

# Processing Aggregation Queries over Encrypted Databases

*Sun S. Chung, Gultekin Ozsoyoglu*

EECS Department, Case Western Reserve University, Cleveland, OH 44106

*(ssc7, tekin)@case.edu*

## Abstract

This paper is about *anti-tamper databases*, where the database contents are encrypted for security in a way to allow efficient query processing directly on the encrypted database. In an earlier work, we proposed a family of open-form and closed-form homomorphism encryption/decryption functions and a computing architecture where (a) the database is encrypted a priori, and (b) for SQL queries expressible in relational algebra, there is no extra query processing cost except for the decryption of the final query output.

In this work, we quantify the additional costs incurred when executing aggregate nested SQL queries over encrypted relational databases, and present detailed experimental results. We observe the crossover points as to when processing a query over an encrypted database is still more advantageous than shipping it over the internet to a secure server housing the original, nonencrypted database, evaluating it and returning the query output to the user.

## 1 Introduction

Advances in mobile computing, web technologies and Internet have elevated significance of data security in computer application systems. Emergence of the “databases-as-a-service” model allows users to retrieve data from anywhere throughout the Internet; as a result, protecting data privacy becomes an imminent challenge. Moreover, mobility of laptops as well as increased storage and power of laptops allows databases with sensitive data to travel everywhere, thus making them easy targets to capture physically, or to compromise by malicious users through bypassing access control or user authentication process in such systems. Especially, in an application service provider (ASP) environment where a database system is provided as a service, the owner of the data and the service provider may be different, and the database server may not be trustworthy; thus the data needs to be protected from the database service provider.

There are two kinds of threats to data privacy, outsider threats such as hackers, interceptors, and insider threats such as malicious employees, business associates who are willing to abuse authorized access to protected data. Recent studies [CSI02] show that a significant percentage of data theft comes from insiders. For example, in a database system that holds sensitive data such as personal medical records or bank account information, the database administrator of the database system has exclusive data access privileges, and there is no protection of data privacy from such inside privileged employees who may not be trustworthy. In addition, there have been substantial efforts in Health Care Industry that try to protect privacy of electronic personal medical information, that may be great values to outsiders, on communication between health care providers and health care payers through the internet.

If legitimate end users (we use this term with the term “clients” interchangeably) are on a secure communication channel, a way to protect data privacy from insider or outsider threats is to encrypt the database and not allow decryption by the server, yet allow query execution directly over the encrypted database without decrypting data by the server. Clearly, in this context, the requirement of high level of security for the encrypted data conflicts with the requirement of efficient query processing and query optimization over the encrypted data. We believe that, as long as the database performance degradation due to encryption is under control, users would choose to use encrypted databases over non-encrypted databases.

The papers by Hacigumus et al [HIM02, HILM02] view the database service provider as an untrustworthy third party; so their approach to protect sensitive data in the server site is also to encrypt the database and allow querying over the encrypted data without decrypting it. While the idea behind Hacigumus et al [HIM02, HILM02] is the same as ours, their approach for the evaluation of queries over encrypted data differs from ours and mainly depends on (i) partitioning the data, and (ii) encryption techniques for evaluating queries via random partitioning methods over the encrypted data. The server filters data based

on the partitions, and defers the rest of the evaluation of the query to the client site where the data is decrypted, and query evaluation is completed. Therefore Hacigumus et al's main concern becomes how much the server can filter the data for efficient query processing.

In an earlier work [OSC03], we considered a database environment where (a) the database contains sensitive data available only to its legitimate users, and (b) the database can be captured by the adversary physically, and a compromise threat exists--by physically accessing the data, and/or using a priori knowledge about the data in the database, and/or breaking user passwords, and accessing the database. Clearly, the database users still have efficiency requirements, and queries to the database still need to be processed within an acceptable time limit. In such an environment, a good way to prevent, delay, or contain the compromise of the protected data in a database is to encrypt the database, and yet allow queries over the encrypted data.

In [OSC03], we proposed *anti-tamper databases* and a query processing architecture, illustrated in Figure 1, with the following properties:

- The database  $DB_E$ , called the *anti-tamper database*, resides at the *anti-tamper site*, and is encrypted a priori using an open-form or closed-form encryption function  $f()$  applied to database attribute values. Thus, if  $DB_E$  becomes available to the adversary, its contents are devoid of semantics and, therefore not directly usable by the adversary.
- To the users, the encryption is transparent. That is, users think that they interact with the original, nonencrypted database  $DB$ , and pose their query  $Q$  against  $DB$ .
- We employ an intermediary software agent, called the (Encryption/Decryption) *Agent*, which we assume to be secure and, in this paper, residing at different site, called the *agent site*. That is, the adversary cannot capture the *Agent* code, and reverse-engineer its encryption/decryption algorithms. Agent rewrites  $Q(DB)$  into the SQL query  $Q_E(DB_E)$  of the encrypted database, and submits it to  $DB_E$ .
- The encryption function  $f()$  is chosen such that (i) it is a group homomorphism from the original database  $DB$  to the anti-tamper database  $DB_E$  with respect to  $S$  where  $S$  is the set of SQL queries  $Q$  on  $DB$  (i.e., when  $f(x)=y$  for all  $x \in DB$ ,  $y \in DB_E$  with the transformed set  $S_E$  of SQL queries  $Q_E(DB_E)$  on  $DB_E$ , for any query  $Q$  in  $S$  on  $DB$ , we have  $Q(DB) = f^{-1}(Q_E(f(DB)))$  [OSC03, RAD78]), (ii) information leakage is minimized, and (iii) inferences are controlled carefully when user have a priori knowledge of some database attribute values.
- The DBMS of the anti-tamper database is not aware of the encryption, and its query engine freely employs its query optimization techniques.
- For SQL queries that are expressible in relational algebra, there is no extra query processing cost except for the decryption of the final query output by the (secure) agent.

However, our approach does incur additional costs for evaluating aggregate SQL queries with or without nested subqueries and GROUP-BY clause. Clearly, depending on (a) database size, (b) the query processing techniques used, (c) implementation techniques used for the algebraic operators of the query, and (d) the selectivities of the algebra operators, at some point, applying our approach of processing the query over the encrypted database and simply decrypting the query output becomes expensive—more expensive than the alternative of shipping the query over a wide-area network to a secure site with the original database, processing the query and returning its output to the user, again over the wide-area network. In this paper, we quantify the additional costs incurred when executing aggregate (nested or unnested) SQL queries over encrypted relational databases, and present detailed experimental results. More specifically, we focus on aggregate queries with Group-By clause, and with correlated nested queries. We define six aggregate query types, and, for each query type, present and analyze query processing techniques over the encrypted database, and derive cost formulas for the query processing. We observe the crossover points as to when processing a query over an encrypted database is still more advantageous than shipping it over the internet to a secure server housing the original, nonencrypted database, evaluating it and returning the query output to the user.

For three types of aggregate and correlated nested queries with Sum Or Ave function, the cost differentials of query processing between encrypted and non-encrypted database increase linearly as join, selection and projection selectivities, and the size of relations increase. Query processing cost of encrypted database for single-block aggregate queries with AVE function is slightly more than double the cost of non-encrypted database query processing. For correlated nested aggregate queries with GROUP BY and AVE at an inner block, the extra cost of encrypted database query processing is 50% more of the cost of non-encrypted database query processing. When correlated nested queries with GROUP-BY clause and AVE function at the outer block are processed over encrypted database without removing correlation, the cost differential becomes higher than those of any other query types because of correlation. Moreover, for all three types of aggregate queries, data transmission costs for computations at the Agent site do not contribute significantly to the overall costs of encrypted database query processing even with very large relation sizes. However, for single-block aggregate queries with AVE function, the crossover point occurs approximately when the transmitted data size becomes over 800 Mbytes when the sizes of relations are very large over 100k pages. From our experimental results, we believe that *anti-tamper database approach* is feasible and effective against insider and outsider threats to database security.

The rest of the paper is organized as follows. In Section 2, we briefly describe the encryption techniques employed in Anti-Tamper Databases. Section 3 reviews the preliminaries of query processing techniques for aggregate queries with Group-By, and correlated nested queries, and summarizes the cost functions. Section 4 presents the cost functions for query processing in encrypted databases. In Section 5, we list six classes of aggregate queries. For each query type, we present encrypted database query processing technique and derive cost formula for the query processing in encrypted database. We mainly discuss about the query types for which encrypted database query processing entails additional costs as compared with query processing in nonencrypted databases, namely, (i) single-block queries with SUM or AVE aggregate queries, (ii) non-correlated nested queries with SUM or AVE in an inner-block query, (iii) correlated nested queries with AVE function at the outermost-level block, and (iv) correlated nested queries with AVE function at an inner-level block. Section 6 shows experimental results. Section 7 concludes.

## 2 Overview of Anti-Tamper Databases

### 2.1 Architecture

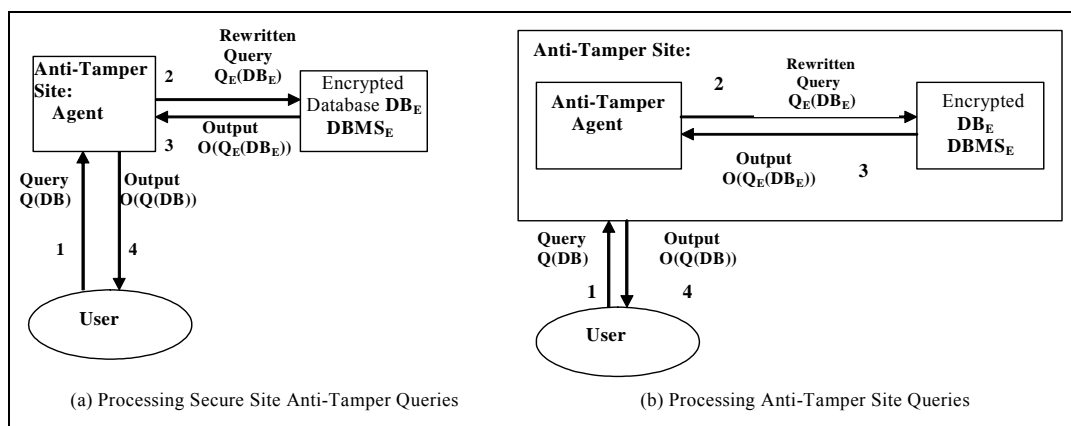


Figure 1. Proposed Architecture of the Anti-Tamper Database System

We propose the following *computing architecture* for our environment. Let the original database DB be encrypted as the *encrypted (anti-tamper) database*  $DB_E$ . Thus, even if  $DB_E$  directly becomes available to the adversary, its contents are devoid of semantics and, therefore not directly usable by the adversary. We employ an intermediary software agent, called the (Encryption/Decryption) *Agent*, which we assume to be secure. That is, the adversary cannot capture the Agent software code, and reverse-engineer its

encryption/decryption algorithms. The Agent is provided to legitimate users as a service as in the ASP model. We Consider two alternative architectures:

- The agent resides at a site different than the site of the anti-tamper database  $DB_E$ , and has significant computational power and storage space. We refer to this site as a *secure site*, and to the Agent as a *secure-site Agent*. There may also be a secure DBMS at the secure site (to be employed in our procedures). We refer to this DBMS as  $DBMS_{Agent}$ .
- The agent resides at the site of the anti-tamper database, and has little computational power. We refer to this Agent as the *anti-tamper-site Agent*.

For both architectures, user queries are processed as follows:

1. The user forms a query  $Q(DB)$  against the original database  $DB$ , and submits it to the Agent.
2. The Agent rewrites the original query  $Q(DB)$  into either
  - i. a single query  $Q_E(DB_E)$  against the encrypted database  $DB_E$  if possible, or
  - ii. a set  $\{Q_{Ei}(DB_E) | 1 \leq i \leq k\}$  of  $k$  different,  $k > 1$ , queries, and submits  $Q_E(DB_E)$  or the query set  $\{Q_{Ei}(DB_E)\}$ , respectively, to the DBMS of the anti-tamper database, referred to as  $DBMS_E$ .
3. The  $DBMS_E$  processes the query  $Q_E(DB_E)$  or the query set  $\{Q_{Ei}(DB_E)\}$ , and returns the output  $O(Q_E(DB_E))$  or the output set  $\{O(Q_{Ei}(DB_E))\}$ , respectively, to the Agent.
4. In the case of a single output  $O(Q_E(DB_E))$ , the Agent decrypts  $O(Q_E(DB_E))$  into  $O(Q(DB))$ , the legitimate output of the original query  $Q$  against the original database  $DB$ , and returns it to the user. In the case of the multiple output set  $\{O(Q_{Ei}(DB_E))\}$ , the Agent decrypts each output  $O(Q_{Ei}(DB_E))$ , performs, if necessary, additional computations with the decrypted output set; and returns the answer  $O(Q(DB))$  to the user.

Note that the encrypting of the original database is transparent and unknown to the legitimate users of the database. That is, there is no extra query specification/transformation burden placed on the legitimate users of the anti-tamper database. We assume that the commercial DBMS that hosts the data does not know that the data in its database is encrypted. This assumption is for convenience in that commercial DBMSs are complex proprietary software systems, and any security technique that is deployable without modifying the DBMS is more desirable over ones that require changes to the DBMS. We give a simple example for this process below.

## 2.2 Running Example

Throughout the paper, we use example queries conforming to a very *simplified* database schema, which is part of an existing medical insurance company database [CPH04, HI96] that contains sensitive information. Personal information about patients are to be protected from both insider and outsider threats.

