Processing Analytical Queries over Encrypted Data

Stephen Tu M. Frans Kaashoek Samuel Madden Nickolai Zeldovich
MIT CSAIL

ABSTRACT

MONOMI is a system for securely executing analytical workloads over sensitive data on an untrusted database server. MONOMI works by encrypting the entire database and running queries over the encrypted data. MONOMI introduces *split client/server query execution*, which can execute arbitrarily complex queries over encrypted data, as well as several techniques that improve performance for such workloads, including *per-row precomputation*, *space-efficient encryption*, *grouped homomorphic addition*, and *pre-filtering*. Since these optimizations are good for some queries but not others, MONOMI introduces a designer for choosing an efficient physical design at the server for a given workload, and a planner to choose an efficient execution plan for a given query at runtime. A prototype of MONOMI running on top of Postgres can execute most of the queries from the TPC-H benchmark with a median overhead of only 1.24× (ranging from 1.03× to 2.33×) compared to an un-encrypted Postgres database where a compromised server would reveal all data.

1. INTRODUCTION

Outsourcing database hosting to a cloud service may offer lower costs, by sharing hardware and administrative staff, and enable elastic scaling [10]. A key problem in outsourcing the storage and processing of data is that parts of the data may be sensitive, such as business secrets, credit card numbers, or other personal information. Storing and processing sensitive data on infrastructure provided by a third party increases the risk of unauthorized disclosure if the infrastructure is compromised by an adversary (who could be an insider from the third party provider itself).

One possible solution to this problem is to encrypt data on the client machine (assumed to be trusted) before uploading it to the server, and process queries by reading back the encrypted data from the server to the client, decrypting the data, and executing the query on the client machine. However, for database queries, and *analytical* workloads in particular, this requires transferring much more data than is needed, since large fractions of a database are read by the query, but the results are typically small aggregate reports or roll-ups.

In this paper, we present the design, implementation, and evaluation of MONOMI, a system for running analytical queries over encrypted data in large databases. Rather than shipping large chunks of encrypted data back from the server, MONOMI evaluates queries on encrypted data at the database server as much as is practical, without giving the database server access to the decryption keys.

MONOMI builds on previous work in execution of database queries on encrypted databases, as described in §2, but MONOMI is the first system that can efficiently and securely execute *analytical* workloads over encrypted data. In particular, these workloads present three main challenges that make it difficult to efficiently process them over encrypted data:

First, queries over large data sets are often bottlenecked by the I/O system, such as reading data from disk or streaming through memory. As a result, encryption schemes that significantly increase the size of the data can slow down query processing. Previous work that focused on transactional workloads cannot support most analytical queries: for example, CryptDB [22] can handle only four out of 22 TPC-H queries, and incurs a median slowdown of 3.50× even for those four queries. Naïvely transferring intermediate results to the client for further processing incurs even higher overheads (as much as 55× for some TPC-H queries) due to the large amount of data.

Second, analytical queries require complex computations, which can be inefficient to perform over encrypted data. In theory, any function can be evaluated over encrypted data by using a recent cryptographic construction called fully homomorphic encryption [12]. However, this construction is prohibitive in practice [13], requiring slowdowns on the order of $10^9\times$. Instead, a practical system must leverage efficient encryption schemes, which can perform only certain computations (e.g., using order-preserving encryption for sorting and comparison [6], or using Paillier to perform addition [11]). Thus, one challenge lies in partitioning the query into parts that can be executed using the available encryption schemes on an untrusted server, and parts that must be executed on the trusted client. Some of these schemes achieve efficiency by revealing additional information to the server, such as the order of items for sorting. As we show in §8.7, few columns require encryption schemes that reveal significant information in practice, and these columns tend to be less sensitive.

Third, some of the techniques for processing queries over encrypted data can speed up certain queries but slow down others, thus requiring careful design of the physical layout and careful planning of each query's execution, for a given database and query mix. For example, Paillier encryption can be used to sum encrypted data on the server, but the cost of decrypting Paillier ciphertexts at the client can be prohibitively high in some situations, such as when computing the sum of a handful of values. As a result, it can be more efficient to decrypt and sum individual data items at the client rather than run aggregates over encrypted data at the server. Similarly, materializing additional columns can improve some queries but slow down others due to increased table sizes.

MONOMI's design addresses these challenges in three ways. First, we introduce *split client/server execution of complex queries*, which executes as much of the query as is practical over encrypted data on the server, and executes the remaining components by shipping encrypted data to a trusted client, which decrypts data and processes queries normally. Although split execution is similar in spirit to distributed query planning, the valid partitionings of query execution are quite different in MONOMI as they depend on the encrypted data available on the server. Second, we introduce a number of techniques that improve performance for certain kinds of queries (but not necessarily for all), including *per-row precomputation*, *space-efficient encryption*, *grouped homomorphic addition*, and *pre-filtering*. Third, we introduce a *designer* for optimizing the physical data layout at

Proceedings of the VLDB Endowment, Vol. 6, No. 5

the server, and a *planner* for deciding how to partition the execution of a query between the client and the server, given a representative mix of analytical queries on a database. The designer and planner are necessary because greedily applying all techniques or greedily executing all computations on the server can lead to excessive space overheads and/or inefficient query execution.

A key contribution of MONOMI lies in showing the importance of an optimizing designer and planner in choosing the best physical design and client/server query execution strategy for encrypted query processing. As more cryptographic schemes and techniques are developed by security researchers, choosing the best ones among increasingly many options will become more difficult to do by hand, and the trade-offs involved in these decisions are often non-intuitive. This is similar to the motivation for physical database designers, which guide users through the technically complex process of choosing indexes, materialized views, compression schemes, and so on.

We have implemented MONOMI's design using Postgres as the backend database server. MONOMI is designed to support any analytical SQL query, and our prototype can handle queries from the standard TPC-H and SSB [19] benchmarks. For concreteness, this paper uses queries from the TPC-H benchmark both as running examples and as the workload for evaluation. The prototype can handle most of TPC-H at scale 10, and achieves a 1.24× median slowdown (ranging from 1.03× to 2.33×) compared to TPC-H running on an unencrypted database (where a compromised server would reveal all data), even when the client and server are separated by a modest 10 Mbit/s network link. Detailed experiments with TPC-H show that MONOMI's techniques are important in achieving good performance, that MONOMI's performance is as much as 39× better than a strawman alternative based on prior work, and that MONOMI's designer and planner perform better than a greedy approach.

2. RELATED WORK

MONOMI is the first system to achieve good performance for analytical queries, such as the TPC-H benchmark, while enforcing strong confidentiality guarantees. MONOMI also introduces new techniques in order to achieve this goal, such as an algorithm for split client/server query execution, cryptographic optimizations tailored to encrypting large analytical data sets, and a designer and planner for achieving an efficient split. These techniques build on prior work, as follows.

Recent work in the cryptography community has shown that it is possible to perform arbitrary computations over encrypted data, using fully homomorphic encryption (FHE) [12]. However, the performance overheads of such constructions are prohibitively high in practice. For instance, recent work on implementing fully homomorphic encryption schemes has shown slowdowns on the order of $10^9\times$ compared to computation on plaintext data, even for relatively straightforward computations [13].

MONOMI builds on CryptDB's design of using specialized encryption schemes to perform certain kinds of computations over encrypted data, such as equality, sorting, and aggregates [22]. CryptDB can handle queries that involve only computation supported by one of these encryption schemes, but few analytical queries fall in this category (e.g., only four out of 22 TPC-H queries), and for those four, CryptDB incurs significant space (4.21×) and time (3.50×) overhead. MONOMI overcomes these limitations by partitioning query execution between a trusted client and the untrusted server, and by using a designer and a planner to choose efficient physical designs and query partitionings. MONOMI also introduces several techniques that improve performance of analytic queries, even for queries that can be completely executed on the server.

Hacıgümüş et al [14] proposed an early approach for executing SQL queries over encrypted data by performing approximate filtering at the server (using bucketing) and performing final query processing at the client, and extended it to handle aggregate queries [15, 16]. At a high level, MONOMI also splits the execution of queries between the client and the server. Hacıgümüş et al can perform only approximate filtering of rows at the server; MONOMI differs in executing many more operations completely at the server (e.g., equality checks, sorting, grouping, and joins), which cuts down on the bandwidth required to transfer intermediate results, and on the client CPU time necessary for client-side query processing. MONOMI also introduces several optimizations that speed up encrypted query processing, and MONOMI uses encryption schemes with provable security properties. Finally, MONOMI introduces a designer that optimizes the server-side physical design and a planner that optimizes query execution plans; for example, the optimal plan for executing some queries may involve sending intermediate results between the client and the server several times to execute different parts of a query. Overall, MONOMI executes 19 out of 22 TPC-H queries at scale 10 with a median slowdown of just 1.24× compared to a plaintext database, while Hacıgümüş et al handle just two out of 22 TPC-H queries at scale 0.1 and do not compare their performance to a plaintext database [15, 16].

Some systems have explored the use of data fragmentation between multiple servers that are assumed not to collude [8], or require the client to store a part of the data [9]. MONOMI does not require any assumptions about non-collusion between servers, and does not require the clients to store any data.

