在数据库服务提供者模型中的加密数据上执行SQL

Executing SQL over Encrypted Data in the Database-Service-Provider Model

Hakan Hacıgümüş[§,1] Bala Iyer[2] Chen Li[1] Sharad Mehrotra[1]

[1]Department of Information and Computer Science, University of California, Irvine, CA 92697, USA
hakanh@acm.org, {chenli,sharad}@ics.uci.edu
[2]IBM Silicon Valley Lab., San Jose, CA 95141, USA, balaiyer@us.ibm.com

ABSTRACT

Rapid advances in networking and Internet technologies have fueled the emergence of the "software as a service" model for enterprise computing. Successful examples of commercially viable software services include rent-a-spreadsheet, electronic mail services, general storage services, disaster protection services. "Database as a Service" model provides users power to create, store, modify, and retrieve data from anywhere in the world, as long as they have access to the Internet. It introduces several challenges, an important issue being data privacy. It is in this context that we specifically address the issue of data privacy.

There are two main privacy issues. First, the owner of the data needs to be assured that the data stored on the service-provider site is protected against data thefts from outsiders. Second, data needs to be protected even from the service providers, if the providers themselves cannot be trusted. In this paper, we focus on the second challenge. Specifically, we explore techniques to execute SQL queries over encrypted data. Our strategy is to process as much of the query as possible at the service providers' site, without having to decrypt the data. Decryption and the remainder of the query processing are performed at the client site. The paper explores an algebraic framework to split the query to minimize the computation at the client site. Results of experiments validating our approach are also presented.

1. INTRODUCTION

The Internet has made it possible for all computers to be connected to one another. The influence of transaction-processing systems and the Internet ushered in the era of e-business. The Internet has also had a profound impact on the software industry. It has facilitated an opportunity to provide software usage over the Internet, and has led to a new category of businesses called "application service providers" or ASPs. ASPs provide worldwide customers the privilege to use software over the Internet. ASPs are staffed by experts in the art of putting together software solutions, using a variety of software products, for familiar business services such as payroll, enterprise resource planning, and customer-relationship marketing. ASPs offer their services over the Internet to small and large worldwide organizations. Since fixed costs are amortized over a large number of users, there is the potential to reduce the service cost even after possibly increased telecommunications overhead.

It is possible to provide storage and file access as services. The natural question is the feasibility of providing the next value-add layer in data management. From the business perspective, database as a service inherits all the advantages of the ASP model, indeed even more, given that a large number of organizations have their own DBMSs. The model allows organizations to leverage hardware and software solutions provided by the service providers, without having to develop them on their own. Perhaps more importantly, it provides a way for organizations to share the expertise of database professionals, thereby cutting the people cost of managing a complex information infrastructure, which is important both for industrial and academic organizations [15].

From the technological angle, the model poses many significant challenges foremost of which is the issue of *data privacy* and *security*. In the database-service-provider model, user data resides on the premises of the database-service provider. Most corporations view their data as a very valuable asset. The service provider would need to provide sufficient security measures to guard data privacy. At least two data-privacy challenges arise. The first challenge is: how do service providers protect themselves from theft of customer data from hackers that break into their site and scan disks? Encryption of stored data is the straightforward solution, but not without challenges. Trade-offs need to be made regarding encryption techniques and the data granularity for encryption. This first challenge was examined by Hacıgümüş et al. [6]. It was found that hardware encryption is superior to software encryption. Encrypting data in bulk reduced the per-byte encryption cost significantly, exposing to startup overheads. Encrypting by row was found preferable to encrypting by field for queries from the TPC-H benchmark [14].

[§] Supported in part by an IBM Ph.D. Fellowship. This work was performed while the author was at IBM.

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA

The second challenge is that of "total" data privacy, which is more complex since it includes protection from the database provider. The requirement is that encrypted data may not be decrypted at the provider site. A straightforward approach is to transmit the requisite encrypted tables from the server (at the provider site) to the client, decrypt the tables, and execute the query at the client. But this approach mitigates almost every advantage of the service-provider model, since now primary data processing has to occur on client machines. It will become clear later, for a large number of queries such as selections, joins, and unions, much of the data processing can be done at the server, and the answers can be computed with little effort by the client.

Our proposed system, whose basic architecture and control flow are shown in Figure 1, is comprised of three fundamental entities. A *user* poses the query to the client. A *server* is hosted by the service provider who stores the encrypted database. The encrypted database is augmented with additional information (which we call the index) allows certain amount of query processing to occur at the server without jeopardizing data privacy. A *client* stores the data at the server. Client[1] also maintains *metadata* for translating user queries to the appropriate representation on the server, and performs post-processing on server query results. Based on the auxiliary information stored, we develop techniques to split an original query over unencrypted relations into (1) a corresponding query over encrypted relations to run on the server, and (2) a client query for post-processing results of the server query. We achieve this goal by developing an algebraic framework for query rewriting over encrypted representation. Finally, we explore the feasibility and effectiveness of our approach by testing the performance of our strategy over numerous queries. Our results show that privacy from service providers can be achieved with reasonable overhead establishing the feasibility of the model.

There is previous work in different research areas some of which are related to our work. Search on encrypted data [2], where only keyword search is supported, and doing arithmetic over encrypted data [10] have been studied in the literature. However functionalities provided by those are very limited and insufficient in executing complex SQL queries over encrypted data.

The rest of the paper is organized as follows. Section 2 presents how data is encrypted and stored on the server. Section 3 discusses how a condition in a query is translated to a condition on the encrypted data at the server. In Section 4 we describe how individual relational operators such as selection, join, set difference, and group by are implemented. Section 5 shows how to rewrite a query by splitting it into a server query and a client query, such that the computation at the client is reduced. Section 6 gives our experimental results on queries from the TPC-H benchmark. We conclude the paper in Section 7.

2. RELATION ENCRYPTION AND STORAGE MODEL

Before we discuss techniques for query processing over encrypted data, let us first discuss how the encrypted data is stored at the server.

For each relation $R(A_1, A_2, \ldots, A_n)$, we store on the server an encrypted relation:

$$R^S(etuple, A_1^S, A_2^S, \ldots, A_n^S)$$

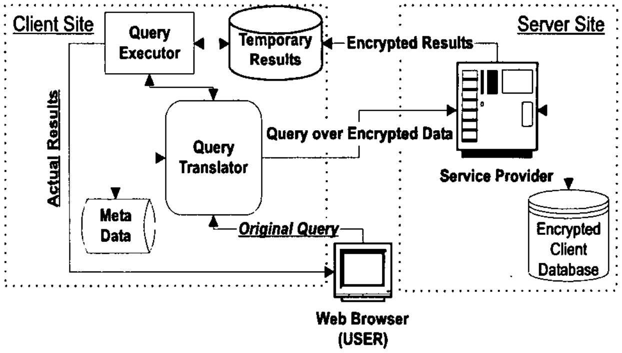


Figure 1: The service-provider architecture.

where the attribute *etuple* (We will explain how *etuple* is defined in Section 2.4.) stores an encrypted string that corresponds to a tuple in relation R.[2] Each attribute A_i^S corresponds to the index for the attribute A_i that will be used for query processing at the server. For example, consider a relation *emp* below that stores information about employees.

eid	ename	salary	addr	did
23	Tom	70K	Maple	40
860	Mary	60K	Main	80
320	John	50K	River	50
875	Jerry	55K	Hopewell	110

The *emp* table is mapped to a corresponding table at the server:

$$emp^S(etuple, eid^S, ename^S, salary^S, addr^S, did^S)$$

It is only necessary to create an index for attributes involve in search and join predicates. In the above example, if we knew that there would be no query that involves attribute *addr* in either a selection or a join, then the index on this attribute need not be created. Without loss of generality, we assume that an index is created over each attribute of the relation.

2.1 Partition Functions

We explain what is stored in attribute A_i^S of R^S for each attribute A_i of R. For this purpose, we will need to develop some notations. We first map the domain of values ($\mathcal{D}_i$) of attribute $R.A_i$ into partitions $\{p_1, \ldots, p_k\}$, such that (1) these partitions taken together cover the whole domain; and (2) any two partitions do not overlap. Formally, we define a function *partition* as follows:

$$partition(R.A_i) = \{p1, p2, \ldots, p_k\}$$

As an example, consider the attribute *eid* of the *emp* table above. Suppose the values of domain of this attribute lie in the range $[0, 1000]$. Assume that the whole range is divided

[1] Often the client and the user might be the same entity.