Patients (pid: integer, pname: string, address: string, age: integer, ssn: integer, ins\_class: integer)

Claims (cid: integer, pid: integer, cname: string, treatment: string, day: dates, totalcost: real)

## 2.3 Example Query and its Rewriting

Note that the Agent rewrites the original query  $Q(DB)$  into either (i) a single query  $Q_E(DB_E)$  against the encrypted database  $DB_E$  if possible, or, (ii) a set  $\{Q_{Ei}(DB_E) | 1 \leq i \leq k\}$  of  $k$  different,  $k > 1$ , queries, and submits  $Q_E(DB_E)$  or the query set  $\{Q_{Ei}(DB_E)\}$ , respectively, to the DBMS of the anti-tamper database, referred to as  $DBMS_E$ . We give an example.

**Example 2.1.** Consider the query  $Q_1$  “Of all the claims for patients over 50 years old, give the age of the oldest patient for each insurance class that has more than 1 patients”, which can be expressed in SQL as in (a).

```

SELECT P.ins_class, Max(P.age)
FROM Patients P, Claims C
WHERE P.pid = C.pid AND P.age > 50
GROUP BY P.ins_class
HAVING COUNT (*) > 1

```

(a) Query Q1(DB)

```

SELECT R.A, Max(R.B)
FROM R, S
WHERE R.C = S.C AND R.B > f2(50)
GROUP BY R.A
HAVING COUNT (*) > 1

```

(b) Query Q1<sub>E</sub>(DB<sub>E</sub>)

Assume that  $f_1()$  and  $f_2()$  are used to encrypt insurance classes and patient ages. Then Patients, Claims, class, age, and pid are encrypted as the characters R, S, A, B and C. Then at the Agent site, Q1(DB) is rewritten into the query Q1<sub>E</sub>(DB<sub>E</sub>) over the encrypted database DB<sub>E</sub> as shown in (b).

Then, Q1<sub>E</sub>(DB<sub>E</sub>) is submitted to DBMS<sub>E</sub> by the Agent, to be executed against the database DB<sub>E</sub>. Upon receiving the output O(Q1<sub>E</sub>(DB<sub>E</sub>)) from DBMS<sub>E</sub>, the Agent decrypts O(Q1<sub>E</sub>(DB<sub>E</sub>)); that is, it decrypts each R.A value  $x$  and each Max(R.B) value  $y$  as  $f_1^{-1}(x)$  and  $f_2^{-1}(y)$ , respectively, and returns the decrypted output as the output O(DB) to the user.

## 2.4 Encryption techniques for Anti-Tamper Databases

In [OSC03], we proposed attribute (field)-level encryption techniques for relational databases.

**Def'n (Order-Preserving Encryption).** Consider (for simplicity, integer) attribute  $V$  of relation  $R$ , and the encryption function  $f()$  for  $V$  values. The encryption function  $f()$  is *order-preserving* if when  $a > b$  for any two  $V$  values  $a$  and  $b$  in the original database  $DB$  then  $f(a) > f(b)$  in the anti-tamper database  $DB_E$ .

In other words, the encryption of attribute  $V$  retains the ordering of values in the domain of  $V$ . Arithmetic comparison operators  $<$ ,  $>$ ,  $\geq$ , and  $\leq$  (but, not  $=$  and  $\neq$ ) of primitive data types (e.g., integers) depend on the total ordering of attribute values.

**Def'n (Difference-Preserving Encryption).** Given (again, for simplicity, integer) attribute  $V$  with nonnegative integer or real values, and an encryption function  $f()$ ,  $f()$  is *difference-preserving* if, for any two attribute  $V$  values  $a$  and  $b$  where  $(a - b) = k$ , we have  $f(a) - f(b) = r * k$ , where  $r$  is a constant.

## 2.5 Order- and Difference-Preserving Open-Form Encryption

Based on the above properties for encryption functions, the Order- and Distance-Preserving Open form encryption technique is summarized as follows:

Consider an integer-valued attribute  $V$  with values  $X$ , to be encrypted using open form order-preserving encryption. Let us assume that the encryption function  $f(X)$  and its inverse are in the form of  $E(K, X)$  and  $D(K, Y)$ , respectively, where  $K$  is the secret key. A family of functions  $Y = E(K, X)$ ,  $X = D(K, Y)$  are defined as follows, where  $X, Y$ , and  $K$  are nonnegative integers:

1.  $K$  is the secret key. Given  $K$ ,  $E$  and  $D$  should be efficiently computable.
2. For all  $X, Y, K$ , we have  $D(K, E(K, X)) = X$ . That is,  $D$  decrypts any number  $X$  encrypted by  $E$  using the same key. We will write  $E_K(X)$  for  $E(K, X)$  and  $D_K(Y)$  for  $D(K, Y)$ ; then we want  $D_K(E_K(X)) = X$ .
3. It should be hard (see inference control) to find  $X$  from  $Y$  or  $Y$  from  $X$  in the absence of knowledge of  $K$ , even assuming complete knowledge of the functions  $E$  and  $D$ .
4. If  $X < X'$  then  $E(K, X) < E(K, X')$  for any  $K$ . (Order-preservation)

Assume that the domain for our function  $E_K$  is the integers from 1 to  $N$ , and the range is the integers from 1 to  $M$ . For fixed  $K$ , let  $y_n = E_K(n)$  for  $1 \leq n \leq N$ . Define a new sequence  $z_n$  by  $z_1 = y_1 - 1$ , and  $z_{n+1} = y_{n+1} - y_n - 1$ . In other words, the  $z_i$ 's are the differences (minus 1) between successive values of the encryption function  $E_K(n)$ . Then we should have  $z_n \geq 0$  for all  $n$ . If we have any order-preserving function we can define the corresponding sequence  $z_n$ ; and the converse is also true: given a sequence  $z_1, z_2, \dots, z_n$  of nonnegative integers, there is a uniquely determined order-preserving function for which  $z_i$ 's are the differences. The algorithms in figure 2 compute the encryption function  $E_K(n)$  and its inverse  $D_K(y_n)$ . Inference control analysis and more details for order-preserving encryption techniques are presented in [OSC03].

**Encryption:**

$E_K(1) := 1 + z_1;$   
 $E_K(n+1) := E_K(n) + 1 + z_{n+1};$

**Decryption:****Input:** Y**Output:**  $D_K(Y)$ **begin**

i:= 0; W := Y;

**while** W > i **do begin** i := i + 1; W := W - z<sub>i</sub>**endwhile;****if** W = i **then** return  $D_K(Y) := i$  **else** output "Failure";**end.**

Figure 2. Encryption and decryption algorithms for order-preserving, integer-to-integer encryption

## 2.6 Order-Preserving Closed-Form Encryption

Closed-form encryption has advantages in that it allows fast anti-tamper-site decryption in case decryption and more computations are needed to generate the output at the Agent site. And another advantage of the closed-form of encryption is its speed: when additional processing at the agent site results in significant query processing time delays, we need a fast decryption algorithm.

One closed-form encryption approach is to use a single polynomial function  $f(x)$  as the encryption function E. Given n as the degree of security, the goal is to find an n degree polynomial that has an inverse function in closed form  $F(x) = C_n X^n + C_{n-1} X^{n-1} + \dots + C_1 X + C_0$ . The problem with this approach is that, in general, the closed form of the inverse function of an arbitrary polynomial function may not exist even if the inverse itself exists. Therefore, we proposed the approach of *multiple encryption functions* [OSC03], which we omit here.

## 3 Overview of Aggregate Query Evaluation in Encrypted Databases

In this section, we give a brief overview of relational database query evaluation techniques for aggregate queries with Group-By and nested queries over encrypted databases and present the cost formulas.

### 3.1 Single-Block Aggregate Queries With Group-By Clause

Consider the query Q1 in Example 2.1 above.

```
Q1:  SELECT      P.ins_class, Max(P.age) As maxage
      FROM        Patients P, Claims C
      WHERE       P.pid = C.pid AND P.age > 50
      GROUP BY   P.ins_class
      HAVING      COUNT (*) > 1
```

Although SQL is a high level declarative language, the traditional query processing systems directly implement the interpretation semantics for queries with GROUP-BY and aggregates, and defer the grouping operation until all joins in FROM and WHERE clauses are done. Note that only the columns that appear in the GROUP BY clause or in Aggregate columns can appear in the HAVING clause. In other word, the columns in HAVING should appear either in the GROUP BY list or as arguments of aggregate operators. Columns in SELECT should appear either in the GROUP BY list or as arguments to aggregate operators. In this paper, we use the term ‘interpretation semantics’ of a query to interpret the query and we use the term ‘execution semantics’ of a query to define the evaluation procedure of the query. The generalized execution semantics of SQL query with GROUP-BY and aggregates is as follows:

1. Construct a joined table of all the tables in the FROM list, and apply the predicates in WHERE clause. Of course, this step is implemented as an expression with Joins and Selections, etc.
2. Eliminate unwanted columns, keep the columns listed in SELECT, GROUP BY.
3. Sort the result by the columns in the GROUP BY clause to identify the groups.
4. Apply the predicates in the HAVING clause to each group.

5. Generate the answer row per the remaining group, plus one or more columns generated by applying aggregate operators to each group.
6. Eliminate the duplicates if there is DISTINCT in the SELECT clause.

Aggregation operators, such as SUM, AVG, etc. are evaluated by scanning the entire relation while maintaining running information about the scanned tuples. The running information for SUM is total of the values retrieved and the running information for AVE is <total, count> of the values retrieved.

When aggregate operators appear in a HAVING clause, generally there are two evaluation algorithms (without using any indexes), one is *sorting* and the other is *hashing*. Sorting algorithm first (1) sorts the relation on the grouping attributes, then (2) scans it again to compute the result of the aggregate operations for each group (while applying any other HAVING predicates, if any). Hashing algorithm first (1) builds a hash table on the grouping attributes with the entries in the form of <grouping-value, running-info of aggregation>, then (2) scans the relation probing the hash table, for each tuple, to find which group the tuple belongs to and updates the running info. When the hash table does not fit into the main memory, (1) the relation is partitioned using a hash function on a grouping-value, and (2) each partition is processed independently by building an in-memory hash table for the tuples in it (updating the running-info).

### 3.2 Correlated Nested Query with Aggregate Functions

Nested queries can be divided into two categories, non-correlated nested queries and correlated nested queries. Non-correlated nested queries can be evaluated by evaluating nested query first, then using it to evaluate the top level query through the nested loops join of the relation in the top level as the outer relation and the computed inner query as the inner relation, however a non-correlated nested query can always be transformed to a single block (non-nested) query. So here, we only consider correlated nested queries. Correlated nested queries have a nested predicate that contain a join predicate that refers to relations in the outer block. We give an example.

```

Q2:  SELECT      AVE(P.age)
      FROM        Patients P
      WHERE       EXISTS ( SELECT      *
                           FROM        Claims C
                           WHERE       C.totalcost > 10,000
                                   AND P.pid = C.pid )

```

This query is correlated since the tuple variable P in the outer block of the query appears in the inner block of the query. The inner block of the query is evaluated for each tuple of Patients in the outer block of the query. Here, we list limitations in optimization of nested query processing:

1. If the same values appear in the correlated field of several outer tuples, then the same subquery is evaluated many times.
2. The approach to the nested subquery is not set-oriented; so join is seen as a scan of the outer relation and a selection of inner subquery of each outer tuple. This excludes other alternatives (e.g., sort-merge join, hash join).
3. In the above query, even if the nested loops join of Patients as the outer relation and the Claims as the inner relation with applying selection on the fly is the best plan, this plan is not considered by the Optimizer because the nested subquery is fully evaluated first.
4. If there is index for pids of Claims, then the best plan would be the index-nested loop join of Patients as the outer loop and Claims as the inner loop; but because of the implicit ordering imposed by the nesting, such a plan is not considered.
5. Even if there is an equivalent query without nesting and correlation (thus proper optimization can be done), recognizing such equivalences is difficult in general.

## 4 Cost Analysis for Query Processing Over an Encrypted Database

First, let us look at how to process queries over an encrypted database, especially for aggregate queries.

**Example 4-1.** Assume that the relation *Claims* and the attribute name *totalcost* are encrypted as *Claims<sub>E</sub>* and *totalcost<sub>E</sub>*, and the attribute *totalcost* is encrypted using an open-form order-preserving encryption function  $f()$ . Then, the SQL query

```
Q3:   SELECT AVG (Claims.totalcost)
      FROM Claims
      WHERE totalcost > 10,000;
```

needs the evaluation of two queries:

```
Q3E-1(DBE):   SELECT ClaimsE.totalcostE FROM ClaimsE WHERE totalcostE > f(10,000); and
Q3E-2(DBE):   SELECT COUNT(ClaimsE.totalcostE) FROM ClaimsE WHERE totalcostE > f(10,000);
```

Without the assumption that  $f()$  is additive (i.e.,  $f(a + b) = f(a) + f(b)$  for any  $x$  and  $y$  values in the attribute,  $f^{-1}(f(x) + f(y)) = f^{-1}(f(x + y)) = (x + y)$ ), the Agent computes the answer of Q2 as follows: Upon receiving  $O_1(Q2_{E-1}(DB_E))$  and  $O_2(Q2_{E-2}(DB_E))$ , it (1) decrypts  $O_1(Q2_{E-1})$ , (2) computes  $SUM(f^{-1}(O_1(Q2_{E-1})))$ , and (3) divides it by  $O_2(Q2_{E-2})$ . When there are aggregate operations in nested SQL queries (with correlated variables), the issue of rewriting the query becomes even more difficult.