An alternative approach to securely outsourcing query processing to a third party is to use trusted hardware [3, 4]. However, such approaches require relatively intrusive changes on behalf of the service provider, which can be either expensive (if using high-end secure processors), or can provide little security against a determined adversary (if using off-the-shelf TPM chips).

MONOMI's designer and planner are similar to previous work on using integer programming to choose the best physical design for a given query workload [2, 21]. The primary difference is that MONOMI is optimizing for the set of encrypted columns that enable efficient operations to execute on the server rather than on the client, instead of optimizing for a set of materialized views or indexes that can be used to efficiently answer a query. The main contribution of this paper lies not in specific techniques for planning, but instead in applying existing planning techniques to the problem of choosing physical designs and query plans for encrypted query processing.

3. OVERVIEW

Analytical workloads are difficult to execute on an untrusted server with access to only encrypted data because they perform complex processing over large data sets. As mentioned in the previous section, fully homomorphic encryption can execute arbitrary queries but incurs $10^9\times$ slowdowns. CryptDB is much more efficient but can execute only four out of 22 TPC-H queries over encrypted data, and incurs significant space and time overhead for the four queries it can execute. Finally, downloading the data to the client for processing requires significant client-side bandwidth and CPU time, and is thus also undesirable.

MONOMI introduces a new approach based on *split client/server execution*. Split execution allows MONOMI to execute a part of the query on the untrusted server over encrypted data. For other parts of the query, which either cannot be computed on the server at all, or that can be more efficiently computed at the client, MONOMI downloads the intermediate results to the client and performs the final computation there.

Encryption scheme	SQL operations	Leakage
Randomized AES + CBC	None	None
Deterministic AES + CMC [17] or FFX [5]	`a = const`, `IN`, `GROUP BY`, equi-join	Duplicates
OPE [6]	`a > const`, `MAX`, `ORDER BY`	Order + partial plaintext [7]
Paillier [20]	`a + b`, `SUM(a)`	None
SEARCH [22, 24]	`a LIKE pattern`	None

Table 1: Encryption schemes used by MONOMI, example SQL operations they allow over encrypted data on the server, and information revealed by each scheme's ciphertexts in the absence of any queries.

For example, suppose an application issues the following query:

```
SELECT SUM(price) AS total
FROM orders
GROUP BY order_id
HAVING total > 100
```

To execute portions of this query on an untrusted server, MONOMI follows CryptDB's approach of using specialized encryption schemes, as shown in Table 1. MONOMI executes most of the example query on the server, as follows. MONOMI encrypts `order_id` with deterministic encryption, which allows the server to find rows that have the same `order_id`, and thus perform the `GROUP BY`. MONOMI encrypts `price` with Paillier, which allows the server to compute the encrypted sum of encrypted values, without having access to the decryption key. However, MONOMI cannot check if `total > 100` at the server, because `total` is encrypted with Paillier, and Paillier does not support order comparison. Thus, MONOMI executes the following query at the server:

```
SELECT PAILLIER_SUM(price_paillier) AS total
FROM orders
GROUP BY order_id_det
```

where `PAILLIER_SUM()` is a MONOMI-supplied UDF that computes the encrypted sum of several Paillier ciphertexts, and `price_paillier` and `order_id_det` are the Paillier and deterministically encrypted columns of the original `price` and `order_id` columns, respectively. Once MONOMI's client library receives the results, it decrypts them, and executes the `HAVING total > 100` clause. Any matching results are sent to the application. §4 describes how MONOMI performs this transformation for arbitrary queries.

Optimization. One key factor in MONOMI's performance lies in the cost of executing computation over encrypted data on the server. MONOMI introduces several optimizations to speed up encrypted data processing for analytical workloads, as we describe in §5.

Another key factor is that the split executions that MONOMI can perform depend on the encryption schemes available at the server, and some splits perform better than others. For example, if a Paillier encryption of `price` was not available in the above example, MONOMI would have to download the entire `price` column to the client. On the other hand, if no query asked for `SUM(price)`, storing a Paillier encryption of `price` wastes storage space. To choose a set of encryption schemes that maximizes query performance, MONOMI uses an optimizing designer. This designer is similar to physical designers used by other databases, and takes as input from the user the kinds of features that are likely to appear in future queries, such as `SUM(price)` in our example. §8.5 evaluates the kinds of query characteristics that the designer needs know in order to achieve good performance. Given a particular physical design, MONOMI uses a planner to choose the best split client/server execution plan, when a new query is issued by the application, as explained in §6.

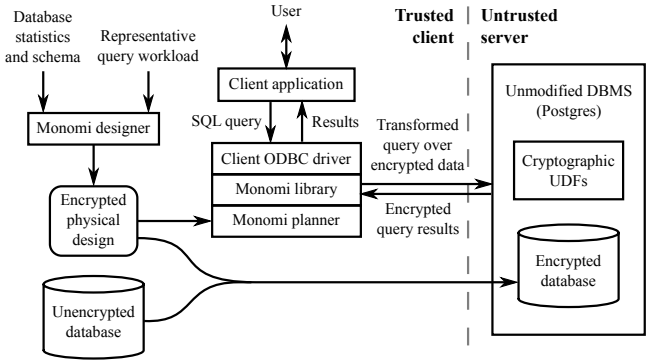

Figure 1: Overall architecture of our MONOMI prototype.

Security. Since MONOMI's design builds on CryptDB, it inherits similar security properties [22]. Although the untrusted server stores only encrypted data, it can still learn information about the underlying plaintext data in three ways, as summarized in Table 1. First, some encryption schemes reveal information necessary to process queries (e.g., deterministic encryption reveals duplicates in order to perform equality checks). Second, some encryption schemes may leak more information than necessary. For example, the order-preserving scheme we use leaks partial information about the plaintext. Newer encryption schemes may be able to reduce this leakage; for instance, a recent order-preserving scheme [23] leaks no unnecessary information beyond the order of elements. Third, the server learns which rows match each predicate computed on the server, such as the rows matching `column LIKE '%keyword%'`.

By combining the above sources of information, an adversarial server may be able to obtain additional information about the rows or the queries using statistical inference techniques. For example, if a column encrypted with a deterministic scheme contains few distinct values, and the server has apriori knowledge of the distribution of these values, the server can infer plaintext values. A server could also combine information about which rows match a single keyword search query with apriori knowledge about keyword distribution to guess the plaintext keywords [18].

One strict guarantee that MONOMI provides is that it never stores plaintext data on the server, and employs only encryption schemes necessary for the application's workload. MONOMI allows the administrator to further restrict the encryption schemes used for especially sensitive columns: for example, requiring order-preserving encryption (MONOMI's weakest scheme) not be used for columns storing credit card or social security numbers.

Using MONOMI. Our MONOMI prototype consists of three major components, as shown in Figure 1. First, during system setup, MONOMI's *designer* runs on a trusted client machine and determines an efficient physical design for an untrusted server. To determine important characteristics of the workload for achieving good performance, our designer takes as input a representative subset of the queries and statistics on the data supplied by the user, such as the TPC-H workload. As we show in §8, MONOMI's designer achieves good performance with only a small subset of queries, as long as they contain the key features of the workload. Users are not required to use the designer, and may instead manually input an encryption strategy, or modify the strategy produced by the designer.

Second, during normal operation, applications issue unmodified SQL queries using the MONOMI *ODBC library*, which is the only component that has access to the decryption keys. The ODBC library uses the *planner* to determine the best split client/server execution plan for the application's query.

Third, given an execution plan, the library issues one or more queries to the *encrypted database*, which does not have access to the decryption keys and can execute operations only over encrypted data. The database runs unmodified DBMS software, such as Postgres, with several user-defined functions (UDFs) provided by MONOMI that implement operations on encrypted data. MONOMI conservatively encrypts all data stored in the database, although in practice non-sensitive data could be stored as plaintext for efficiency. Once the client library receives intermediate results from the database, decrypts them, and executes any remaining operations that could not be efficiently performed at the server, the results are sent to the application, as if executed on a standard SQL database.

4. SPLIT CLIENT/SERVER EXECUTION

In order to execute queries that cannot be computed on the server alone, MONOMI partitions the execution of each query across the untrusted server, which has access only to encrypted data, and a trusted client machine, which has access to the decryption keys necessary to decrypt the encrypted data.

Consider TPC-H query 11, as shown in Figure 2. This query requires checking whether a `SUM()` for each group is greater than a sub-select expression that computes its own `SUM()` and multiplies it by a constant. MONOMI's encryption schemes cannot support such queries directly over encrypted data, as described in the previous section, because addition and comparison involve incompatible encryption schemes, and because no efficient encryption scheme allows multiplication of two encrypted values.

```
SELECT   ps_partkey,
         SUM(ps_supplycost * ps_availqty) AS value
FROM     partsupp JOIN supplier JOIN nation
WHERE    n_name = :1
GROUP BY ps_partkey
HAVING   SUM(ps_supplycost * ps_availqty) > (
  SELECT SUM(ps_supplycost * ps_availqty) * 0.0001
  FROM   partsupp JOIN supplier JOIN nation
  WHERE  n_name = :1 )
ORDER BY value DESC;
```

Figure 2: TPC-H query 11, with join clauses omitted for brevity.

