Answering Aggregation Queries in a Secure System Model

Tingjian Ge
Department of Computer Science
Brown University
tige@cs.brown.edu

Stan Zdonik
Department of Computer Science
Brown University
sbz@cs.brown.edu

ABSTRACT

As more sensitive data is captured in electronic form, security becomes more and more important. Data encryption is the main technique for achieving security. While in the past enterprises were hesitant to implement database encryption because of the very high cost, complexity, and performance degradation, they now have to face the ever-growing risk of data theft as well as emerging legislative requirements. Data encryption can be done at multiple tiers within the enterprise. Different choices on *where* to encrypt the data offer different security features that protect against different attacks. One class of attack that needs to be taken seriously is the compromise of the database server, its software or administrator. A secure way to address this threat is for a DBMS to directly process queries on the ciphertext, without decryption. We conduct a comprehensive study on answering SUM and AVG aggregation queries in such a system model by using a secure homomorphic encryption scheme in a novel way. We demonstrate that the performance of such a solution is comparable to a traditional symmetric encryption scheme (e.g., DES) in which each value is decrypted and the computation is performed on the plaintext. Clearly this traditional encryption scheme is not a viable solution to the problem because the server must have access to the secret key and the plaintext, which violates our system model and security requirements. We study the problem in the setting of a read-optimized DBMS for data warehousing applications, in which SUM and AVG are frequent and crucial.

1. INTRODUCTION

1.1 Motivation

In the past, enterprises were hesitant to implement database encryption because of the very high cost, complexity, and performance degradation. With the ever-growing risk of data theft and emerging legislative requirements like California's SB 1386 and NY State's Information Security Breach and Notification Act, enterprises must now report any data breach that has happened when unencrypted data is compromised. Rather than risk public damage to a brand or legal and possibly government (FTC) intervention, database encryption is now a top priority [30].

VLDB '07, September 23-28, 2007, Vienna, Austria.

There are a number of points of security attack in database systems, including storage media theft, application-level compromises, malicious DBAs, wiretapping client-server communication channels, key theft, and breaking into DBMS runtime environment. It was reported by several sources, including the draft of *National Strategy to Secure Cyberspace* announced by the White House [29], about 70% of security breaches are attributable to internal users. Thus, "separation of duty", which involves having a separate *Security Administrator* (SA), has become a golden rule. The DBA's role is to perform usual DBA tasks. The SA would administer privileges and permissions and manage keys and decryption, etc. For example, the "Database Vault" feature in a recent Oracle release [20] is such a step towards limiting a DBA's privileges.

1.2 System Model and the Problem

Consider the issue of where to encrypt the data in a consolidated environment of applications and databases. We have two categories: "*inside-the-box*" and "*outside-the-box*" encryption. In the former, encryption and decryption are performed by the database server in its runtime environment (i.e., "the box"), which is managed and inspected by the DBA. In the latter, encryption is handled outside the database server environment. In light of the previous discussion, outside-the-box is certainly more desirable for security; frequently it is a must. Note that the security administrator (SA) is considered to be outside the "box" (i.e., outside the database server environment). When a client application itself performs encryption and decryption, the client takes the role of SA. Generally, we call the secure agent that handles encryption and decryption the *Key Holder*. We assume the adversary cannot compromise the Key Holder. This system model is shown in Figure 1.

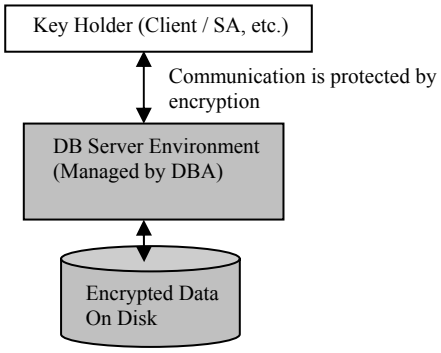

Figure 1: Illustrating the outside-the-box encryption model. The sensitive data at the server and on disk is encrypted (as shown in grey).

Compared to inside-the-box encryption, this model has the following advantages in data security:

- It follows the rule of "separation of duties". The administrative and security duties are divided between the DBA and the SA.
- Sensitive data from the DBMS logs, configuration files, temporary tables, memory, etc. is secure at the server.
- Communication between the database server and outside environment is protected as well.
- Key management becomes easier. Since the storage of keys is restricted to the Key Holder, we do not need to manage keys at the server. We do not need to deal with key theft at the server or key distribution in general.

Note that the secure system model we described is simply a *general abstraction* of the *data encryption part* of the "Database-As-a-Service" (a.k.a. database outsourcing) proposed in [11] (an example of such a business is [31]). Database outsourcing is an instance of the outside-the-box encryption due to the privacy and trust issues [3]. Outside-the-box encryption also implies that even when a database is not outsourced, we still have a separate entity, *SA* (who may not be the client and is certainly not the DBA), that handles encryption. Of course, database outsourcing also includes a usage model with issues besides data encryption.

An equally critical issue is query processing in such a model. Clearly, it is desirable for most of the query processing to occur at the database server. One would also like to minimize the communication cost between the server and the Key Holder (say, in returning query results). However, it is difficult to process a query at a server that does not have access to the decryption key nor the plaintext. While there has been work on performing comparison and indexing directly on the ciphertext, thus handling some SQL query types, so far there is no general, secure solution for the SUM and AVG aggregate queries in such a system model. This paper gives a comprehensive solution to this problem.

1.3 Related Work

Pioneering work in this area includes [2, 3, 10, 12, 13, 14, 17, 24]. Most of this work shares the goal of supporting efficient query processing on encrypted data. Much of this research addresses the problem of indexing ciphertext. For example, Agrawal et al [2] propose an order preserving encryption scheme (OPES) by which indexes can be built directly on ciphertext. We can use this scheme in our "un-trusted server" model (with its assumption that the adversary does not know the plaintext value distributions). The authors of [2] state that OPES can handle directly (without decryption) any interesting SQL query types, *except* SUM and AVG. Indeed, any approach that can handle comparison and indexing on ciphertext likely has this property. Our techniques proposed in this paper are complimentary to theirs, and therefore, taken together, they provide a complete solution. As a matter of fact, in some column stores such as C-Store [25, 27], two ciphertexts for the same column can coexist redundantly. In general, only one ciphertext would be accessed for each query, thus the I/O cost stays about the same. Moreover, as pointed out in [4, 26], SUM and AVG aggregate queries are very frequent and important in data warehousing, OLAP, and large database applications, the target of column-oriented (read-optimized) systems, such as C-Store [25, 27].

There have been proposals of using homomorphism in the database context. In particular, Hacigumus et al [12] propose to handle SUM and AVG using a particular homomorphic encryption function based upon the so-called Privacy Homomorphism [22]. The usage is simple and straightforward as each value is encrypted separately. Unfortunately, as pointed out by Mykletun and Tsudik [17], its encryption scheme is insecure, demonstrated by its vulnerability to a basic ciphertext-only attack. Instead, the authors in [17] describe a simple alternative for supporting aggregation queries *assuming* a bucketization scheme in [10] (i.e., splitting the attribute domain into a set of buckets). This essentially assumes two things:

1. A bucketization scheme must be deployed at the server;
2. The bucketization scheme is indeed secure.

While assumption (1) indicates the usage limitation of this simple approach, there has been some work (e.g., [2, 13]) that negates assumption (2). Large buckets are not a feasible solution, as an equijoin becomes a cross-product of buckets, with the result size (and client effort) growing rapidly with larger buckets (as shown in [10]). On the other hand, a fine partitioning makes the scheme vulnerable to estimation exposure. The relative size of buckets reveals information about the distribution of the data. Moreover, relationships between fields in a tuple can be revealed. More details can be found in [2, 13]. Further, a fine partitioning also makes the solution in [17] for SUM and AVG very costly (in both communication and post-processing at the client), as one encrypted value for *each* potentially qualified bucket must be sent to the client for decryption.