Cost metric  $C_{QE}$  as the processing cost of query  $Q_E$  over the encrypted database reflects the overhead coming from query transformation that may rewrite the original query into two or more queries if necessary, and the transmission of data to the Agent site for additional computations. Therefore, we define  $C_{QE}$  as follows:

$$C_{QE} = C_{\text{anti-tamper}} + C_{\text{agent}} + C_t$$

where  $C_{\text{anti-tamper}}$  is the query processing cost at anti-tamper site,  $C_{\text{agent}}$  is the query processing cost at agent site, and  $C_t$  is the data transmission cost between the anti-tamper site and the agent site over a network. We do not assume that any of the attributes in encrypted database is sorted or has an index.

Although an accurate cost model should employ more complex factors like CPU time and memory for the cost analysis, considering today's fast-growing computational power and cheap memory cost, our cost analysis mainly focuses on the disk I/O cost. The cost model here relies on formulas to estimate the I/O costs of query execution techniques over the encrypted database. For example, in the case query splitting is required, the extra cost for splitting queries should be estimated properly. Furthermore, given that the proposed transformation techniques here for encrypted database query processing rely on moving around the GROUP-BY clause with aggregates, we need to estimate the size of the relations after group-by is moved.

### 4.1 Cost of Aggregate Query Evaluation

We summarize the query processing cost of aggregate queries in relational DBMSs. We assume that there is no index for attributes, and the files are not sorted. The expected cost of processing the query Q1 in 3.1 depends on the costs of the join, and the grouping and aggregation operators. Let  $N_p$ ,  $N_c$  be the numbers of data pages (disk blocks) in the relations Patients and Claims respectively. Assume that there are  $R_p$ ,  $R_c$  records (tuples) per page for Patients and Claims, and it takes on the average  $T$  time to process a record (tuple), and the average time of  $P$  to read or write a page. Then the I/O cost to scan the relation Patients is  $N_p*(T*R_p + P)$ , the relation Claims is  $N_c*(T*R_c + P)$ . To simplify the presentation, from now on, we concentrate only on the disk access costs and use  $N_p$ ,  $N_c$  instead of  $N_p*(T*R_p + P)$ ,  $N_c*(T*R_c + P)$ , respectively. Throughout this paper, we use the terminology as defined in Table 1.

Term	Definition
$C_Q$	Query processing cost over non-encrypted relational databases
$C_{QE}$	Query processing cost over encrypted databases
$C_{\text{anti-tamper}}$	Query processing cost over encrypted database at anti-tamper site
$C_{\text{agent}}$	Query processing cost over encrypted database at agent site
$C_t$	Data transmission cost between anti-tamper site and agent over a network
$R_t$	The number of records to be transmitted from anti-tamper site to agent for computation at agent site
$R_p$	The number of records (tuples) per page in relation Patients
$R_c$	The number of records (tuples) per page in relation Claims
$s_j$	Join Selectivity
$s_p$	Selectivity of Predicate
$r_p$	Projection ratio: Ratio of the size of temporary table after projection
$g_f$	GROUP-BY ratio: Ratio of the size of temporary table after GROUP-BY over the size before-GROUP-BY table
$N_C$	The number of pages of relation Claims
$N_P$	The number of pages of relation Patients
$N_{c-p}$	The number of partitions (groups of duplicates) in relation Claims
$pr_1$	Ratio of the number of pages of duplicates in a partition over relation Claims, $0 < pr_1 \leq 1$
$pr_2$	Rate of the number of pages of duplicates in a partition over relation Patients, $0 < pr_2 \leq 1$

**Table 1.** Terminology used in Cost formulas

A. *Cost of join*:  $j(N_p, N_c)$  represents cost of join of the relations Patients and Claims in Q1.

1) Nested-loops join ( $j_N$ ) (with relation Patients as the inner relation and relation Claims as outer relation): the total cost with page-at-a-time access is  $(N_c + N_c * N_p)$ .

2) Block nested loops join ( $j_{BN}$ ) with  $B + 2$  buffer pages (one to scan the larger relation, another to write): When the smaller relation (outer relation) does not fit into memory, then break  $N_p$  into blocks that fit into  $B$  buffer pages, build a hash table in main memory for each block of Patients, then scan  $N_c$  (Claims as inner relation) for each block of Patients, thus the total cost for this join is  $N_p + N_c * \lceil N_p/B \rceil$ .

3) Sort-Merge join ( $j_{SM}$ ): In general, total cost for sort-merge join is  $N_c \lg N_c + N_p \lg N_p + N_c + N_p$  (sorting cost for Claims is  $N_c \lg N_c$ , for Patients is  $N_p \lg N_p$ , and the cost for merging phase is  $N_c + N_p$ , if there is no repetitive merging of duplicates). If there are a lot of duplicates (partitions) in each relation, in merging phase, a partition (of duplicates) in RHS (right hand side) relation is repeatedly scanned for each tuple in the corresponding current partition in LHS (left hand side) relation, in which case, merging phase costs the number of pages in RHS relation partition times the number of tuples in LHS relation partition.  $pr_1$  in Table 1 is ratio of the number of pages of duplicates in a partition over relation Claims, that is,  $(pr_1 * N_c * R_c)$  is the number of tuples (of duplicates) in a partition in the relation Claims. If join is on key attributes of the relation Claims, then there is no repetitive merging over the corresponding partition of Patients, that is,  $pr_1 = 1 / (N_c * R_c)$  and  $N_{c-p} = 0$ . In the similar way for  $pr_2$ ,  $(pr_2 * N_p)$  is the number of pages (of duplicates) of a partition in the relation Patients, thus,

$(N_c + N_p) < N_{c-p} * [(pr_1 * N_c * R_c) * (pr_2 * N_p)] < (N_c * N_p)$ . The cost formula derived here is a block-based cost, which is based on Block-based Reread (BR-n) optimization which includes combining the merging phase of sorting with merging phase of join [LGS02].

4) Hash join ( $J_H$ ): Read and write (for each relation) for building a hash table for both relations in disk costs  $2(N_c + N_p)$ , plus read each partition of both relations into the memory, and apply the second hash function for finding matching costs  $N_c + N_p$ . Therefore, the total cost for hash join is  $3(N_c + N_p)$ .

*B. Cost of Projection  $P(N_c)$* : represents cost of projection over relation Claims in Q1.

1) Sort-based Projection over Claims: Cost for projection over Claims is  $N_c$  (for scanning Claims to remove all the unwanted columns) +  $T \log T$  (for sorting the result table) +  $T$  (for scanning to eliminate duplicates) where  $T$  is the size of the temporary relation after projection, which is a fraction of  $N_c$ . At best, sort-based projection can be improved by doing the first phase of merge sort, and then, duplicate elimination can be done at the merging phase of sort, in which case, total cost would be  $N_c + T \log T$  [RR98].

2) Hash-based Projection over Claims (with one input buffer and  $B-1$  output buffer): Total cost for hash based projection is  $N_c$  (to remove all the unwanted columns and to apply a hash function for each tuple of Claims) +  $2T$  ( $T$  to write the result of the hash table, and  $T$  to duplicate elimination) when  $B >$

$\sqrt{f * T}$  where  $T$  is the size of temporary relation after projection, which is a fraction of  $N_c$  and  $f$  is a factor to capture the small increase in size between the partition and the hash table for the partition [RR98].

*C. Cost of Aggregations*

Aggregation operators can be used in combination with a GROUP-BY clause as in Q1. If we sort the temporary relation that results after projection in Q1 on the grouping attributes and then scan it again to compute the result of the aggregation operation for each group, then the I/O cost is just the cost of the sort [RR98], so it is  $T \log_2 T$  where  $T$  is a temporary relation resulted after projection. When the table is sorted on the grouping attributes or aggregating without grouping, then the cost of aggregation is just  $T$ , the cost of scanning the table.

We define the following functions for cost formulas over relation Claims and relation Patients in Table 2 below:

Function	Definition	Cost Formula
$J(N_c, N_p)$	Cost of join of relation C and relation P	Either one of $J_{BN}(N_c, N_p)$ , $J_{SM}(N_c, N_p, N_{c-p}, pr_1, pr_2)$ , or $J_H(N_c, N_p)$
$J_N(N_c, N_p)$	Cost of Nested Loops Join of relation C and relation P	$N_c + N_c * N_p$
$J_{BN}(N_c, N_p)$	Cost of Block Nested Loops Join of relation C and relation P	$N_c + N_p * \lceil N_c / (B-2) \rceil$ (where $B$ is the number of buffer pages)
$J_{SM}(N_c, N_p, N_{c-p}, pr_1, pr_2)$	Cost of Sort-Merge Join of relation C and relation P	$N_c \log N_c + N_p \lg N_p$ $+ [N_{c-p} * (pr_1 * N_c * R_c) * (pr_2 * N_p)]$
$J_H(N_c, N_p)$	Cost of Hash Join of relation C and relation P	$3 * (N_c + N_p)$
$P(N_c)$	Cost of Projection over relation C	Either one of $P_S(N_c)$ or $P_H(N_c)$
$P_S(N_c)$	Cost of Sort-based Projection Method over relation C	$N_c + (r_p * N_c) \lg(r_p * N_c)$
$P_H(N_c)$	Cost of Hash-based Projection	$N_c + 2 * (r_p * N_c)$

	Method over relation C	
--	------------------------	--

**Table 2.** Definition of Cost Functions with C as Relation Claims, P as Relation Patients

## 5 Aggregate Queries And Their Cost Differences

For single-block queries with Min, Max, or Count, and non-correlated nested queries with Min, Max, or Count, the evaluation costs over encrypted database are the same as those over non-encrypted relational databases. In this section, we define 5 query types. For each query type, we give an example query, present the query processing technique over encrypted database, and then we derive cost differential between encrypted database query processing and non-encrypted database query processing. We focus on aggregate queries that require extra query processing for encrypted databases.

Assume that each relation and attribute is encrypted as follows:

Patients  $\rightarrow$  Patients<sub>E</sub>, Claims  $\rightarrow$  Claims<sub>E</sub>, pid  $\rightarrow$  pid<sub>E</sub>, pname  $\rightarrow$  pname<sub>E</sub>, address  $\rightarrow$  address<sub>E</sub>, ssn  $\rightarrow$  ssn<sub>E</sub>, age  $\rightarrow$  age<sub>E</sub>, insclass  $\rightarrow$  ins\_class<sub>E</sub>, cid  $\rightarrow$  cid<sub>E</sub>, cname  $\rightarrow$  cname<sub>E</sub>, treatment  $\rightarrow$  treatment<sub>E</sub>, day  $\rightarrow$  day<sub>E</sub>, totalcost  $\rightarrow$  totalcost<sub>E</sub>, and  $f_e$  is an encryption function for the attribute values.

### 5.1 Single Query Block with Min, Max, Count

Query Type 1:

```
SELECT      AGG(Rj.Ah)
FROM        Ri, Rj
WHERE       Ri.Ah = C
```

Where AGG is an aggregate function among Count, Min, and Max. As for the predicates in WHERE clause, the simple predicates of the form  $[R_i.A_k \theta C]$  are allowed for single-relation queries, and for multiple-relation queries, the simple predicates can only be extended into only join Predicate, that is,  $C$  may be in the form of  $R_j.A_t$ , that is,  $[R_i.A_h \theta C]$  may be in the form of  $[R_i.A_h \theta R_j.A_t]$ , where  $\theta$  is a scalar operator or a set membership operator.

Q4:

```
SELECT      MAX(P.age)
FROM        Patients P, Claims C
WHERE       P.pid = C.pid AND C.totalcost  $\geq$  10,000;
```

Q4 is rewritten to QE4 to query over encrypted database as follow:

QE4:

```
SELECT      MAX(P.ageE)
FROM        PatientsE P, ClaimsE C
WHERE       P.pidE = C.pidE AND C.totalcostE  $\geq$   $f_e(10,000)$ ;
```

Execution Semantics of Q4 at DBMS:

- 1) Apply the predicate  $C.totalcost \geq 10,000$  over Claims, call the result Temp1.
- 2) Do join of Temp1 and Patients on pid, call the result Temp2.
- 3) Project pid and age, call the result Temp3.
- 4) Scan the computed result Temp3 maintaining the running information for MAX(P.age).