To answer such queries, MONOMI partitions the execution of the query between the client and the server, by constructing a SQL operator tree for the query, consisting of regular SQL operators and decryption operators that execute on the client, as well as `RemoteSQL` operators that execute a SQL statement over encrypted data on the untrusted server. For example, Figure 3 shows a potential split query plan for executing TPC-H query 11.

The general algorithm for computing a split query plan is presented in Algorithm 1. This algorithm can handle arbitrary SQL queries, but we explain its operation using TPC-H query 11 as an example, as follows.

Lines 6–13 try to find a way to execute the `WHERE n_name=:1` clause on the server. The REWRITESERVER$(expr, E, enctype)$ function returns an expression for computing the value of *expr* at the server, encrypted with *enctype*, given a set of encrypted columns E. For *enctype*=PLAIN, REWRITESERVER produces an expression which generates the plaintext value of *expr*. In our example, this translates `n_name=:1` to `n_name_DET=`*`encrypt(:1)`*, which produces the same (plaintext) boolean value without revealing `n_name`.

Lines 14–18 try to move the `GROUP BY` clause to the server, by using REWRITESERVER to find a deterministic encryption of the group keys (in our example, `ps_partkey`) by passing in a DET *enctype*.

input : Q, an abstract syntax tree (AST) for the query.
E, a description of the available encrypted columns.
output : P, a plan node for query, such as the one in Figure 3.

```
1  for s in subqueries in Q.relations do
2      p ← GENERATEQUERYPLAN(Q, E)
3      Replace s with p in Q.relations
4  RemoteQ ← Q                          // Query to run over encrypted data
5  LocalFilters ← []
6  for c in Q.join_clauses || [Q.where_clause] do
7      c' ← REWRITESERVER(c, E, PLAIN)
8      if c' ≠ Nil then                 // c computable on server
9          Replace c with c' in RemoteQ
10     else
11         Remove c from RemoteQ
12         Add c to LocalFilters
13         Add EXPRS(c) to RemoteQ.projections
14 RemoteQ.group_by.keys ← []
15 for k in Q.group_by.keys do
16     k' ← REWRITESERVER(k, E, DET)
17     if k' ≠ Nil then
18         Add k' to RemoteQ.group_by.keys
19 LocalGroupBy ← Nil
20 LocalHaving ← Nil
21 if RemoteQ.group_by.keys.size = Q.group_by.keys.size then
                                        // GROUP BY pushed onto server
22     f' ← REWRITESERVER(Q.group_by.having, E, PLAIN)
23     RemoteQ.group_by.having ← f'
24     if f' = Nil and Q.group_by.having ≠ Nil then
25         LocalHaving ← Q.group_by.having
26         Add EXPRS(LocalHaving) to RemoteQ.projections
27 else                                 // Compute GROUP BY on client
28     RemoteQ.group_by ← Nil
29     LocalGroupBy ← Q.group_by
30     for k in LocalGroupBy.keys || [LocalGroupBy.having] do
31         Add EXPRS(k) to RemoteQ.projections
32 for p in Q.projections do
33     p' ← REWRITESERVER(p, E, ANY)
34     if p' = Nil then
35         Replace p with EXPRS(p) in RemoteQ.projections
36     else
37         Replace p with p' in RemoteQ.projections
38 P ← LOCALDECRYPT(REMOTESQL(RemoteQ))
39 for f in LocalFilters do
40     P ← LOCALFILTER(P, f)
41 if LocalHaving ≠ Nil then
                                        // Client-side computation of just HAVING
42     P ← LOCALGROUPFILTER(P, LocalHaving)
43 else if LocalGroupBy ≠ Nil then
                                        // Client-side GROUP BY and maybe HAVING
44     P ← LOCALGROUPBY(P, LocalGroupBy)
45 return P
```

Algorithm 1: Pseudo-code for GENERATEQUERYPLAN. For brevity, the pseudo-code assumes that the query is a `SELECT` statement and does not have `ORDER BY` and `LIMIT` components. The pseudo-code also does not keep track of the positions of projections necessary to reconstruct expressions on the client.

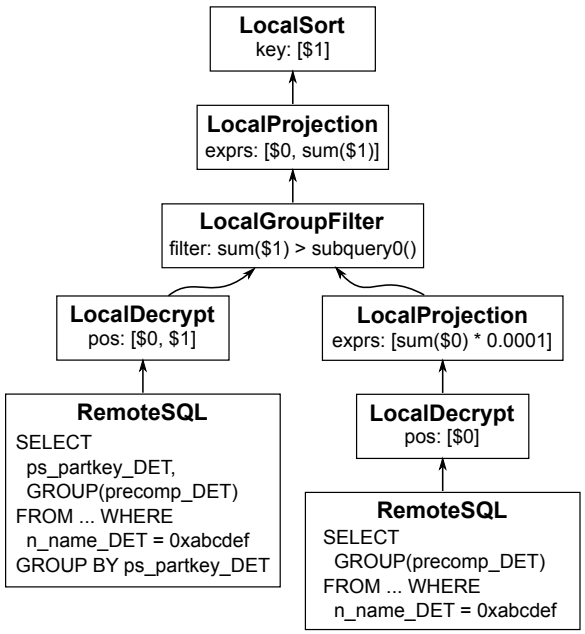

Figure 3: Example split query plan for TPC-H query 11. "Local" operators run on the trusted client, and "Remote" operators run on the untrusted server. `GROUP()` denotes the concatenation of all values from each `GROUP BY` group, `0xabcdef` denotes the deterministic encryption of the argument `:1`, and `precomp_DET` denotes the deterministic encryption of the precomputed expression `ps_supplycost * ps_availqty`. `$n` refers to the *n*-th column of the child operator.

For our example, REWRITESERVER returns `ps_partkey_DET`, which is placed into *RemoteQ*, the query that will run on the server.

Lines 22–26 try to push the `HAVING` clause to the server, assuming that the `GROUP BY` can be performed on the server. Since in our example the `HAVING` clause cannot be computed at the server, REWRITESERVER returns **Nil**. To execute the `HAVING` clause on the client, line 26 uses the EXPRS(*expr*) helper function to determine what sub-expressions can be computed on the server in order to then compute the entire *expr* on the client, and adds those expressions to the list of projections fetched by *RemoteQ*.

Since the `HAVING` clause involves a sub-select, EXPRS recursively calls back to GENERATEQUERYPLAN, which determines how to run the sub-select on the server. This eventually returns the `RemoteSQL`, `LocalDecrypt`, and `LocalProjection` operators shown in the lower right branch of Figure 3.

Lines 32–37 determine how to best fetch the projections from the server, passing the ANY *enctype* to REWRITESERVER since any encryption scheme would suffice. In our example, this translates `ps_partkey` to `ps_partkey_DET`, and `SUM(..)` into `GROUP(precomp_DET)`. Here, we assume the server has a precomputed deterministic encryption of `ps_supplycost*ps_availqty`, denoted by `precomp_DET`, but does not have its homomorphic encryption (as specified in the *E* argument). Thus, the `GROUP()` operator concatenates all values from the group, and the `SUM()` will be computed at the client. The planner, described in §6, may choose this set of encryption schemes *E* if computing the `SUM()` at the client is faster than decrypting a homomorphic ciphertext.

Finally, the algorithm constructs the local part of the query plan. Line 38 constructs the `RemoteSQL` operator coupled with a `LocalDecrypt` operator, as shown in the two branches in Figure 3. Lines 39–40 apply any remaining `WHERE` or `JOIN` clauses. Lines 41–44 add operators for client-side `GROUP BY` or `HAVING` operations.

5. OPTIMIZATION TECHNIQUES

To speed up query execution over encrypted data, MONOMI introduces several new techniques, described in the rest of this section. We use example queries from the TPC-H benchmark to illustrate their benefits, although these techniques can be applied to arbitrary queries. MONOMI's designer and planner automatically use these techniques as appropriate to achieve good performance, without requiring the application developer to perform manual query rewriting.

5.1 Per-row precomputation

Although split client/server execution allows MONOMI to compute the answer for any query, executing an operator on the client can require downloading a large amount of intermediate data. For example, consider TPC-H query 11 again, as shown in Figure 2. This query requires computing `SUM(ps_supplycost*ps_availqty)`, but Paillier encryption cannot perform multiplication of two values. As another example, TPC-H queries 8 and 9 require grouping aggregates by `EXTRACT(YEAR FROM o_orderdate)`, which cannot be computed from encryptions of `o_orderdate` without making the encryption schemes less secure and less space-efficient.

To optimize the execution of such queries, MONOMI employs per-row precomputation, in which the designer materializes an additional column in a table, containing the encrypted value of an expression that depends only on other columns in the same row. These materialized expressions can then be used at query runtime. For instance, in query 11, MONOMI can materialize an additional column storing the homomorphic encryption of `ps_supplycost*ps_availqty`, which allows the server to compute the entire `SUM()`. This allows the client to download one encrypted aggregate value, instead of downloading all `ps_supplycost` and `ps_availqty` values.

To determine which expressions should be materialized on the server, MONOMI's designer considers operators that cannot be performed over encryptions of individual columns alone. It considers only operators that involve columns from the same table, and does not consider aggregates, since they span multiple rows. For such operators, it finds the simplest expression (i.e., lowest in the operator tree) that, if precomputed in a separate column, would allow the operation to execute on the server. We describe how MONOMI chooses which candidate expressions to actually materialize in §6.