[2] Note that we could alternatively have chosen to encrypt at the attribute level instead of the row level. Each alternative has its own pros and cons. We point the interested readers to [6] for a detailed description. The rest of this paper assumes encryption is done at the row level.

into 5 partitions[3]: [0, 200], (200, 400], (400, 600], (600, 800], and (800, 1000]. That is:

$$partition(emp.eid) = \{[0,200], (200,400], (400,600], (600,800], (800,1000]\}$$

Different attributes may be partitioned using different partition functions. It should be clear that the partition of attribute A_i corresponds to a splitting of its domain into a set of buckets. Any histogram-construction technique, such as MaxDiff, equi-width, or equi-depth [9], could be used to create partitioning of attributes. In the examples used to explain our strategy, for simplicity, we will assume the equi-width partitioning. Extension of our strategy to other partitioning methods is relatively straightforward, though it will require changes to some of the notations developed. For example, unlike equi-width case where a value maps to only a single histogram bin, in equi-depth it may map to multiple buckets. Our notation assumes that each value maps to a single bucket. In the experimental section, besides using the equi-width we will also evaluate our strategy under the equi-depth partitioning.

In the above example, an equi-width histogram was illustrated. Note that when the domain of an attribute corresponds to a field over which ordering is well defined (e.g., the *eid* attribute), we will assume that a partition p_i is a continuous range. We use $p_i.low$ and $p_i.high$ to denote the lower and upper boundary of the partition, respectively.

2.2 Identification Functions

Furthermore, we define an identification function called *ident* to assign an identifier $ident_{R.A_i}(p_j)$ to each partition p_j of attribute A_i. Figure 2 shows the identifiers assigned to the 5 partitions of the attribute *emp.eid*. For instance, $ident_{emp.eid}([0,200]) = 2$, and $ident_{emp.eid}((800,1000]) = 4$.

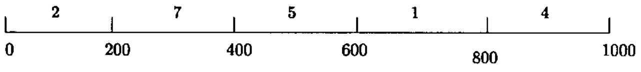


Figure 2: Partition and identification functions of *emp.eid*.

The *ident* function value for a partition is unique, that is, $ident_{R.A_i}(p_j) \neq ident_{R.A_i}(p_l)$, if $j \neq l$. For this purpose, a collision-free hash function that utilizes properties of the partition may be used as an *ident* function. For example, in the case where a partition corresponds to a numeric range, the hash function may use the start and/or end values of a range.

2.3 Mapping Functions

Given the above partition and identification functions, we define a mapping function $Map_{R.A_i}$ that maps a value v in the domain of attribute A_i to the identifier of the partition to which v belongs: $Map_{R.A_i}(v) = ident_{R.A_i}(p_j)$, where p_j is the partition that contains v.

In the example above, the following table shows some values of the mapping function for attribute *emp.eid*. For instance, $Map_{emp.eid}(23) = 2$, $Map_{emp.eid}(860) = 4$, and $Map_{emp.eid}(875) = 4$.

eid value v	23	860	320	875
$Map_{emp.eid}(v)$	2	4	7	4

We further classify two types of mapping functions:

1. *Order preserving*: A mapping function $Map_{R.A_i}$ is called *order preserving* if for any two values v_i and v_j in the domain of A_i, if $v_i < v_j$, then $Map_{R.A_i}(v_i) \leq Map_{R.A_i}(v_j)$.
2. *Random*: A mapping function is called *random* if it is not order preserving.

A random mapping function provides superior privacy compared to its corresponding order-preserving mapping. However, as we will see later, whether a mapping function is order preserving or not affects how we translate a query into queries on the client and server. Query translation is simplified using a order-preserving mapping function. We will develop translation strategies for both types of mapping functions.

We further define three more mapping functions that will help us in translating queries over the encrypted representation. While the first function defined holds over any attribute, the latter two hold for the attributes whose domain values exhibit total order. Application of the mapping function to a value v, greater than the maximum value in the domain, v_{max}, returns $Map_{R.A_i}(v_{max})$. Similarly, application of the mapping function to a value v, less than the minimum value in the domain, v_{mim}, returns $Map_{R.A_i}(v_{min})$.

Let S be a subset of values in the domain of attribute A_i, and v be a value in the domain. We define the following mapping functions on the partitions associated with A_i:

$$Map_{R.A_i}(S) = \{ident_{R.A_i}(p_j) | p_j \cap S \neq \emptyset\}$$

$$Map^{>}_{R.A_i}(v) = \{ident_{R.A_i}(p_j) | p_j.low \geq v\}$$

$$Map^{<}_{R.A_i}(v) = \{ident_{R.A_i}(p_j) | p_j.high \leq v\}$$

Essentially, $Map_{R.A_i}(S)$ is the set of identifiers of partitions whose ranges may overlap with the values in S. The result of $Map^{>}_{R.A_i}(v)$ is the set of identifiers corresponding to partitions whose ranges may contain a value not less than v. Likewise, $Map^{<}_{R.A_i}(v)$ is the set of identifiers corresponding to partitions whose ranges may contain a value not greater than v.

2.4 Storing Encrypted Data

We now have enough notations to specify how to store the encrypted relation R^S on the server. For each tuple $t = \langle a_1, a_2, \ldots, a_n \rangle$ in R, the relation R^S stores a tuple:

$$\langle encrypt(\{a_1, a_2, \ldots, a_n\}), Map_{R.A_1}(a_1), Map_{R.A_2}(a_2), \ldots, Map_{R.A_n}(a_n) \rangle$$

where *encrypt* is the function used to encrypt a tuple of the relation. For instance, the following is the encrypted relation emp^S stored on the server:

etuple	eid^S	$ename^S$	$salary^S$	$addr^S$	did^S
1100110011110010...	2	19	81	18	2
1000000000011101...	4	31	59	41	4
1111101000010001...	7	7	7	22	2
1010101010111110...	4	71	49	22	4

The first column *etuple* contains the string corresponding to the encrypted tuples in *emp*. For instance, the first tuple is encrypted to "1100110011110010..." that is equal to

[3] Note that it is not necessary to create all of the partitions at the beginning. They can be created as the values are inserted inτo the database.

$encrypt(23, Tom, 70K, Maple, 40)$. The second is encrypted to "1000000000011101..." equal to $encrypt(860, Mary, 60K, Main, 80)$. We treat the encryption function as a black box in our discussion. Any block cipher technique such as AES [1], RSA [11], Blowfish [12], DES [3] etc., can be used to encrypt the tuples.

The second column corresponds to the index on the employee ids. For example, value for attribute *eid* in the first tuple is 23, and its corresponding partition is $[0, 200]$. Since this partition is identified to 2, we store the value "2" as the identifier of the *eid* for this tuple. Similarly, we store the identifier "4" for the second employee id 860. In the table above, we use different mapping functions for different attributes. The mapping functions for the *ename*, *salary*, *addr*, and *did* attributes are not shown, but they are assumed to generate the identifiers listed in the table.

In general, we use the notation "E" ("Encrypt") to map a relation R to its encrypted representation. That is, given relation $R(A_1, A_2, \ldots, A_n)$, relation $E(R)$ is $R^S(etuple, A_1^S, A_2^S, \ldots, A_n^S)$. In the above example, $E(emp)$ is the table emp^S.

2.5 Decryption Functions

Given the operator E that maps a relation to its encrypted representation, we define its inverse operator D that maps the encrypted representation to its corresponding unencrypted representation. That is, $D(R^S) = R$. In the example above, $D(emp^S) = emp$. The D operator may also be applied on query expressions. A query expression consists of multiple tables related by arbitrary relational operators (e.g., joins, selections, etc.)