5) Generate the output.

$$C_{Q4} = Nc + J(N_{Temp1}, Np) + P(N_{Temp2}) + N_{Temp3}$$

where  $N_{Temp1} = s_p * Nc$  with selectivity of predicate  $s_p$

$$N_{Temp2} = s_j * (N_{Temp1} * Np) \text{ with join selectivity of } s_j$$

$$N_{Temp3} = r_p * N_{Temp2} = r_p * (s_j * s_p * Nc * Np) \text{ with projection rate } r_p$$

In case sort-merge join and sort-based projection are employed:

$$\begin{aligned} C_{Q4} &= Nc + J_{SM}(s_p * Nc, Np, Nc_p, pr_1, pr_2) + P_H(N_{Temp2}) + N_{Temp3} \\ &= Nc + (s_p * Nc)lg(s_p * Nc) + NplgNp + (s_j * s_p + 3r_p * s_j * s_p + Nc_p * pr_1 * sp * pr_2 * Rc) * Nc * Np \end{aligned}$$

Execution Semantics of QE4 at DBMS<sub>E</sub>:

- 1) Apply the predicate  $C.totalcost_E \geq f_e(10,000)$  over  $C_E$ , call the result Temp1.
- 2) Do sort-merge join of Temp1 and Patients<sub>E</sub> on pid<sub>E</sub>, call the result Temp2.
- 3) Project pid<sub>E</sub> and age<sub>E</sub>, call the result Temp3.
- 4) Scan the computed result Temp3 maintaining the running information for MAX(P.age<sub>E</sub>).
- 5) Generate the output.

$$C_{QE4} = C_{anti-tamper} + C_{agent} + C_t$$

$$C_{anti-tamper} = Nc + J(N_{Temp1}, Np) + P(N_{Temp2}) + N_{Temp3}$$

$$= Nc + J_{SM}(N_{Temp1}, Np, Nc_p, pr_1, pr_2) + P_H(N_{Temp2}) + N_{Temp3}$$

$$= Nc + (s_p * Nc)lg(s_p * Nc) + NplgNp + (s_j * s_p + 3r_p * s_j * s_p + Nc_p * pr_1 * sp * pr_2 * Rc) * Nc * Np$$

where sort-merge join and hash-based projection are employed. Since query transformation for encrypted database is directly one-to-one transformation, there is no extra query processing cost for encrypted database at anti-tamper site, no computation at agent site, and no transmission cost, so  $C_{agent} = 0$ ,  $C_t = 0$ . Thus,  $C_{QE4}$  is same as  $C_{Q4}$ . Therefore,  $C_{QE4} - C_{Q4} = 0$ , that is, there is no extra cost for processing Query Type 1 over encrypted database.

## 5.2 Single Query Block with Sum or Ave as Aggregate Functions

Query Type 2:

```
SELECT      AGG(Rj.Ah)
FROM        Ri, Rj
WHERE       [Ri.Ah = C]
```

where AGG is an aggregate function among Sum, Ave. As for the predicates in WHERE clause, the simple predicates of the form  $[R_i.A_k \theta C]$  are allowed for single-relation queries, and for multiple-relation queries, the simple predicates can only be extended into only join Predicate, that is,  $C$  may be in the form of  $R_j.A_t$ , that is,  $[R_i.A_h \theta C]$  may be in the form of  $[R_i.A_h \theta R_j.A_t]$ , where  $\theta$  is a scalar operator or a set membership operator.

```
Q5:  SELECT      AVE(P.age)
      FROM        Patients P, Claims C
      WHERE       P.pid = C.pid AND C.totalcost ≥ 10,000;
```

Execution Semantics of Q5 at DBMS:

- 1) Apply the predicate  $C.totalcost \geq 10,000$  over Claims, call the result Temp1.

- 2) Do join of Temp1 and Patients on pid, call the result Temp2.
- 3) Project pid and age, call the result Temp3.
- 4) Scan the computed result Temp3 maintaining the running information for AVE(P.age).
- 5) Generate the output.

$$C_{Q5} = Nc + J(N_{Temp1}, Np) + P(N_{Temp2}) + N_{Temp3}$$

In case sort-merge join and sort-based projection are employed,

$$C_{Q5} = Nc + J_{SM}(s_p * Nc, Np, Nc_p, pr_1, pr_2) + P_H(N_{Temp2}) + N_{Temp3}$$

$$= Nc + (s_p * Nc) \lg(s_p * Nc) + Np \lg Np + (s_j * s_p + 3r_p * s_j * s_p + Nc_p * pr_1 * sp * pr_2 * Rc) * Nc * Np$$

where  $N_{Temp1} = s_p * Nc$  with selectivity of the predicate  $s_p$

$N_{Temp2} = s_j * (N_{Temp1} * Np)$  with selectivity of join predicate  $s_j$

$N_{Temp3} = r_p * N_{Temp2} = r_p * (s_j * s_p * Nc * Np)$  with projection rate  $r_p$

Q5 is rewritten to QE5 for the encrypted database as follow:

```
QE5:  SELECT      AVE(P.ageE)
      FROM        PatientsE P, ClaimsE C
      WHERE       P.pidE = C.pidE AND C.totalcostE ≥ fe(10,000);
```

Since there is AVE in Q5, query splitting is needed to compute AVE(P.age<sub>E</sub>), so Q5 is rewritten to QE5-1 and QE5-2 for processing over encrypted database as follows:

```
QE5-1: SELECT      COUNT(P.ageE)
      FROM        PatientsE P, ClaimsE C
      WHERE       P.pidE = C.pidE AND C.totalcostE ≥ fe(10,000);
```

```
QE5-2: SELECT      P.pidE, P.ageE
      FROM        PatientsE P, ClaimsE C
      WHERE       P.pidE = C.pidE AND C.totalcostE ≥ fe(10,000);
```

Note that P.pid<sub>E</sub> is added in the select list in QE5-2 to get all the ages of the selected patients.

Execution Semantics of QE5-1 at DBMS<sub>E</sub>:

- 1) Apply the predicate  $C.totalcost_E \geq f_e(10,000)$  over Claims<sub>E</sub>, call the result Temp1.
- 2) Do sort-merge join of Temp1 and Patients<sub>E</sub> on pid<sub>E</sub>, call the result Temp2.
- 3) Project pid<sub>E</sub> and age<sub>E</sub>, call the result Temp3.
- 4) Scan the computed result Temp3 maintaining the running information for COUNT(P.age<sub>E</sub>).
- 5) Generate the output.

C<sub>QE5-1</sub> is the same as C<sub>Q5</sub> as follow:

$$C_{QE5-1} = Nc + J(N_{Temp1}, Np) + P(N_{Temp2}) + N_{Temp3}$$

$$= Nc + J_{SM}(s_p * Nc, Np, Nc_p, pr_1, pr_2) + P_H(N_{Temp2}) + N_{Temp3}$$

$$= Nc + (s_p * Nc) \lg(s_p * Nc) + Np \lg Np + (s_j * s_p + 3r_p * s_j * s_p + Nc_p * pr_1 * sp * pr_2 * Rc) * Nc * Np$$

Execution Semantics of QE5-2 at DBMS<sub>E</sub>:

- 1) Apply the predicate  $C.totalcost_E \geq f_c(10,000)$  over Claims<sub>E</sub>, call the result Temp1.
- 2) Do sort-merge join of Temp1 and Patients<sub>E</sub> on pid<sub>E</sub>, call the result Temp2.
- 3) Project pname<sub>E</sub> and age<sub>E</sub>, call the result Temp3.
- 4) Generate the output.

$$C_{QE5-2} = Nc + J(N_{Temp1}, Np) + P(N_{Temp2})$$

$$\text{Thus, } C_{QE5\text{anti-tamper}} = C_{QE5-1} + C_{QE5-2} = 2*[Nc + J(N_{Temp1}, Np) + P(N_{Temp2})] + N_{Temp3}$$

After obtaining the result of QE5-1 and QE5-2, the Agent needs to (1)decrypt P.age<sub>E</sub> to P.age, (2)compute SUM(P.age), and 3)compute AVE(P.age) using COUNT(P.age<sub>E</sub>), thus  $C_{agent} = 2*Cost \text{ for Scan Temp3} = 2*N_{Temp3} = 2*(r_p*s_j*s_p*Nc*Np)$ . In addition, there is extra cost for transmitting all P.age<sub>E</sub> to the agent site, so  $C_t$  is transmitted data size, that is, the cost for transmitting  $N_{Temp3}$  ( $= r_p*s_j*s_p*Nc*Np$ ). Thus, total cost of processing QE5 over encrypted database is the sum of all the above.

$$C_{QE4} = C_{QE5-1} + C_{QE5-2} + C_{agent} + C_t$$

$$= 2*[Nc + (s_p*Nc)lg(s_p*Nc) + Np lg Np + (s_j*s_p + 2*r_p*s_j*s_p + N_{c-p}*pr_1*sp*pr_2*Rc)*Nc*Np] + 4*r_p*s_j*s_p*Nc*Np$$

Therefore, cost difference of processing Q5 is as follows:

$$C_{QE5} - C_{Q5} = C_{QE5-2} + C_{agent} + C_t$$

$$= Nc + (s_p*Nc)lg(s_p*Nc) + Np log Np + (s_j*s_p + 2*r_p*s_j*s_p + N_{c-p}*pr_1*sp*pr_2*Rc)*Nc*Np + 2*r_p*s_j*s_p*Nc*Np + C_t$$

This implies that extra cost for Query Type 2 is the cost of processing QE4-2 +  $2*N_{Temp3} + C_t$ .

### 5.3 Non-Correlated Nested Queries without SUM or AVE

Query Type 3:

```
SELECT      AGG(Ri.Am)
FROM        Ri
WHERE [Ri.Ah = Ci]
           Ri.Am θ (SELECT      AGG(Rj.Ah)
                   FROM        Rj
                   WHERE        [Rj.As = Ck]);
```

where AGG is either one of COUNT, MIN, and MAX.

An example of Query type 3 and the rewritten query of Q6 are as follows:

Q6:

```
SELECT      MAX(P.age)
FROM        Patients P
WHERE       P.pid IN ( SELECT      C.pid
                       FROM        Claims C
                       WHERE        C.totalcost >= 10,000 );
```

Execution Semantics of Query at DBMS For Q6:

- 1) Compute the nested subquery first from the relation Claims, then name the computed result as Temp2.
  - 1-1) Scan Claims to compute the predicate, call the result Temp1
  - 1-2) Project pid, call it Temp2
- 2) Do the nested loops join of Patients as the outer relation and Temp2 as the inner relation on pid, call the result Temp3.
- 3) Project pid and age, call the result Temp4.
- 4) Scan the computed result maintaining the running information for MAX(P.age).
- 5) Generate the output.

$$\begin{aligned}
 C_{Q6} &= N_c + P_H(N_{Temp1}) + J_{BN}(N_p, N_{Temp2}) + P_H(N_{Temp3}) + N_{Temp4} \\
 &= N_c + N_{Temp1} + 2 * N_{Temp2} + N_p + N_{Temp2} * \lceil N_p / (B-2) \rceil + N_{Temp3} + 2 * N_{Temp4} + N_{Temp4} \\
 &= (1 + s_p + 2 * r_p * s_p) N_c + N_p + (r_p * s_p * \lceil 1 / (B-2) \rceil + (s_j * r_p * s_p + 3 * r_p^2 * s_j * s_p)) N_c * N_p
 \end{aligned}$$

where Size of Temp1,  $N_{Temp1} = s_p * N_c$  with selectivity of the predicate  $s_p$

Size of Temp2,  $N_{Temp2} = r_p * N_{Temp1} = r_p * s_p * N_c$  with project rate  $r_p$

Size of Temp3,  $N_{Temp3} = s_j * N_p * N_{Temp2} = s_j * (r_p * s_p * N_c) * N_p$  with selectivity of join predicate  $s_j$

Size of Temp4,  $N_{Temp4} = r_p * N_{Temp3} = r_p * s_j * (r_p * s_p * N_c) * N_p$  with projection rate  $r_p$

And Hash-based projection method is employed.

Q6 is rewritten to QE6 as follows:

QE6:

```

SELECT      MAX(P.ageE)
FROM        PatientsE P
WHERE       P.pidE IN      (SELECT      C.pidE
                             FROM        ClaimsE C
                             WHERE       C.totalcost ≥ fc(10,000));

```

Execution semantics for QE6 is same as that of Q6, and there is no extra cost for encrypted database at anti-tamper site, no computation cost at agent site, so no transmission cost for processing QE6, so  $C_{agent} = 0$ ,  $C_t = 0$ . Thus, the total cost for processing QE3 is same as Q6. Therefore, there is no extra cost for processing Query Type 3 over encrypted database.

#### 5.4 Non-Correlated Nested Queries with SUM or AVE in the Nested Query

Q7:

```

SELECT      AVE(P.age)
FROM        Patients P
WHERE       P.pid IN (
               SELECT      C.pid)
             FROM        Claims C
             WHERE       C.totalcost ≥ 10,000 );

```

Non-correlated nested queries can always be rewritten to the equivalent unnested queries, which then become of Query Type 2. For example, Q7 above is equivalent to Q5 in Query Type2. Thus, the query evaluation technique for Q7 over encrypted database is the same as that for Q5. However, there are non-correlated nested queries that cannot always be rewritten to the equivalent unnested queries, which can be expressed as Query Type 4 as follow:

Query Type 4:

```

SELECT      Ri.Am, Ri.Ak
FROM        Ri
WHERE       [Ri.Ah = Ci],
           Ri.Am > (SELECT      AGG(Rj.Ah)
                   FROM        Rj
                   WHERE       [Rj.As = Ck]);

```

Where AGG is either SUM or AVE.

