TrustedDB: A Trusted Hardware based Database with Privacy and Data Confidentiality

Sumeet Bajaj, Radu Sion

Abstract—Traditionally, as soon as confidentiality becomes a concern, data is encrypted before outsourcing to a service provider. Any software-based cryptographic constructs then deployed, for server-side query processing on the encrypted data, inherently limit query expressiveness. Here, we introduce TrustedDB, an outsourced database prototype that allows clients to execute SQL queries with privacy and under regulatory compliance constraints by leveraging server-hosted, tamper-proof trusted hardware in critical query processing stages, thereby removing any limitations on the type of supported queries. Despite the cost overhead and performance limitations of trusted hardware, we show that the costs per query are orders of magnitude lower than any (existing or) potential future software-only mechanisms. TrustedDB is built and runs on actual hardware, and its performance and costs are evaluated here.

Index Terms—Database architectures, security, Privacy, Special-purpose Hardware.

1 Introduction

Although the benefits of outsourcing and clouds are well known [41], significant challenges yet lie in the path of large-scale adoption since such services often require their customers to inherently trust the provider with full access to the outsourced datasets. Numerous instances of illicit insider behavior or data leaks have left clients reluctant to place sensitive data under the control of a remote, third-party provider, without practical assurances of *privacy* and *confidentiality*, especially in business, healthcare and government frameworks. Moreover, today's privacy guarantees for such services are at best declarative and subject customers to unreasonable fine-print clauses. E.g., allowing the server operator to use customer behavior and content for commercial profiling or governmental surveillance purposes.

Existing research addresses several such security aspects, including access privacy and searches on encrypted data. In most of these efforts data is encrypted before outsourcing. Once encrypted however, inherent limitations in the types of primitive operations that can be performed on encrypted data lead to fundamental expressiveness and practicality constraints.

Recent theoretical cryptography results provide hope by proving the existence of universal homomorphisms, i.e., encryption mechanisms that allow computation of arbitrary functions without decrypting the inputs [43]. Unfortunately actual instances of such mechanisms seem to be decades away from being practical [17].

Ideas have also been proposed to leverage tamper-proof hardware to privately process data server-side, ranging from smart-card deployment [25] in healthcare, to more general database operations [23], [32], [26].

Yet, common wisdom so far has been that trusted hardware is generally impractical due to its performance limitations and higher acquisition costs. As a result, with very few exceptions [25], these efforts have stopped short of proposing or building full - fledged database processing engines.

However, recent insights [9] into the cost-performance trade-off seem to suggest that things stand somewhat differently. Specifically, at scale, in outsourced contexts, computation inside secure processors is orders of magnitude cheaper than any equivalent cryptographic operation performed on the provider's unsecured server hardware, despite the overall greater acquisition cost of secure hardware.

This is so because the overheads for cryptography that allows some processing by the server on encrypted data are extremely high even for simple operations. This fact is rooted not in cipher implementation inefficiencies but rather in fundamental cryptographic hardness assumptions and constructs, such as trapdoor functions. Moreover, this is unlikely to change anytime soon as none of the current primitives have, in the past half-century. New mathematical hardness problems will need to be discovered to allow hope of more efficient cryptography.

As a result, we posit that a full-fledged, privacy enabling secure database leveraging server-side trusted hardware can be built and run at a fraction of the cost of any (existing or future) cryptography-enabled private data processing on common server hardware. We validate this by designing and building TrustedDB, a SQL database processing engine that makes use of tamper-proof cryptographic coprocessors such as the IBM 4764 [3] in close proximity to the outsourced data.