1.4 Overview of Our Solution

In contrast to [12], we propose to use a secure modern homomorphic encryption scheme, which typically operates on a much larger (encryption) block size (say 2K bits) than single numeric data values. The simple solution of encrypting only one value in an encryption block, which makes it trivial to apply the homomorphic property for SUM and AVG, is highly inefficient Instead, we propose an interesting way to manipulate multiple data values in large encryption blocks. Such manipulation handles complex and realistic scenarios such as predicates in queries, compression of data, overflows, and more complex numeric data types (float), etc. Therefore, our work is the first comprehensive study of using a *secure*, *modern* homomorphic scheme to compute SUM and AVG in an un-trusted server environment. Specifically, our contributions are:

- A solution for computing SUM and AVG queries in the described system model. The queries can be general (e.g., with predicates), and data can be first compressed, and then encrypted.
- A randomized algorithm plus its analysis to improve the performance of our solution.
- A technique for using this scheme on floating point numbers.
- An actual implementation of a homomorphic encryption scheme and computation of SUM and AVG in a real database system. We then test the viability of the solution in terms of performance.

The basic idea underlying our solution is to densely pack data values in an encryption block, and perform computation directly on the ciphertext using a secure homomorphic encryption scheme in a novel way. This approach fits with our system and security model because the database server performs the bulk of the computation without having access to the secret key or the sensitive data. In the end, a constant number of ciphertext blocks are passed back to the trusted agent (i.e., the Key Holder) to perform a final decryption and simple calculation of the final result. Moreover, because of the

dense packing of values to reduce the number of modular multiplications and the minimization of the number of expensive decryption operations, the performance is acceptable. We demonstrate that the performance of SUM and AVG with our approach is comparable to that of using a traditional symmetric encryption scheme, such as DES [6]. The latter is not a viable solution in our system and security model, as the database server must have access to the secret key or plaintext.

We study the problem in the context of an open-source column-oriented DBMS called C-Store [25, 27]. C-Store is a read-optimized relational DBMS. The most salient difference between it and a traditional "row-store" is the way that it stores data by column rather than by row and the way that it uses sorting and compression [1, 25]. Our encryption techniques fit naturally with the design of C-Store. We will also discuss how to apply our techniques to row-store systems.

The rest of the paper is organized as follows. In Section 2, we briefly introduce the background needed for the paper. We present our basic solution of doing SUM and AVG over encrypted data in Section 3. Section 4 presents a randomized approach to further improve our algorithm, and we then illustrate how we can apply our technique for floating-point data in Section 5. In Section 6, we discuss special usage scenarios, some other system aspects, and usage in row stores. And in Section 7 we show some experimental results to verify the performance viability of the solution. Finally, we conclude the paper and point out some future work in Section 8.

2. BACKGROUND

2.1 Compression in C-Store

In C-Store, data can be stored either *uncompressed* or compressed using one of the *three compression methods*: Run length encoding (RLE), Bitmap encoding, and Delta encoding [18, 23, 25]. Data compression can clearly save I/O cost, but it can also increase CPU costs if the database has to decompress a lot of data to process the query. The key to the high performance of C-Store is that it can often evaluate significant pieces of a query without decompressing the data [1].

When encryption is combined with compression, the data is first compressed using any of the C-Store compression schemes, and then an encryption algorithm is run on the compressed data. C-Store's goal is to evaluate queries without decompressing the data. Thus, in fact, our scheme often requires no decompression, and minimum decryption (at the end of query processing).

2.2 Homomorphic Encryption

Homomorphic encryption is a well-known technique in cryptography. The additive homomorphic property of a homomorphic cryptosystem is:

$$enc(a+b) = enc(a) \times enc(b)$$

where *a* and *b* are two plaintext message blocks, and "*enc*" is the encryption function that takes a plaintext message block (and an encryption key, which we omit) and returns the ciphertext block. Thus, it is clear that in the above equation, "+" operates on the plaintext, and "×" operates on the ciphertext. An example of such an encryption scheme is the Paillier system [21]. In our experiments, we use a generalized version of the original Paillier system [5]. Such a system is provably secure under some number theoretic assumption, which is commonly regarded as more secure than those that are not provably secure (e.g., commonly used block ciphers), which are more prone to attacks as computational power improves.

Homomorphic encryption has been used in voting systems, etc [5]. In this work, we use it to handle most of the computational work of SUM and AVG queries while keeping data in ciphertext form. The generalized Paillier cryptosystem suits our needs because other than the homomorphic property, it has a nice property that by dynamically adjusting the encryption block size, the ciphertext expansion factor can be close to 1. That is, there is a parameter *s* (proportional to the block size), such that the ciphertext expansion factor is $\frac{s+1}{s}$. For example, if $s = 4$, the expansion factor is 1.25. Clearly the bigger *s* is, the smaller the expansion factor. However, a large *s* value causes big encryption block size, which slows down cryptographic operations on the data. Thus there is a tradeoff.

3. A SOLUTION FOR SUM AND AVG

3.1 The Basic Building Block

In this section, we describe a simple basic building block, which uses the homomorphic encryption property to efficiently compute SUM or AVG over an encrypted column. For now we assume that there are no predicates in the query, nor any compression on the column (i.e., the encoding type is uncompressed). For example, SELECT AVG(*salary*) FROM *employees*, where *salary* is not compressed, but encrypted.

Note that for a typical homomorphic encryption scheme, such as the generalized Paillier system, the size of a plaintext block has to be sufficiently big (e.g., at least 1024 bits) to be secure. Thus we consider the operations $(enc, dec, +, \times)$ as operating on big binary numbers. As a result, a plaintext block needs to hold more than one data value in general. For example, if the plaintext block size is 2048 bits, it should hold $2048 / 32 = 64$ integer values each of which is 32 bits.

We first show how we encrypt the data. Typically only a small fraction of the columns are sensitive and in need of encryption. One should be able to specify which columns to encrypt. In this way, a query referencing a non-sensitive column does not have to go through the expensive decryption operation on that column. Just as in Oracle [19], we provide column-level encryption. Recall that C-Store organizes and stores data column-wise. We subdivide the column data into encryption blocks which have to be relatively large (such as 2K bits) to be secure. Each such block, then, contains multiple column values.

Let us illustrate this through an example. Suppose each plaintext encryption block can hold 64 data values, and the whole column of the table fits in 32K such blocks, which implies that the whole table has 64 * 32K = 2M records. We use the generalized Paillier scheme to do one encryption on each block, and get 32K ciphertext blocks, as shown in Figure 2. Suppose each value has 32 bits, then a block has 32 * 64 = 2K bits. In the encryption operation, each block is treated as a 2K bit number, and is converted to a ciphertext block. Note that in a plaintext block, we can imagine there is an invisible boundary between two values, each of which is 32 bits. But because each encryption block is treated as one big number during encryption, there are no value boundaries within a ciphertext block (i.e., it no longer holds that each 32-bit sub-block is the encryption of an integer, but rather, information of all values are "mingled" together). Next we give *Algorithm 1* to compute SUM (illustrated in Figure 2), followed by a toy example (Figure 3).

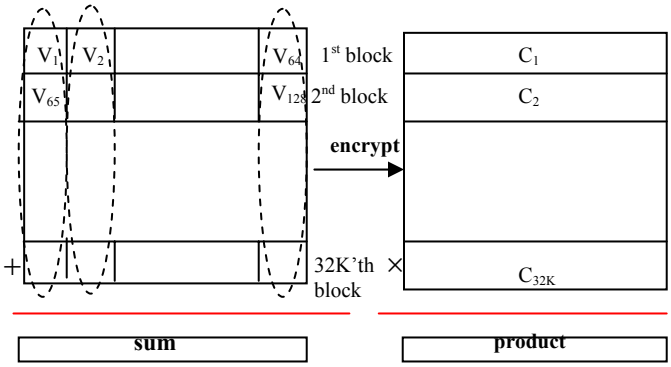

Figure 2: Organizing values into encryption blocks, encrypting using the generalized Paillier scheme, and using Algorithm 1 to sum them. The left-hand side is plaintext. Each block is shown as one row, containing 64 actual values. Encryption treats each block as one large 2K bit number, and converts the 32K plaintext blocks into 32K ciphertext blocks on the right-hand side. The product of the 32K ciphertext large numbers, when decrypted back, corresponds to the sum of the 32K plaintext large (2K-bit) numbers.

Algorithm 1.

Input: c_i (for $i=1,...,n$), each of which is the encryption of a block of k values v_{ij} (for $j=1,...,k$), decryption key K.

Output: The sum of all $n \cdot k$ *values:* $\sum_{i,j} v_{ij}$

(1) Compute $c = \prod_{i=1}^{n} c_i$. Note that under the generalized Paillier cryptosystem, these are modular multiplications.

(2) Compute $s = dec(K, c)$, the decryption of c using key K.

(3) Split s into k values of equal bit length s_i (for $i=1,...,k$), such that $s = s_1 \circ s_2 \circ ... \circ s_k$, where the operator $\circ$ is "bit-string concatenation" treating s_i as a bit-string (instead of a binary number).