Since SQL donot allow that any attribute appears with aggregate operation in the same SELECT clause unless the query contains a GROUP-BY, Query Type 4 above cannot be transformed to an equivalent single block query. SQL92 allow this type of queries. Non-correlated subquery is evaluated first, in this case the result is a single valued table, then it is converted to a value to be used to evaluate the top-level query.

```

Q8:  SELECT      P1.pname, P1.age
      FROM        Patients P1
      WHERE       P1.age > ( SELECT      AVE(P2.age)
                           FROM        Patients P2
                           WHERE       P.insclass = '1' );

```

Q8 is transformed to QE8 for encrypted database as follows:

```

QE8:  SELECT      P1.pnameE, P1.ageE
      FROM        PatientsE P1
      WHERE       P1.ageE > ( SELECT      AVE(P2.ageE)
                           FROM        PatientsE P2
                           WHERE       P2.ins_class.E = fc(1);

```

Since there is AVE in the nested query of QE8, query splitting is needed to compute AVE(P.age<sub>E</sub>). However, since AVE is in the nested query and non-correlated subquery is always evaluated independently, the subquery of Q8 is rewritten to QESub8-1 and QESub8-2 as follows;

```

QESub8-1:  SELECT      COUNT(P2.ageE)
            FROM        PatientsE P2
            WHERE       P2.ins_class.E = fc(1);

```

```

QESub8-2:  SELECT      P2.pidE, P2.ageE
            FROM        PatientsE P2
            WHERE       P2.ins_class.E = fc(1);

```

After obtaining the results of QESub8-1 and QESub8-2, the Agent computes AVE(P2.age), say the computed value is x, then submits QE8-1 as follow:

```

QE8-1:  SELECT      P1.pname, P1.ageE
        FROM        PatientsE P1
        WHERE       P1.ageE > f(x);

```

Execution Semantics of Query at DBMS For Q8:

- 1) Compute the nested subquery first from the relation Patients, then name the computed result as Temp3.
  - 1-1) Scan Patients to compute the predicate, call the result Temp1
  - 1-2) Project age, call it Temp2
  - 1-3) Scan the computed result Temp2 maintaining the running information for AVE(P2.age).
- 2) Convert the result Temp3 to a single value, then scan Patients applying the predicate with the converted value. Call the result Temp4.
- 3) Project pname and age, call the result Temp5.
- 4) Generate the output.

$$\begin{aligned}
 C_{Q6} &= N_p + P_H(N_{Temp1}) + N_{Temp2} + N_p + P_H(N_{Temp4}) \\
 &= N_p + N_{Temp1} + 2 * N_{Temp2} + N_{Temp2} + N_p + N_{Temp4} + 2 * N_{Temp5} \\
 &= N_p * (2 + 2 * s_p) + 5 * r_p * s_p * N_p
 \end{aligned}$$

where Size of Temp1,  $N_{Temp1} = s_p * N_p$  with selectivity of the predicate  $s_p$ ,  
 Size of Temp2,  $N_{Temp2} = r_p * N_{Temp1} = r_p * s_p * N_p$  with project rate  $r_p$   
 Size of Temp3,  $N_{Temp3} = 1$  (a single valued table)  
 Size of Temp4,  $N_{Temp4} = s_p * N_p$   
 Size of Temp5,  $N_{Temp5} = r_p * N_{Temp4} = r_p * s_p * N_p$  and,  
 Hash-based projection is employed

Execution Semantics of Query at DBMS<sub>E</sub> For QESub8-1 is the same as the subquery of Q8, QESub8-2 is the same as QESub8-1 except the step for scanning Temp2 for COUNT, and QE8-1 is the same as the top-level query of Q8.

$$\begin{aligned}
 C_{QESub8-1} &= N_p + P_H(N_{Temp1}) + N_{Temp2} = N_p * (1 + s_p + 3 * r_p * s_p) \\
 C_{QESub8-2} &= N_p + P_H(N_{Temp1}) = N_p * (1 + s_p + 2 * r_p * s_p) \\
 C_{QE8-1} &= N_p + P_H(N_{Temp4}) = N_p + N_{Temp4} + 2 * N_{Temp5} = N_p * (1 + s_p + 2 * r_p * s_p)
 \end{aligned}$$

$$\text{Therefore, } C_{\text{anti-tamper}} = C_{QESub8-1} + C_{QESub8-2} + C_{QE8-1} = 3 * N_p * (1 + s_p) + 7 * N_p * r_p * s_p$$

Computations at Agent Site:

1. Upon obtaining the result of QESub8-1 and QESub8-2, the Agent 1) decrypts P.age<sub>E</sub> to P.age, 2) computes SUM(P.age), 3) computes AVE(P.age) using COUNT(P2.age<sub>E</sub>), thus  $C_{\text{agent}} = 2 * \text{Cost}$  for Scanning Temp2 =  $2 * N_{Temp2} = 2 * r_p * s_p * N_p$ .
2. After computing AVE(P.age) = x, the agent encrypts x, then submits QE8-1 with f(x).

In addition, there is extra cost for transmitting all P.age<sub>E</sub> to the agent site, so  $C_t$  is the cost for transmitting  $N_{Temp2}$  ( $= r_p * s_p * N_p$ ). Thus, the total cost of processing QE8 is the sum of all the above.

$$\begin{aligned}
 C_{QE8} &= C_{QESub8-1} + C_{QESub8-2} + C_{QE8-1} + C_{\text{agent}} + C_t \\
 &= 3 * N_p * (1 + s_p) + 3 * N_p * r_p * s_p + 9 * N_p * r_p * s_p + C_t
 \end{aligned}$$

Therefore, cost differential of Query Type 2 is

$$C_{QE8} - C_{Q8} = Np*(1 + s_p + 4*r_p*s_p) + C_t$$

## 5.5 Nested and Correlated Queries with AVG at Top level query

Query Type 5:

```

SELECT      Ri.Ak, AGG(Ri.Am)
FROM        Ri
WHERE       [Ri.Ah = Cj]
GROUP BY   Ri.Ak
HAVING      Ri.Am θ (SELECT      AGG(Rj.Ah)
                        FROM        Rj
                        WHERE       Rj.At = Ri.Ak AND
                        [Rj.As = Ck]);

```

Where AGG(R<sub>i</sub>.A<sub>m</sub>) in outer block query is either Sum or Ave, and AGG(R<sub>i</sub>.A<sub>m</sub>) in inner block query is either Min, Max or COUNT.

Q: Find the average age of patients who are of adult age 21 for each insurance class that has 100 or more patients of age 50 or more.

```

Q9:  SELECT      P.ins_class, AVE(P.age) AS avgage
      FROM        Patients P
      WHERE       P.age ≥ 21
      GROUP BY   P.ins_class
      HAVING      100 < ( SELECT      COUNT(*)
                        FROM        Patients P2
                        WHERE       P2.ins_class = P.ins_class AND P2.age > 50);

```

Note that only the columns that appear in GROUP BY or Aggregate Columns in the top level query can appear in the correlated subquery in the HAVING clause. Q9 is a correlated nested query because the variable P in the top-level appears in its subquery. Q9 is rewritten to QE9-1 and QE9-2. Query execution technique for Query Type 5 over encrypted database is applied without removing correlation.

QE9-1:

```

SELECT      P.ins_classE, COUNT(*)
FROM        PatientsE P
WHERE       P.ageE ≥ fc (21)
GROUP BY   P.ins_classE
HAVING      100 < ( SELECT      COUNT(*)
                        FROM        PatientsE P2
                        WHERE       P2.ins_classE = P.ins_classE AND P2.ageE > fc(50));

```

```

QE9-2: SELECT      P.ins_classE, P.ageE, P.pidE
      FROM        PatientsE P
      WHERE       P.ageE ≥ fc (21)
      GROUP BY   P.ins_classE, P.ageE, P.pidE
      HAVING      100 < ( SELECT      COUNT(*)
                        FROM        PatientsE P2

```

WHERE P2.ins\_class<sub>E</sub> = P.ins\_class<sub>E</sub> AND P2.age<sub>E</sub> >f<sub>e</sub>(50));

Note that the key P.pid<sub>E</sub> with P.age<sub>E</sub> should be added in the GROUP-BY clause in QE9-2 to obtain all the qualified P.age<sub>E</sub> values. This is because only the columns that appear in the GROUP-BY clause can appear in the SELECT clause, and P.pid<sub>E</sub> should be added in the GROUP-BY clause to get all the same age<sub>E</sub> values.

Query Execution Semantics of Q9 at DBMS:

1. Compute Top level query
  - 1-1) From Patients, apply the predicate in WHERE, call the result Temp1.
  - 1-2) Project ins\_class, pid, age, call the result Temp2.
  - 1-3) Sort the remaining tuples on ins\_class, call the sorted table Temp3.
  - 1-4) For each group of Temp3 in Top level, compute subquery.
    - 1-4-1) From Patients, evaluate the predicates in the sub query, call it Temp4
    - 1-4-2) Compute COUNT(\*) in subquery.
    - 1-4-3) Apply the predicate in HAVING with the computed subquery result, call it Temp5.
2. Scan Temp5 in the top level query maintaining the running info to compute AVE(P.age) in the top level query.
3. Output as in the SELECT clause in the top level query.

$$C_{Q9} = N_p + P_H(N_{Temp1}) + N_{Temp2} * \lg N_{Temp2} + g_f * N_{Temp3} * (N_p + N_{Temp4}) + N_{Temp5}$$

$$= (1 + s_p + 2 * r_p * s_p + s_p^2 * g_f * r_p) N_p + (r_p * s_p * N_p) \lg(r_p * s_p * N_p)$$

$$+ (1 + s_j * s_p) * g_f * r_p * s_p N_p^2$$

where size of Temp1  $N_{Temp1} = s_p * N_p$  with selectivity of the predicate  $s_p$ ,

size of Temp2  $N_{Temp2} = r_p * N_{Temp1} = r_p * s_p * N_p$  with projection rate  $r_p$ ,

size of Temp3  $N_{Temp3} = N_{Temp2} = r_p * s_p * N_p$  and sorted,

size of Temp4  $N_{Temp4} = s_j * s_p * N_p$ ,

and size of Temp5  $N_{Temp5} = s_p * g_f * N_{Temp3} = s_p * g_f * (r_p * s_p * N_p)$ .

Note that the cost for subquery would be  $N_p$  (to scan Patients for two predicates) +  $P(N_{Temp4})$  (for projection) +  $r_p * N_{Temp4}$  (for COUNT), however, since there is only COUNT(\*) in the SELECT clause in Q9, there is no need for projection. Therefore, the cost for subquery is  $N_p + N_{Temp4}$ . So, the total cost for the step1-4) is  $g_f * N_{Temp3} * (N_p + N_{Temp4}) = g_f * (r_p * s_p * N_p) * (N_p + s_j * s_p * N_p)$ .

Query Execution Semantics of QE9-1 at DBMS<sub>E</sub> is same as Q9.

$$C_{QE9-1} = N_p + P(N_{Temp1}) + \text{Sort}(N_{Temp2}) + g_f * N_{Temp3} * (N_p + N_{Temp4}) + N_{Temp5}$$

When hash-based projection is employed and  $\text{Sort}(N) = N \lg N$ ,

$$C_{QE9-1} = N_p + P_H(N_{Temp1}) + N_{Temp2} * \lg N_{Temp2} + g_f * N_{Temp3} * (N_p + N_{Temp4}) + N_{Temp5}$$

$$= (1 + s_p + 2 * r_p * s_p + s_p * g_f * r_p * s_p) * N_p + (r_p * s_p * N_p) \lg(r_p * s_p * N_p) + (1 + s_j * s_p) * g_f * r_p * s_p N_p^2$$

Query Execution Semantics of QE9-2 at DBMS<sub>E</sub> is similar to QE9-1 except that there is no last step to scan Temp5 for COUNT,

$$C_{QE9-2} = N_p + P(N_{Temp1}) + \text{Sort}(N_{Temp2}) + g_f * N_{Temp3} * (N_p + N_{Temp4})$$

$$= N_p + P_H(N_{Temp1}) + N_{Temp2} * \lg N_{Temp2} + 1 * N_{Temp3} * (N_p + N_{Temp4})$$

$$= (1 + s_p + 2 * r_p * s_p) N_p + (r_p * s_p * N_p) \lg(r_p * s_p * N_p) + (1 + s_j * s_p) * r_p * s_p N_p^2$$

where  $N_{Temp1} = s_p * N_p$ ,  $N_{Temp2} = r_p * s_p * N_p$ ,  $N_{Temp3} = N_{Temp2} = r_p * s_p * N_p$  and Sorted,  $N_{Temp4} = s_j * s_p * N_p$ , and  $N_{Temp5} = s_p * N_{Temp3} = s_p * (r_p * s_p * N_p)$  ( From here, we call it  $N_{Temp5-2}$  to distinguish it from  $N_{Temp5}$  in  $C_{Q9}$ ).

Note that the total cost of step1-4) for QE9-2 is  $N_{Temp3} * (N_p + N_{Temp4}) = 1 * (r_p * s_p * N_p) * (N_p + s_j * s_p * N_p)$  because we add P.pid<sub>E</sub> and P.age<sub>E</sub> in the GROUP-BY clause, since P.pname<sub>E</sub> is the key attribute of Patients<sub>E</sub>, so gf = 1 in QE9-2. Thus, the cost of processing QE9 at the anti-tamper site is

$$C_{anti-tamper} = C_{QE9-1} + C_{QE9-2}.$$

In addition, the cost at the Agent site  $C_{agent}$  = cost for decrypting P.age<sub>E</sub> in the output of QE9-2 + cost for computing SUM(P.age) =  $2 * N_{Temp5-2} = 2 * (s_p * r_p * s_p * N_p)$ , and the cost for transmitting all the qualified P.age<sub>E</sub> values to the Agent site  $C_t$  = cost for transmitting all  $N_{Temp5-2}$  (=  $s_p * r_p * s_p * N_p$ ). Thus, the total cost for processing QE9 over encrypted database is:

$$\begin{aligned} C_{QE9} &= C_{anti-tamper} + C_{agent} + C_t = C_{QE9-1} + C_{QE9-2} + C_{agent} + C_t \\ &= N_p + P(N_{Temp1}) + Sort(N_{Temp2}) + g_f * N_{Temp3} * (N_p + N_{Temp4}) + N_{Temp5} \\ &\quad + N_p + P(N_{Temp1}) + Sort(N_{Temp2}) + N_{Temp3} * (N_p + N_{Temp4}) \\ &\quad + 2 * N_{Temp5-2} + C_t \end{aligned}$$

Therefore, cost differential for Query Type 5 is

$$\begin{aligned} C_{QE9} - C_{Q9} &= C_{QE9-2} + 2 * N_{Temp5-2} + C_t \\ &= N_p + P(N_{Temp1}) + Sort(N_{Temp2}) + N_{Temp3} * (N_p + N_{Temp4}) + 2 * N_{Temp5-2} + C_t \\ &= (1 + s_p + 2 * r_p * s_p) N_p + (r_p * s_p * N_p) \lg(r_p * s_p * N_p) + (1 + s_j * s_p) * r_p * s_p N_p^2 + 3 * (s_p^2 * r_p * N_p) + C_t \end{aligned}$$

## 5.6 Nested and Correlated Queries with AVG at Nested Subquery

Q: Find the number of patients who are of adult age 21 or above for each insurance class for which the average age is more than 40 for such patients.