Tamper resistant designs however are significantly constrained in both computational ability and memory capacity which makes implementing fully featured

- *Sumeet Bajaj is with the Computer Science Department at Stony Brook University, Stony Brook, NY. E-mail: sbajaj@cs.stonybrook.edu..*
- *Radu Sion is with the Computer Science Department at Stony Brook University, Stony Brook, NY. E-mail: sion@cs.stonybrook.edu..*

Digital Object Indentifier 10.1109/TKDE.2013.38

database solutions using secure coprocessors (SCPUs) very challenging. TrustedDB achieves this by utilizing common unsecured server resources to the maximum extent possible. E.g., TrustedDB enables the SCPU to transparently access external storage while preserving data confidentiality with on-the-fly encryption. This eliminates the limitations on the size of databases that can be supported. Moreover, client queries are pre-processed to identify sensitive components to be run inside the SCPU. Non-sensitive operations are off-loaded to the untrusted host server. This greatly improves performance and reduces the cost of transactions.

Overall, despite the overheads and performance limitations of trusted hardware, the costs of running TrustedDB are orders of magnitude lower than any (existing or) potential future cryptography-only mechanisms. Moreover, it does not limit query expressiveness.

The contributions of this paper are threefold: (i) the introduction of new cost models and insights that explain and quantify the advantages of deploying trusted hardware for data processing, (ii) the design, development, and evaluation of TrustedDB, a trusted hardware based relational database with full data confidentiality, and (iii) detailed query optimization techniques in a trusted hardware-based query execution model.

2 The Real Costs of Security

As soon as confidentiality becomes a concern, data needs to be encrypted before outsourcing. Once encrypted, solutions can be envisioned that: (A) straightforwardly transfer data back to the client where it can be decrypted and queried, (B) deploy cryptographic constructs server-side to process encrypted data, and (C) process encrypted data server-side inside tamper-proof enclosures of trusted hardware.

In this section we will compare the per-transaction costs of each of these cases. This is possible in view of novel results of Chen et al. [9] that allow such quantification. We will show that, at scale, in outsourced contexts, (C) computation inside secure hardware processors is orders of magnitude cheaper than any equivalent cryptographic operation performed on the provider's unsecured common server hardware (B). Moreover, due to the extremely high cost of networking as compared with computation, the overhead of transferring even a small subset of the data back to the client for decryption and processing in (A) is overall significantly more expensive than (C).

The main intuition behind this has to do with the amortized cost of CPU cycles in both trusted and common hardware, as well as the cost of data transfer. Due to economies of scale, provider-hosted CPU cycles are 1-2 orders of magnitude cheaper than that of clients and of trusted hardware. The cost of a CPU cycle in trusted hardware (56+ picocents[1], discussed below) becomes thus of the same order as the cost of a traditional client CPU cycle at (e.g., 14-27 picocents for small businesses) including acquisition and operating costs.

Additionally, when data is hosted far from its accessing clients, the extremely expensive network traffic often dominates. E.g, transferring a single bit of data over a network costs upwards of 3500 picocents [9].

Finally, cryptography that would allow processing on encrypted data demands extremely large numbers of cycles even for very simple operations such as addition. This limitation is rooted in fundamental cryptographic hardness assumptions and constructs, such as cryptographic trapdoors, the cheapest we have so far being at least as expensive as modular multiplication [31], which comes at a price-tag of upwards of tens of thousands of picocents per operation [9].

The above insights lead to (C) being a significantly more cost-efficient solution than (A) and (B). We now detail.

2.1 Cost of Primitives

Compute Cycles and Networks. In [9] Chen et al. derived the cost of compute cycles for a set of environments ranging from individual homes with a few PCs (H) to large enterprises and compute clouds (L) (M,L=medium,large sized business). These costs include a number of factors, such as hardware (server, networking), building (floor space leasing), energy (electricity), service (personnel, maintenance) etc. Their main thesis is that, due to economies of scale and favorable operating parameters, per-cycle costs decrease dramatically when run in large compute providers' infrastructures.

Fig. 1. CPU cycle costs (picocent).

H	S	M	L
5	14-27	2	<0.5

The resulting cpu cycle costs (figure 1) range from 27 picocents for a small business environment to less than half of a picocent for large cloud providers. Network service costs range from a few hundred picocents per bit for *non-dedicated* service to thousands of picocents in the case of medium sized businesses. Detailed numbers are available in [39], [9], [40].