As it will be clear later, the general schema of an encrypted relation or the result of relational operators amongst encrypted relations, R_i^S is:

$$\langle R_1^S.etuple, R_2^S.etuple, \ldots, R_1^S.A_1^S, R_1^S.A_2^S, \ldots, R_2^S.A_1^S, R_2^S.A_2^S, \ldots\rangle$$

When the decryption operator D is applied to R_i^S, it strips off the index values $(R_1^S.A_1^S, R_1^S.A_2^S, \ldots, R_2^S.A_1^S, R_2^S.A_2^S, \ldots)$ and decrypts $(R_1^S.etuple, R_2^S.etuple, \ldots)$ to their unencrypted attribute values.

As an example, assume that another table defined as *mgr* (*mid*, *did*) was also stored in the database. The corresponding encrypted representation $E(mgr)$ will be a table mgr^S ($etuple$, mid^S, did^S). Suppose we were to compute a join between tables emp^S and mgr^S on their did^S attributes. The resulting relation $temp^S$ will contain the attributes $\langle emp^S.etuple, eid^S, ename^S, salary^S, addr^S, emp^S.did^S, mgr^S.etuple, mid^S, mgr^S.did^S\rangle$. If we are to decrypt the $temp^S$ relation using the D operator to compute $D(temp^S)$, the corresponding table will contain the attributes

$$(eid, ename, salary, addr, emp.did, mid, mgr.did)$$

That is, $D(temp^S)$ will decrypt *all* of the encrypted columns in $temp^S$ and drop the auxiliary columns corresponding to the indices.

3. MAPPING CONDITIONS Map_{cond}

In this section we study how to translate specific query conditions in operations (such as selections and joins) to corresponding conditions over the server-side representation. This translation function is called Map_{cond}. Once we know how conditions are translated, we will be ready to discuss how relational operators are translated over the server-side implementation, and how query trees are translated.

For each relation, the server side stores the encrypted tuples, along with the attribute indices determined by their mapping functions. Meanwhile, the client stores the meta data about the specific indices, such as the information about the partitioning of attributes, the mapping functions, etc. The client utilizes this information to translate a given query Q to its server-side representation Q^S, which is then executed by the server. We consider query conditions characterized by the following grammar rules:

- Condition ← Attribute *op* Value;
- Condition ← Attribute *op* Attribute;
- Condition ← (Condition ∨ Condition) | (Condition ∧ Condition) | (¬ Condition);.

Allowed operations for *op* include $\{=, <, >, \leq, \geq\}$.

In the discussion below we will use the following tables to illustrate the translation.

```
emp(eid, ename, salary, addr, did, pid)
mgr(mid, did, mname)
proj(pid, pname, did, budget)
```

Attribute = Value: Such a condition arises in selection operations. The mapping is defined as follows:

$$Map_{cond}(A_i = v) \Rightarrow A_i^S = Map_{A_i}(v)$$

As defined in Section 2.3, function Map_{A_i} maps v to the identifier of A_i's partition that contains with value v. For instance, consider the *emp* table above, we have:

$$Map_{cond}(eid = 860) \Rightarrow eid^S = 4$$

since $eid = 860$ is mapped to 4 by the mapping function of this attribute.

Attribute < Value: Such a condition arises in selection operations. The attribute must have a well defined ordering over which the "<" operator is defined. Depending upon whether or not the mapping function Map_{A_i} of the attribute is order-preserving or random, different translations are possible.[4]

- Order preserving: In this case, the translation is straightforward:

$$Map_{cond}(A_i < v) \Rightarrow A_i^S \leq Map_{A_i}(v)$$

- Random: The translation is a little complex. We check if the attribute value representation A_i^S lies in any of the partitions that may contain a value v' where $v' < v$. Formally, the translation is:

$$Map_{cond}(A_i < v) \Rightarrow A_i^S \in Map_{A_i}^{<}(v)$$

For instance, the following condition is translated:

$$Map_{cond}(eid < 280) \Rightarrow eid^S \in \{2, 7\}$$

since all employee ids less than 280 have two partitions $[0, 200]$ and $(200, 400]$, whose identifiers are $\{2, 7\}$.

[4] Note that we can always use the mapping defined in the random case to translate conditions involving order-preserving attributes. We differentiate between the two cases since the translation (as well as the query processing) is easier for the former case.

Attribute > Value: This condition is symmetric with the previous one. As before we differentiate whether or not the mapping function is order preserving. The translation is as follows:

- Order preserving: $Map_{cond}(A_i > v) \Rightarrow A_i^S \geq Map_{A_i}(v)$;
- Random: $Map_{cond}(A_i > v) \Rightarrow A_i^S \in Map^{>}_{A_i}(v)$.

For instance, the following condition is translated:

$$Map_{cond}(eid > 650) \Rightarrow eid^S \in \{1, 4\}$$

since all employee ids greater than 650 are mapped to identifiers: $\{1, 4\}$.

Attribute1 = Attribute2: Such a condition might arise in a join. The two attributes can be from two different tables, or from two instances of the same table. The condition can also arise in a selection, and the two attributes can be from the same table. The following is the translation:

$$Map_{cond}(A_i = A_j) \Rightarrow \bigvee_{\varphi}(A_i^S = ident_{A_i}(p_k) \wedge A_j^S = ident_{A_j}(p_l))$$

where φ is $p_k \in partition(A_i), p_l \in partition(A_j), p_k \cap p_l \neq \emptyset$. That is, we consider all possible pairs of partitions of A_i and A_j that overlap. For each pair (p_k, p_l), we have a condition on the identifiers of these two partitions: $A_i^S = ident_{A_i}(p_k) \wedge A_j^S = ident_{A_j}(p_l)$. Finally we take the disjunction of these conditions. The intuition is that each pair of partitions may provide some values of A_i and A_j that can satisfy the condition $A_i = A_j$.

Partitions	$Ident_{emp.did}$	Partitions	$Ident_{mgr.did}$
[0,100]	2	[0,200]	9
(100,200]	4	(200,400]	8
(200,300]	3		
(300,400]	1		

For instance, the table above shows the partition and identification functions of two attributes *emp.did* and *mgr.did*. Then condition *emp.did* = *mgr.did* is translated to the following condition C_1:

$$\begin{array}{rl} C_1: & (emp^S.did^S = 2 \wedge mgr^S.did^S = 9) \\ \vee & (emp^S.did^S = 4 \wedge mgr^S.did^S = 9) \\ \vee & (emp^S.did^S = 3 \wedge mgr^S.did^S = 8) \\ \vee & (emp^S.did^S = 1 \wedge mgr^S.did^S = 8). \end{array}$$

Attribute1 < Attribute2: Again such a condition might arise in either a join or in a selection. Let us assume that the condition is $A_i < A_j$. Just as in translating conditions with inequality operator seen previously, the mapping of the condition depends upon whether or not the mapping functions of the attributes A_i and A_j are order preserving or random. We specify the translation for each in turn.

- Map_{A_j} is order preserving: In such a case we list out all the partitions of A_i and identify all the partitions of A_j that satisfy the ordering condition. Specifically, the mapping is as follows:

$$Map_{cond}(A_i < A_j) \Rightarrow \bigvee_{p \in partition(A_i)} (A_i^S = ident_{A_i}(p) \wedge A_j^S \geq Map_{A_j}(p.low))$$

- Map_{A_i} is order preserving: If A_i is order preserving, we can do the translation in a symmetric way with the roles of A_i and A_j reversed. The mapping will be as follows:

$$Map_{cond}(A_i < A_j) \Rightarrow \bigvee_{p \in partition(A_j)} (A_j^S = ident_{A_j}(p) \wedge A_i^S \leq Map_{A_i}(p.high))$$

- Both Map_{A_i} and Map_{A_j} are order preserving: In this case we have a choice of using either of the above two mappings. Our choice is based on the specific partitioning of A_i and A_j. We can do the translation as follows:

$$Map_{cond}(A_i < A_j) \Rightarrow \bigvee_{\varphi}(Map_{A_i}(p_k.low) \leq Map_{A_j}(p_l.high))$$

where φ is $p_k \in partition(A_i), p_l \in partition(A_j)$.

- Both Map_{A_i} and Map_{A_j} are random: We have the following translation:

$$Map_{cond}(A_i < A_j) \Rightarrow \bigvee_{\varphi}(A_i^S = ident_{A_i}(p_k) \wedge A_j^S = ident_{A_j}(p_l))$$

where φ is $p_k \in partition(A_i), p_l \in partition(A_j), p_l.high \geq p_k.low$. That is, we consider all pairs of partitions of A_i and A_j that could satisfy the condition. For each pair, we have a condition corresponding to the pair of their identifiers. We take the disjunction of these conditions.

