* pair $[Key, V_s] \in L_i$ **do**
  Find share $sh([Key, V_s], P_j)$ of each data source P_j for $[Key, V_s]$ with Algorithm 1 using a random polynomial $q(x)$ where $q(x) = a_{k-1}x^{k-1} + ... + V_s$ and the hash function H such that $sh([Key, V_s], P_j) = [H(Key), q(x_j)]$.
  Add $sh([Key, V_s], P_j)$ into $share(L_i, P_j)$.
end for

for For each data source P_j **do**
  Send $share(L_i, P_j)$ to data source P_j
end for

Intermediate-Computation Phase

After receiving their shares from the data sources, $P_1, ..., P_m$, each data source, P_i, calculates intermediate result lists, $IR(L_1, P_i)$,...,$IR(L_m, P_i)$, corresponding to the lists

$share(L_1, P_i)$,...,$share(L_m, P_i)$ respectively. The kth element of $share(L_j, P_i)$ is a key-value pair i.e., $share(L_j, P_i)[k] = [H(Key), Value^*]$ which is the share of P_i from the $[Key, Value]$ pair in L_j ($share(L_j, P_i)[k][1] = H(Key)$ and $share(L_j, P_i)[k][2] = Value^*$). P_i computes the intermediate result lists as follows:

$IR(L_j, P_i)[k][1] = share(L_j, P_i)[k][1]$

$IR(L_j, P_i)[k][2] = \sum_{h=1}^{m}(share(L_h, P_i)[g][2]$ s.t. $\exists\ g$ where $share(L_h, P_i)[g][1] = IR(L_j, P_i)[k][1])$, i.e., $INTER - RES_i$.

In Example 2, P_1 would have lists $share(L_1, P_1)$, $share(L_2, P_1)$, $share(L_3, P_1)$ and $share(L_4, P_1)$ where

$share(L_1, P_1) = \{[H(6565), 120], [H(7070), 320], [H(8080), 400]\}$

$share(L_2, P_1) = \{[H(6565), 100], [H(8080), 600]\}$

$share(L_3, P_1) = \{[H(6565), 3500], [H(7070), 900], [H(8080), 90]\}$

$share(L_4, P_1) = \{[H(6565), 110], [H(7070), 80]\}$

Then, in the intermediate computation phase, P_1will compute $IR(L_1, P_1)$ $IR(L_2, P_1)$, $IR(L_3, P_1)$ and $IR(L_4, P_1)$ and send them to data sources P_1, P_2, P_3 and P_4 respectively. For example, $IR(L_3, P_1)$ is computed as follows: Since $H(6565)$ exists in all lists, the values associated with it, 120, 100, 3500 and 110 in $share(L_1, P_1)$ through $share(L_4, P_1)$ respectively, are added. Therefore, $IR(L_3, P_1)[1] = H(6565)$ and $IR(L_3, P_1)[2] = 120{+}100{+}3500{+}110 = 3830$. The same calculation is performed for all items in the list resulting in $IR(L_3, P_1) = \{[H(6565), 3830], [H(7070), 4900], [H(8080), 1090]\}$.

The intermediate computation process at data source P_i is summarized in Algorithm 3.

Algorithmus 3. Intermediate Computation Phase

Input:
$Share_L$: Set of share lists, $Share_L = \{share(L_1, P_i), .., share(L_m, P_i)\}$;
Output:
Set of intermediate result lists $\{IR(L_1, P_i), .., IR(L_m, P_i)\}$ to send back to the data sources $P = \{P_1, ..., P_m\}$ respectively;
Procedure:
for each list $share(L_k, P_i) \in Share_L$ **do**
  for $j = 1; j \leq |share(L_k, P_i)|$ **do**
    $IR(L_k, P_i)[j][1] = share(L_k, P_i)[j][1]$
    if $share(L_k, P_i)[j][1] = share(L_l, P_i)[o][1]$ such that $\exists\ l$ and o where $l \leq m$ and $1 \leq o \leq |share(L_l, P_i)|$ **then**
      $IR(L_k, P_i)[j][2] = IR(L_k, P_i)[j][2] + share(L_l, P_i)[o][2]$
    end if
  end for
end for
Send $IR(L_1, P_i), ..., IR(L_m, P_i)$ to $P_1, ..., P_m$ respectively