Also, the work in [39], [9] derives the cost of x86-equivalent CPU cycles inside cloud-hosted SCPUs such as the IBM 4764 to be $\approx$56 picocents. We note that while this is indeed much higher than the < 0.5 picocent cost of a cycle on commodity hardware, it is comparable to the cost of cycles in CPUs hosted in small sized enterprises (14-27 picocents).

2.2 Comparison

Given these data points we now compare the A, B and C alternatives discussed above. We consider the following simple scenario. A client outsources a encrypted dataset composed of integers to a provider. The encrypted data is then subjected to a simple aggregation (SUM) query in which the server is to add all the integers without

1. 1 US picocent = 10^{-14} USD

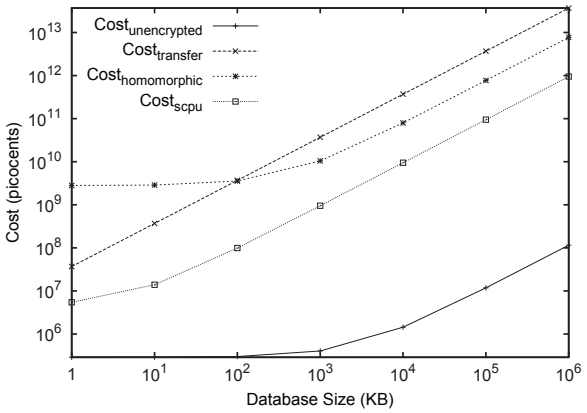


Fig. 2. Comparison of outsourced aggregation query solutions.

decryption and return the result to the client. We chose this mechanism not only for its illustrative simplicity but also because SUM aggregation is one of the very few types of queries for which non-hardware solutions have been proposed. This allows us to directly compare with existing work. Later in section 2.3 we also generalize for arbitrary queries. Figure 2 summarizes the cost analysis that follows.

Querying un-encrypted data. No confidentiality. As a baseline consider the most prevalent scenario today, in which the client's data is stored un-encrypted with the service provider. Client queries are executed entirely on the provider's side and only the results are transferred back. Although this is the most cost - effective solution it offers no data confidentiality. The lower bound cost of query execution in this case is as follows[2]:

$$Cost_{unencrypted} = 2 \cdot D \cdot C_{bit\_transmit} + \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_server} \cdot \eta_{addition} \quad (1)$$

where N is the size of the entire database in bits, $D = 32$ (32 bit integers), C_{cycle_server} is the cost of one CPU cycle on server hardware, $\eta_{addition} = 1$ is the average number of CPU cycles required for an addition operation [22]. $C_{bit\_transmit}$ is the cost of transmitting 1 bit of data from the service provider to the client.

(A) Transferring encrypted data to client. The first baseline solution for data confidentiality works by transferring the entire database to the client. The client then decrypts and aggregates the data. The cost of this alternative becomes

$$Cost_{transfer} = N \cdot (C_{bit\_transmit} + C_{bit\_decryption}) + \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_client} \cdot \eta_{addition} \quad (2)$$

Where $C_{bit\_decryption} = 8$ picocents is the normalized cost of decrypting one bit with AES-128 and $C_{cycle\_client} = 2$ picocents is the cost of a single client CPU cycle respectively in medium-sized (M) enterprises. Naturally we observe that here the cost of transferring the database to the client dominates.

2. The cost of reading data from storage into main memory is a common factor in all solutions and thus not included here

(B) Cryptography. Traditional additive homomorphisms [33], [28], [29] have been used in existing work [19], [42], to allow servers to run aggregation queries over encrypted data. These allow the computation of the encryption of the sum of a set of encrypted values without requiring their decryption in the process.

Existing homomorphisms require the equivalent work of at least a modular multiplication in performing their corresponding operation, such as addition. Moreover, for security, this modular multiplication needs to be performed in fields with a large modulus. For efficiency [42] goes one step further and proposes to perform aggregation in parallel by simultaneously adding multiple 32-bit integer values. They achieve this by adding two 1024-bit chunks of encrypted data at a time. Due to the properties of the Paillier cryptosystem, each such addition involves one 2048-bit modular multiplication[3].