For example, condition C_2 : *emp.did* < *Dept.did* is translated to:

$$\begin{array}{rl} C_2: & (emp^S.did^S = 2 \wedge mgr^S.did^S = 9) \\ \vee & (emp^S.did^S = 2 \wedge mgr^S.did^S = 8) \\ \vee & (emp^S.did^S = 4 \wedge mgr^S.did^S = 9) \\ \vee & (emp^S.did^S = 4 \wedge mgr^S.did^S = 8) \\ \vee & (emp^S.did^S = 3 \wedge mgr^S.did^S = 8) \\ \vee & (emp^S.did^S = 1 \wedge mgr^S.did^S = 8). \end{array}$$

Condition $emp^S.did^S = 4 \wedge mgr^S.did^S = 9$ is included, since partition (100, 200] for attribute *emp.did* and partition (200, 400] for attribute *mgr.did* can provide pairs of values that satisfy *emp.did* < *mgr.did*.

For condition *Attribute*1 > *Attribute*2, the Map_{cond} mapping is same as the mapping of *Attribute*2 < *Attribute*1, as described above with the roles of the attributes reversed.

Condition1 ∨ Condition2, Condition1 ∧ Condition2: The translation of the two composite conditions is given as follows:

$$Map_{cond}(Condition1 \vee Condition2) \Rightarrow Map_{cond}(Condition1) \vee Map_{cond}(Condition2)$$

$$Map_{cond}(Condition1 \wedge Condition2) \Rightarrow Map_{cond}(Condition1) \wedge Map_{cond}(Condition2)$$

Translation of $Map_{cond}(\neg Condition)$ treatment is more involved since negated queries are not monotonic and their correct translation requires more notation. This discussion can be found in [5].

Operator $\leq$ follows the same mapping as $<$ and operator $\geq$ follows the same mapping as $>$. Conditions that involve more than one attribute and operator are not discussed.

4. IMPLEMENTING RELATIONAL OPERATORS OVER ENCRYPTED RELATIONS

In this section we describe how individual relational operators (such as selections, joins, set difference, and grouping operators) can be implemented in the proposed database architecture. Our strategy is to partition the computation of the operators across the client and the server. Specifically, we will attempt to compute a superset of answers generated by the operator using the attribute indices stored at the server. These answers will then be filtered at the client after decryption to generate the true results. We will attempt to minimize the work done at the client as much as possible. Furthermore, we will try to ensure as much as possible that operators executed on the client side are such that they can be applied to the tuples arriving over the answer stream as soon as they arrive (without a need to store them). The purpose is to guarantee that the client-side operators can be efficiently implemented. The implementation of operators developed in this section will be used in the following section, where we develop an algebraic framework for rewriting SQL queries for the purpose of splitting the query computation across the client and the server.

For explaining the implementation of operators, we will consider the following two simplified relations of those in the previous section:

```
emp(eid, did), mgr(mid, did)
```

In the previous sections we have given the *Map* functions of *emp.eid*, *emp.did*, and *mgr.did*. For simplicity, we assume that the *Map* function of *mgr.mid* is the same as that of *emp.eid*, as shown in Figure 3. In addition, we use R and T to denote two relations, and use the operator notations in [4].

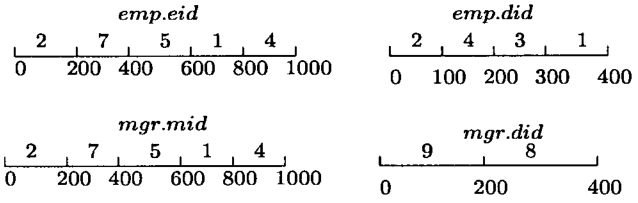


Figure 3: Partition and identification functions for four attributes.

The Selection Operator (σ): Consider a selection operation $\sigma_C(R)$ on a relation R, where C is a condition specified on one or more of the attributes $A_1, A_2, \ldots, A_n$ of R. A straightforward implementation of such an operator in our environment is to transmit the relation R^S from the server to the client. Then the client decrypts the result using the D operator, and implements the selection. This strategy, however, pushes the entire work of implementing the selection to the client. In addition, the entire encrypted relation needs to be transmitted from the server to the client. An alternative mechanism is to partially compute the selection operator at the server using the indices associated with the attributes in C, and push the results to the client. The client decrypts the results and filters out tuples that do not satisfy C. Specifically, the operator can be rewritten as follows:

$$\sigma_C(R) = \sigma_C\Big(D(\sigma^S_{Map_{cond}(C)}(R^S))\Big)$$

In the above notation, we adorn the σ operator that executes at the server with a superscript "S" to highlight the fact that the select operator executes at the server. All non-adorned operators are assumed to execute at the client. The decryption operator D will only keep the attribute *etuple* of R^S, and drop all the other A^S_i attributes. We explain the above implementation using an example $\sigma_{eid<395 \wedge did=140}(emp)$. Based on the definition of $Map_{cond}(C)$ discussed in the previous section, the above selection operation will be translated into $\sigma_C\Big(D(\sigma^S_{C'}(emp^S))\Big)$, where the condition C' on the server is:

$$C' = Map_{cond}(C) = (eid^S \in [2,7] \wedge did^S = 4)$$

The Join Operator ($\bowtie$): Consider a join operation $R \underset{C}{\bowtie} S$. The join condition C could be either equality conditions (in which case the join corresponds to an equijoin), or could be more general conditions (resulting in theta-joins). The above join operation can be implemented as follows:

$$R \underset{C}{\bowtie} T = \sigma_C\Big(D(R^S \underset{Map_{cond}(C)}{\bowtie^S} T^S)\Big)$$

As before, the S adornment on the join operator emphasizes the fact that the join is to be executed at the server. For instance, join operation $emp \underset{emp.did=mgr.did}{\bowtie} mgr$ is translated to:

$$\sigma_C\Big(D\big(emp^S \underset{C'}{\bowtie^S} mgr^S\big)\Big)$$

where the condition C' on the server is condition C_1 defined in Section 3.

The Grouping and Aggregation Operator (γ): A grouping and aggregation operation is denoted by $\gamma_L(R)$, where $L = L_G \cup L_A$. L_G refers to a list of attributes on which the grouping is performed, and L_A corresponds to a set of aggregation operations. As an example, the operation $\gamma_{C,COUNT(B) \to F}(R)$ means that we create groups using attribute C of relation R, and for each group compute the *count*(B) function. That is, $L_G = \{C\}$, and $L_A = \{COUNT(B) \to F\}$. The resulting relation will contain two attributes C and F. A tuple in the result will have an entry for each distinct value of C, and the number of tuples in the group reported as attribute F. If $L_A = \emptyset$, only grouping is performed.

Implementation of the grouping operator $\gamma_L(R)$ can be achieved as follows:

$$\gamma_L(R) = \gamma_L\Big(D(\gamma^S_{L'}(R^S))\Big), where\ L' = \{A^S_i | A_i \in L_G\}$$

That is, the server will group the encrypted tuples based on the attributes of L_G. The server does *not* perform any aggregation corresponding to L_A, since it does not have any values for those attributes in L_A. The results of $\gamma^S_{L'}$ are returned to the client, which performs the grouping operation γ_L. This operation can be implemented very efficiently, since every tuple belonging to a single group of γ_L will be in a single $\gamma^S_{L'}$ group computed by the server. As a result, the client only needs to consider tuples in a single $\gamma^S_{L'}$ group when computing the groups corresponding to γ_L. Of course, the aggregation functions specified in L_A will be computed at the client, since their computation requires that tuples be first decrypted.

We explain the implementation using the example below.

$$\gamma_{did,COUNT(eid)\rightarrow F}(emp)$$

That is, we want to find the number of employees in each department. Let L denote "$did, COUNT(eid) \rightarrow F$." The operation is translated to:

$$\gamma_L\Big(D\big(\gamma^S_{did^S}(emp^S)\big)\Big)$$

That is, we first do a grouping on the did^S attribute on the server. After the grouped tuples are returned to the client, we decrypt the data, and perform the grouping operation on the *did* attribute. This step can be done efficiently, since all the tuples with the same *did* have already been grouped by the server. Finally, we perform the aggregation *count*(*eid*) to count the number of employee ids for each *did*.