Final-Computation Phase

In the final computation phase, data source P_i retrieves its intermediate result lists, $IR(L_i, P_1)$,...,$IR(L_i, P_m)$ from all m data sources. Since all data sources compute the sum of their shares for a specific *Key*, the kth entry of an intermediate list contains $H(Key)$ and the sum of shares for *Key*. Therefore, for a *Key-Value* pair in L_i, the corresponding entry k in the intermediate result lists are: $IR(L_i, P_1)[k] = [H(Key), INTER - RES_1]$, $IR(L_i, P_2)[k] = [H(Key), INTER - RES_2]$, ..., $IR(L_i, P_m)[k] = [H(Key), INTER - RES_m]$.

In the final computation phase, data sources calculate the sum for each *Key* from the m intermediate results. Since all data sources use a random polynomial degree of $m-1$ and compute the shares of all data sources using m points, $X = \{x_1, x_2, .., x_m\}$, these result in a polynomial $P(x) = a_{m-1}x^{m-1} + ... + a_1x^1 + a_0$ where constant term, a_0, is the sum of the values for *Key* and $P(x_i) = INTER - RES_i$. The coefficients of $P(x)$ and thus the sum of the values could be computed because the values of $P(x)$ are known at m different points ($P(x_i) = INTER - RES_i$).

Proof of Correctness
A data source P_j constructs a random polynomial $a_j x_i^{m-1} + b_j x_i^{m-2} + ... + Value$ to hide the secret values for each $[Key, Value]$ pair. After generating this random polynomial, it computes the share of P_i as $(H(Key), [a_{P_j} x_i^{m-1} + b_{P_j} x_i^{m-2} + ... + v_{P_j}])$ for each secret key-value pair, where $v_{P_j} = Value$ and sends the shares of the other data sources. After P_i receives the shares from all m data sources, it sends the sum of values which have the same key. Without loss of generality, assume l of the m data sources have the same *Key* with the secret values v_1 through v_l respectively. Then the sum for that *Key* is in the following form:

$$\begin{array}{c} a_1 x_i^{m-1} + b_1 x_i^{m-2} ... + v_1 + \\ a_2 x_i^{m-1} + b_2 x_i^{m-2} ... + v_2 + \\ \vdots \\ a_l x_i^{m-1} + b_l x_i^{m-2} ... + v_l \end{array}$$

Therefore, P_i sends its results $INTER - RES_i = (a_1 + a_2 + ... + a_l)x_i^{m-1} + ... + SUM$ to the parties having *Key* in their lists, where SUM is the sum of the secret values ($SUM = v_1 + v_2 + ... + v_l$) for the values that have the same key, *Key*.

Each data source receives m results from each of the data sources (including itself) for each key in its *[Key-Value]* list:

$$\begin{array}{c} INTER - RES_1 = (a_1 + a_2 + ... + a_l)x_1^{m-1} + ... + SUM \\ INTER - RES_2 = (a_1 + a_2 + ... + a_l)x_2^{m-1} + ... + SUM \\ \vdots \\ INTER - RES_m = (a_1 + a_2 + ... + a_l)x_m^{m-1} + ... + SUM \end{array}$$

Since $X = \{x_1, x_2, .., x_m\}$ is known by all data sources, there are a total of m unknown coefficients including SUM and m equations in the above system of equations. Therefore, SUM can be derived by using the above equations. The data source, P_j, cannot know the value of the other data sources, since the coefficients of the polynomials used by other data sources are not known by P_j.

For the average query, P_i sends $INTER - RES_i = [(a_1 + a_2 + ... + a_l)\ x_i^{m-1} + ... + SUM]/l$ where $INTER - RES_i = \frac{(a_1+a_2+...+a_l)}{l} x_i^{m-1} + ... + AVG)$ and $AVG = \frac{v_1+v_2+...+v_l}{l}$. Therefore, each data source receives m results:

$$\begin{array}{c} INTER - RES_1 = \frac{(a_1+a_2+...+a_l)}{l} x_1^{m-1} + ... + AVG \\ INTER - RES_2 = \frac{(a_1+a_2+...+a_l)}{l} x_2^{m-1} + ... + AVG \\ \vdots \\ INTER - RES_m = \frac{(a_1+a_2+...+a_l)}{l} x_m^{m-1} + ... + AVG \end{array}$$

Again, since $X = \{x_1, x_2, .., x_n\}$ is known by the data sources, there are m unknown coefficients including AVG and m equations and thus, AVG can be derived from the above equations.