Query Type 6:

```
SELECT Ri.Ak, AGG(Ri.Am)
FROM      Ri
WHERE     [Ri.Ah = Ci]
GROUP BY  Ri.Ak
HAVING    Ri.Am > θ
          (SELECT  AGG(Rj.Ah)
           FROM    Rj
           WHERE   Rj.At = Ri.Ak,
                 [Rj.As = Ck]);
```

Q10:

```
SELECT P.ins_class, COUNT(*)
FROM    Patients P
WHERE   P.age ≥ 21
GROUP BY P.ins_class
HAVING  40 <
        (SELECT  AVE(P2.age)
         FROM    Patients P2
         WHERE   P2.ins_class = P.ins_class,
                 AND P2.age ≥ 21);
```

where AGG(R<sub>i</sub>.A<sub>m</sub>) in outer block query is either Min, Max or COUNT, and AGG(R<sub>i</sub>.A<sub>m</sub>) in inner query block is either Sum or Ave.

Execution Semantics of Q10 at DBMS and the processing cost  $C_{Q10}$  are as follows:

1. Compute Top level query
  - 1-1) From Patients, apply the predicate in WHERE, call the result Temp1.
  - 1-2) Project ins\_class, age, pid, call the result Temp2.
  - 1-3) Sort the remaining tuples on ins\_class, call the sorted table Temp3.
  - 1-4) For each group of Temp3 in Top level
    - 1-4-1) From Patients, evaluate the predicates in WHERE in the subquery, call it Temp4.
    - 1-4-2) Project P2.pid, P2.age, P2.ins\_class, call it Temp5.
    - 1-4-3) Scan Temp5 to compute AVE(P2.age) in the subquery.
    - 1-4-4) Apply the predicate in HAVING with the computed subquery result.

2. Scan the computed table Temp6 in Top level maintaining the running info for COUNT(P.age) in Top level query.
3. Generate the output with each different value of insclass and the number of values of age for each group of insclass in Top-level query.

$$\begin{aligned}
C_{Q10} &= Np + P(N_{Temp1}) + \text{Sort}(N_{Temp2}) + g_f * N_{Temp3} * (Np + P(N_{Temp4}) + N_{Temp5}) + N_{Temp6} \\
&= Np + P_H(N_{Temp1}) + N_{Temp2} * l_g N_{Temp2} + g_f * N_{Temp3} * (Np + P_H(N_{Temp4}) + N_{Temp5}) + N_{Temp6} \\
&= (1 + s_p + 2 * s_p + s_p^2 * g_f) Np + (s_p * Np) \log(s_p * Np) + (1 + s_j * s_p + 3 * r_p * s_j * s_p) * g_f * s_p * Np^2
\end{aligned}$$

where size of Temp1  $N_{Temp1} = s_p * Np$ ,  
size of Temp2  $N_{Temp2} = r_p * N_{Temp1} = r_p * s_p * Np$ ,  
size of Temp3  $N_{Temp3} = N_{Temp2} = r_p * s_p * Np$  and sorted,  
size of Temp4  $N_{Temp4} = s_j * s_p * Np$ ,  
size of Temp5  $N_{Temp5} = r_p * N_{Temp4}$ , and  
size of Temp6  $N_{Temp6} = s_p * g_f * N_{Temp3} = s_p * g_f * (1 * s_p * Np)$ .

Since there is COUNT(\*) in the SELECT clause in the top level query,  $r_p = 1$  in  $N_{Temp2}$ ,  $N_{Temp3}$ , and  $N_{Temp6}$ .

However, there are two problems to evaluate Query Type 6 over encrypted database. First, query splitting transformation can not be applied directly to Query Type 6 because of the Having clause as shown in QE10-1\* and QE10-2\*, which are not valid queries. Second, AVE(P2.age<sub>E</sub>) in the correlated nested query of Q10 should be evaluated first for each insurance class for the HAVING clause in the top level query.

QE10-1\*:

```

SELECT P.ins_classE, COUNT(*)
FROM      PatientsE P
WHERE     P.ageE ≥ fe(21)
GROUP BY  P.ins_classE
HAVING    fe(40) <
          (SELECT P2.pidE, P2.ageE
           FROM   PatientsE P2
           WHERE  P2.ins_classE = P.ins_classE,
                AND P2.ageE ≥ fe(21));

```

QE10-2\*:

```

SELECT P.ins_classE, COUNT(*)
FROM      PatientsE P
WHERE     P.ageE ≥ fe(21)
GROUP BY  P.ins_classE
HAVING    fe(40) <
          (SELECT COUNT(P2.ageE)
           FROM   PatientsE P2
           WHERE  P2.ins_classE = P.ins_classE,
                AND P2.ageE ≥ fe(21));

```

A way to process such query over encrypted database is that first split the top level query without the HAVING clause and the subquery, and then, for each group i in the top level, split the subquery into two queries to compute AVE(P2.AGE) as follows:

QETop10-1:

```

SELECT      P.ins_classE
FROM        PatientsE P
WHERE       P.ageE ≥ fe(21)
GROUP BY    P.ins_classE

```

QEiTop10-2

```

SELECT      P.ins_classE, COUNT(P.ageE)
FROM        PatientsE P
WHERE       P.ins_classE = i AND
           P.ageE ≥ fe(21)

```

QEiSub10-1:

```

SELECT      P2.pidE, P2.ageE
FROM        PatientsE P2
WHERE       P2.ins_classE = i
           AND P2.ageE ≥ fe(21)

```

QEiSub10-2:

```

SELECT      COUNT(P2.ageE)
FROM        PatientsE P2
WHERE       P2.ins_classE = i
           AND P2.ageE ≥ fe(21)

```

Computations at the Agent site: The agent

1. Submits QETop10 to DBMS<sub>E</sub>,
2. For each n different value of P.ins\_class<sub>E</sub> in the SELECT clause, after obtaining the result of QETop10,
  - 2-1 Submits QESub10-1 and QESub10-2 to DBMS<sub>E</sub>.
  - 2-2 Upon obtaining the results of QESub10-1 and QE10-2 from the server,
  - 2-3 Decrypts values of P.age<sub>E</sub> from the result of QESub10-1.
  - 2-4 Computes AVE(P.age) with the result of QESub10-2.
  - 2-5 If the predicate in the HAVING clause in the top level query is satisfied with the result of the step 1-4, submits QEiTop10-2.
3. Union all the results of QEiTop10-2.

$$C_{\text{anti-tamper}} = C_{\text{QETop10-1}} + n * (C_{\text{QEiSub10-1}} + C_{\text{QEiSub10-2}}) + r * C_{\text{QEiTop10-2}}$$

where n is the number of different values of P.ins\_class, and r is the number of values of P.ins\_class<sub>E</sub> that satisfy the predicate in the HAVING clause.

$$\begin{aligned} C_{\text{QETop10-1}} &= Np + P_H(s_p * Np) + (s_p * Np) \log(s_p * Np) \\ &= Np + s_p * Np + r_p * s_p * Np + (s_p * Np) \log(s_p * Np) \end{aligned}$$

$$\begin{aligned} C_{\text{QEiSub10-1}} &= Np + P_H(s_p * s_p * Np) \\ &= Np + s_p * Np + 2 * r_p * s_p * s_p * Np \end{aligned}$$

$$\begin{aligned} C_{\text{QEiSub10-2}} &= Np + P_H(sp * sp * Np) + sp * sp * Np \\ &= Np + sp * sp * Np + 2 * r_p * sp * sp * Np + r_p * sp * sp * Np \end{aligned}$$

$$\begin{aligned} C_{\text{QEiTop10-2}} &= Np + P_H(sp * sp * Np) + sp * sp * Np \\ &= Np + sp * sp * Np + 2 * r_p * sp * sp * Np + r_p * sp * sp * Np \end{aligned}$$

$$\begin{aligned} C_{\text{Agent}} &= n * (\text{cost for decrypting } r_p * sp * sp * Np + \text{cost for computing AVE of } r_p * sp * sp * Np) \\ &= n * 2 * r_p * sp * sp * Np \end{aligned}$$

$$\begin{aligned} C_t &= n * (\text{cost for transmitting the result of QEiSub10-2}) \\ &\quad + r * (\text{cost for transmitting the result of QEiTop10-2}) \\ &= \text{Cost for transmitting } n * (r_p * sp * sp * Np) + r * (r_p * sp * sp * Np) \end{aligned}$$

However, repetitive agent site computations and submissions to the anti-tamper site for each different value of insurance class over a network leads to unacceptably inefficient query processing and expensive extra query processing cost.

An alternative way to deal with this problem is to remove the correlation, that is, to rewrite the query Q10 to an equivalent non-correlated unnested query as follows:

<p>QE10:</p> <pre> SELECT      P.ins_class<sub>E</sub>, COUNT(*) FROM        Patients<sub>E</sub> P WHERE       P.age<sub>E</sub> ≥ f<sub>c</sub>(21) GROUP BY   P.ins_class<sub>E</sub> HAVING     f<sub>c</sub>(40) &lt; ( SELECT    AVE(P2.age<sub>E</sub>) FROM        Patients<sub>E</sub> P2 WHERE       P2.ins_class<sub>E</sub> = P.ins_class<sub>E</sub> AND P2.age<sub>E</sub> ≥ f<sub>c</sub>(21));</pre>	<p>QE10R:</p> <pre> SELECT P.ins_class<sub>E</sub>, COUNT(*) FROM Patients<sub>E</sub> P, ( SELECT P2.ins_class<sub>E</sub> AS ins_classdone<sub>E</sub>, AVE(P2.age<sub>E</sub>) AS aveage FROM Patients<sub>E</sub> P2 WHERE P2.age<sub>E</sub> ≥ f<sub>c</sub>(21) GROUP BY P2.ins_class<sub>E</sub> ) AS NEW WHERE P.age<sub>E</sub> ≥ f<sub>c</sub>(21) AND NEW.aveage &gt; f<sub>c</sub>(40) AND P.ins_class<sub>E</sub> = NEW.ins_classdone<sub>E</sub> ;</pre>
--	--

We adopt an execution strategy originally proposed by Kim [KW82] to transform Query Type6 to an equivalent non-correlated query. It is possible to aggregate directly over the Patients<sub>E</sub> table to obtain AVE(P2.age<sub>E</sub>) per insclass and later join with another Patients<sub>E</sub> table on the combined conditions. This

strategy pushes aggregation below join. The SQL query uses the derived table NEW to join, not a subquery [GJ01]. We split the subquery of QE10R to compute AVE(P2.age<sub>E</sub>) as follows:

QE10RSub1:	QE10RSub2:
SELECT P2.ins_class <sub>E</sub> , COUNT(P2.age <sub>E</sub> )	SELECT P2.ins_class <sub>E</sub> , P2.age <sub>E</sub> , P2.pid <sub>E</sub>
FROM Patients <sub>E</sub> P2	FROM Patients <sub>E</sub> P2
WHERE P2.age <sub>E</sub> ≥ f <sub>c</sub> (21)	WHERE P2.age <sub>E</sub> ≥ f <sub>c</sub> (21)
GROUP BY P2.ins_class <sub>E</sub>	GROUP BY P2.ins_class <sub>E</sub> , P2.age <sub>E</sub> , P2.pid <sub>E</sub>

Note that P2.age<sub>E</sub>, P2.pid<sub>E</sub> are added in the GROUP BY clause to get all the qualified values of P2.age<sub>E</sub>. After computing subquery of QE10R, the agent resubmits QE10R with the computed result of the subquery as the table NEW.

Execution Semantics of QE10RSub1:

1. Scan Patients<sub>E</sub> to apply the predicate, call the result Temp1.
2. Project ins\_class<sub>E</sub>, age<sub>E</sub>, called the result Temp2.
3. Sort on the GROUP-BY column ins\_class<sub>E</sub>, call the result Temp3.
4. Scan the Temp3 to compute COUNT(P2.age<sub>E</sub>), call the subquery result Temp4.

Execution Semantics of QE10RSub2:

1. Scan Patients<sub>E</sub> to apply the predicate, call the result Temp1.
2. Project ins\_class<sub>E</sub>, age<sub>E</sub>, pid<sub>E</sub> called the result Temp2.
3. Sort on the GROUP-BY columns ins\_class<sub>E</sub>, age<sub>E</sub>, pid<sub>E</sub>, call the result Temp3.