The server then computes the encrypted sum of all such large integers, which is equivalent to a single modular multiplication of the encrypted values modulo 2048, and returns the result to the client. The client decrypts the 2048 bit result into a 1024 bit plain-text, splits this into 32 integers of 32 bits each, and computes their sum. The cost of this scheme is given by

$$Cost_{homomorphic} = \frac{B_h}{D} \cdot \left(\left(\frac{N}{B_h} - 1\right) \cdot C_{modular\_mul} + 2 \cdot B_h \cdot C_{bit\_transmit} + C_{homomorphic\_dec} + \left(\frac{B_h}{D} - 1\right) \cdot C_{cycle\_client} \cdot \eta_{addition}\right) \quad (3)$$

where $B_h = 1024$ is the plain-text block size and $C_{modular\_mul}$ is the cost of performing a single modular multiplication modulo 2048 on the server. $C_{homomorphic\_dec}$ is the cost of performing the single decryption on client and involves modular multiplication and exponentiation.

(C) SCPUs. A possible use of a SCPU is to perform the aggregation fully within it. The result can then be re-encrypted and transmitted back to the client.

In addition to the core CPU processing costs (which can be computed directly from the cost of SCPU cycles), data transfer overheads are incurred i.e., to bring encrypted data into the SCPU and then transfer the encrypted results back to the host server. The total cost of the solution becomes

$$Cost_{scpu} = \left\lceil \frac{N}{B_s} \right\rceil \cdot (\delta_{srv} \cdot C_{cycle\_srv} + \delta_{scpu} \cdot C_{cycle\_scpu}) + N \cdot C_{bit\_decryption\_scpu} + \left(\frac{N}{D} - 1\right) \cdot C_{cycle\_scpu} \cdot \eta_{addition\_scpu} + B_c \cdot C_{bit\_encryption\_scpu} + B_c \cdot C_{bit\_transmit} + B_c \cdot C_{bit\_decyption\_client} \quad (4)$$

Where δ_{srv} and δ_{scpu} are the server and SCPU cycles used to setup data transfer and include the cost of setting

3. To process n-bit plain-texts, Paillier operates in $n^2 = 2048$ bit fields for 1024 bit plain-texts. Cipher-texts are 2048 bit.

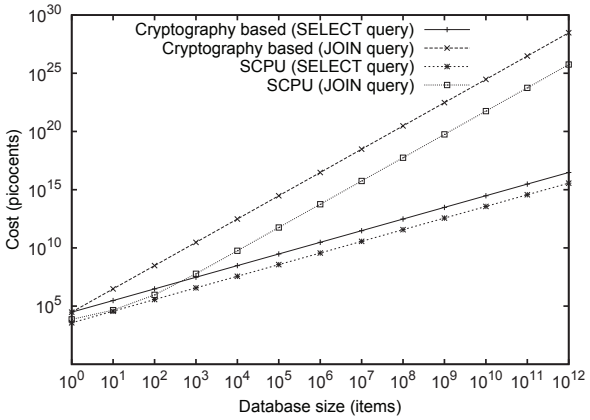


Fig. 3. SCPU is 1-3 orders of magnitude cheaper than cryptography.

up and handling DMA interrupts. C_{cycle_scpu} is the cost of a SCPU cycle. $B_s = 64KB$ is the block size of data transmitted between the server and the SCPU in one round and B_c is the cipher block size (128 bits for AES). $\eta_{addition\_scpu} = 2$ is the number of cycles per addition operation in the SCPU for 64 bit addition (on a 32 bit architecture).

Figure 2 shows the cost relationship between the solutions. It can be seen that for any data set sizes $Cost_{scpu} < Cost_{homomorphic}$ and $Cost_{scpu} < Cost_{transfer}$. We also note that for data sets of size $< 100KB$, the cost of client-side homomorphic decryptions (which involves modular exponentiation) dominates and exceeds the data transmission cost in $Cost_{transfer}$. Overall, the use of SCPUs is the most efficient from a cost-centric point of view, by more than an order of magnitude when compared with cryptographic alternatives.