Row-Based Aggregation in Data Warehouses
After the query is posed, data sources create lists of *[Key,Value]* pairs using their fact and dimension tables so that row-based aggregation can be performed over them with the above technique. All information in the dimension table about a tuple in the fact table is used to form a *Key* for that tuple. The tuple from a fact table is added into the list as *[Key,Value]* pairs where *Value* is the value associated with that tuple. For example, data source P_2 in Figure 4 creates *[Key-Value]* pairs as follows: for a tuple with SSN 6565, it retrieves other information about 6565 from the customers table such as name, surname and address. Then, it combines those information to create the *Key* for this tuple and the amount is used as the *Value*.

Properties of the Algorithm
Data sources use a one-way hash function to hide *Key*, and thus all of the data sources will learn $H(Key)$. Only those data sources which have *Key* would be able to know *Key* and its existence at data source P_i. In addition, P_i uses Shamir's secret sharing to hide the value associated with *Key* from other data sources. It uses a polynomial degree of $m-1$ and m random points to compute shares of the m data sources. Then, it keeps one of these shares for itself and sends the remaining $m-1$ shares to the other parties. Since all of the m shares are needed to reveal the secret value in Shamir's secret sharing method, the other data sources would not be able to compute the value, even if they combine their shares coming from P_i.

In general, for any *Key* at any data source P_j, any data source P_i can prevent execution of aggregation for that *Key*. Since one of the m shares is sent to P_i, P_i can prevent aggregation on *Key* by not sending the intermediate result to the other data sources. Therefore, other data sources would not be able to learn SUM for *Key*. Using this property, ABACUS allows data sources to control sharing the value of *Key* with other data sources. This might be needed since if only two data sources have *Key*, performing row-based aggregation will result in revealing the values to these two data sources (the result is the sum of the two values, and since these data source know their values, they can figure out the other value from the result). Note that, if *Key* exists in only one data source, then the owner can protect it from other data sources This can easily be done by preventing aggregation on *Key*. In addition to these, data sources cannot figure out something from their shares using the distribution of values since they are random values (i.e., a random polynomial is used for each item in the list to compute the shares).

4.2 Column-Based Aggregation

Enterprises might want to know the size of the market and some statistical information about the market where they compete. In addition, they might be interested in expenditures of their customers such as the ratio of their expenditures in their companies to

their total expenditures in the market. In other words, a company might want to know how much it satisfies the needs of customers. Therefore, companies might be willing to collaborate to perform these kinds of operations however, they might not want to reveal extra information, for example a company might not want to reveal how much it satisfies the needs of its customers. One way to compute the market size in a privacy preserving manner is to aggregate the expenditures of all customers in that market. Formally, data sources $P_1, P_2, .., P_m$ might want to know sum of their local sums $LS_1, LS_2, ..., LS_m$ respectively, and the global sum $GS = LS_1 + ... + LS_m$, without revealing their local sums. This problem could be solved with the technique discussed in Section 4.1. However, in a competitive environment it is unrealistic to expect enterprises to share their local sums. For example, a big company with 1000 customers might not be willing to share its local sum which is the sum of its 1000 customers with a small company with 10 customers. Instead, it might want to collaborate for the common customers to compute their total expenditures, so that both companies could learn how much they satisfy the needs of their customers. However, during this process they do not want to reveal any additional information. In order to satisfy these needs, we introduce *column-based aggregation*.

Formally, the column-based aggregation query processing problem is defined as follows:

> Let $T_1, T_2, ..., T_m$ be the tables stored by a set of data warehouses $P = \{P_1, P_2, ..., P_m\}$ $(m \geq 3)$ respectively containing a key and a value field. The data source P_i would like to learn the aggregation of values for all *Keys* in T_i, i.e., $\sum_{\forall Key \in T_i} \sum_{k=1}^{m} (Value\ s.t.\ \exists [Key, Value] \in T_k \wedge \exists Key \in T_i)$. Then the problem is to obtain the answer by only providing the aggregation result to P_i while revealing only the common *Keys* to other data sources.