(4) Output $\sum_{i=1}^{k} s_i$ (treating s_i as a binary number).

Note that in step 3, this algorithm produces the sums of the vertical slices through each block (i.e., the sums of the i'th position in each block). The algorithm assumes for now that these sums do not overflow the 32 bit representation for integers. We will relax this shortly.

Figure 3 shows a toy example in which we have 3 encryption blocks (n=3), each containing 4 values (k=4). As we discussed, in an actual system, n and k are much bigger. Step 0 is the encryption of plaintext blocks (left side) to ciphertext (right side). Step 1,2, and 3 roughly correspond to the same step numbers used in Algorithm 1. In step 1, we do a modular multiplication on the ciphertext blocks, followed by a decryption of the product in step 2. Finally in step 3 we split the resulting plaintext block into 4 (i.e., k) values (18,15,10, and 22), corresponding to the sums of the four vertical slices of the three plaintext blocks. Thus adding them together produces the total sum, 65.

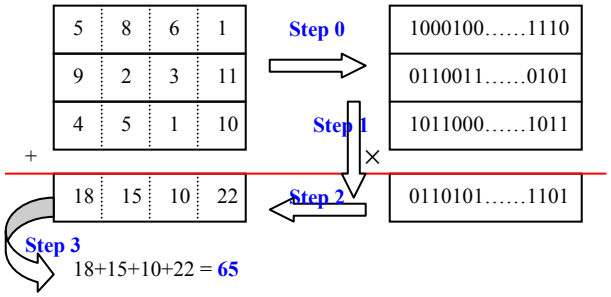

Figure 3: An illustrative toy example of Algorithm 1.

Theorem 1: Assume no "overflow" when adding up a vertical "slice" to get $\sum_{i=1}^{n} v_{ij}$ (for $1 \le j \le k$). That is, $\sum_{i=1}^{n} v_{ij}$ does not overflow the number of bits *for* each value in a block, thus no carry bits between vertical slices (See Figure 2). Algorithm 1 correctly computes the sum of all $n \cdot k$ values.

Proof: Let p_i $(1 \le i \le n)$ be the *i'th* plaintext block, i.e., $p_i = v_{i1} \circ v_{i2} \circ ... \circ v_{ik}$. From homomorphic property, after step (2) of the algorithm, $s = \sum_{i=1}^{n} p_i = \sum_{i=1}^{n} v_{i1} \circ v_{i2} \circ ... \circ v_{ik}$. Because of the assumption of no overflow between slices, $s_j = \sum_{i=1}^{n} v_{ij}$ $(1 \le j \le k)$. Thus, finally the output $\sum_{i=1}^{k} s_i = \sum_{i=1}^{n} \sum_{j=1}^{k} v_{ij}$. ■

In Algorithm 1, we only do one decryption in step (2), and *n-1* modular multiplications (a multiplication is much cheaper than decryption), as opposed to having to decrypt every data value if we were not using a homomorphic encryption scheme. It is clear that the computation of AVG would be similar.

Note that the heavy computation work happens in step (1) (where *n* can be huge) and is carried out at the database server in the system model described in Section 1. Steps (2), (3), and (4) beginning with a decryption are executed at the secure agent (Key Holder).

3.2 Handling Overflows

In Theorem 1 we made an assumption that there is no overflow in the sums of vertical slices. However, overflows are possible. If overflows did happen, there would be carry bits between two vertical slice boundaries when the plaintext blocks are added up. Whether or not overflows happen depends on the actual value range, and the number of records.

Example 1. *Suppose the average value in each vertical slice is within 1024, and the column type is INT (32 bits). This allows us to have* $\frac{2^{32}}{1024} = 4M$ *encryption blocks. If each block has 64 values, there can be 256M values. Even un-compressed, the table can have 256M records. Compression allows even more records to be accommodated.* ■

The above suggests that overflow may be rare, yet we still need to handle it. One approach is to simply leave some extra space preceding each plaintext value in an encryption block. This approach, albeit simple, has the downside of increasing the ciphertext size. A more sophisticated approach is to organize the encryption blocks into groups, shown next.

As illustrated in Figure 4, Algorithm 1 is applied within each group. Each group is similar to what is shown in Figure 2. The difference is that we also maintain a row vector of partial sums (sums of values in each vertical slice) for each group. In the ciphertext world, the row vector corresponds to the (modular) product of the ciphertext blocks in the group. Thus, in Figure 4, for each of the two groups, we only show its plaintext representation. We dynamically maintain the partial sums row vector during the insertion of values. By keeping track of the sums of vertical slices we ensure that no overflow happens for each slice of a group. We stop filling a slice when the next value *v* would cause it to overflow. In such a case, we try to place *v* in the next slice (to the right). This continues until *v* cannot be placed in any slice which means that the group is full. When a group is full, a new one is created. Thus this approach has less overhead on ciphertext size and is more efficient in I/O. In the toy example of Figure 3, in the plaintext (left side), the *(18, 15, 10, 22)* row vector is what we maintain on the side when we encrypt values. Suppose the capacity of each of the four positions is *25*, and the next value to insert is *8*. Then the first vertical slice would overflow (*18+8 > 25*) if we put the new value there. Hence we put the new value into the second slice and the partial sum row vector becomes (*18, 23, 10, 22)*.

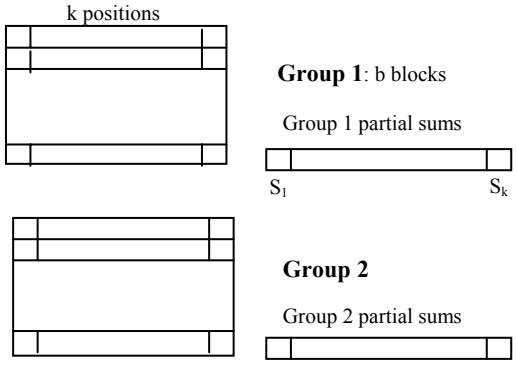

Figure 4: Organizing encryption blocks into groups.

We note that the overflow handling technique introduced in this section applies to the general cases discussed in the following sections as well. We do not explicitly show it in the algorithms for simplicity.

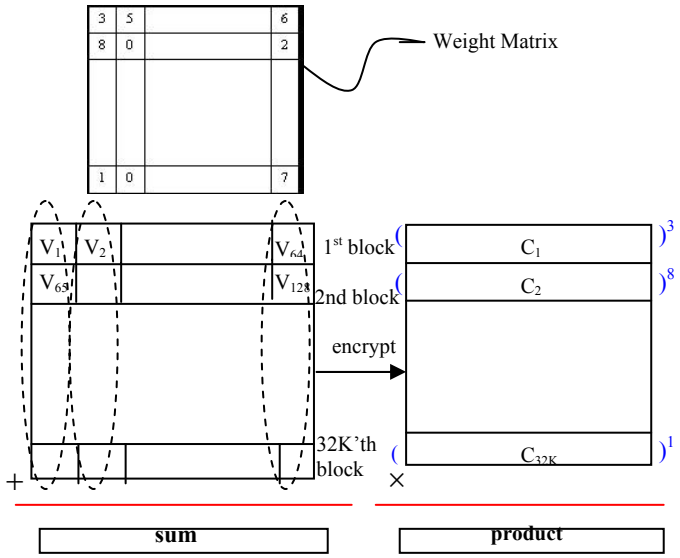

Figure 5: Illustrating Alg. 2, computing the 1st vertical slice.

3.3 An Extension of Algorithm 1

A typical query will select a subset of the rows of a table and then aggregates over that subset. Therefore, in this context some values in a block have no contribution to the sum. Moreover, in many columns, some values may appear many times and a compression method, such as *Run Length Encoding*, makes it easy to know the number of repetitions. We can capture this by associating an integer weight with each value to be summed. We extend Algorithm 1 and add a "weight matrix", as shown in Figure 5. Each value in the plaintext blocks has a weight specified by an integer at the corresponding position in the weight matrix, and we want to compute the weighted sum, i.e., the sum of each value multiplied by its weight. Interestingly, this small extension works for predicates in queries as we will discuss in Section 3.4 and for compressed data. The following algorithm describes the extension.

Algorithm 2.

Input: c_i (for $i=1,\ldots,n$), each of which is the encryption of a block of k values v_{ij} (for $j=1,\ldots,k$) of the same bit length, decryption key K, and the weight matrix w_{ij} (for $i=1,\ldots,n, j=1,\ldots,k$).