The Sorting Operator (τ): A sorting operation $\tau_L(R)$ can be implemented similarly to the grouping operator. That is, we first sort on partition ids at the server. The strategy to implement $\tau_L(R)$ is as follows:

$$\tau_L(R) = \tau_L\Big(D\big(\gamma^S_{L'}(R^S)\big)\Big)$$

where L' = list of A^S_i corresponding to the A_i in the list L of attributes.

That is, we do a grouping operation $\gamma^S_{L'}$ on the encrypted attributes L' of those in L. If the mapping functions of the attributes in L are all order preserving, this grouping $\gamma^S_{L'}$ operation can be replaced by a corresponding sorting operation $\tau^S_{L'}$. After the results are returned to the client, we call the decryption function D, and perform the τ_L operation by sorting the tuples on attributes L.

Note that the amount of work done at the client to compute τ_L in postprocessing depends upon whether or not the attributes listed in L have order-preserving mappings. If the attributes have order-preserving mappings, then the results returned by the server are presorted upto within a partition. Thus, sorting the results is a simple local operation over a single partition. Alternatively, even if the mapping is not order preserving, it is useful to compute γ^S at the server to reduce the amount of client work. Since the tuples have been grouped by the server, τ_L can be implemented efficiently using a merge-sort algorithm.

For example, the sorting operation $\tau_{eid}(emp)$ can be implemented as follows:

$$\tau_{eid}\Big(D\big(\gamma^S_{eid}(emp^S)\big)\Big)$$

where $L = \{eid\}$. That is, we first perform a grouping operation γ_{eid} on the emp^S relation on the server. The client decrypts the returned tuples, and applies the sorting operation τ_{emp}.

The Duplicate-Elimination Operator (δ): The duplicate-elimination operator δ is implemented similarly to the grouping operator:

$$\delta(R) = \delta\Big(D\big(\gamma_L(R^S)\big)\Big)$$

where L = list of all attributes A^S_i where A_i is an attribute in R.

That is, we first group the encrypted tuples on the server using all the attributes in R^S. After the results are returned and decrypted at the client, we perform the duplicate elimination operation δ. For example, the operation $\delta(emp)$ is translated to:

$$\delta\Big(D\big(\gamma_{eid^S,did^S}(emp^S)\big)\Big)$$

The Set Difference Operator ($-$): Implementation of the difference operation $R-T$ at the server is difficult since, without first decrypting the relations R and T, it is impossible to tell whether or not a given tuple of R also appears in S. However, the indices stored at the server can still be used to meaningfully reduce the amount of work done at the client. In the following we assume that relations R and T are set difference compatible and are defined over attributes $A_1, A_2, \ldots, A_n$ and $B_1, B_2, \ldots, B_n$ respectively. The following rule can be used to implement the set difference operator:

$$R - T = \pi_{R.A_1,\ldots,R.A_n}(R') - \pi_{T.B_1,\ldots,T.B_n}(R')$$

$$R' = D\big(\gamma^S_L(R^S \mathbin{⟕}^S_{Map_{cond}\left(\wedge_{i=1,\ldots,n}(R.A_i=T.B_i)\right)} T^S)\big)$$

where $L = \{A^S_1, A^S_2, \ldots, A^S_n, B^S_1, B^S_2, \ldots, B^S_n\}$.

Once again, the symbol S as a superscript of the left-outer join emphasizes (denoted ⟕) that the operator is implemented on the server side. We illustrate the above rule through an example. Suppose we want to compute $emp - mgr$, that is, we want to find all the employees who are not managers. The query is translated to the following query:

$$\pi_{emp.eid,emp.did}(R') - \pi_{mgr.mid,mgr.did}(R')$$

$$R' = D\big(\gamma^S_{eid^S,emp^S.did^S,mid^S,mgr^S.did^S}(emp^S \mathbin{⟕}^S_{C'} mgr^S)\big)$$

The condition C' is: $Map_{cond}(emp.eid = mgr.mid) \wedge C_1$, where C_1 is defined in Section 3. (See Attribute1 = Attribute2 case)

A few observations about the above implementation of the set-difference operator are noteworthy. First, the grouping of the results based on index attributes is not necessary – that is, the translation would be correct even without the grouping operator. The reason for including the grouping operator is that it can significantly reduce the computation on the client. For example, due to the grouping operator, all the tuples that have a NULL value for T^S attributes will be grouped together. When the resulting tuples of the set difference operator arrive at the client, such tuples can be decrypted and the corresponding R tuple immediately returned as an answer. The reason is that there are no matching tuples of T that could cause the potential elimination of these tuples of R. Hence, the projection and the subsequent set difference implemented on the client side may only be restricted to those tuples for which the corresponding T value is not NULL.

Furthermore, in computing the projection to the attributes of R and S and the subsequent set difference between the two projections we only need to consider a single group formed by γ^S_L operator at a time. That is, a T tuple from a different group will not eliminate an R tuple from another group. Thus, performing the grouping at the server side, while not necessary, could significantly reduce the computation at the client.

Second, even with the above optimization, the implementation of the set-difference operator using the outer-join on the server should be used with care. A naive strategy is to transmit the entire relations R^S and T^S to the client, which decrypts them and computes the set difference. This naive strategy might be cheaper than the previous strategy since the size of the outerjoin might be quadratic resulting in high transmission and decryption cost compared to the strategy of transmitting the two relations individually and computing the set difference at the client. Which strategy is used depends upon the content of the relations. Selecting the specific strategy depends upon integrating our framework into a cost-based query optimizer, which is beyond the scope of this paper.

The Union Operator ($\cup$): There are essentially two different union operators based on the bag and the set semantics. The former does not eliminate duplicates, while the latter does. The implementation of the union operator based on bag semantics is straightforward:

$$R \cup T = D(R^S \cup^S T^S)$$

If we wish to compute the union under the set semantics, it can be computed as follows:

$$R \cup T = \delta\Big(D\big(\tau_L^S(R^S \cup^S T^S)\big)\Big)$$

where L = list of all attributes A_i^S where A_i is an attribute in R.

While the implementation of the union operator on the server side (that is, $\cup^S$) is straightforward, there is one technical challenge that arises. Specifically, since tuples in $R^S \cup^S T^S$ could originate either from R^S or T^S, to be able to apply the correct decryption function at the client, as well as to correctly interpret the values in the index attributes of the result at the server, we will store an additional attribute in the result of the union that will determine the origin of the tuple (that is, whether it originates from R^S or T^S). Adding such an attribute is crucial for the correct implementation, but we will ignore it in the discussion to keep the algebra simple. The full version of the paper [5] illustrates the need for maintaining the additional attribute and the resulting modifications to the mapping functions and developed algebra.

The Projection Operator (π): Since each tuple in a relation R is encrypted together into a single string in the *etuple* attribute of relation R^S at the server, a projection π is not implemented at the server. As a result, to compute $\pi_L(R)$, where L is a set of attributes, the strategy is to transmit the complete relation R^S to the client, decrypt the relation at the client, and then compute the projection. That is,

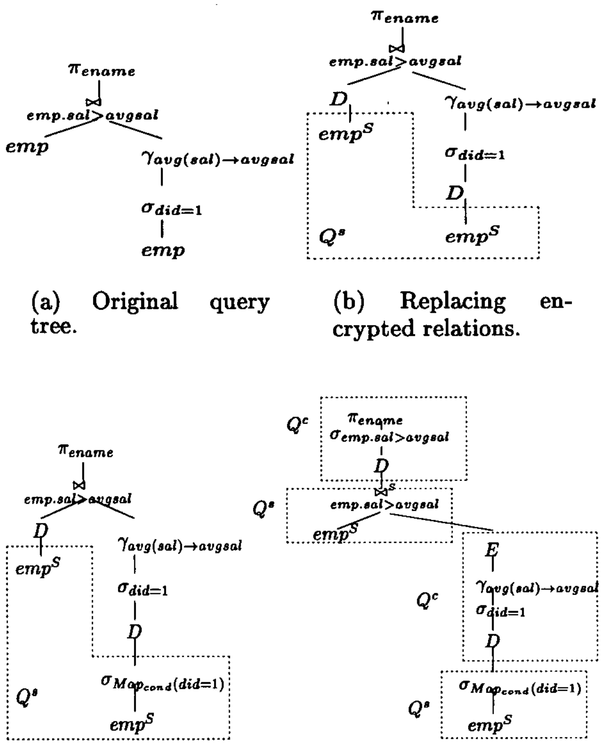
$$\pi_L(R) = \pi_L\big(D(R^S)\big)$$

For instance, we have $\pi_{eid}(emp) = \pi_{eid}\big(D(emp^S)\big)$.