The goal of the query processing is to compute *column-based aggregation* such that the data source posing the query, P_i, would only know the result of the query, $\sum_{\forall Key \in T_i} \sum_{k=1}^{m} (Value\ s.t.\ \exists [Key, Value] \in T_k \wedge \exists Key \in T_i)$, while other data sources would only know the *Keys* in T_i if they have those *Keys*. The query processing consists of three steps:

- *Intersection Phase:* Data source P_i sends the list of hash values of *Keys* in T_i. On P_j receiving this list, P_j computes the common keys in tables T_i and T_j (by hashing its keys in T_j and comparing them with the list coming from P_i).
- *Local Aggregation for Intersection Phase:* Data source P_j, computes the local sum of values, *local sum*, for the common keys between P_i and P_j. Formally, the local sum, LS_j, at data source P_j is: $LS_j = \sum_{\forall Key \in T_i} (Value\ s.t.\ \exists [Key, Value] \in T_j \wedge \exists Key \in T_i)$.
- *Global Aggregation Phase:* Data sources compute the *global sum*, GS, which is the sum of *local sum* of m data sources. They compute $GS = \sum_{i=1}^{m} LS_i$ without revealing the local sums with the technique discussed in Section 4.1 (One could think of the data sources, $P_1, ..., P_m$, have the following *[Key,Value]* pairs $[P_i, LS_1]$,...,$[P_i, LS_m]$ respectively and they want to compute the *row-based aggregation* for the key P_i, which is the global sum).

The proposed query processing method computes column-based aggregation queries correctly. The answer to the column-based aggregation query for data source P_i is $\sum_{\forall Key \in T_i} \sum_{k=1}^{m} (Value\ s.t.\ \exists [Key, Value] \in T_k \wedge \exists Key \in T_i)$. The proposed technique computes the local sum at each data source in *local aggregation for intersection phase* where $LS_j = \sum_{\forall Key \in T_i} (Value\ s.t.\ \exists [Key, Value] \in T_j \wedge \exists Key \in T_i)$. Then, in *global aggregation phase* the sum of all the local aggregations are computed as answer which is $\sum_{k=1}^{m} LS_k$, i.e., $\sum_{\forall Key \in T_i} \sum_{k=1}^{m} (Value\, s.t.\, \exists [Key, Value] \in T_k \wedge \exists Key \in T_i)$.

At the end of the query processing other data sources will only know their common *Keys* with the query poser P_i and P_i will only know the result of the column based aggregation query result but nothing else. After *intersection phase*, the other data sources will know the common elements between P_i and them but nothing else, since one-way hash function is used to hide *Keys*. During *local aggregation for intersection phase*, the data sources would compute their local aggregates. Then, in the *global aggregation phase*, they compute the sum of the local aggregations without revealing their local aggregations to anybody with the *row-based aggregation*. Therefore, P_i would only know the global aggregation result, which is column based aggregation result but not the local aggregations. And the other data sources would not know any other local aggregation and the global aggregation results unless $P_i$wants them to know (Note that if P_i does not send its intermediate result to other data sources, they cannot compute the global sum in the row-based aggregation in Section 4.1).

5 Analytical Evaluation

In this section, we compute the query response times for the proposed query processing techniques. The query responses time for intersection and join queries are studied in Section 5.1. Then, the query execution costs of row-based aggregation and column-based aggregation queries are calculated in Section 5.2. Finally, we show the query response times of the queries over a sample scenario to demonstrate the scalability of our technique in Section 5.3.

5.1 Cost of Intersection and Join Query Processing

Data source P_i hashes its list and sends to m data sources. Then, it compares its list with other datasources to find the intersection. Therefore, the computation cost is the cost of hashing the list and the cost of comparisons. Let C_h be the cost of hashing a single item and every hashed word is b bits long. The computation time for hashing is: $C_h \times |L_i|$. The number of comparisons to compare the hashed list, L_i, with the other lists coming from other data sources is (assuming lists are sorted) less than $(m-1) \times |L_i|$ without loss of generality assume L_i is the longest list. The time needed for this comparison is: $\frac{(m-1) \times |L_i|}{CPU\ Speed}$ seconds. Therefore, total computation time is: $C_h \times |L_i| + \frac{(m-1) \times |L_1|}{CPU\ Speed}$ The communication time is the sum of the time needed to send its own hashed list and the time to receive the $m-1$ hashed lists from other data sources. Therefore, total communication time is: $\frac{b \times (|L_1| + ... + |L_m|)}{Bandwidth}$. The query response time, the sum of the computation and communication cost, is:

$$C_h \times |L_i| + \frac{(m-1) \times |L_i|}{CPU\ Speed} + \frac{b \times (|L_1| + ... + |L_m|)}{Bandwidth}$$