5.2 Space-efficient encryption

Database queries that involve table scans—which are common in analytic workloads—are often bottlenecked by I/O, such as the rate at which data can read from disk. As a result, the performance of such queries under MONOMI is sensitive to the size of the encrypted on-disk data. To optimize the performance of such queries, MONOMI uses encryption schemes that minimize *ciphertext expansion*, which is the length increase of ciphertext values compared to the length of the original plaintext values, while still preserving security.

For example, consider the encrypted column storing the precomputed `EXTRACT(YEAR FROM o_orderdate)` expression from TPC-H queries 8 and 9, as mentioned above. To allow grouping by the extracted year, the value must be encrypted deterministically, but using the standard AES or Blowfish algorithm would produce a 128- or 64-bit ciphertext even for 8-, 16-, or 32-bit integers.

To minimize ciphertext expansion for deterministic encryption of such small data types, MONOMI uses the FFX block cipher mode of operation [5], which encrypts *n*-bit plaintexts to *n*-bit ciphertexts, as long as *n* is less than the block cipher width (e.g., 128 bits for AES). For values that are longer than the block cipher width, the ciphertext stealing (CTS) mode of block cipher operation, combined with CMC mode [17], provides a suitable alternative. These techniques

reduce the size of the `lineitem` table from the TPC-H benchmark by $\sim 30\%$ when using only deterministic encryption.

Most homomorphic encryption schemes, including Paillier as used by MONOMI, operate over large plaintext and ciphertext values, such as 1,024 and 2,048 bits respectively for Paillier. Such large plaintext and ciphertext sizes are necessary to achieve an adequate level of security. To make efficient use of these 1,024-bit plaintext payloads, MONOMI uses techniques proposed by Ge and Zdonik [11] to pack multiple integer values into a single 1,024-bit Paillier plaintext. MONOMI both packs values from multiple columns in a single row, and packs values from multiple rows, into a single Paillier plaintext. This packing scheme allows us to reduce the per-row space overhead of Paillier ciphertexts for a single 64-bit column by $\sim 90\%$. We also use techniques described by Ge and Zdonik to compute aggregates over these packed ciphertexts, along with an additional optimization that we describe next.

5.3 Grouped homomorphic addition

To compute `SUM()` or `AVG()` aggregates over encrypted values, MONOMI uses the Paillier homomorphic encryption scheme. In Paillier, computing an encrypted version of the sum of two values, $E(a+b)$, requires multiplying the encryptions of the two values modulo a 2,048-bit public key: $E(a+b) = E(a) \times E(b)$. These modular multiplications can become computationally expensive, especially for queries that require aggregating a large number of rows, such as TPC-H query 1:

```
SELECT SUM(l_quantity), SUM(l_extendedprice), ..
FROM lineitem WHERE .. GROUP BY ...
```

Such queries require one modular multiplication per row per `SUM()` operator in the naïve case. The complete TPC-H query 1 computes seven distinct aggregates for each matching row from the `lineitem` table, and the `lineitem` table contains a large number of rows.

To avoid performing separate modular multiplications per row for each column being aggregated, MONOMI implements a *grouped homomorphic addition* optimization, whereby columns which are aggregated together are packed into Paillier plaintexts in a particular fashion that allows aggregates for all columns to be computed simultaneously with a single modular multiplication. MONOMI's designer considers only layouts which pack all columns being aggregated for a particular query into a single group.

Consider a query that asks for the `SUM()` aggregates of k columns from one table, denoted by a_1 through a_k for some row a. Let the Paillier plaintext for row a be the concatenation of the values from each of these columns $(a_1 \| a_2 \| \ldots \| a_k)$, and store the Paillier-encrypted version of this concatenation on the server for that row. With this physical layout at the server, the `SUM()` for all of the k columns can be derived by multiplying together the Paillier ciphertext from each row in the result set, which can be computed with one modular multiplication per row. This works because, arithmetically, $(a_1 \| \ldots \| a_k) + (b_1 \| \ldots \| b_k) = (a_1 + b_1) \| \ldots \| (a_k + b_k)$, and thus $E(a_1 \| \ldots \| a_k) \times E(b_1 \| \ldots \| b_k) = E((a_1 + a_2) \| \ldots \| (a_k + b_k))$. Given the product of all row ciphertexts, the client can extract the aggregate of column i from position i in the decrypted plaintext.

One potential problem is that the sum of values from one column, $a_i + b_i + \ldots$, can overflow the width of that column's type and spill over into the next column. To avoid this problem we add zero padding between columns in the concatenation, where the number of zeroes is log base 2 of the maximum number of rows in the table (which we assume is $\sim 2^{27}$ in our experiments).

To reduce ciphertext expansion, MONOMI combines grouped homomorphic addition with the packing scheme from §5.2. Values from multiple rows are packed together into a single 1,024-bit plaintext, and each row's value is the concatenation of all aggregatable columns from that row, as described above. For simplicity, we do not split a single row across multiple 1,024-bit plaintexts. While this slightly increases ciphertext expansion (by not using some of the bits in the 1,024-bit plaintext), it keeps the number of modular multiplications necessary for aggregation low, because a given column always appears at the same offsets in every plaintext.

5.4 Conservative pre-filtering

Analytic workloads often filter large amounts of data on the server to return a small amount of data to the client. To achieve efficiency for such queries over encrypted data, MONOMI must avoid sending all of the data to the client, and instead apply filtering to encrypted data on the server. However, some filter predicates cannot be efficiently computed over encrypted data. For example, consider the following sub-select expression from TPC-H query 18:

```
SELECT l_orderkey FROM lineitem
GROUP BY l_orderkey
HAVING SUM(l_quantity) > :1
```

Downloading all `lineitem` groups to the client is time-consuming, but the Paillier scheme necessary to compute the `SUM()` expression is incompatible with the order-preserving encryption necessary to check if `SUM(l_quantity) > :1` on the server.

To achieve reasonable performance for filtering predicates that cannot be computed on the server, MONOMI generates a conservative estimate of the predicate, to pre-filter intermediate results on the server, and applies the exact predicate on the client. For TPC-H query 18, MONOMI executes the following query on the server:

```
SELECT l_orderkey_det,
       PAILLIER_SUM(l_quantity_paillier)
FROM lineitem
GROUP BY l_orderkey_det
HAVING MAX(l_quantity_ope) > encrypt_ope(m)
       OR COUNT(*) > (:1 / m)
```

where `m` is an arbitrary positive integer. The `HAVING` clause executed on the server computes the superset of rows that would have matched the original `HAVING` clause: if a group could have matched the original clause, then either it had at least one value larger than `m`, or it had at least `(:1/m)` rows in the group, each of which was `m` or less. Once the server performs this filtering and sends the intermediate results to the client, the client applies the exact filter to the decrypted results, and returns only the matching groups to the application. MONOMI estimates of the maximum value of a column (e.g., `l_quantity`) during setup and uses that as `m`.

6. DESIGNER AND PLANNER

The optimizations described in §5 do not always apply to every query. In particular, as we show in §8, a greedy application of the techniques or a greedy execution of all parts of the query on the server does not necessarily yield the best performance and can waste space. In this section, we describe the physical database designer we developed that selects the best set of expressions to precompute, the best set of encryption schemes to create for each column (regular and precomputed), and which columns to include in homomorphic ciphertexts, as described in §5.1–§5.3. The MONOMI designer uses standard techniques for optimization, but applies them in the context of encrypted query processing. To evaluate the performance implications of different physical designs, the designer uses a query planner to choose the best query execution strategy given a physical

design. This planner is also used at runtime to choose a query plan for a new query from the application.

6.1 Input and output

The goal of MONOMI's designer is to decide how to encrypt the data (*physical design*). MONOMI's planner, on the other hand, determines how to best execute queries given a particular physical design. We envision the use case of the designer to be similar to automated index selection and materialized view selection tools.

MONOMI's designer is invoked during the setup phase, when the user is preparing to load his database into MONOMI. The user provides the designer with a query workload $Q_1, Q_2, ..., Q_n$. The query workload does not need to exhaustively enumerate every possible query the user will run, but it should be representative of the operations that the user is expecting to perform over the data. The user also provides a sample of the data that will be loaded into the database, which is used for estimating statistics about the data, and need not be the exact data that will be eventually loaded on the server. The user can also specify a space constraint factor S, which controls how much space MONOMI's designer can consume.

Given these inputs, the designer returns to the user a physical design for the server, which is a set of (encrypted) columns to materialize for each table, including pre-computed and Paillier columns.