5. ALGEBRAIC FRAMEWORK FOR QUERY SPLITTING

Given a query Q, our purpose in this section is to develop a strategy to split the computation of Q across the server and the client. The server will use the implementation of the relational operators discussed in the previous section to compute as much of the query as possible, relegating the remainder of the computation to the client. Our objective is to come up with the "best" query plan for Q that minimizes the execution cost. In our setting, the cost of a query consists of many components – the I/O and CPU cost of evaluating the query at the server, the network transmission cost, and the I/O and CPU cost at the client. A variety of possibilities exist. For example, consider the following query over the *emp* table that retrieves employees whose salary is greater that the average salary of employees in the department identified by *did* = 1.

```
SELECT emp.name FROM emp
     WHERE  emp.salary > (SELECT AVG(salary)
     FROM emp WHERE did = 1);
```



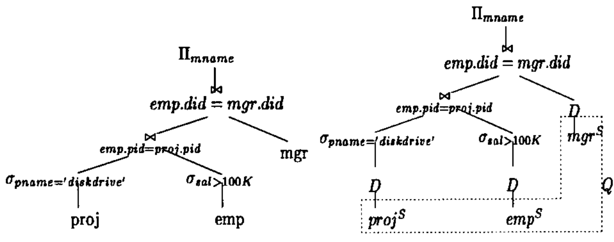
(a) Original query tree.

(b) Replacing encrypted relations.

(c) Doing selection at server.

(d) Multiple interactions between Client and Server.

Figure 4: Query plans for employees who make more than average salary of emloyees who are in did=1

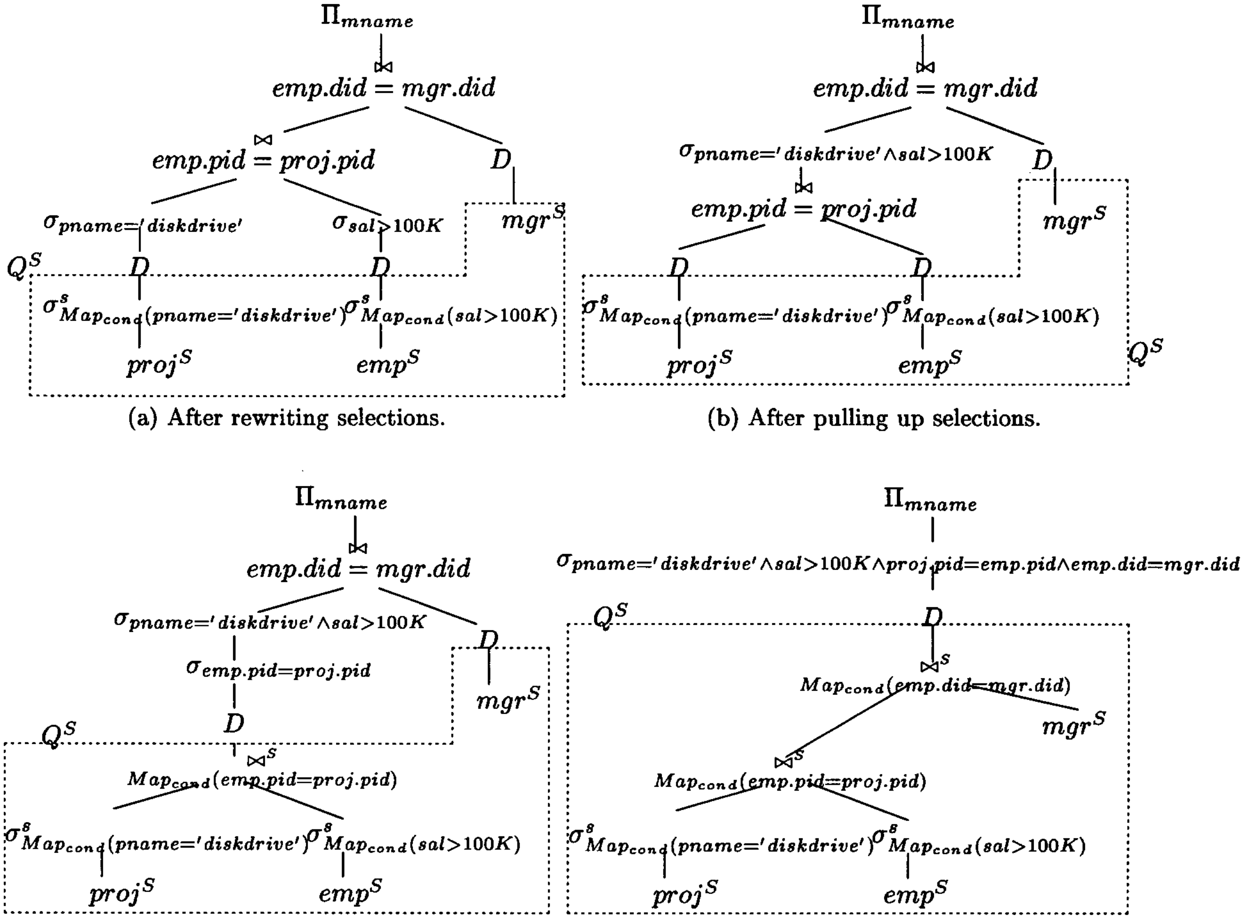


(a) Original query tree.

(b) After replacing with encrypted relations.

Figure 5: Evaluation of a join query.

The corresponding query tree and some of the evaluation strategies are illustrated in Figures 4(a) to (d). The first strategy (Figure 4(b)) is to simply transmit the *emp* table to the client, which evaluates the query. An alternative



Πmname
⋈ emp.did = mgr.did
⋈ emp.pid = proj.pid
σpname='diskdrive'
σsal>100K
D
mgr^S
Q^S
σ^s Mapcond(pname='diskdrive')
σ^s Mapcond(sal>100K)
proj^S
emp^S
σpname='diskdrive'∧sal>100K
σemp.pid=proj.pid
⋈^S Mapcond(emp.pid=proj.pid)
⋈^S Mapcond(emp.did=mgr.did)
σpname='diskdrive'∧sal>100K∧proj.pid=emp.pid∧emp.did=mgr.did

(a) After rewriting selections.

(b) After pulling up selections.

(c) After rewriting join.

(d) The final query tree.

Figure 6: Query rewriting heuristic for join queries

strategy (Figure 4(c)) is to compute part of the inner query at the server, which selects (as many as possible) tuples corresponding to $Map_{cond}(did = 1)$. The server sends to the client the encrypted version of the *emp* table, i.e., emp^S, along with the encrypted representation of the set of tuples that satisfy the inner query. The client decrypts the tuples to evaluate the remainder of the query. Yet another possibility (Figure 4(d)) is to evaluate the inner query at the server. That is, select the tuples corresponding to the employees that work in department $did = 1$. The results are shipped to the client, which decrypts the tuples and computes average salary. The average salary is encrypted by the client and shipped back to the server, which then computes the join at the server. Finally, the results are decrypted at the client.

5.1 Heuristic Rules to Separate Queries

It should immediately be obvious that a rich set of possibilities exist in evaluating a query in our framework, and that the decision of the exact query plan should be cost based. This topic, however, is outside the scope of this paper. Our attempt is primarily to establish the feasibility of the proposed model, and cost-based optimization is relegated to future work. Instead, in this section, we will restrict ourselves to a simpler task – we will explore heuristic rules that allow for a given query tree to be split into two parts – the server part (referred to as Q^S) that executes at the server first, and the client part (referred to as Q^C) that executes at the client based on the results of the query evaluated at the server. Our objective will be to minimize the computation in Q^C. That is, we would attempt to rewrite the query tree, such that most of the effort of evaluating the query occurs at the server, and the client does least amount of work.