Output: The weighted sum of all $n \cdot k$ values: $\sum_{i,j} v_{ij} \cdot w_{ij}$

(1) $sum = 0$
(2) **For** each vertical slice $j = 1$ to k, **do**
(3) Compute $c = \prod_{i=1}^{n} c_i^{w_{ij}}$. Note that in the generalized Paillier cryptosystem, these are modular multiplications and exponentiations.
(4) Compute $s = dec(K, c)$, the decryption of c using key K.
(5) Split s into k values of equal bit length $s_i (1 \le i \le k)$, i.e., $s = s_1 \circ s_2 \circ \ldots \circ s_k$.
(6) $sum = sum + s_j$
(7) **End For** loop.
(8) Output *sum*.

The complication here is that each of the k values in an encryption block in general has different weights, thus we cannot simply raise the ciphertext block to a power of a uniform weight. Instead, we do it in k loops, and the *j'th* loop takes the *j'th* value in "*s*" and discards the rest. In the toy example of Figure 3, suppose the 1st column of the weight matrix is ***2, 0, 1***, the 2nd column is ***1, 3, 0***, the 3rd is *1, 2, 5*, and the last column is *6, 0, 1*. Let the three blocks (rows) of ciphertext be c_1, c_2, and c_3. Then Algorithm 2 proceeds in four loops (one for each vertical slice). In the 1st loop, we compute $c = (c_1)^2 (c_2)^0 (c_3)^1$, decrypt it, and then only take the value in the 1st position (i.e., s_1 in step 5 and 6 of the algorithm). As we will show in Theorem 2, indeed $s_1 = 5\times2+9\times0+4\times1=14$, the weighted sum of the first vertical slice of the plaintext in Figure 3. In the same manner, the 2nd loop starts by computing $c = (c_1)^1 (c_2)^3 (c_3)^0$, and so on. After four loops, we get the weighted sum of the 12 values.

Theorem 2: Assume there's no "overflow" when adding up a vertical "slice" to get $\sum_{i=1}^{n} v_{ij} \cdot w_{ij}$ (for $1 \le j \le k$). That is, $\sum_{i=1}^{n} v_{ij} \cdot w_{ij}$ does not exceed the *space for* each value in the plaintext encryption block (See Figure 5). Algorithm 2 correctly computes the weighted sum $\sum_{i,j} v_{ij} \cdot w_{ij}$.

Proof Idea: The idea of this proof is similar to that of Theorem 1, so we omit the details. One additional point is that the homomorphic property states that $enc(a+b) = enc(a) \times enc(b)$, it follows $enc(a \times w) = (enc(a))^w$, where the modular exponentiation operates on the ciphertext. Algorithm 2 has *k* loops (outer loop), and each loop computes a partial weighted sum of a vertical slice in Figure 5. Thus, the *j'th* loop takes the *j'th* value in "*s*" and discards the rest. ■

Note that compared to encryption or decryption, modular multiplication is much cheaper. We will discuss the performance improvement in Section 4. Also note that when we implement Algorithm 2, we can carry out the *k* loops in parallel, i.e., incrementally compute the *k* products at the same time block by block. Thus we only need to read the ciphertext from the disk once.

3.4 Handling Predicates

We discuss how to use Algorithm 2 to handle predicates in the query. We classify predicates into two categories: (1) those that do *not* reference a sensitive, encrypted column and (2) those that do.

Example 2. *Suppose we have a schema:* Employees *(*name *VARCHAR(50),* age *INT,* salary *INT ENCRYPTED,* company *VARCHAR(50)), where only the* "salary" *column is encrypted. Consider these queries:*

Q1: *SELECT AVG(salary) FROM employees WHERE age > 35 AND company = 'SUN'*

Q2: *SELECT AVG(salary) FROM employees WHERE salary > 60000 AND company ='MICROSOFT'*

Using our classification, Q1's predicates are in the first category, while Q2 has a predicate in the second category. ■

We first consider the case in which all predicates are in the first category. Many DBMS's support bitmap indices. Evaluating a predicate on a column with a bitmap index results in a bit-string, in which each bit indicates whether a row in the table is qualified (*1* indicates the record is qualified, and *0* otherwise). Then this bit-string is used to "mask" the aggregated column (*salary*), and the resulting aggregate is computed over the masked rows. It is easy to see how we can derive this bit-string with other forms of index as well. In our context, this bit-string is essentially a one-dimensional form of the "weight matrix" in Algorithm 2 (treating each *bit* value 0/1 as the weight of the attribute value). That is, we have 1's and 0's in the weight matrix. It is easy to verify that Algorithm 2 indeed computes the correct SUM (likewise, AVG) of the *qualified* records. Note that the weight matrix of Algorithm 2 does not necessarily require extra space overhead since it is simply the bit-string that the DBMS (e.g., C-Store) would compute anyway as a result of predicate evaluation.

In the toy example of Figure 3, we have 12 attribute values, thus 12 records in the table. Suppose the first 4 records use the 1st encryption block to store *salary*, the next 4 records use the 2nd encryption block, and so on. Suppose evaluating a predicate results in a 12-bit bit-string *1001,0110,1000*, indicating that records 1, 4, 6, 7, and 9 satisfy the predicate. Then the weight matrix derived from the bit-string, in row major form, is (1,0,0,1; 0,1,1,0; 1,0,0,0).

For a query that has a predicate of category 2 (such as *Q2* of Example 2), the encrypted column in the predicate may or may not be the column being aggregated. Like a plaintext column, an index can be built on the encrypted column using a scheme that handles indexing on ciphertext (discussed in Section 1.3). Note that if the column in the predicate is also the one being aggregated (such as *Q2*), the sensitive column is encrypted differently for SUM/AVG (using a homomorphic scheme) than in the index. Now the predicate can be evaluated efficiently without decryption, resulting in a bit-string just as a category 1 predicate does. Then we proceed using Algorithm 2 to compute the SUM/AVG. We will discuss issues such as storage and updates in Section 6.2.

Let us illustrate this through *Q2* of Example 2. We build an index on the *salary* column. In the index, values are encrypted using OPES [2]. There is also an index on the plaintext *company* column. The *salary* column is also encrypted using a homomorphic scheme as described. Using the 1st index to evaluate the predicate *salary>60000* results in a bit-string B_1 indicating which records satisfy this predicate. Similarly, using the 2nd index for *company ='MICROSOFT'* will result in a bit-string B_2. Let *B* be the bitwise AND of B_1 and B_2. Then we treat *B* as the weight matrix and proceed using Algorithm 2.

Since the index is directly built on the ciphertext and searching it does not involve decryption, searching an index on ciphertext has the same cost as searching a plaintext index. Therefore, the query performance is exactly the same as a query with a category 1 predicate with the same selectivity.

For a group-by query:

SELECT AVG(salary) FROM employees GROUP BY company

We can use the index on *company* to get a bit-string for each distinct *company* value. Again we use each of those bit-strings to compute the AVG of a group. Now, suppose the query also has a HAVING clause:

SELECT AVG(salary) FROM employees GROUP BY company HAVING AVG(salary) > 60000

Then the index on ciphertext would not help us on the HAVING clause here. What the database server gets from a homomorphic scheme is simply a ciphertext AVG value for each company. For such a query, we have to resort to a post-processing step at the Key Holder to filter out some groups after decryption. We assume, however, that the number of possible groups is manageable.

3.5 Allowing Compression

A sensitive column can be first compressed by any encoding type supported by C-Store, and then encrypted using a homomorphic scheme; we show how we can still use Algorithm 2 to compute SUM or AVG efficiently. Thus we save both I/O costs (by compression) and CPU costs (by avoiding decompression and by minimizing the number of decryptions).

In Section 2.1, we mentioned the three compression methods that C-Store supports. We first consider a sorted column encrypted after being *RLE* compressed. Recall that the RLE compression produces pairs (***v, n***), where ***v*** is the data value, and ***n*** is the number of repetitions. We put all ***v*** values from the pairs in the encryption

blocks and encrypt them separately as described earlier using a homomorphic scheme. The ***n*** values go into the weight matrix and are either encrypted separately, or are left in the clear, depending on the security requirement of the application. It is clear that Algorithm 2 gives the correct result. Note that the weight matrix of Algorithm 2 does not have extra space overhead here as well, as it is part of the representation of compressed data.