2.3 Generalized Argument

Recall that current cryptographic constructs are based on trapdoor functions [18]. Currently viable trapdoors are based on modular exponentiation in large fields (e.g., 2048 bit modular operations) and viable homomorphisms involve a trapdoor for computing the ciphertexts. Additionally, the homomorphic operation itself involves processing these encrypted values at the server in large fields, while respecting the underlying encryption trapdoor, incurring at least the cost of a modular multiplication [33], [28], [29]. This fundamental cryptography has not improved in efficiency in decades and would require the invention of new mathematical tools before such improvements are possible.

Thus, overall, for large scale, efficient deployments, (e.g., clouds) where CPU cycles are extremely cheap (e.g., 0.45 picocents/cycle), performing the cheapest, least secure homomorphic operations (modular multiplication) comes at a price-tag of at least 30,000 picocents [9] even for values as small as 32-bit (e.g., salaries, zip-codes).

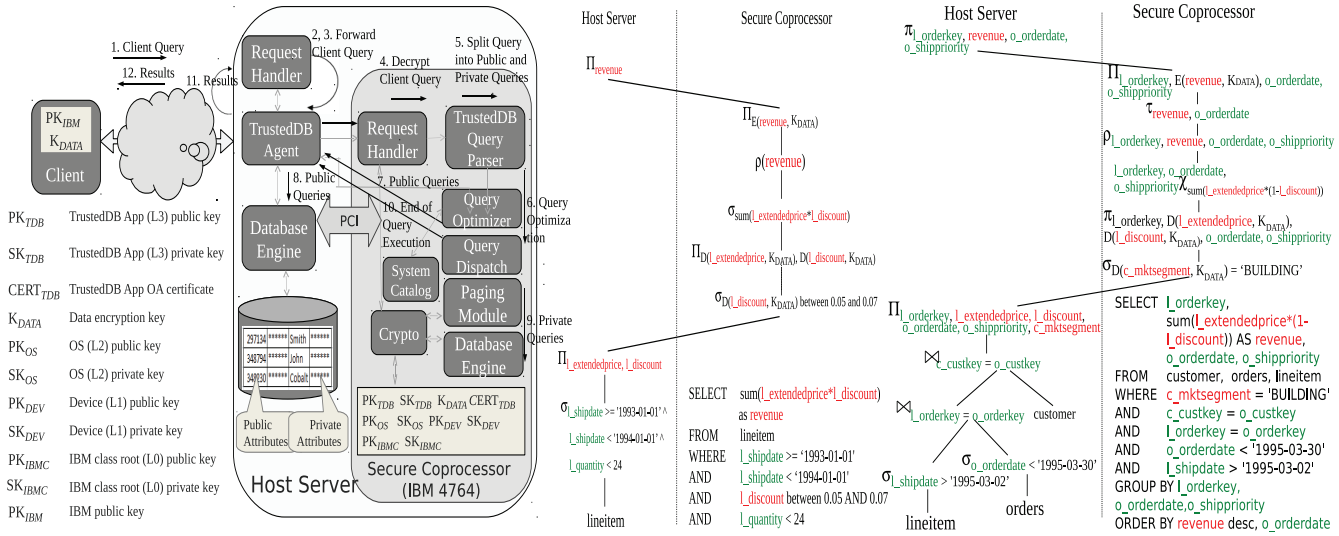
Thus, even if we assume that in future developments homomorphisms will be invented that can allow full Turing Machine languages to be run under the encryption envelope, unless new trapdoor math is discovered each operation will yet cost at least 30,000 picocents when run on efficient servers. By comparison, SCPUs process data at a cost of 56 picocents/cycle. This is a difference of several orders of magnitude in cost. We also note that, while ECC signatures (e.g., even the weak ECC-192) may be faster, ECC-based trapdoors would be even more expensive, as they would require two point multiplications, coming at a price-tag of least 780,000 cycles ([11] page 402).