We illustrate our ideas using examples. As a first example, consider the following query that computes the names of the managers of those employees working on project "diskdrive" whose salary is more than 100K.

```
SELECT  mname
     FROM    emp, mgr, proj
     WHERE   proj.pname = 'diskdrive'
      AND proj.pid = emp.pid
      AND emp.sal > 100K
      AND emp.did = mgr.did;
```

The first step is to convert the above query into a corresponding query tree, and to manipulate the query tree to generate a good plan (using the standard query rewrite laws of relational algebra [13]). Figure 5(a) shows the query tree in which the two selections have been pushed down to relations *proj* and *emp*. Since relations are encrypted and stored on the server, we first replace each relation R in the query with encrypted relation R^S. The resulting tree is shown in Figure 5(b).

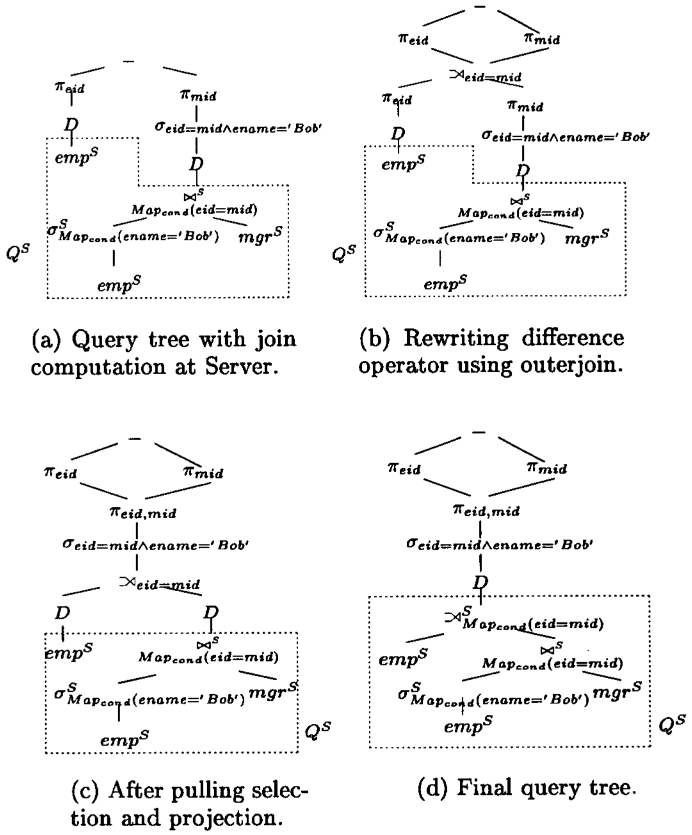
As it stands, the current query tree requires the entire relations $proj^S$, emp^S, and mgr^S to be sent to the client that will decrypt the relations to evaluate the query. We next replace the selection operations by their implementation listed in the previous section resulting in the query tree shown in Figure 6(a). Notice that in the corresponding tree, the server is participating in the evaluation of the two selection conditions. Since our objective is to perform as much of the computation at the server as possible, we next pull up the two client-side selection conditions $\sigma_{pname='diskdrive'}$ and $\sigma_{sal=100K}$ above the join operator $\underset{emp.pid=proj.pid}{\bowtie}$ using the standard rewrite rules involving selections in relational algebra [13]. The new query tree is shown in Figure 6(b). We can now rewrite the query tree again using the join implementation discussed in the previous section, such that $\underset{emp.pid=proj.pid}{\bowtie}$ is executed at the server. Figure 6(c) shows the query tree after the rewriting. Finally, we pull the two selections $\sigma_{pname='diskdrive' \wedge sal=100K}$ and $\sigma_{emp.pid=proj.pid}$ above the join operator $\underset{emp.did=mgr.did}{\bowtie}$. Then we replace the join operator based on the implementation discussed in the previous section, and get the final query tree, as shown Figure 6(d).

Notice that in the tree of Figure 6(d), much of the work of query processing is done at the server. The results obtained from the server are decrypted and filtered at the client. Our success in splitting the query Q into the server-side Q^S and client-side Q^C depended on (1) being able to pull the selection operations above other relational operations higher in the query tree; and (2) repeatedly rewriting the higher-level operations using the operator implementations listed in the previous section.

There are situations when the selection operator cannot be pulled up the query tree as it is illustrated in the following example, which uses a set-difference operator. Consider a query that retrieves the set of employees who do not work for the manager named "Bob." The corresponding SQL query is shown below:

```
SELECT  ename FROM emp
WHERE   eid NOT IN (
        SELECT eid FROM emp, mgr
        WHERE eid = mid AND ename = 'Bob');
```

Using the strategy discussed above, we can easily convert the query into the query tree shown in Figure 7(a). If we are to execute the query plan illustrated in Figure 7(a), the server will submit to the client the relation emp^S, as well as the encrypted answers generated by the $\bowtie^S$ operator. The projections followed by the set-difference operator will be implemented at the client. Notice since the selection and projection operators cannot be pulled above the set-difference operator, it is difficult to apply the implementation of the set-difference operator discussed in the previous section to evaluate the set difference at the server. The trick is to rewrite the set-difference operator using the left-outer join operator $⟕$ (similar to the implementation of set difference discussed in the previous section). Using the rewrite law for set difference, the corresponding tree is modified to the query tree shown in Figure 7(b). We can now pull the selections and projections above the outer join, resulting in the query tree shown in Figure 7(c). Finally, this tree can be manipulated using the operator implementation discussed in the previous section, resulting in the final tree shown in Figure 7(d). The final tree performs much of the query computation at the server, and the results are decrypted and filtered, and the final answer is evaluated at the client.



(a) Query tree with join computation at Server.

(b) Rewriting difference operator using outerjoin.

(c) After pulling selection and projection.

(d) Final query tree.

Figure 7: Query rewriting for a set-difference query.

6. EXPERIMENTAL EVALUATION

We have conducted experiments to show the validity and the effectiveness of the architecture proposed in this paper. In this section, we present our experimental results.

We ran the tests by utilizing TPC-H benchmark [14]. TPC-H benchmark database is created at scale factor 0.01 and 0.1, which are also referred to as the 10 MB and 100 MB database respectively. The experiments were conducted on two IBM Intel-based personal computers with Pentium III 700 MHz processors with 256 MB RAM. One of the computers performed as the server, and another one performed as the client according to our client/server architecture. Relevant software components used were IBM DB2 v7.1 and Microsoft Windows 2000 as the operating system.

Relations: While the TPC-H benchmark includes multiple tables, of particular interest in our experimental study are the `lineitem`, `customer`, and `order` tables. We partitioned the following attributes for these tables:

Lineitem	:	l_shipdate, l_discount, l_quantity
Orders	:	o_orderdate,o_custkey,o_shippriority
Customer	:	c_custkey

Partitions have been created based on the partitioning criteria described below. To encrypt the rows of the relations, we used the Blowfish encryption algorithm [12] implemented in Java.

Partitioning Algorithm: We used equi-width and equi-depth histograms [9] to partition the data for two different classes of queries. Equi-width and equi-depth histograms have been widely used and investigated in the context of selectivity estimation in databases [7, 8]. Detailed description of constructing equi-depth histograms is given in [9].

Queries : We considered two different queries from TPC-H suite to present the evaluation of the different aspects of the architecture. The first query, as shown in Figure 8, is a selection query from a single table, and it is not involved join operations. The second query, as shown in Figure 9, is a modified version of TPC-H query number 3, denoted Q3. This query involves a join operation between two tables, `customer` and `orders`. We first successfully rewrote the given queries using the rewriting rules described in the paper, and then executed the translated queries in the client/server architecture with different partitioning schemes.

```
select sum(l_extendedprice * l_discount) as revenue
from tpcd.lineitem
where l_shipdate >= date ('1994-01-01')
and l_shipdate < date ('1994-01-01') + 1 year
and l_discount between 0.06 - 0.01 and 0.06 + 0.01
and l_quantity < 24
```

Figure 8: Query used for first set of experiments, based on Q6 from TPC-H benchmark.

```
select o_orderdate, o_shippriority from tpcd.customer,tpcd.orders
where c_custkey = o_custkey and o_orderdate < date ('1995-03-15')
group by o_orderdate, o_shippriority order by o_orderdate
```

Figure 9: Query used for second set of experiments, based on Q3 from TPC-H benchmark.