A similar approach also applies to *Bitmap encoding* (which has a *value* part and *bitmap*). We encrypt the *value* parts together in blocks, and by simply counting the set-bits in the *bitmaps* we get the weights.

Delta encoding is a little different. In Delta encoding, we have a sequence of values ($base, inc_1, inc_2, \ldots, inc_n$) corresponding to the $n+1$ actual values in the column: $base$, $(base+inc_1)$, $(base+inc_1+inc_2), \ldots$, and $(base+inc_1+ \ldots+inc_n)$. Their sum is $base\times(n+1)+inc_1\times n+inc_2\times(n-1)+\ldots+inc_n\times 1$. Therefore, we can simply put ($base, inc_1, inc_2, \ldots, inc_n$) sequences in encryption blocks, and $n+1, n, n-1, \ldots, 1$ in the weight matrix. Figure 6 illustrates this.

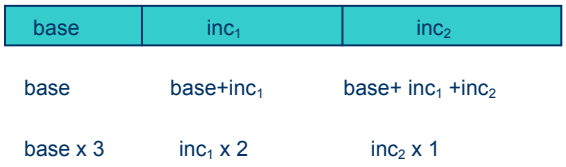

Figure 6: Using Alg. 2 for encrypted and Delta-encoded data.

Finally, we can combine the discussions in Section 3.4 and 3.5. We allow data to be compressed and encrypted, and queries to have predicates. It is analogous to the case of having predicates with uncompressed data that we discussed in Section 3.4: some of the weight values resulting from compression are reduced (possibly not all the way to 0), as some records are filtered out by the predicates.

Note that in the system model of Section 1, the database server passes a constant number (k) of ciphertext blocks to the secure agent (Key Holder), which does the final decryption and addition.

4. A RANDOMIZED ALGORITHM TO FURTHER IMPROVE PERFORMANCE

We introduce and analyze a randomized technique to further improve the performance of Algorithm 2.

4.1 The Randomized Algorithm

Recall what happens in Algorithm 2. We have k rounds (where k is the number of column values in an encryption block), and in each round we essentially compute a partial sum over a vertical slice of the plaintext values. For simplicity of presentation, we show the algorithm for uncompressed columns and with predicates in the query. This can be easily extended to include compression types.

The cost of Algorithm 2 is: a constant number (k) of decryptions plus $n \cdot k \cdot p$ modular multiplications, where n is the number of encryption blocks, and p is the combined selectivity of the predicates. We can see that, as the number of records in the table grows, n grows, hence the cost of the modular multiplications grows linearly, whereas the cost of the decryptions stays constant. Therefore, a mechanism to further lower the cost of the modular multiplications would give us additional benefit. Our randomized algorithm does just that by exploiting pre-computation and sharing of intermediate results among sub-tasks. In Section 4.2, we present a probabilistic analysis of this algorithm.

The high level idea is that we divide the encryption blocks into groups each of size s (encryption blocks). We call each of these groups a *segment*. The computation of all k vertical slices is carried out in parallel one segment after another. For each segment, using the same amount of space as the original ciphertext (s blocks), we pre-compute and store s modular multiplications of uniformly random subsets of the s blocks (i.e., s product values out of the 2^s values in total). Accordingly, for each product, we store an s-bit value *identifying* the subset (a "1" in i'th bit indicates the i'th block of the segment gets selected to be in the subset and included in the precomputed product). The computation of each vertical slice within a segment tries to use both the pre-computed values, and the results of already computed vertical slices within the segment. Note that as in Algorithm 2, a slice corresponds to a slice of "weights", or in the plaintext world, a slice of values to be summed. In the ciphertext world, the multiplication is always carried out on the whole "wide" ciphertext blocks. Figure 7 shows pictorially how the computation of k vertical slices proceeds in a parallel manner. The algorithm follows.

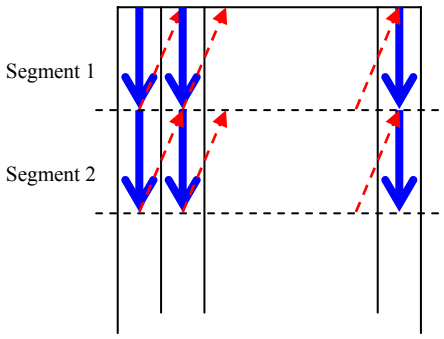

Figure 7: Illustrating parallel computation of vertical slices.

Algorithm 3.

For each segment,
  For i from 0 to k-1, (each vertical slice)
    From the 0/1 weight matrix in Alg.2, extract an s-bit value ***t*** (i.e., the actual subset to be multiplied) corresponding to this segment and vertical slice.
    Consider $s+i$ (s-bit) values $b_j\ (0 \le j \le s+i-1)$ which are the s values identifying the pre-computed subsets and i values identifying the actual subsets computed for previous slices (in previous i loops). Find one value b^* with minimum Hamming distance with ***t***.
    From the product value identified by b^*, which is either pre-computed or computed in previous i loops, derive the needed product identified by ***t***. Specifically, if b^* and ***t*** match exactly, nothing needs to be done. Else, for a bit change 0→1 from b^* to ***t***, we multiply some ciphertext block value; for a bit change 1→ 0, we multiply the (modular) inverse of some value (which is also pre-computed).
  End for i loop.
  Accumulate k product values (one for each vertical slice) across segments (by doing multiplications).

End for each segment.

In Algorithm 3, to compute a product block value corresponding to some vertical slice, we look at two sources for speedups:

1. *s* pre-computed product values for the segment
2. *i* product values just computed for previous slices of the same segment (in previous loops; *i* is the loop index).

We look for the "closest" match from these two sources by comparing the *identifying* bitmaps to find the minimum Hamming distance. For a bit difference, we either need to do a modular multiplication (for a 0→1 change) or a modular division (i.e., multiplying the inverse, for a 1→0 change).

We show a toy example. Let *k=4*, as the example in Figure 3. But unlike Figure 3, we have many more blocks, and let the segment size *s=4*. Consider the 1st segment. Let the four blocks of ciphertext in this segment be c_1, c_2, c_3, and c_4. Let the first four rows of weight matrix in Algorithm 2 or 3 (corresponding to this segment) be

$$\begin{matrix}1&0&1&1\\0&1&0&1\\1&1&1&0\\1&0&1&0\end{matrix}$$. Let $\begin{matrix}c_2c_4\\c_1c_2c_3\\c_1c_4\\c_2c_3\end{matrix}$ be the pre-computed modular products of uniformly random subsets of the *4 ciphertext* blocks. The 1st column of the weight matrix indicates we need to compute $c_1c_3c_4$. By seeking the minimum Hamming distance, we know we can use the pre-computed c_1c_4 and multiply it by c_3. The 2nd column of the weight matrix requires computing c_2c_3 and that is immediately available from the same pre-computation. The 3rd column requires computing $c_1c_3c_4$ again and that is available as we just got it for the 1st column (i.e., in a previous *i* loop of algorithm 3). The 4th column requires computing c_1c_2 and we obtain it from $(c_1c_2c_3)/c_3$. Likewise, Algorithm 3 then proceeds to the next segment for the *k* products.

4.2 The Analysis

We next analyze the cost of Algorithm 3. We first compute the expected number of modular multiplications M that it needs to perform per segment. Let M_i denote the number of multiplications performed for vertical slice *i* of the segment. We have,

$$E(M) = \sum_{i=0}^{k-1} E(M_i) \qquad (1)$$

As M_i is a discrete random variable with *non-negative* values, we have (from [15])[1]

$$E(M_i) = \sum_{r=1}^{\infty} \Pr(M_i \geq r) = \sum_{r=1}^{s} \Pr(M_i \geq r) \qquad (2)$$

Let random variables M_{ij} denote the Hamming distance between the s-bit value t (identifying the i'th vertical slice) and value $b_j\,(0 \leq j \leq s+i-1)$ as described in Algorithm 3. Thus from Algorithm 3, we have

$$\Pr(M_i \geq r) = \Pr[\min_{0 \leq j \leq s+i-1} M_{ij} \geq r] = (\Pr[M_{ij} \geq r])^{s+i} \qquad (3)$$

where the last equality follows from the fact that the *s* pre-computed subset products are chosen uniformly at random, and we assume the subset identifiers of already computed vertical slices are also uniformly random. As we'll discuss later, we may make them correlated to further enhance performance, but that can only reduce the total number of multiplications.