Yet, this is not entirely accurate, as we also need to account for the fact that SCPUs need to read data in before processing. The SCPUs considered here feature a decryption throughput of about 10-14 MB/second for AES decryption [3], confirmed also by our benchmarks. This limits the ability to process data. E.g, comparing two 32-bit integers as in a JOIN operation becomes dominated not by the single-cycle conditional JUMP CPU operation but by the cost of decryption. At 166-200 megacycles/second this results in the SCPU having to idly wait anywhere between 47 and 80 cycles for decryption to happen in the crypto engine module before it can process the data. This in effect results in an amortized SCPU cost of between 2632 and 4480 picocents (3556 picocents on average) for each operation which reduces the above 3 orders of magnitude difference to only one order of magnitude, still in favor of SCPUs [4].

The above holds even for the case when the SCPU has only enough memory for the two compared values. Further, in the presence of significantly higher, realistic amounts of SCPU memory (e.g., $M = 32$MB for 4764-001), optimizations can be achieved for certain types of queries such as relational JOINs. The SCPU can read in and decrypt entire data pages instead of single data items and run the JOIN query over as many of the decrypted data pages as would fit in memory at one time. This results in significant savings. To illustrate, consider a page size of P 32-bit words and a simple JOIN algorithm for two tables of size N 32-bit integers each (we're just concerned with the join attribute). Then the SCPU will perform a number of $(N/P)^2 + (N/P)$ page fetches each involving also a page data decryption at a cost of $P \cdot 3556$ picocents. Thus we get a total cost of $(\frac{N^2}{P} + N) \cdot 3556 + N^2 \cdot 56$. For reasonable sizes, e.g., $P = M/2/4 = 4$ million, this cost becomes 3+ orders of magnitude lower than the $N^2 \cdot 30000$ picocent cost incurred in the cryptography-based case.

Cost vs. Performance. Given these 3+ orders of magnitude cost advantages of the SCPU over cryptography-based mechanisms, we expect that for the above discussed aggregation query mechanism [42], the SCPU's overall *performance* will also be at least comparable if not better despite the CPU speed handicap. We experimentally evaluated this hypothesis and achieved a throughput of about 1.07 million tuples/second for the

4. The cost can be reduced further by up to 50% if instead of AES, a cipher is built using a faster cryptographic hash as a pseudo-random function [6].



(a) TrustedDB architecture. (b) Query Plan for Q6. (c) Query Plan for Q3.

Fig. 4. TrustedDB architecture and query plans. Green and Red indicate public & private attributes respectively.

SCPU. By contrast, in [42] best-case scenario throughputs range between 0.58 and 0.92 million tuples/second and at much higher overall cost.

Conclusion Summary. Figure 3 compares SCPU based query processing with the most ideal cryptography based mechanisms employing a single modular multiplication. Note that such idealistic crypto mechanisms have not been invented yet, but even if they were as Figure 3 illustrates, for linear processing queries (e.g., SELECT) the SCPU is roughly one+ order of magnitude cheaper. For JOIN queries, the SCPU costs drop even further even when assuming no available memory. Finally, in the presence of realistic amounts of memory, due to increased overhead amortization, the SCPU is multiple orders of magnitude cheaper than software-only cryptographic solutions on legacy hardware.

We note that the above conclusion *may not* apply to targeted niche scenarios. E.g., it is entirely possible that by maintaining client pre-computed data server-side, processing only a pre-defined set of queries or by supporting minimal query classes (such as only range queries) specifically designed niche solutions may turn out to be cheaper than general-purpose full-fledged SCPU-backed databases like TrustedDB.

3 Architecture

Figure 4(a) depicts the TrustedDB architecture. In the following we discuss some of the key elements.

Overview. To overcome SCPU storage limitations, the outsourced data is stored at the host provider's site. Query processing engines are run on both the server and in the SCPU. Attributes in the database are classified as being either public or private. Private attributes are encrypted and can only be decrypted by the client or by the SCPU.

Since the entire database resides outside the SCPU, its size is not bound by SCPU memory limitations. Pages that need to be accessed by the SCPU-side query processing are pulled in on demand by the Paging Module.