6.2 Algorithm without constraints

MONOMI's designer works by initially considering each query in isolation, performing the following steps for each query Q_i:

1. The designer considers all of the operations in Q_i, including all of the expressions in the `WHERE` and `HAVING` clauses, any `ORDER BY` and `GROUP BY` clauses, etc. For each operation, the designer determines what expression and encryption scheme would allow that operation to execute on the server. The expression might be a column, or it may be an expression that could be precomputed per row. The encryption scheme depends on the operation being performed (such as deterministic encryption, Paillier encryption, or order-preserving encryption). The set of these $\langle value, scheme \rangle$ pairs for Q_i is called $EncSet_i$, and we define $\mathscr{E}$ to be the set of all possible such pairs; that is, $EncSet_i \subseteq \mathscr{E}$.
   For example, a `WHERE x = :1` clause generates a $\langle \texttt{x}, DET \rangle$ pair, referring to the `x` column, and an `ORDER BY x+y` clause generates a $\langle \texttt{x+y}, OPE \rangle$ pair, referring to a precomputed `x+y` value.
2. The designer invokes the planner to determine how to best execute Q_i given the encryption schemes in $EncSet_i$. The planner does so by computing $PowSet_i$, the power set that contains all subsets of $EncSet_i$. It then constructs an execution plan for Q_i for each element of the power set (denoted $PowSet_i[j]$), where the execution plan describes what parts of the query would be executed on the server, and what parts would be executed on the client, using Algorithm 1. An example execution plan is illustrated in Figure 3.
3. For each of the execution plans above, the planner uses a cost model to estimate how much time it would take to execute that plan. The planner chooses the fastest execution plan for Q_i, and we denote the corresponding subset of encryption types used by that plan as $BestSet_i$; that is, $BestSet_i \in PowSet_i$.

Once the designer uses the planner to compute $BestSet_i$ for each query, the designer takes the union of the encryption schemes required by each query and uses that as the chosen physical design.

6.3 EncSet pruning heuristics

The algorithm described above performs exhaustive enumeration of all possible subsets of encryption schemes (i.e., all of $PowSet_i$). Although this will find the optimal plan, the number of distinct subsets that have to be considered can grow quite large. For example, consider a query that has a `WHERE col1 > :1 OR col2 > :2` clause. The naïve algorithm above would separately consider storing an OPE encryption of `col1` but not `col2`, of `col2` and not `col1`, and finally of both. However, if any part of the clause cannot be evaluated on the server, the client must still fetch the entire table from the server and apply the `WHERE` clause locally.

To avoid considering such unnecessary subsets, MONOMI applies a simple pruning heuristic. In particular, MONOMI considers only subsets of encryption schemes where either all of the encryptions necessary for a given unit are present, or all of them are absent. A query unit corresponds to a `WHERE` or `HAVING` clause, a `GROUP BY` clause, etc. We make a special case for `WHERE` clauses however, where we treat each top level conjunction as a separate unit.

6.4 Cost model

We model the cost to execute a given query plan as the sum of three main components: execution time on the server, transfer time of data over the network link, and post-processing time on the client.

MONOMI estimates the execution time on the server by asking the Postgres query optimizer for cost estimates to run the one or more server queries. The network transfer time is computed based on a user-provided estimate of the network bandwidth.

To estimate the post-processing time on the client, we focus on decryption cost. We query the Postgres database to obtain estimates of the cardinality and row-length of the result set, and then estimate decryption time based on the encryption type used. Since decryption costs are comparable across different modern systems, we estimate the per-byte decryption cost of each scheme (random, homomorphic, order preserving, and deterministic encryption) by running a profiler that decrypts a small amount of data when MONOMI is first launched.

6.5 With constraints

The designer algorithm in §6.2 chooses the lowest-cost plan without considering any constraints. For example, it does not take into account the server space overhead of the chosen plan, which can be high because the encryption scheme chosen by the designer requires a lot of space, or because the designer chose to precompute several additional columns. This subsection present an Integer Linear Program (ILP) formulation that chooses the lowest-cost plan subject to constraints, such as a user-supplied server space budget S. Client CPU and bandwidth use is not considered here, since it is already included in the cost of query execution.

As before, the designer computes $EncSet_i$ for each Q_i, and $PowSet_i$ (all possible subsets of $EncSet_i$). The ILP formulation minimizes the total cost c over all of the queries, as a function of the chosen plan for each query, represented by the bit matrix $x_{i,j}$:

$$c(x) = \sum_{i=1}^{n} \sum_{j=1}^{||PowSet_i||} cost(i,j)\, x_{i,j}$$

where n is the number of queries, $x_{i,j}$ is 1 if $PowSet_i[j]$ is chosen for Q_i (and 0 otherwise), and $cost(i,j)$ is MONOMI's estimate for the cost of executing Q_i using the j-th candidate scheme of $PowSet_i$. Here, the designer assumes that all queries are weighted equally, although it would be straightforward to support variable weights.

The designer minimizes this overall cost function under two constraints. First, to enforce that each query has one plan chosen, we require that exactly one element of $PowSet_i$ is chosen for each Q_i:

$$\forall i, \quad \sum_{j=1}^{||PowSet_i||} x_{i,j} = 1$$

Note that ≥ 1 would have worked here also, since the minimum cost solution will necessarily involve picking only a single candidate query plan (query costs are all positive numbers).

Second, bounding the amount of space used on the server requires another constraint. A naïve formulation would be as follows:

$$\sum_{i=1}^{n} \sum_{j=1}^{||PowSet_i||} \sum_{k \in PowSet_i[j]} encsize(k) \cdot x_{i,j} \leq S \times plainsize$$

where $encsize(k)$ is the size in bytes of the encrypted column k ($k \in \mathscr{E}$), $plainsize$ is the absolute size of the plaintext database, and $S \geq 1$ is supplied as a space overhead factor by the user. Specifying $S = 1$ is equal to saying only deterministic encryption is allowed.

The above space constraint is too simplistic, because it double-counts the space of encryption schemes required for multiple queries. To avoid this problem, we extend the formulation with a new binary decision variable that tracks whether each of the possible encryption schemes is used: $e_k \in \{0,1\}$ where $k \in \mathscr{E}$. We now replace the above space constraint with two constraints. The first of these constraints limits the total space used by all of the encryption schemes:

$$\sum_{k \in \mathscr{E}} e_k \cdot encsize(k) \leq S \times plainsize$$

The second constraint now must ensure that the encryption schemes needed for each query are reflected in e_k:

$$\forall i,\ \forall j,\ ||PowSet_i[j]|| \cdot x_{i,j} - \sum_{k \in PowSet_i[j]} e_k \leq 0$$

If Q_i does not use plan j, then $x_{i,j} = 0$ and the above constraint is always true. If Q_i does use plan j, then the above constraint is satisfied only if all of the encryption schemes required by that query (i.e., $PowSet_i[j]$) are enabled in e_k. Note that the ILP is always feasible as long as $S \geq 1$, because using only deterministic encryption for every column is always a valid (albeit not very interesting) solution.

Solving this ILP identifies a set of encrypted columns that ensures each query has a viable plan and that the minimum cost set of plans that meets the space budget is selected. The end result is a physical design, as well as a best execution plan for each query. For TPC-H, our formulation used 713 variables and 612 constraints.

Our ILP formulation is an approximation for row-stores, and is more accurate for column-stores. The ILP leaves out cross-query interactions which affect row-stores: if one plan requires an additional encrypted column, other plans that perform sequential scans over that table but do not use the new column are slowed down. Column-stores avoid this problem. It is possible to account for this non-linear interaction in row-stores using a *mixed-integer-quadratic-program* (MIQP)[1]; we do not use this formulation because there are no robust open-source MIQP solvers and the added benefit is minimal.

7. IMPLEMENTATION

MONOMI is implemented according to the design shown in Figure 1, and uses the Postgres database as the backend. The designer and planner are implemented in $\sim 8,000$ lines of Scala. The client library and the server-side UDFs are implemented in $\sim 4,000$ lines of C++. We use OpenSSL for cryptography and NTL for infinite-precision numerical arithmetic. The client library assumes that all intermediate results sent from the server to the client will fit in memory, which is true for all of the TPC-H queries we considered.

In the encrypted Postgres database, each table in the original schema maps to a single table in the encrypted schema. This encrypted table contains one or more copies of every column in the original table, based on the number of encryption schemes chosen for that table during setup. We chose this representation rather than placing each encrypted column into a separate table because we found that the cost of additional joins to reconstruct tuples at query time outweighed the benefits of narrower tables (others report similar results for emulating column-stores in row-oriented databases [1]).

The packed Paillier ciphertexts for each group are kept in separate files on the local file system. We do this because efficiently mapping a value in row i to rows $i, i+1, ..., i+n$ in the same table is an unnatural fit for the relational model. In order for each row in a table to access its corresponding Paillier ciphertext (if one exists), we add an additional `row_id` column for each table requiring it. We then supply this `row_id` to the homomorphic aggregate UDF, which can then compute an offset into the ciphertext file and read the Paillier ciphertext from there. We currently do not support transactionally inserting into both a table and its corresponding ciphertext file(s).

All columns in MONOMI are encrypted with at most deterministic encryption. Randomized encryption would be simple to support, requiring an extra 64-bit initialization vector (IV) in each row.

MONOMI is designed to support arbitrary SQL queries, but our implementation has a few limitations that prevent it from handling some constructs, including three out of the 22 TPC-H queries. We do not support views, which prevents us from running TPC-H query 15. We also do not support text pattern matching with two or more patterns, such as `c LIKE '%foo%bar%'`, although we support single patterns, such as `c LIKE '%foo%'`. This prevents us from running TPC-H queries 13 and 16. However, there is nothing fundamental in our design that prohibits implementing these features.