To compute $\Pr[M_{ij} \geq r]$, imagine that we fix the s-bit value *t* (identifying the i'th vertical slice), and enumerate the cases that the uniformly random value b_j differs from *t*. Thus,

$$\Pr[M_{ij} \geq r] = \frac{\sum_{u=r}^{s} \binom{s}{u}}{2^s} \qquad (4)$$

Finally, combining (1) to (4), we have,

$$E(M) = \sum_{i=0}^{k-1} \sum_{r=1}^{s} \left(\frac{\sum_{u=r}^{s} \binom{s}{u}}{2^s} \right)^{s+i} \qquad (5)$$

From (5), we can compute the expected number of multiplications within a segment, given *k, s* values. Now suppose the previous Algorithm 2 were used, it is easy to see that the number of multiplications within a segment would be $k \cdot s \cdot p$, where *p* is the *combined* selectivity of all predicates on other columns. Likewise, for Algorithm 3, from (5), we can further compute a "percentage" value $\frac{E(M)}{k \cdot s}$, and compare it with *p*. Actually, to be precise, we also need to count the cumulative multiplications "across" segment boundaries. For an upper bound, we simply assume every vertical slice of a segment is a non-empty subset (product). Hence the additional percentage due to this is no more than $\frac{n \cdot k}{n \cdot s \cdot k} = \frac{1}{s}$, where *n* is the total number of segments. Therefore, the multiplication percentage of Algorithm 3, p^*, satisfies

$$E(p^*) \leq \frac{E(M)}{k \cdot s} + \frac{1}{s}$$

For a given *k* value, we can determine the optimal *s* value (segment size) that minimizes the p^* value bound (A simple program will do). For example, if *k=64*, the best upper bound is 0.27 when *s=7*. Therefore, on average, when the combined selectivity of predicates on other columns is greater than 0.27, Algorithm 3 performs better than Algorithm 2. The nice thing about it is that regardless of how close the selectivity is to 1, Algorithm 3 "stabilizes" the number of multiplications, as if the selectivity were staying at 0.27. In contrast, Algorithm 2 will proportionally perform a lot more multiplications as the selectivity increases.

Algorithm 3 typically reads a subset of pre-computed *s* products, and a small fraction ($\frac{M}{k \cdot s}$, which has expected value 0.12 in the example of k=64 and s=7) of the original ciphertext block values (or the inverse). At run time, according to the actual selectivity of the predicates, we can compare the costs and decide whether to kick off Algorithm 3 for a segment, or just use Algorithm 2. For example, when the selectivity is very low, Algorithm 2 performs fewer multiplications and is chosen.

Note that we could further improve Algorithm 3 using other techniques. For instance, if we know a predicate is most likely on *some* column, we can organize encrypted column values in the vertical slices by clustering on the "predicate column". This way, even though the bit vectors of the vertical slices are not uniformly

[1] Intuitively, for r from 1 upwards, accumulatively, $\Pr(M_i \geq r)$ is the probability that we add 1 to the expectation.

random, "reuses" of product values are much more likely because the *k* target subset identifiers probably have either all 1's, or all 0's (due to the clustering).

5. ON FLOATING-POINT NUMBERS

Observant readers may notice that the techniques we introduced of using homomorphic encryption to do SUM and AVG only work with integer types (For negative integers, one can use a "bias" to reduce to the non-negative value case). For REAL or DOUBLE, this is inherently much harder, due to the separation of the "exponent" and the "significant" part of a number. Given that floating point numbers are often a requirement, we now show how the homomorphic approach can also be made to work on floating-point numbers.

5.1 Some Observations and the Basic Idea

The IEEE 754 floating-point standard has single precision and double precision number formats. Throughout this section, we illustrate the ideas on single precision floating point numbers. With straightforward changes they work with double precision as well. The actual value of a single precision number is $(-1)^s(1+significant)\times 2^{(exponent-127)}$, where the *significant* and *exponent* parts are 23 and 8 bits, respectively.

Observation 1: If we add two numbers that differ by at least 24 in their exponents, the result is simply the bigger of the two numbers. This is because when we shift the significant part of the number with a smaller exponent to the right during the normalization step of the addition, all significant bits are shifted out. For example, 1.2763×2^{15} plus 1.18×2^{-9} is simply 1.2763×2^{15}, as their exponents (15 and –9) differ by 24.

Observation 2: SQL does not restrict the order of the numbers upon which *SUM* or *AVG* is applied. In particular, if we knew the maximum exponent in the list of numbers we are aggregating, we would only need to consider the subset of numbers whose exponents differ by no more than 23 from the maximum exponent.

This is interesting in two ways:

- It makes SUM and AVG faster and more efficient, if significantly fewer numbers need to be summed due to this.
- As we describe next, it enables the homomorphic encryption technique.

Note that computing *SUM* or *AVG* on the same set of numbers, but in a different order, we may get a result of different precision. But there is no requirement on how an order should be selected, and even if we could determine an order, it might be impossible or very inefficient to enforce it. We also note that in practical applications an attribute with values whose exponents differ by 24 or more is arguably rare. However we still need to be able to handle this case for completeness.

For the sake of simplicity, we assume, for now, that the exponents are almost uniformly distributed in [0, 255] (i.e. [-127, 128] after subtracting the bias). Also, for now, we just consider positive numbers (s=0). The idea is that we still use groups of encryption blocks, similar to what we do to handle overflows in Section 3.

Let us consider one example of grouping. We may divide the encrypted blocks into 32 groups (G_0, …, G_{31}), each covering 8 values of the exponent range [0, 255] (say, G_0 covers [248, 255], G_1 covers [240, 247], and so on). Imagine that a predicate on another column selects a subset of the encrypted column values to be summed. The maximum exponent in this subset of values falls in one of the groups, say G_i, according to the 8-value range that it covers. Once we can determine that, we use *only* group G_i to compute the sum using the algorithms we presented earlier. This is shown in Figure 8. Thus there are two key questions:

1. What values are encrypted in each group?
2. How can one determine which group to use at run-time to process a query given that values are encrypted?

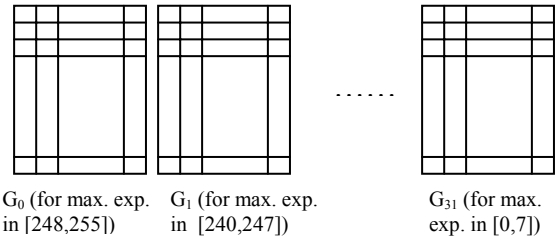

Figure 8: Illustrating groups of encryption blocks for floating-point numbers.

5.2 What Values Are in Each Group?

We start by answering the first question above. Let us consider group G_i covering *maximum* exponent range [e, e+7]. From Observation 2, in order for a set of numbers whose maximum exponent is in [e, e+7] to use encrypted values in group G_i *only*, G_i needs to contain all numbers whose exponents are in [e-23, e+7]. We encrypt the 24-bit significant part (the default leading bit "1" and 23-bit *significant* part as in the IEEE 754 floating-point standard) of these numbers normalized to having exponent *e*. That is to say,

1. For a number whose *exponent* is exactly *e*, we encrypt its 24-bit significant as it is.
2. For a number with *exponent* $e+j\ (1\le j\le 7)$, its 24-bit significant part is shifted *left j* bits (i.e., multiplied by 2^j) and then encrypted. Thus, we encrypt *24 + j* bits.
3. For a number with *exponent* $e-j\ (1\le j\le 23)$, its 24-bit significant part is shifted *right j* bits (i.e., divided by 2^j) and then encrypted. Thus, we encrypt *24 - j* bits.

For example, consider G_0, which covers maximum exponent range [248, 255]. For numbers with significant part *S* and with exponent 248, we simply put "$1\circ S$" (24 bits) into a plaintext block for encryption. (Note that the small circle denotes concatenation.) But for numbers with exponent 249, 250, … and 255, we shift "$1\circ S$" *left* 1, 2, … and 7 bits, respectively; and for numbers with exponent 225, 226, … and 247, we shift "$1\circ S$" *right* 23, 22, … and 1 bit(s), respectively. We then put the "normalized" significant part into plaintext blocks for encryption.