6.1 Experiment 1

In the first set of experiments, we studied the components of the query-execution time in our architecture. We conducted these tests with increasing number of buckets. Figure 10(a)(b) show the results of the tests. It is shown that network communication cost and client-site query-execution time significantly decrease with the increase in the number of buckets. The reason is due to the decreasing number of rows returned by the server. When the number of buckets that partition the data increases, the server has a better capability to filter out more false rows, which do not satisfy the selection predicates. While network cost and client-site query-execution time decrease sharply, it is not the case for the server-site query-execution time. In the experiments, the selectivity of the query is approximately 18%. Because of the possibility of prefetch batch I/O's, doing a table scan remained as the best choice for the database optimizer. Hence, independently from the number of buckets, predicate evaluation is performed via a sequential table scan, causing the steady behavior in server-site query-execution time.

In these experiments we also compared the query-execution times in our architecture with the case of having a single server, which performs all the functions described in the architecture. The former represents total data privacy, while the latter represents row-level data encryption/decryption, where the server is trusted to decrypt the data. Figure 10(b) shows this comparison. Again we present the results for different numbers of buckets. The first bar in the figure shows the query-execution time for the single-server setup, where the server selects the *etuple* columns from the encrypted tables, and performs the real query on the selected rows. The second and third bars show the query-execution times for the server side and client side respectively when the query is executed in our architecture.[5] These experiments show that our architecture does not introduce significant overhead due to the proposed communication protocol between client and server.

6.2 Experiment 2

In the second set of experiments, we studied queries that include join operations. Experiments are based on the modified version of Q3 (Figure 9) in the TPC-H benchmark. Figure 11(a) to (c) show the client-side, server-side, and the total query-execution times for increasing number of buckets on join attributes, namely, `c_custkey` and `o_custkey` in the query. The figure illustrates that query response times decrease very sharply with the increasing number of buckets. As was explained in the previous experiment, this behavior is primarily since with increasing number of buckets the server is better able to eliminate tuples which would otherwise have to be decrypted and filtered at the client. The performance is significantly improved for both client and server side queries. Although the client-side query-execution time also shows steep decrease, it is greater than the server-side query-execution time. The reason is due to the dominant cost of decryption performed at the client site. To express this fact, Figure 11(b) shows the query response times of client-side query-execution time with decryption and client-side query-execution time without decryption, which is plotted by removing decryption cost from the query-execution time.

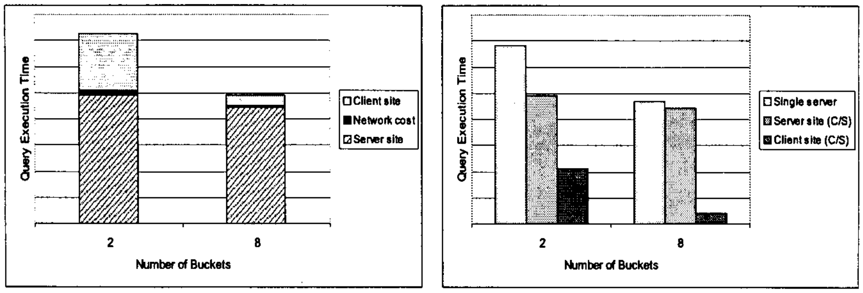
As was studied in our first set experiments, Figure 11(c) shows the total query-execution times for single-server and client-server architectures. The results of these tests are also consistent with the previous ones.

7. CONCLUSIONS

Application Service Provider (ASP) model for enterprise computing has emerged with the rise of Internet technologies. In the ASP model, a service provider can provide software as a service to a very large client-base over the Internet.

Unlike many other services, however, databases are special. Data is a precious resource of an enterprise. As a result, privacy and security of data at the service-provider site is paramount. In this paper, we addressed a specific data-privacy challenge – what if the owner of the database does not trust the service provider with the data? Our solution is to store the data at the service provider after encrypting it, which can only be decrypted by the owner. We have developed techniques using which the bulk of the work of executing the SQL queries can be done by the service provider without the need to decrypt the stored data. The technique deploys a "coarse index", which allows partial execution of an SQL query on the provider side. The result of this query is sent to the client. The correct result of the query is found by decrypting the data, and executing a compensation query at the client site. We proposed technique to operate the SQL query, and split it into a server query and a

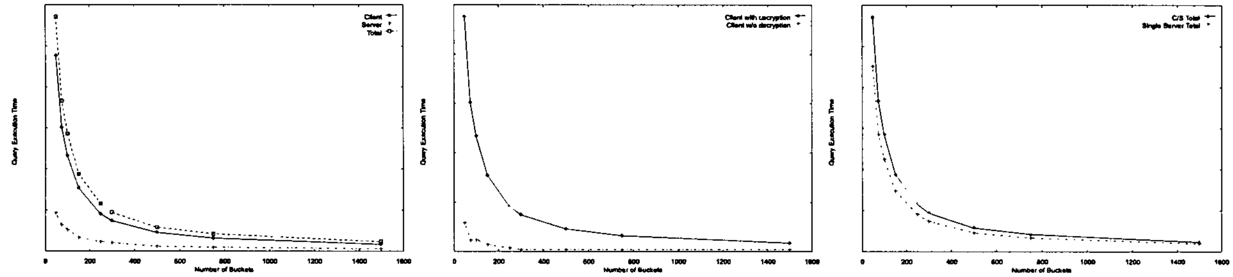
[5] The client-site query-execution time also includes the network communication cost required to transfer selected rows by the server.



(a) Cost factors for the query-execution time.

(b) Comparison for Client-Server strategy v.s. the single-server strategy.

Figure 10: Effect of number of buckets on nonjoin queries.



(a) Client, server, and total query-execution times.

(b) Effect of decryption performed on the client site.

(c) Comparison for client-server strategy v.s. single-server strategy

Figure 11: Effect of number of buckets on join query.

client query. The service provider retains the responsibility to manage the persistence of the data. The client gets total privacy, and the cost of cooperating in query execution with the service provider. The client does not need to manage data persistence, thus continues to benefit from the system-management service of the database service provider.

8. REFERENCES

[1] AES. Advanced Encryption Standard. *National Institute of Science and Technology, FIPS 197*, 2001.

[2] D. Song and D. Wagner and A. Perrig. Search on encrypted data. In *Proc. of IEEE SRSP*, 2000.

[3] DES. Data Encryption Standard. *FIPS PUB 46, Federal Information Processing Standards Publication*, 1977.

[4] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice Hall, 2002.

[5] H. Hacıgümüş, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over Encrypted Data in Database-Service-Provider Model. Technical Report TR-DB-02-02, Database Research Group at University of California, Irvine, 2002.

[6] H. Hacıgümüş, B. Iyer, and S. Mehrotra. Providing Database as a Service. In *Proc. of ICDE*, 2002.

[7] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query answers. In *Proc. of VLDB*, pages 174–185, 1999.

[8] H. V. Jagadish, H. Jin, B. C. Ooi, and K.-L. Tan. Global optimization of histograms. In *Proc. of ACM SIGMOD*, 2001.

[9] G. Piatatetsky-Shapiro and C. Connell. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of ACM SIGMOD*, 1984.

[10] R. L. Rivest and L. M. Adleman and M.L. Dertouzos. On Data Banks and Privacy Homomorphisms. In *Foundations of Secure Computation*, pages 169–178, 1978.

[11] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[12] B. Schneier. Description of a new variable-length key, block cipher (blowfish), fast software encryption. In *Cambridge Security Workshop Proceedings*, 1994.

[13] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts, 3rd Edition.* McGraw-Hill Book Company, 1997.

[14] TPC-H. *Benchmark Specification.* http://www.tpc.org.

[15] M. Winslett and J. D. Ullman. Jeffrey D. Ullman speaks out on the future of higher education, startups, database theory, and more. *SIGMOD Record*, 30(3), 2001.