8. EVALUATION

To evaluate MONOMI, we answer the following questions, in the respective subsections:

- Can MONOMI efficiently execute an analytical workload over encrypted data on an untrusted server? (§8.2)
- How much do MONOMI's optimization techniques, designer, and planner matter in achieving good performance? (§8.3)
- What are the overheads for MONOMI in terms of space and client-side CPU time? (§8.4)
- What workload features are important for MONOMI's designer, and how sensitive is it to having only a subset of the queries during setup? (§8.5)
- Is MONOMI's ILP formulation necessary for achieving good performance under a space constraint? (§8.6)
- What security level does MONOMI provide? (§8.7)

8.1 Experimental setup

To answer these questions, we use MONOMI to run the TPC-H 2.14 benchmark over a TPC-H scale 10 data set. We evaluated 19 of the 22 queries in the benchmark: as described in §7, MONOMI cannot execute queries 13, 15, and 16. Each reported query runtime is the median of three runs of that query. The setup phase (i.e., running MONOMI's ILP designer) took 52 seconds for this TPC-H workload.

The experimental setup consisted of two machines: a client running the MONOMI client library, and a server running unmodified Postgres 8.4. The client machine has four 4-core 2.2 GHz Intel Xeon E5520 processors, and the server machine has four 4-core 2.4 GHz Intel Xeon E5530 processors. Both machines have 24 GB of RAM. The server has six 7,200 RPM disks setup in a RAID 5 configuration. The client and server run Linux 2.6.38 and 2.6.35 respectively.

To ensure that queries access disk (as is typically the case for large analytical workloads), we limit the amount of memory available to

[1] In fact, our ILP formulation is a direct result from setting the quadratic term in the MIQP formulation to zero.

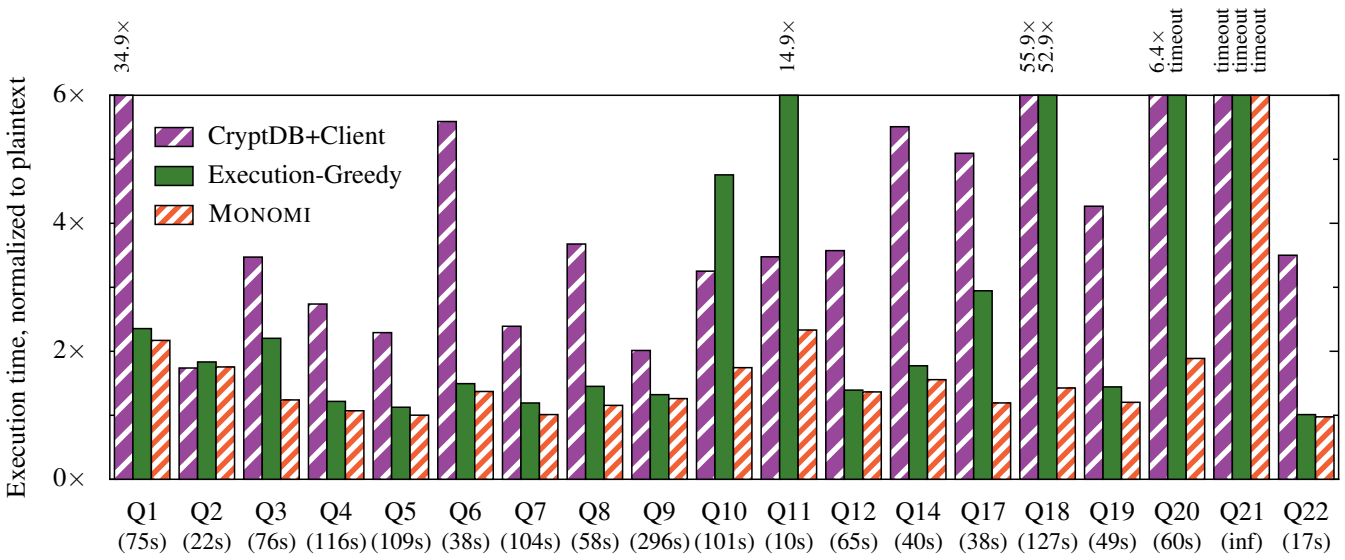

Figure 4: Execution time of TPC-H queries under various systems, normalized to the execution time of plaintext Postgres (shown in parentheses). Query 21 times out on all systems at scale 10 due to correlated subqueries, but incurs a 1.04× overhead with MONOMI relative to plaintext at scale 1.

the Postgres server to 8 GB of RAM on the server, and flush both the OS buffer cache and the Postgres buffer pool before running each query. Results using 4 GB and 24 GB of RAM are similar. To simulate a wide-area network connection between the client and a server running in a remote data center, we use the `tc` command in Linux to throttle the network bandwidth between the two machines to 10 Mbit/s. We use `ssh -C -c blowfish` to compress network traffic between the client and server machines for all experiments.

We use multiple cores to speed up decryption (on the client) and homomorphic multiplication (on the server), using at most 8 threads. We further speed up decryption on the client by caching the decryptions of repeating ciphertexts, using a cache size of 512 elements with a random eviction policy.

In all physical designs, we use the primary key definitions specified by the TPC-H benchmark. For encrypted designs, we index over the deterministic version of the column. We create only one secondary index over the `o_custkey` column, which we applied to all physical designs. We cluster each table on its primary key.

Queries 17, 20, and 21 cause trouble for the Postgres optimizer: they involve correlated subqueries, which the optimizer is unable to handle efficiently. We modified the query text for queries 17 and 20 to work around this issue by de-correlating the subqueries via explicit joins. We used the modified texts both as inputs to MONOMI, and to execute the plaintext query for our experiments. Query 21 had no obvious re-write, so we left it in its original form.

To provide a fair comparison between MONOMI and plaintext query processing, we make a few modifications to the plaintext schema. First, we replace all `DECIMAL` data types with regular integers, because integer arithmetic is much faster. Additionally, we found that `SUM()` was a CPU bottleneck for some of the plaintext queries, because it was emulating infinite size arithmetic in software. Thus, we replace `SUM()` with a custom aggregate UDF which performs regular integer addition without regard for overflow.

Finally, we modify the plaintext execution for TPC-H query 18 to speed up plaintext performance. Query 18 contains an outer `SELECT` statement with a predicate of the form `o_orderkey IN ( subselect )` . We run query 18 as two separate queries, fully materializing the inner subselect on the client. This is more efficient because it allows the Postgres optimizer to use a bitmap index scan on the `orders` table when answering the outer `SELECT` statement.

8.2 Overall efficiency

To understand how much overhead MONOMI imposes for securely executing queries over encrypted data, we compare the runtime of TPC-H queries using a plaintext Postgres database to using MONOMI with space constraint factor $S = 2$. The **MONOMI** bars in Figure 4 show the slowdown imposed by MONOMI. The median overhead is only 1.24×, and ranges from 1.03× for query 7 to 2.33× for query 11. We believe these modest overheads show that analytical queries over encrypted data are practical in many cases.

To the best of our knowledge, MONOMI is the first system that can efficiently execute TPC-H queries over encrypted data, making it difficult to construct a head-to-head comparison with state-of-the-art approaches for encrypted query processing. For example, the only TPC-H queries that CryptDB can execute are 2, 4, 12, and 22. Nonetheless, to provide some form of comparison, we constructed a modified version of CryptDB that is able to execute the same TPC-H queries as MONOMI. This modified version of CryptDB, which we call **CryptDB+Client**, executes as much of the query on the server as possible, using only techniques found in the original CryptDB design, and executes the rest of the query on the client using Algorithm 1 (which is something that CryptDB did not support). The bar **CryptDB+Client** in Figure 4 shows that MONOMI outperforms this approach by 3.16× in the median case.

8.3 Technique performance

The previous subsection shows that MONOMI achieves good performance for TPC-H queries. To understand what techniques contribute to MONOMI's effectiveness, we measure the time taken to execute TPC-H queries by allowing the designer to use one additional technique from §5 per experiment, and finally adding the optimizing planner. Figure 5 shows the mean and geometric mean execution times for the TPC-H queries with different optimizations applied cumulatively, and Figure 6 provides the execution time for example queries that benefit the most from each of the optimizations.

In these experiments, we start with the **CryptDB+Client** system from §8.2, and introduce one technique at a time. We use a greedy physical design and plan formulation for this experiment—that is, for each new technique we introduce, we always apply it if possible.

The first optimization, **+Col packing**, refers to packing multiple integer values from the same row into a single Paillier ciphertext.

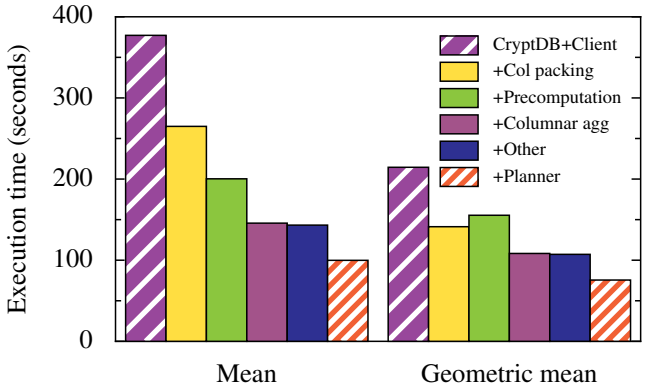

Figure 5: Aggregate execution time for a range of configurations, each adding one technique on top of the previous one.