Consider a list of floating-point numbers we want to sum. The maximum exponent must fall in one of the groups. Once we can discover the group, we can use that group *only* to compute the significant value of the sum (with normalized exponent *e*), applying our algorithms for summing integers. For example, suppose our list of numbers to sum is 1.25×2^9, 1.5×2^{12}, and 1.375×2^6. We first determine that the maximum exponent (12) falls in group G_{14}. From the group partition we also know we need to normalize the numbers to have exponent 9 before we add them up, by shifting the significant part of the 2nd number *left* 3 bits and by shifting the

significant part of the 3rd number *right* 3 bits. We then encrypt the normalized significant parts of group G_{14} using a homomorphic scheme as described earlier (clearly we need to note the bit position that separates the integer and fractional parts of the numbers).

Note that this process discards the numbers (if any) whose exponents are too small to contribute to the sum.

5.3 Which Group to Use for a Query?

Now we tackle the second question in Section 5.1. First we must realize that the trivial solution of storing the exponent of each value and finding the maximum among the set of records being summed is not secure because we cannot expose value range information. Also decrypting each exponent at run-time would be too costly.

In the following, we use bitmaps and talk about them in terms of the set of records that they represent. We will use set operators on bitmaps. For example, $\cup$ would correspond to bitwise OR. We use [0] to denote a bitmap of all zeros.

Bitmaps are compact and their operations (such as bitwise AND) are fast given today's hardware. Suppose that we have a bitmap M_i for each encryption group G_i indicating which records are in that group's *maximum* exponent range (i.e., M_i contains records whose values' exponents are in [e_i, e_i+7]). Further, suppose that *P* is the bitmap produced by evaluating a given query predicate. We look for the smallest *i*, such that ($M_i \cap P$) $\neq$ [0]. Recall that a smaller index *i* corresponds to a group with higher exponent range.

To be more efficient, we use a binary search. Let the number of groups be *n*. We define *n* bitmaps $S_0 = M_0$, $S_1 = M_0 \cup M_1$, $S_{n-1} = M_0 \cup M_1 \cup ... \cup M_{n-1}$. Note that $S_0 \subseteq S_1 \subseteq ... \subseteq S_{n-1}$. The algorithm is as follows. We invoke it with parameters (0, n-1).

Algorithm *BinaryFindGroup (low, high)*.
Input: P, $S_0, \cdots S_{n-1}$.
Output: The group to use for SUM or AVG.

If $low \geq high$, RETURN *low*.
Let $i = \left\lfloor \frac{low + high}{2} \right\rfloor$
If P AND $S_i \neq$ [0],
  Then RETURN *BinaryFindGroup (low, i)*.
  Else RETURN *BinaryFindGroup (i+1, high)*.
End.

Once we determine which group to use, we need to determine *all* the values in the group that are also in P. For this we need another bitmap T_i for each group G_i. T_i contains the record positions of *all* values in G_i that we need to consider (with exponents in [e_i-23, e_i+7], and not just [e_i, e_i+7] as in M_i). Therefore, AND'ing T_i with P gives a bitmap of all the values that we should sum.

6. DISCUSSION

6.1 Other Operations

So far we only deal with SUM and AVG on a single (encrypted) column. There are situations in which an application may require an aggregate over a more complex expression. Consider the following two queries:

Q1: SELECT SUM ($2 * salary$) FROM *employees*
Q2: SELECT SUM ($price * quantity$) FROM *products*

For *Q2*, suppose the *price* column is encrypted, while the *quantity* column is not. In many (though not all) cases, we can still apply our techniques. For example, if the sum of an expression can be converted to an equivalent expression on the sum of the column (such as in *Q1*), then we can still use the techniques we introduced to first compute the sum of the column. For *Q2*, if the *quantity* column is of integer type, then we put it into the "weight matrix" of Algorithm 3. Not all expressions can be handled this way. For instance, suppose the *quantity* column were also encrypted, then Q2 could not be processed using our algorithms.

For JOIN (which is beyond the scope of this paper), there are two cases. If the join predicate is on a sensitive, encrypted column, then handling this is an open question. As far as we know, none of the existing solutions can completely handle this without decryption (even with OPES). If the join predicate is not on a sensitive column and the result set contains a column encrypted with a homomorphic scheme, the database server may have to put "pointers" in the result set, pointing to values at specific positions in ciphertext blocks passed to the Key Holder (as a ciphertext block contains multiple values).

6.2 Update and Storage

In the scheme we described, updates on individual values would require a whole encryption block to be re-encrypted. The increased cost does not include much I/O (as an encryption block is still typically much smaller than a page), but consists mostly of re-encrypting a block typically larger than an individual value (CPU cost). However, this is not a serious issue for many OLAP applications. Recall that data warehouse systems (e.g., C-Store) are read-optimized. Analytical processing in decision support differs from online transaction processing in that it involves very complex queries (often with aggregates) and *few or no updates* [26]. Also, updates in a system like C-Store are performed in large batches [4, 26]. Thus, individual updates are not a concern.

As discussed in Section 3.4, if the predicate in an aggregate query is on the column being aggregated, we need to build an index on that column which is encrypted with OPES. In this case the column is encrypted in two ways. This does not affect query performance (compared to a query with a predicate on a plaintext column), but takes more disk space. It does not seem to be a serious issue as the cost of disk space has been falling rapidly in recent years, and the trend continues. In addition, the aggressive compression techniques in C-Store allow us to support storing columns in different ways (e.g., in different sort-orders) without an explosion in space [25]. However, if space is really an issue, we can resort to a *sparse* B+ tree index. C-Store organizes columns into *projections* (sets of columns) and each projection has a *sort-key* [25]. We can sort the sensitive aggregate column before applying homomorphic encryption and then build a sparse page-level index over the encrypted column. The first *plaintext* value of each page is *also* OPES encrypted and the sparse index is built using those values as keys. It is then clear how we can perform a range query with such an index, and compute SUM or AVG afterwards. For example, consider a range query such as "**SELECT AVG(*salary*) FROM t1 WHERE *salary* > 60,000 AND *salary* < 500,000**". The initial answer will be imprecise because the first and last pages used may

contain values outside the range. The database server must pass these first and last pages to the Key Holder (as well as the total number of values used for the tentative AVG result) for post-processing to make the final result accurate.

6.3 Usage in Row Store Systems

Although we conducted the work in a column-oriented database system, the same techniques can be applied to a row store system. In a row store, the homomorphic encryption ciphertext would be stored "outside" the table, much like an "index", except that it is for the computation of SUM and AVG, not for search.

7. EXPERIMENTS

7.1 Setup

Our experiments were conducted using C-Store on Debian Linux. We implemented the generalized Paillier system using the GMP library [28] (edition 4.2). We enhanced the C-Store code to support encryption with different schemes, such as DES and generalized Paillier. We also changed C-Store's code for aggregates and implemented and evaluated Algorithm 1, 2, and 3 as described in the paper. The algorithms were implemented in C++. The experiments were run on a Linux workstation with an AMD Athlon-64 2Ghz processor and 512 MB memory.

The goal of the experiments is to verify the viability of the solution in terms of performance (Certainly we have also verified that the computation result is actually correct, i.e., consistent with that of plaintext). To the best of our knowledge, using a homomorphic scheme seems to be the only solution so far to securely compute SUM and AVG without having access to the decryption key and the plaintext. Yet it is still necessary to verify that the performance of such a solution is acceptable.

7.2 Experiment 1

In this experiment, we evaluated the performance of an AVG query on different database sizes but with fixed selectivity (25%) using homomorphic encryption and Algorithm 2, with overflow handling as described in Section 3.2. We compared its performance with the C-Store system using both DES encryption and no encryption at all. The plaintext of *salary* is generated uniformly at random in the range of 20,000 to 200,000 (As the bulk of the computation is on ciphertext, the performance does not have much dependency on the actual plaintext values). We experimented on a category (1) query **SELECT AVG(*salary*) FROM *employees* WHERE *age* > ?**, as well as a category (2) query **SELECT AVG(*salary*) FROM *employees* WHERE *salary* > ?**. The selectivity of the predicates in both queries is fixed at 25%. The two categories are described in Section 3.4 (i.e., based on whether it references an encrypted column). One index is built on the plaintext *age* and another is built on the *salary* column encrypted using an order-preserving scheme [2]. As we discussed in Section 3.4, their query running times are about the same since the selectivity is the same. Hence we only plot one set of curves.

Figure 9 shows the result. We can see that, with a state-of-the-art homomorphic encryption scheme (generalized Paillier), C-Store runs slightly faster than using DES for encryption. This is due to the saving in the decryption cost during execution. We see that the cost of using homomorphic encryption, albeit lower than using DES, is still much higher than that of the plaintext (i.e., when the column is not encrypted at all). The reason is that although the decryption cost is now constant, there is a cost of modular multiplications, which is proportional to the number of records. Dense packing of values in encryption blocks reduces the number of modular multiplications. As expected, we find that the cost of final decryption and addition at the Key Holder is negligible compared to the whole cost. Thus we do not plot it separately.