Query execution entails a set of stages. (0) In the first stage a client defines a database schema and partially populates it. Sensitive attributes are marked using the SENSITIVE keyword which the client layer transparently processes by encrypting the corresponding attributes:

```
CREATE TABLE customer(ID integer primary key,
Name char(72) SENSITIVE, Address char(120) SENSITIVE);
```

(1) Later, a client sends a query request to the host server through a standard SQL interface. The query is transparently encrypted at the client site using the public key of the SCPU. The host server thus cannot decrypt the query. (2) The host server forwards the encrypted query to the Request Handler inside the SCPU. (3) The Request Handler decrypts the query and forwards it to the Query Parser. The query is parsed generating a set of plans. Each plan is constructed by rewriting the original client query into a set of sub-queries, and, according to their target data set classification, each sub-query in the plan is identified as being either public or private. (4) The Query Optimizer then estimates the execution costs of each of the plans and selects the best plan (one with least cost) for execution forwarding it to the dispatcher. (5) The Query Dispatcher forwards the public queries to the host server and the private queries to the SCPU database engine while handling dependencies. The net result is that the maximum possible work is run on the host server's cheap cycles. (6) The final query result is assembled, encrypted, digitally signed by the SCPU Query Dispatcher, and sent to the client.

Query Parsing and Execution. Sensitive attributes can occur anywhere within a query, e.g., in SELECT, WHERE or GROUP-BY clauses, in aggregation operators, or within sub-queries. The Query Parser's job is then.

(a) To ensure that any processing involving private attributes is done within the SCPU. All private attributes are encrypted using a shared data encryption keys be-

tween the client and the SCPU, hence the host server cannot decipher these attributes (section 5).
(b) To optimize the rewrite of the client query such that most of the work is performed on the host server.

To exemplify how public and private queries are generated from the original client query we use examples from the TPC-H benchmark [2]. TPC-H does not specify any classification of attributes based on security. Therefore, we define a attribute set classification into private (encrypted) and public (non-encrypted). In brief, all attributes that convey identifying information about customers, suppliers and parts are considered private. The resulting query plans, including rewrites into main CPU and SCPU components for TPC-H queries Q3 and Q6 are illustrated in Figure 4.

For queries that have WHERE clause conditions on public attributes, the server can first SELECT all the tuples that meet the criteria. The private attributes' queries are then performed inside the SCPU on these intermediate results, to yield the final result. E.g., query $Q6$ of the TPC-H benchmark is processed as shown in Figure 4(b). The host server first executes a public query that filters all tuples which fall within the desired $ship\ date$ and $quantity$ range, both of these being public attributes. The result from this public query is then used by the SCPU to perform the aggregation on the private attributes $extended\ price$ and $discount$. While performing the aggregation the private attributes are decrypted inside the SCPU. Since the aggregation operation results in a new attribute composing of private attributes it is re-encrypted within the SCPU before sending to the client.

Note that the execution of private queries depends on the results from the execution of public queries and vice-a-versa even though they execute in separate database engines. This is made possible by the TrustedDB Query Dispatcher in conjunction with the Paging Module.

Data manipulation queries (INSERT, UPDATE) also undergo a rewrite. Moreover, any new generated attribute values are re-encrypted within the SCPU before updating the database. For illustration, consider the query `UPDATE EMPLOYEES SET SALARY = SALARY + 2000 WHERE ZIP = 98239`. If `SALARY` is a private attribute then the query works by first decrypting the `SALARY` attribute, performing the addition and then re-encrypting the updated values and is executed within the SCPU. On the other hand if `SALARY` is a public attribute then the query will be executed entirely by the host server.

We refer the reader to [39] for more detailed explanation of query processing including group by and nested queries. In this work we focus on query optimization techniques in a trusted hardware model.