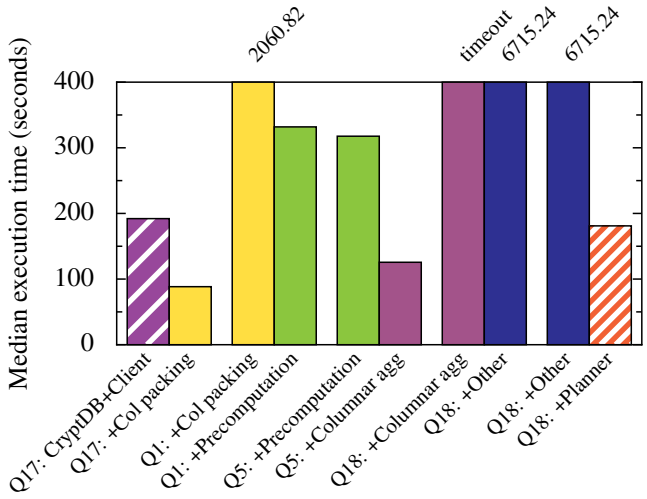

Figure 6: Running times of queries before and after a specific optimization is applied. For each optimization, we show the query that benefits the most from that optimization.

The runtime of query 17 is reduced by this optimization because it involves scanning the large `lineitem` table, and scans are much faster when there are fewer large Paillier ciphertexts stored per row.

The **+Precomputation** optimization refers to materializing encrypted values of certain expressions on the server. Query 1 benefits the most from this optimization because it involves aggregates over expressions that themselves cannot be computed at the server, such as `SUM(l_extendedprice * (1 - l_discount))`. Precomputing the Paillier encryption of `l_extendedprice * (1 - l_discount)` allows the server to compute the entire aggregate.

The **+Columnar agg** optimization refers to packing integer values from *multiple* rows into a single Paillier ciphertext. As described in §7, we split the storage of Paillier ciphertexts into a separate file, rather than storing them in the row itself. Query 5 benefits from this optimization because it computes an aggregate on the `lineitem` table, and packing multiple rows into a single ciphertext significantly reduces the amount of data read from disk.

The **+Other** optimization refers to pre-filtering and other optimizations. Query 18 benefits the most, because pre-filtering helps the server filter out some results, which both reduces the size of the intermediate results sent to the client, and the amount of work the server has to do to generate these intermediate results.

The last optimization, **+Planner**, uses MONOMI's planner to choose the best execution plan for each query. This is in contrast to a greedy execution strategy, which greedily executes all computation on the server whenever possible. This significantly improves the throughput of query 18, because it involves computing an aggregate for many distinct groups, where each group has few rows. The cost of decrypting the homomorphic ciphertext is higher than the cost of sending over the individual rows and aggregating them on the client.

In fact, the planner improves the performance of many TPC-H queries, and does not hurt any of them, as shown in Figure 4. The **Execution-Greedy** bar in Figure 4 corresponds to the **+Other** optimization level, and the performance of **MONOMI** is often better, and never worse than, the performance of **Execution-Greedy**.

8.4 Space and CPU overheads

Table 2 shows the amount of disk space used on the server by a plaintext database, MONOMI, and the two alternatives described previously: **CryptDB+Client** and **Execution-Greedy**. As can be seen from this data, MONOMI incurs only a 1.72× space overhead. **CryptDB+Client** incurs a much higher space overhead because it does not use space-efficient encryption schemes. **Execution-Greedy** has a slightly higher space overhead than MONOMI because it materializes some additional columns on the server that MONOMI's designer does not choose.

System	Size (GB)	Relative to plaintext
Plaintext	17.10	–
CryptDB+Client	71.98	4.21×
Execution-Greedy	32.55	1.90×
MONOMI	29.38	1.72×

Table 2: Server space requirements for the TPC-H workload, under several systems, and the overhead compared to plaintext.

To evaluate the CPU overhead imposed by MONOMI, we consider two scenarios. In the first scenario, the user runs the query on a local database (using plaintext Postgres, since the machine is trusted). In the second scenario, the user runs the query on a remote untrusted machine using MONOMI, and uses the local machine to run the MONOMI client library. Figure 7 shows the ratio of the CPU time spent by the client library in scenario two divided by the total time spent by the plaintext database in scenario one.

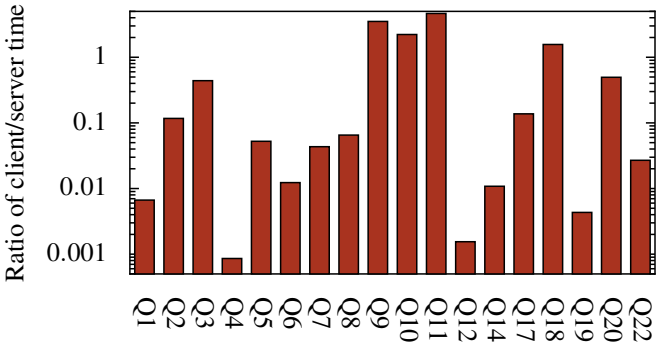

Figure 7: Ratio of the client CPU time required to execute a query using MONOMI compared to the client time that would be required to execute the query using a local Postgres server.

Ideally, this ratio would always be less than one, indicating that it always takes less CPU time to outsource query processing using MONOMI. For most queries, this is true, except for queries 9, 10, 11, and 18. In those queries, the MONOMI client library spends a lot of time in decryption. We believe this is acceptable because it is easy to parallelize decryption across many cores on the client (whereas

it is hard to parallelize Postgres), and because MONOMI does not require the client machines to store any data.

8.5 Sensitivity to designer input

The physical design used for MONOMI in Figure 4 was generated by handing the entire TPC-H workload to MONOMI's designer. In practice, however, the physical design is generated by feeding the designer a representative set of queries. This section provides insights on what properties the representative subset should have.

One option is to randomly choose k queries, which puts the least amount of burden on the administrator. Unfortunately, this yields poor performance for small values of k: for example, the median TPC-H cost estimate out of all designs from all possible $k = 4$ input queries, according to MONOMI's planner, is $\sim 10^6\times$ higher than the optimal design with all input queries, and executing TPC-H with this design times out. The reason is that if query 19 (which involves the large `lineitem` table joined with the `part` table) is not included, then the query runs entirely on the client, and the overhead of downloading the entire table to the client kills performance. Thus, it is important for the administrator to choose the right set of input queries to the designer in order to achieve good performance.

A natural question to ask is how many queries does the administrator have to provide to the designer at a minimum, in order to achieve good performance, even if the administrator manually chooses the best queries? To answer this question, we measured the best performance that MONOMI can obtain on TPC-H using $k = 0, 1, \ldots, 19$ input queries to the designer. In particular, for each k, we enumerated all n-choose-k subsets of the $n = 19$ total TPC-H queries we could support, and picked the k queries that yielded the lowest cost estimate on the *entire* TPC-H workload. We then implemented the best physical design for each k, and benchmarked the performance. Figure 8 shows the results. As the figure shows, using as few as four queries for the physical database design allows MONOMI to match the performance of having the entire query workload (Figure 4). Thus, as long as the administrator chooses the right representative queries, few queries are necessary to achieve good performance.

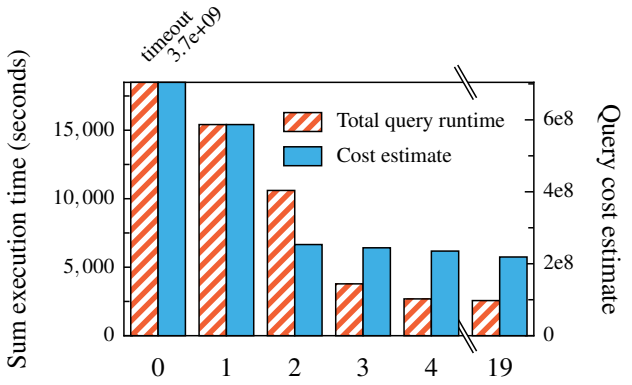

Figure 8: Total TPC-H workload execution time on a physical design chosen using the best $0 \le k \le 4$ and $k = 19$ queries as input to MONOMI's designer, along with the cost estimate from MONOMI's designer.

To understand how the administrator should pick these representative queries, we analyzed the $k = 4$ case to explore what characteristics these queries have that make them crucial to achieving good performance. In the $k = 4$ case, using TPC-H queries 1, 4, 9, and 19 as input to the designer resulted in the best overall performance. Query 1 involves aggregation over expressions which other TPC-H queries share, including several expressions which require pre-computation. Queries 4, 9, and 19 have (sub-)queries that contain very selective `WHERE` clauses over the `lineitem` table (the largest relation in TPC-H), which require specialized encryption schemes (such as OPE or keyword search) for computing the `WHERE` clause on the server. Thus, it is important for the administrator to choose queries that perform representative types of significant filtering and aggregation on the server.

These experiments assumed deterministic encryption as the default encryption scheme for certain types of columns. These types of columns include primary/foreign keys and enumerations/categories (such as `l_shipmode` from the `lineitem` table). The former is obviously important for joins; the latter is important for `GROUP BY` operations on categories, which are common in TPC-H.

8.6 Space constraints

The experiments so far have not stressed the space budget. To evaluate how well MONOMI's ILP formulation trades off performance for space, we compare the performance of TPC-H queries under a large space budget ($S = 2$) with their performance under a smaller space budget ($S = 1.4$) using two approaches: the first is MONOMI's ILP designer, and the second is a **Space-Greedy** approach that deletes the largest column until the space budget is satisfied.