7.3 Experiment 2

In contrast to the first experiment, we now fix the data size to be 50M records, but vary the selectivity of the predicates from 5% up to 65% and compare the query run time using Algorithm 2 to that of DES. The first two bars in each group of three bars in Figure 10 show the result.

We find that the performance difference (ratio) between DES and generalized Paillier (using Algorithm 2) is quite consistent across different selectivities. The reason is that with Algorithm 2, while the decryption cost is constant, the modular multiplication cost is proportional to the number of *qualified* records selected by the predicates and is the dominant part of the CPU cost when the number of records is large. With DES encryption, the CPU cost, dominated by decryption, is also roughly proportional to the number of *qualified* records.

7.4 Experiment 3

Experiment 3 is the same as the previous experiment, except that we change Algorithm 2 to Algorithm 3 and look at the improvement under different selectivities of the predicate (with the same data size). Figure 10 also shows this result.

When the selectivity is low (25% or below), Algorithm 3 is no better than Algorithm 2. In fact, the execution engine should revert to Algorithm 2 when the resulting number of modular multiplications from using Algorithm 3 is no smaller, as we discussed in Section 4.2. When the selectivity is high (in our experiment, 35% or more), Algorithm 3 begins to dominate, and we can see that performance roughly "stabilizes" as the selectivity goes up, whereas with Algorithm 2, the run time is proportional to selectivity. This is the power of using randomness in Algorithm 3, which we also mathematically analyzed in Section 4.2.

7.5 Experiment 4

We now try to evaluate the different choices of "segment" size in Algorithm 3, in the same setting as experiment 3 with the selectivity fixed to 50%. From Figure 11, we see that within the range from 5 to 16 (for parameter *s*), the performance of Algorithm 3 is relatively insensitive to the segment size and is in the optimal range. This matches our analysis in Section 4.2 (where we computed that the optimal segment size to be 7).

7.6 Some Comments

Homomorphic encryption is still a very promising area in cryptography. As cryptography advances, we expect to see more advanced homomorphic schemes that are not only provably secure, but also are faster than today's schemes.

The speed of our algorithms crucially depends on the efficiency of the underlying implementation of the big number arithmetic library. In our case, we use the GMP library (edition 4.2), which is generally believed to be fast. However, there is one potential optimization on a large number of modular multiplications by using the Montgomery algorithm [16], which is currently not done in GMP. As GMP improves, we expect the performance of our algorithms will improve accordingly.

Our encryption techniques are unique in their ability to compute SUM and AVG at the server without needing to decrypt the ciphertext. Without this ability, in the secure system model, we would have to compute the SUM and AVG at the Key Holder, which is in general infeasible due to the communication cost and the computing resource constraints at the Key Holder. In order to make this technique viable in practice, we need only show that the performance of our algorithms is competitive with previous approaches. The experiments clearly indicate that this is the case.

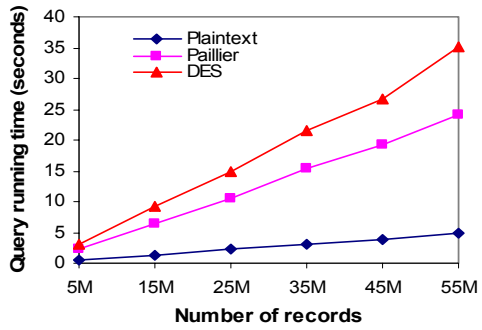

Figure 9: Performance of Alg. 2 and comparisons.

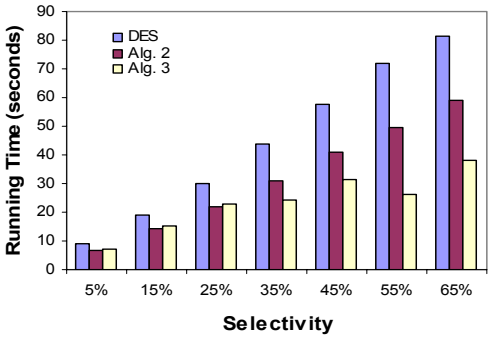

Figure 10: Algorithms under different selectivities.

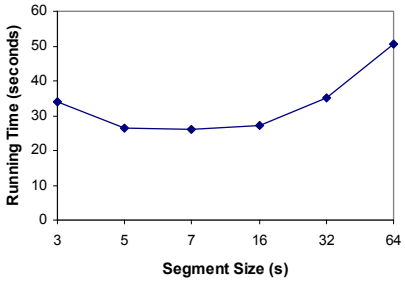

Figure 11: Alg. 3 and different segment sizes.

8. CONCLUSIONS

In this paper, we first discuss the choices of deploying encryption in a consolidated environment of applications and databases. We then point out a secure system model compliant to the acknowledged security principles, including separation of duty. In such an un-trusted server environment, we give a comprehensive study for computing SUM and AVG using a secure modern homomorphic scheme that operates in big blocks. Combining this with other schemes that handle comparison and indexing (for other query types), we approach a nearly complete solution.

9. ACKNOWLEDGMENTS & REFERENCES

This work was supported by the NSF, under the grants IIS-0086057 and IIS-0325838, and a gift from Vertica Systems, Inc.

[1] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD* 2006.
[2] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu. Order preserving encryption for numeric data. In *SIGMOD*, 2004.
[3] A. Ceselli, E. Damiani, S. Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Modeling and assessing inference exposure in encrypted databases,. In *ACM TISSEC,* 2005.
[4] S. Chaudhuri and U. Dayal, An overview of data warehousing and OLAP technology, *ACM SIGMOD Record, 26:1*, 1997.
[5] I. Damgard, and M. Jurik: A generalisation, a simplification and some applications of Paillier's probabilistic public-key system, *PKC 2001*.
[6] DES. Data Encryption Standard. *FIPS PUB 46,* 1977.
[7] P. Gibbons and Y. Matias, New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, 1998.
[8] O. Goldreich. *Foundations of Cryptography*. Cambridge University Press, 2003.
[9] S. Goldwasser and S. Micali. Probabilistic Encryption. In *J. of Computer and System Sciences*, Vol. 28, April 1984.
[10] H. Hacigumus, B. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in database-service-provider model. *SIGMOD*, 2002.
[11] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, March 2002.
[12] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted databases. In *DASFAA*, 2004.
[13] M. Kantarcioglu and C. Clifton. Security issues in querying encrypted data. In *IFIP WG 11.3 on Data and Applications Security*. 2005.
[14] F. Li, M. Hadjieleftheriou, G. Kollios, L. Reyzin. Dynamic Authenticated Index Structures for Outsourced Databases. *SIGMOD 2006*.
[15] M. Mitzenmacher, E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge Univ. Press, 2005.
[16] P. L. Montgomery, Modular Multiplication without trial division, *Mathematics of Computation*, 44, pp. 519-521, 1985.
[17] E. Mykletun, G. Tsudik. Aggregation queries in the database-as-a-service model. *IFIP WG 11.3 on Data and Application Security* 2006.
[18] P. O'Neil and D. Quass. Improved Query Performance with Variant Indexes, In *Proceedings of SIGMOD*, 1997.
[19] Oracle Corporation. *Database Encryption in Oracle 8i*, August 2000.
[20] Oracle Corp. *Oracle Database 10g Release 2, Database Vault*. 2006.
[21] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT'99,* 1999.
[22] R. Rivest, L. Adleman, and M. Dertouzous. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978.
[23] M. Roth, S. Van Horn: Database Compression. In *SIGMOD,* 1993.
[24] D. X. Song, D. Wagner, A. Perrig. Practical techniques for searches on encrypted data. In *IEEE Symp. on Security and Privacy*, 2000.
[25] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, S. Zdonik. C-Store: A Column Oriented DBMS. In *VLDB 2005*.
[26] J. Widom, Research problems in data warehousing, In *CIKM*, 1995.
[27] http://db.csail.mit.edu/projects/cstore/.
[28] http://www.swox.com/gmp/.
[29] http://www.whitehouse.gov/pcipb/cyberstrategy-draft.pdf.
[30] http://www.vormetric.com/documents/FINALPart2Database EncryptionCoreGuardvsColumnLevelWhitePaper7.pdf.
[31] http://www.datafix.com/.