In the aggregated join, the first step is aggregated intersection. After this first step, data source P_j sends the related tuples to P_i. The query response time is sum of the cost of aggregated intersection and the cost of sending related tuples. Therefore, the query response time for aggregated join is:

$$\text{The cost of intersection} + \frac{m \times |L| \times v}{Bandwidth}$$

where v is the size of a tuple $t \in T_R$.

5.2 Cost of Aggregation Query Processing

The Cost of Row-Based Aggregation Query Processing
In the distribution phase, data sources compute the hash value of keys and the shares of m data sources. Therefore, the computation cost is $\frac{m \times |L|}{CPU speed} + C_h \times |L|$. The communication cost is sending these shares to other data sources and receiving shares from other data sources, which is $\frac{2 \times m \times |L| \times b}{Bandwidth}$, where b is the size of a *Key-Value* pair.

In the local aggregation phase, the computation cost is scanning all lists and adding the values for a specific key (computation of intermediate result lists). The amount of addition is less than $m \times |L|$. Thus the cost of computation in the local aggregation phase is $\frac{m \times |L|}{CPU speed}$ (assuming that lists are sorted and are of the same size). After this computation, P_i sends intermediate results lists to m data sources and receive its intermediate result lists from m data sources. The communication cost for this operation is $\frac{2 \times m \times |L| \times b}{Bandwidth}$ (note that the size of intermediate lists is equal to the size of lists).

In the final aggregation phase, P_i solves an equation system for each element in the list. Thus, the computation time is $|L_i| \times C_{eq}$, where C_{eq} is the cost of solving an equation with m unknowns.

The query response time for row-based aggregation query is (without loss of generality, assume all lists are size of $|L|$):

$$\approx |L| \times C_h + \frac{4 \times m \times b \times |L|}{Bandwidth} + \frac{2 \times m \times |L|}{CPU speed} + |L| \times C_{eq}.$$

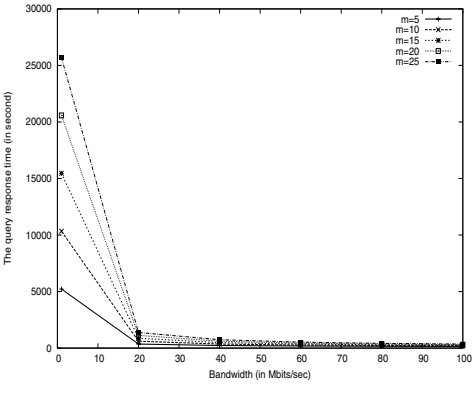

Fig. 5. The Query Response Time for Intersection Queries

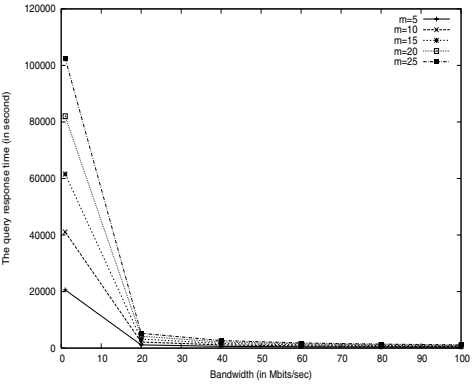

Fig. 6. The Query Response Time for Row-based Aggregation Queries

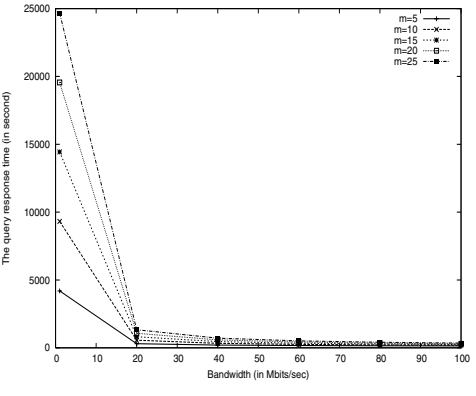

Fig. 7. The Query Response Time for Column-based Aggregation Queries

The Cost of Column-Based Aggregation Query Processing

The column-based aggregation query processing consists of three phases: 1) intersection phase 2) local aggregation for intersection 3) global aggregation. In the intersection phase, data source P_i sends its hashed lists to m data sources. The communication and computation cost for this phase is:

$$C_h \times |L_i| + \frac{m \times b \times |L_i|}{Bandwidth}$$

The cost of computation in local aggregation for intersection phase is $\frac{|L_i|}{CPUspeed}$ (data sources calculates the sum of values in the intersection). Remember that there is no communication in this phase. The cost of global aggregation phase is negligible since the cost of the summation of m values using m parties is negligible in this context. Therefore, the cost of *column-based aggregation* query processing is:

$$\approx C_h \times |L_i| + \frac{m \times b \times |L_i|}{Bandwidth} + \frac{|L_i|}{CPUspeed}.$$

5.3 Query Response Times over a Sample Scenario

We demonstrate the query response time of ABACUS for intersection and row-based and column based aggregation queries over a sample scenario to show that it is scalable and efficient. We compute the response times for queries in an environment where m data warehouses each of which with a dimension table and a fact table size of 1 million. We execute the queries over these data warehouses by varying the bandwidth and the number of data warehouses involved, m. Figures 5, 6, and 7 show the query response time for intersection, row-based aggregation and column-based aggregation queries. During these calculations we take the size of key-value pair, b, as 1024 bits, the cost of hashing, C_h, as 10^{-4} [5] seconds and the cost of solving an equation, C_{eq}, as 10^{-5} seconds (the time needed to solve an equation system with 20 unknowns in Matlab). The analytical evaluations and the results over the sample scenario demonstrate that ABACUS is scalable in terms of the number of parties participating in queries and the cost is increasing linearly with the number of parties involved. In addition as results show that the query processing is communication intensive operation since ABACUS uses light-weight computations.

6 Conclusion

In this paper, we propose a distributed middleware, ABACUS, to perform intersection, join and aggregation queries over multiple private data warehouses in a privacy preserving manner. In addition to this, we present new types of aggregation queries which are needed for privacy preserving data sharing. Analytical evaluations demonstrate that the proposed scheme is efficient and scalable.

References

1. G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, N. Mishra, R. Motwani, U. Srivastava, D. Thomas, J. Widom, and Y. Xu. Enabling privacy for the paranoids. In *Proc. of the 30th Int'l Conference on Very Large Databases VLDB*, pages 708–719, Aug 2004.
2. R. Agrawal, A. Evfimievski, and R. Srikant. Information sharing across private databases. In *Proc. of the 2003 ACM SIGMOD international conference on on Management of data*, pages 86–97, 2003.
3. S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano. Semantic integration of heterogeneous information sources. *Data Knowl. Eng.*, 36(3):215–249, 2001.
4. U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. In *IEEE Transactions on Software Engineering*, volume 10, pages 628–644, 1984.
5. P. Ganesan, R. Venugopalan, P. Peddabachagari, A. Dean, F. Mueller, and M. Sichitiu. Analyzing and modeling encryption overhead for sensor network nodes. In *Proc. of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 151–159. ACM Press, 2003.
6. O. Goldreich. Secure multi-party computation. *Working Draft*, jun 2001.
7. L. M. Haas, R. J. Miller, B. Niswonger, M. T. Roth, P. M. Schwarz, and E. L. Wimmers. Transforming heterogeneous data with database middleware: Beyond integration. *IEEE Data Engineering Bulletin*, 22(1):31–36, 1999.

8. A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. of the 19th ICDE*, pages 505–516, 2003.
9. A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *Proc. of the 2003 ACM SIGMOD*, pages 325–336, 2003.
10. M. W. N. Jefferies, C. Mitchell. A proposed architecture for trusted third party services. *Cryptography Policy and Algorithms Conference*, July 1995.
11. M. Naor and K. Nissim. Communication preserving protocols for secure function evaluation. In *ACM Symposium on Theory of Computing*, pages 590–599, 2001.
12. Secure Hash Standart. http://www.itl.nist.gov/fipspubs/fip180-1.htm.
13. A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.