Execution Semantics of QE10RTop with the computed subquery Temp4 as the table NEW:

1. Scan Patients<sub>E</sub>, apply two predicates in the WHERE clause, call the result Temp5.  
Project Temp5, call it Temp6.
2. Join Temp4 with Temp6 on P.ins\_class<sub>E</sub> = NEW.ins\_classdone<sub>E</sub> call the result Temp7.
3. Scan Temp7 to compute COUNT(\*).

Where  $N_{Temp1} = s_p * N_p$ ,  $N_{Temp2} = r_p * s_p * N_p$ ,  $N_{Temp3} = N_{Temp2}$  and sorted,  $N_{Temp4} = g_f * r_p * s_p * N_p$ ,  $N_{Temp5} = s_p * s_p * N_p$ ,  $N_{Temp6} = r_p * s_p * s_p * N_p$ , and  $N_{Temp7} = s_j * r_p * s_p * s_p * N_p * g_f * r_p * s_p * N_p$ .

$$C_{QE10RSub1} = N_p + P(s_p * N_p) + \text{Sort}(r_p * s_p * N_p) + r_p * s_p * N_p$$

$$= N_p + s_p * N_p + 2 * r_p * s_p * N_p + (r_p * s_p * N_p) \lg(r_p * s_p * N_p) + r_p * s_p * N_p$$

$$C_{QE10RSub2} = N_p + P(s_p * N_p) + \text{Sort}(r_p * s_p * N_p)$$

$$= N_p + s_p * N_p + 2 * r_p * s_p * N_p + (r_p * s_p * N_p) \lg(r_p * s_p * N_p)$$

$$C_{QE10RTop} = N_p + P(N_{Temp5}) + J(N_{Temp6}, N_{Temp4}) + N_{Temp7}$$

$$= N_p + P(s_p * s_p * N_p) + J(s_p * s_p * r_p * N_p, r_p * s_p * N_p) + s_j * r_p * r_p * s_p * s_p * s_p * N_p * N_p$$

$$= N_p + s_p * s_p * N_p + 2 * r_p * s_p * s_p * N_p + (s_p * s_p * N_p) + (r_p * s_p * N_p) * \lceil (s_p * s_p * N_p) / (B-2) \rceil$$

$$+ g_f * r_p * s_j * r_p * s_p * s_p * s_p * N_p * N_p$$

$$C_{anti-tamper} = C_{QE10RSub1} + C_{QE10RSub2} + C_{QE10R}$$

$$= 2 * [N_p + P(N_{Temp1}) + \text{Sort}(N_{Temp2})] + N_{Temp3} + N_p + P(N_{Temp5}) + J(N_{Temp6}, N_{Temp4}) + N_{Temp7}$$

$$C_{Agent} = \text{Cost for decrypting Temp3} + \text{Cost for computing Temp3}$$

$$= 2 * r_p * s_p * N_p$$

$$C_t = \text{cost for transmitting the result of QE10RSub1 and QE10RSub2}$$

$$+ \text{cost for transmitting the computed result from the agent to the anti-tamper site}$$

$$= \text{cost for transmitting } 2 * N_{Temp3} + N_{Temp4} (= 2 * r_p * s_p * N_p + g_f * r_p * s_p * N_p)$$

Thus, the total cost of encrypted database query processing of QE10R is

$$C_{QE10} = C_{QE10RSub1} + C_{QE10RSub2} + C_{QE10RTop} + C_{Agent} + C_t.$$

Therefore, cost differential between encrypted and non-encrypted database query processing of Q10 is

$$C_{QE10R} - C_{Q10R} = C_{QE10R} - (C_{Q10RSub} + C_{Q10RTop}) = C_{QE10RSub2} + C_{Agent} + C_t.$$

Note that the cost differential between encrypted and non-encrypted database query processing of Q10 above is derived by comparing the cost of processing QE10R over encrypted database to the cost of processing Q10R instead of the original query Q10 with the followings:

Q10R:

```

SELECT P.ins_class, COUNT(*)
FROM Patients P,
     (SELECT P2.ins_class AS ins_classdone,
          AVE(P2.age) AS aveage
     FROM Patients P2
     WHERE P2.age ≥ 21
     GROUP BY P2.ins_class
    ) As NEW
WHERE P.age ≥ 21
     AND NEW.aveage > 40 AND
     P.ins_class = NEW.ins_classdone

```

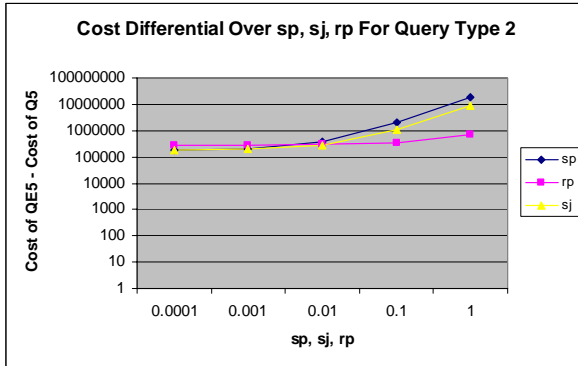
The cost of processing Q10R over non-encrypted database is the sum of the cost of subquery of Q8R and the cost of processing the top-level query of Q10R.

$$\begin{aligned}
C_{Q10R} &= C_{Q10RSub} + C_{Q8RTop} \\
&= N_p + P_H(s_p * N_p) + \text{Sort}(r_p * s_p * N_p) + r_p * s_p * N_p + N_p + P(N_{Temp5}) + J(N_{Temp6}, N_{Temp4}) + N_{Temp7} \\
&= N_p + s_p * N_p + 2 * r_p * s_p * N_p + (r_p * s_p * N_p) \lg(r_p * s_p * N_p) + r_p * s_p * N_p + N_p + s_p * s_p * N_p + 2 * r_p * s_p * s_p * N_p + \\
&\quad (r_p * s_p * s_p * N_p) + (g_f * r_p * s_p * N_p) * \lceil (r_p * s_p * s_p * N_p) / (B-2) \rceil + s_j * g_f * r_p * r_p * s_p * s_p * s_p * N_p * N_p
\end{aligned}$$

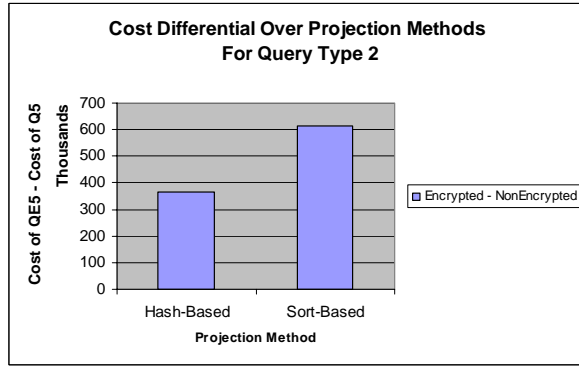
## 6. Evaluation of Query Processing Costs Due to Encryption

In this section, we report experimental results about how cost differentials between encrypted and non-encrypted database query processing change when various parameters in cost formulas change for the query types. For Query Type 1 and 3, there is no extra cost for encrypted database query processing. For Query Type 4, encrypted database query processing technique is the same as that of Query Type 2. In this section, we focus on analysis of the query types that require extra query processing for encrypted database; so mainly on Query Type 2, 4 and 5. The experiments are performed based on the cost formulas and the parameters in Table 1 and Table 2 in section 3; so the query processing cost is measured by disk I/O cost, which is the number of pages processed, and  $C_t$  data transmission cost to evaluate a query.

Cost differentials for two projection methods and three join methods are observed. Each experiment is performed with parameter values as follows:  $N_p = 10,000$  pages,  $N_c = 50,000$  pages, predicate selectivity  $sp = 1\%$ , join selectivity  $sj = 2\%$ , projection ratio  $rp = 20\%$ , and buffer size  $B = 300$  pages.



**Figure 3.** Cost Differential Between Encrypted and Nonencrypted Over  $sp$ ,  $sj$ , and  $rp$  For Query Type 2



**Figure 4.** Cost Differential Over Projection Methods For Query Type 2

*Observation 1* (Figure 3): For Query Type 2, the cost of encrypted database query processing is slightly more than double the processing cost of the non-encrypted database query. This is because the original query Q5 is split into two queries QE5-1 and QE5-2, each of which does almost the same processing except  $r_p * s_j * s_p * N_c * N_p$  less for QE5-2, and the additional computation at agent site costs  $2 * (r_p * s_j * s_p * N_c * N_p)$ .

*Observation 2* (Figure 3): For Query Type 2, the cost differential between encrypted and non-encrypted databases linearly increases in selectivities  $s_p$ ,  $s_j$ , and  $r_p$ .

*Observation 3* (Figure 3): Cost differential is more sensitive over predicate selectivity  $s_p$  than projection ratio  $r_p$  for Query Type 2.

This is because the effect of  $s_p$  starts to contribute earlier than  $r_p$  to the query evaluation, so it affects the size of the intermediate tables more than  $r_p$  does.

*Observation 4* (Figure 4): Cost differential is higher when sort-based projection is employed over hash-based projection for Query Type 2.

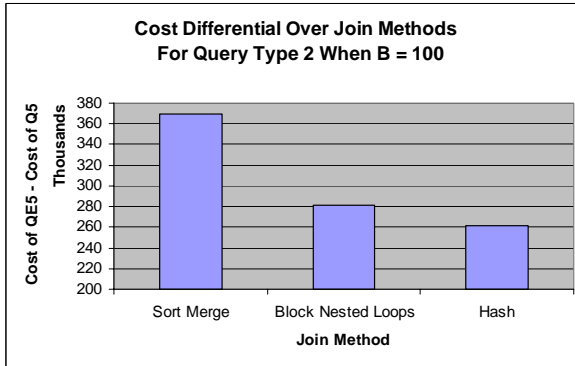
This is because sort-based projection costs more than hash-based projection for this type of queries.

*Observation 5* (Figure 5-1): For Query Type 2, cost differential is 30% and 50% more when sort-merge join is used for Query Type 2 than block nested loops or hash join with buffer size  $B = 100$ .

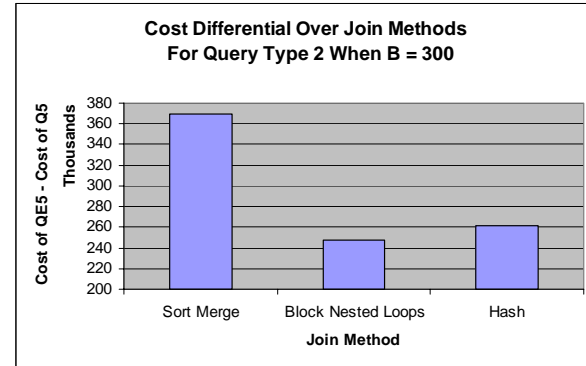
This is because the cost formula employed for sort-merge join counts each repetition of duplicates, and there are a lot of duplicates on pid in Claims as the parameters  $pr_1$ ,  $pr_2$  and  $N_{c-p}$  increase.

*Observation 6* (Figure 5-1, Figure 5-2): For Query Type 2, block nested loop join is the best when buffer size is around 300 while hash join becomes the best when the buffer size is decreased down to about 100 pages.

Obviously this is because block nested loops join takes advantage of buffer usage, and for hash join, we assume that hash tables fit in the main memory.



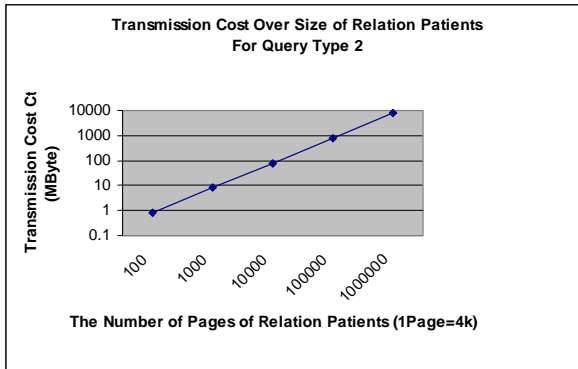
**Figure. 5-1.** Cost Differential Between Encrypted and Nonencrypted Over Join Methods For Query Type 2 When B = 100



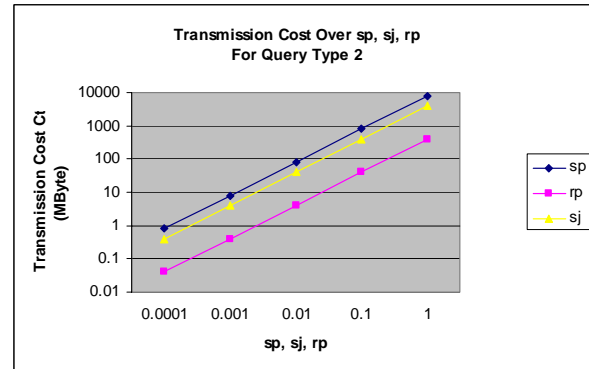
**Figure. 5-2.** Cost Differential Between Encrypted and Nonencrypted Over Join Methods For Query Type 2 When B = 300

*Observation 7* (Figure 6): For Query Type2, when the first (Patients) relation size reaches 10k pages (with 4kbyte page size and 50k pages of the other relation Claims),  $C_t$ , that is, the transmitted data size becomes 80MB. If the first relation size is very large (say over 100k pages),  $C_t$  becomes more than 800Mbytes, which makes our anti-tamper approach unfeasible.

*Observation 8* (Figure 7): For Query Type2, we see the same crossover point for  $C_t$  between 80MB and 800 MB when  $s_p$  is between 1% and 10%, and  $s_j$  is between 2% and 20 %. Note that even when  $r_p$  is 100%,  $C_t$  is still within a feasible range.