We expect to see few differences between $S = 2$ and $S = 1.4$ because, as we saw in §8.4, MONOMI incurs a space overhead of only $1.72\times$, and indeed, most queries have the same running time. Figure 9 shows the results for the four queries affected by the space budget change. Both the **Space-Greedy** algorithm and MONOMI's ILP designer decide to drop the largest homomorphic column from the `lineitem` table, which significantly reduces the performance of query 1. The **Space-Greedy** algorithm decides to drop the OPE encryption of the `l_discount` column from the `lineitem` table, which significantly slows down query 6 because it can no longer apply a fairly selective predicate involving `l_discount` on the server.

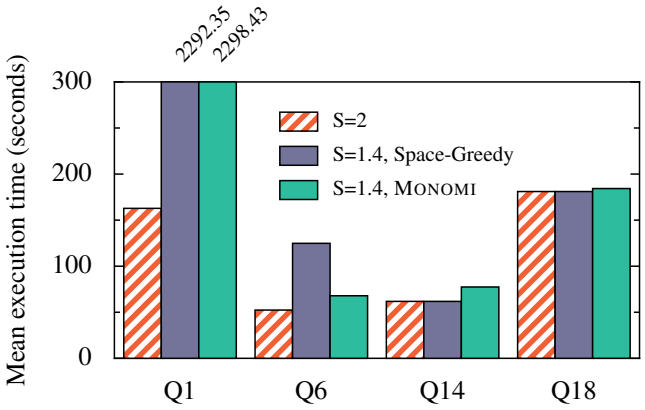

Figure 9: Execution times of queries that were affected by reducing the space budget from $S = 2$ to $S = 1.4$.

MONOMI's ILP designer instead drops two more homomorphic columns from the `lineitem` table, replacing one of them with a precomputed deterministic column. It also removes the OPE encryption of the `o_totalprice` column from the `orders` table. Collectively, this slows down queries 6, 14, and 18 by a much smaller amount.

8.7 Security

To understand the level of security that MONOMI provides, it is important to consider the encryption schemes chosen by MONOMI, since they leak different amounts of information, as shown in Table 1. The worst is OPE, which reveals order, followed by DET. Table 3 shows the encryption schemes that MONOMI chooses for the TPC-H workload. MONOMI never reveals plaintext to the server. The weakest encryption scheme used, OPE, is used relatively infrequently.

Table	Total # columns	RND, HOM, or SEARCH	DET	OPE
customer	8+1	0	7+1	1
lineitem	16+4	3+2	8+2	5
nation	4	1	2	1
orders	9+1	3	4+1	2
part	9	4	4	1
partsupp	6	2	3	1
region	3	1	2	0
supplier	7	4	2	1

Table 3: Number of distinct columns in the TPC-H tables encrypted by MONOMI under each of the encryption schemes shown. For each column, we consider only the weakest encryption scheme used. Numbers after a plus sign indicate encryptions of precomputed expressions.

Most uses of OPE come from the `lineitem` table, where MONOMI reveals order for five distinct columns. A common use of OPE is for date fields, which may be less sensitive. The next weakest scheme, DET, is used in quite a few columns, but reveals only duplicates, which may be less of a concern. Finally, many columns can remain encrypted with the most secure schemes, where a malicious server would learn nothing from the ciphertext other than its size.

9. DISCUSSION

A natural question for analytical workloads is how well MONOMI's techniques would apply to a column-store database. We expect that MONOMI's techniques would generalize well; a column-store would be less sensitive to the increased table width imposed by adding additional encrypted columns, and so it should have lower overheads relative to an unencrypted system than a row-store. Currently, MONOMI's designer does not consider table width when considering different designs, so it would be well suited for a column-store.

A second question has to do with how MONOMI would evolve physical designs over time, as the workload changes. It is straightforward to re-run the designer and compute a new database design, but in the current implementation installing the new design requires that the client download and decrypt each table that needs to be updated (via a `SELECT * ` query), and then re-insert each encrypted row using the new encrypted design. Exploring more efficient physical design evolution for encrypted databases is an area for future work.

Finally, MONOMI's designer does not take into account any security constraints. While it is difficult to quantitatively reason about trade-offs between performance and security, it would be straightforward to allow the administrator to specify minimum security thresholds for each column during the setup phase.

10. CONCLUSION

This paper presented MONOMI, a system for securely executing analytical queries over confidential data using an untrusted server. MONOMI protects data confidentiality by executing most query operations over encrypted data. MONOMI uses *split client/server query execution* to perform queries that cannot be efficiently computed over encrypted data alone. MONOMI introduces *per-row precomputation*, *space-efficient encryption*, *grouped homomorphic addition*, and *pre-filtering* to improve performance of analytical queries over encrypted data. Finally, MONOMI shows that using a designer and planner improves performance over greedy algorithms, by choosing efficient physical designs and query execution plans. On the TPC-H benchmark at scale 10, MONOMI achieves a median runtime overhead of just 1.24× with a 1.72× server space overhead.

ACKNOWLEDGMENTS

We thank Raluca Ada Popa, Adam Marcus, Dan Ports, Eugene Wu, and the anonymous reviewers for their feedback. This work was supported by NSF award IIS-1065219 and by Google.

REFERENCES

[1] D. J. Abadi, S. R. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *Proc. of SIGMOD*, pages 967–980, Vancouver, Canada, June 2008.

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *Proc. of the 26th VLDB*, pages 496–505, Cairo, Egypt, Sept. 2000.

[3] A. Arasu, S. Blanas, K. Eguro, R. Kaushik, D. Kossmann, R. Ramamurthy, and R. Venkatesan. Orthogonal security with Cipherbase. In *Proc. of the 6th CIDR*, Asilomar, CA, Jan. 2013.

[4] S. Bajaj and R. Sion. TrustedDB: a trusted hardware based database with privacy and data confidentiality. In *Proc. of SIGMOD*, pages 205–216, Athens, Greece, June 2011.

[5] M. Bellare, P. Rogaway, and T. Spies. Addendum to "The FFX mode of operation for format-preserving encryption". `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/proposedmodes/ffx/ffx-spec2.pdf`, Sept. 2010.

[6] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving symmetric encryption. In *Proc. of the 28th EUROCRYPT*, pages 224–241, Cologne, Germany, Apr. 2009.

[7] A. Boldyreva, N. Chenette, Y. Lee, and A. O'Neill. Order-preserving encryption revisited: Improved security analysis and alternative solutions. In *Advances in Cryptology (CRYPTO)*, pages 578–595, Aug. 2011.

[8] S. S. M. Chow, J.-H. Lee, and L. Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. In *Proc. of the 16th NDSS*, Feb. 2009.

[9] V. Ciriani, S. D. C. di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati. Keep a few: Outsourcing data while maintaining confidentiality. In *Proc. of the 14th ESORICS*, pages 440–455, Sept. 2009.

[10] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi. Zephyr: Live migration in shared nothing databases for elastic cloud platforms. In *Proc. of SIGMOD*, pages 301–312, Athens, Greece, June 2011.

[11] T. Ge and S. B. Zdonik. Answering aggregation queries in a secure system model. In *Proc. of the 33rd VLDB*, pages 519–530, Vienna, Austria, Sept. 2007.

[12] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. of the 41st STOC*, pages 169–178, Bethesda, MD, May–June 2009.

[13] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. Cryptology ePrint Archive, Report 2012/099, June 2012.

[14] H. Hacıgümüş, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proc. of SIGMOD*, pages 216–227, Madison, WI, June 2002.

[15] H. Hacıgümüş, B. R. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *DASFAA*, pages 125–136, Mar. 2004.

[16] H. Hacıgümüş, B. R. Iyer, and S. Mehrotra. Query optimization in encrypted database systems. In *DASFAA*, pages 43–55, Apr. 2005.

[17] S. Halevi and P. Rogaway. A tweakable enciphering mode. In *Advances in Cryptology (CRYPTO)*, pages 482–499, Aug. 2003.

[18] R. Kumar, J. Novak, B. Pang, and A. Tomkins. On anonymizing query logs via token-based hashing. In *Proc. of the 16th International World Wide Web Conference*, pages 629–638, Banff, Canada, May 2007.

[19] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. `http://www.cs.umb.edu/~poneil/StarSchemaB.pdf`, Jan. 2007.

[20] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proc. of the 18th EUROCRYPT*, pages 223–238, Prague, Czech Republic, May 1999.

[21] S. Papadomanolakis and A. Ailamaki. An integer linear programming approach to database design. In *Proc. of the 23rd ICDE*, pages 442–449, Istanbul, Turkey, Apr. 2007.

[22] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan. CryptDB: Protecting confidentiality with encrypted query processing. In *Proc. of the 23rd SOSP*, pages 85–100, Cascais, Portugal, Oct. 2011.

[23] R. A. Popa, F. H. Li, and N. Zeldovich. An ideal-security protocol for order-preserving encoding. In *Proc. of the 34th IEEE Symposium on Security and Privacy*, San Francisco, CA, May 2013.

[24] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *Proc. of the 21st IEEE Symposium on Security and Privacy*, pages 44–55, Oakland, CA, May 2000.