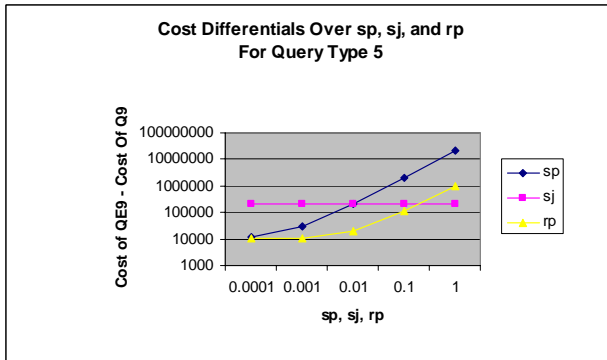
**Figure 6.** Transmission Cost Over Size of Relation Patients For Query Type 2



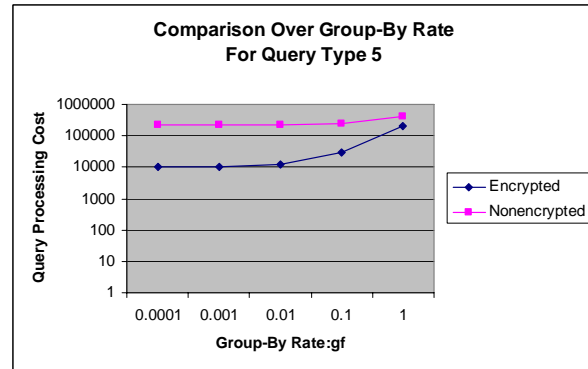
**Figure 7.** Transmission Cost Over  $s_p$ ,  $s_j$ , and  $r_p$  For Query Type 2

*Observation 9* (Figure 8): For Query Type 5, the change in  $r_p$  is more significant than the change in  $s_j$  because  $s_p * r_p$  in each tuple in the top-level is repeated for the subquery in QE9-2.

*Observation 10* (Figure 8): For Query Type 5, join selectivity  $s_j$  has no effect on the cost differential. This is because the effect of  $s_j$  on Q9 cancels out the effect of  $s_j$  on QE9-1.



**Figure 8.** Cost Differential Over  $s_p$ ,  $s_j$ , and  $r_p$  For Query Type 5



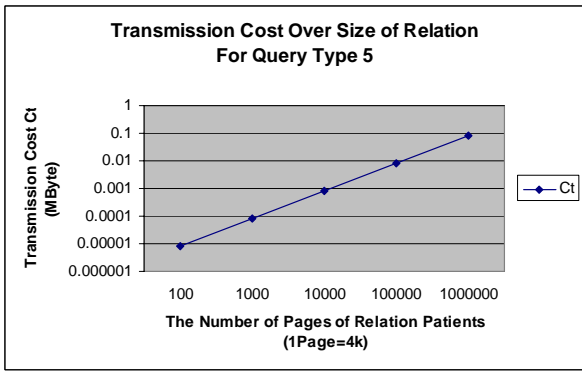
**Figure 9.** Comparison of Query Processing Cost Between Encrypted and Nonencrypted Over  $g_f$  For Query Type 5

*Observation 11* (Figure 9): For Query Type 5, the cost of Q9 and the cost of QE9 linearly increase as Group-By ratio  $g_f$  increases, however  $g_f$  has no effect on the cost differential. This is because (a) the effect of  $g_f$  on the cost of Q9 cancels out the effect of  $g_f$  on the cost of QE9-1, and (b)  $g_f$  has no effect on the cost of QE9-2 because  $pid_E$ , which is the key, is added to the GROUP-BY clause in the rewritten QE9-2 to obtain all the qualified  $age_E$  values for computation at the Agent site later.

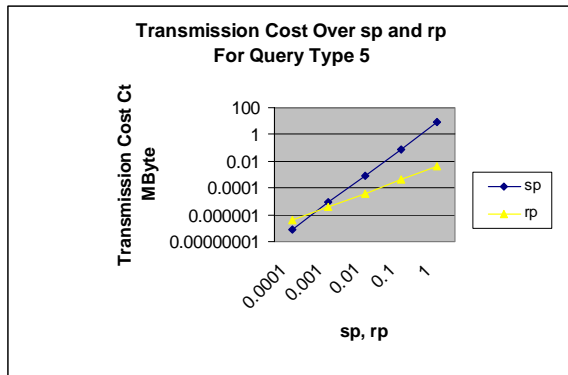
*Observation 12* (Figure 10): For Query Type 5, the transmission cost  $C_t$  is mostly less than 1MB, thus always feasible even with very large page input relation sizes with over 1 Mpages.

*Observation 13* (Figure 11): For Query Type 5, the transmission cost  $C_t$  is mostly less than 8MB, thus always feasible even when  $s_p$ , or  $r_p = 100\%$ .

This is because QE9 contains the Group-By clause, which reduces the final table size to be transmitted significantly.



**Figure 10.** Transmission Cost Over Size of Relation For Query Type 5

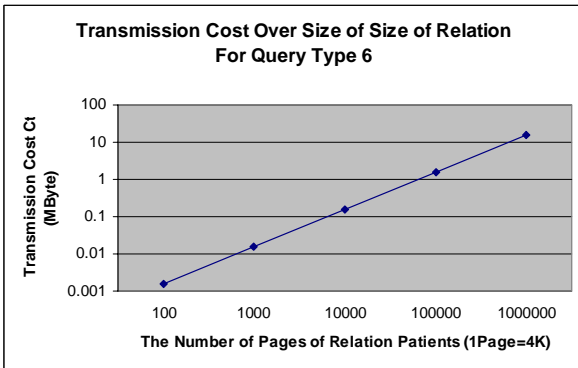


**Figure 11.** Transmission Cost Over sp and rp For Query Type 5

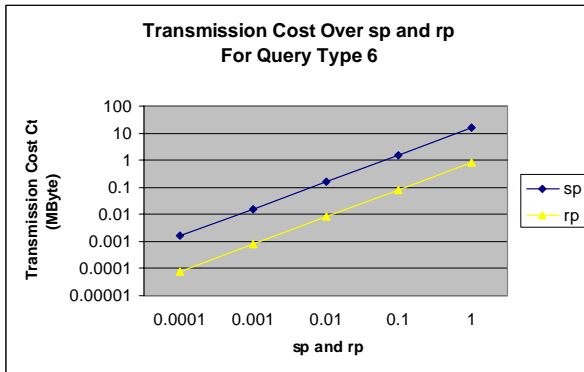
*Observation 14* (Figure 12): For Query Types 6,  $C_t$  is mostly less than about 16MB even with very large input relation sizes (about 1Mpages).

*Observation 15* (Figure 13): For Query Types 6,  $C_t$  is mostly less than about 16MB even when  $s_p$ , or  $r_p=100\%$ .

This is again because QE10R contains the Group-By clause, which reduces the final table size to be transmitted significantly.



**Figure 12.** Transmission Cost Over Size of Relation For Query Type 6



**Figure 13.** Transmission Cost Over sp, rp For Query Type 6

*Observation 16* (Figure 14): For Query Type 6, changes to the cost differentials due to change in parameters  $s_p$ ,  $s_j$ ,  $r_p$  are less significant than the other query types.

This is because the correlation is removed in the rewritten query QE10R; so the cost differential is relatively small (the extra cost contains the cost for processing the subquery and the costs for the agent site computation and transmitting a very small size data).

*Observation 17* (Figure 15): For Query Type 6, the cost differential over the two projection methods is less significant than the other query types.

This is again because the correlation is removed in the rewritten query QE10R; so the cost differential is relatively small.

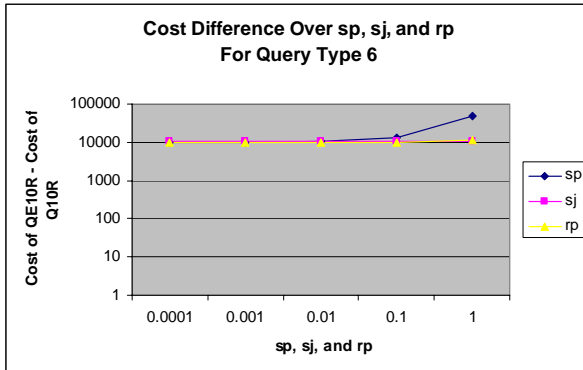


Figure 14. Cost Differentials Over sp and rp For Query Type 6

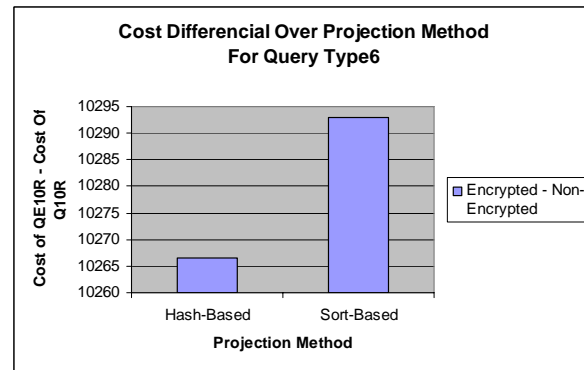


Figure 15. Cost Differentials Over Projection Method For Query Type 6

## 6 Conclusion

For the query types that require extra query processing over encrypted database, the cost differentials of query processing between non-encrypted and encrypted database increase linearly in sp, sj, rp, and the size of relations. The cost of encrypted database for correlated nested queries with GROUP BY clause and AVE in top-level is most expensive of all the query types due to the repetition of evaluation of the correlated subquery for each GROUP BY arguments. For correlated nested aggregate queries with GROUP BY and AVE in the subquery, the cost of encrypted database query processing is 50% more than the cost of non-encrypted database when correlation is removed. In addition, for the query types, data transmission cost for computation at the Agent site does not contribute significantly to the overall cost of encrypted database query processing. However, for single block aggregate queries with AVE, the crossover point occurs approximately when the transmitted data size becomes over 800 Mbytes when the sizes of relations are very large over 100k pages. From our experimental results, we conclude that the anti-tamper database approach is feasible and effective for securing database against insider and outsider threats.

## 7. References

- [AF02] D. Asonov, J.C. Freytag, "Querying Databases Privately", in the Proceedings of the 28<sup>th</sup> VLDB Conference, HongKong, china, 2002.
- [CO05] Chung S. S., Ozsoyoglu G., "Processing Aggregate Queries Over Encrypted Relational Databases", Technical Report, Case Western Reserve University, 2005.  
<http://art.case.edu/anti-tamper.pdf>
- [CPH04] B. Kiraly, A. Podgursky, and S. Hoffman, "Security Vulnerabilities and Conflicts of Interest in the Provider-Clearinghouse\*-Payer Model", In the PORTIA workshop, Stanford, May, 2004.
- [CS98] S. Chaudhuri, "An Overview of Query Optimization in Relational Systems", in the Proceedings of PODS 1998 Seattle WA, U.S.A..
- [CS95] S. Chaudhuri, Kyuseok Shim, "An Overview of Cost-based Optimization of Queries with Aggregates", IEEE DE Bulletin, Sep. 1995. (Special Issue on Query Processing)
- [CS94] S. Chaudhuri, Kyuseok Shim, "Including Group-By in Query Optimization", in the Proceedings of VLDB, Santiago, 1994.
- [CS96] S. Chaudhuri, Kyuseok Shim, "Optimizing Queries with Aggregate Views", in the Proceedings of EDBT, Avignon, 1996.

- [CSI02] Computer Security Institute, CSI/FBI Computer Crime and Security Survey.
- [DU87] U. Dayal, "Of Nest and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers", in the Proceedings of the 13<sup>th</sup> VLDB, Brighton, 1987.
- [GJ01] C. A. Galindo-Legaria, M. M. Joshi, "Orthogonal Optimization of Subqueries and Aggregation", in the Proceedings of ACM SIGMOD 2001 May 21 – 24, Santa Barbara, California, USA.
- [HI96] Health Care Claim Institutional 837, Health Insurance Claim Form. Health Insurance Portability and Accountability Act (HIPAA), 1996.
- [HILM02] Hakan Hacigumus, Balakrishna R. Iyer, Li Chen, Sharad Mehrotra, "Executing SQL over Encrypted Data in the Database-Service-Provider Model", In the Proceedings of ACM SIGMOD, June, 2002. Madison WI.
- [KW82] W. Kim, "On Optimizing an SQL-like Nested Query", ACM Transaction on Databases Systems, vol. 7, No. 3, September 1982, pp. 443-469.
- [LGS02] W. Li, D. Gao, and R. Snodgrass, "Skew Handling Techniques in Sort-Merge Join", In the Proceedings of 2002 ACM SIGMOD, June 2002, Madison Wisconsin.
- [OSC03] Ozsoyoglu G., Singer D., Chung S., "Anti-Tamper Databases: Querying Encrypted Databases", In the Proceedings of IFIP11.3 Conference 2003 on Database Security, August 3 - 6, Estate Park, Colorado, USA.
- [RAD78] R. L. Rivest, L. Adleman and M.L. Dertouzos, On data banks and privacy homomorphisms, in R. A. DeMillo et al., eds., Foundations of Secure Computation, Academic Press, New York, 1978, 169-179.
- [RR98] R. Ramakrishnan, "Databases Management Systems", McGraw-Hill, Inc. U.S.A. 1998.
- [SPG79] P.G. Selinger et.al., "Access Path Selection in a Relational Database Management" in the Proceedings of the ACM SIGMOD Conference on Management of Data, June 79.