3.1 Query Optimization

3.1.1 Model

As per section 3, due to the un-availability of storage within the SCPU the entire database is stored at the server. However, the attribute classification into public (non-encrypted) and private (encrypted) introduces a strict vertical partitioning (although logical) of the database between the server and the SCPU. The requirement to be adhered to is that any processing on private attributes must be done within the confinements of the SCPU. This partitioning of data resembles a federated database rather than a stand alone DBMS. Since this partitioning is dictated by the security requirements of the application and all data resides on the server existing techniques [35], [14] are ruled out. The following sections describe query optimization in TrustedDB within the scope of this logical partitioning.

3.1.2 Overview

At a high level query optimization in a database system works as follows.
(i) The *Query Plan Generator* constructs possibly multiple plans for the client query.
(ii) For each constructed plan the *Query Cost Estimator* computes an estimate of the execution cost of that plan.
(iii) The best plan i.e., one with the least cost, is then selected and passed on to the *Query Plan Interpretor* for execution.

The query optimization process in TrustedDB works similarly with key differences in the *Query Cost Estimator* due to the logical partitioning of data mentioned above. However, we note that all optimizations possible in a traditional DBMS with no private attributes are still applicable to public sub-queries executed on the server. We refer the reader to [24] for details of these existing optimizations.

In the following sections we only discuss cases which are unique to TrustedDB and trusted hardware based designs alike.

Metric. The key in query optimization is estimating the costs of various logical query plans. A common metric utilized in comparing query plan costs has been disk I/O [38], [36] which is justified since disk access is the most expensive operation and should be minimized. In the trusted hardware model of TrustedDB an additional significant I/O cost is introduced i.e., the server↔SCPU data transfer. Moreover, disk access on the server and the Server↔SCPU communication have different costs. In addition we also need to consider the disparity between the computational abilities of the server and the SCPU. To combine all these factors we use execution time as the metric for cost estimation. Note that from this point onwards any reference to the cost of a query plan refers to its execution time.

Also, the goal of the *Query Optimizer* is not to measure query execution times with high accuracy but only to correctly compare query plans based on an estimation. To clarify, assume that a query Q has two valid execution plans P_A and P_B. Then, if the real execution times of P_A and P_B are such that $\mathcal{ET}_{real}(P_A) > \mathcal{ET}_{real}(P_B)$, then it suffices for the *Query Optimizer* to estimate $\mathcal{ET}_{est}(P_A) > \mathcal{ET}_{est}(P_B)$ although the values for $\mathcal{ET}_{real}(P_i)$ and

$\mathcal{ET}_{est}(P_i)$ may not be close.

Approach. To illustrate the query optimization in TrustedDB we take the following approach for each case.
(1) First, we present two alternative plans for the case.
(2) Next, we estimate the execution times ($\mathcal{ET}$) for each of the two plans.
(3) We analyze the estimations of step (2) for selection of the best plan.

Then, in section 4 we experimentally verify query optimization techniques.

Note that in the running system multiple (>2) plans could be considered. We limit ourselves to only two here, for brevity. Also, steps (1-3) from above are performed by the TrustedDB *Query Optimizer* (figure 4(a)).

3.1.3 System Catalog

Any query plan is composed of multiple individual execution steps. To estimate the cost of the entire plan it is essential to estimate the cost of individual steps and aggregate them. In order to estimate these costs the *Query Cost Estimator* needs access to some key information. E.g., the availability of an index or the knowledge of possible distinct values of an attribute. These sets of information are collected and stored in the *System Catalog*. Most available DBMS today have some form of periodically updated *System Catalog*. Figures 5-6 and tables 1-3 give a partial view of the System Catalog maintained by TrustedDB. Later, in section 3.1.5 we will see how this information is used in estimating plan execution times. System Catalog content is categorized as follows.

Fig. 5. System Configuration parameters.

	Server		SCPU	
CPU	η_s	3.4 GHz	η_t	233 MHz
Memory	Υ_s	4 GB	Υ_t	32 MB

(a) **System Configuration** (Figure 5).
These are the available compute capacities of the system hardware. This information is unlikely to change frequently and is configured during setup.

(b) **Benchmarked Parameters** (Table 1).
As part of query execution many basic operations are performed which add to the overall execution time. Benchmarks are employed to determine the average execution times for these operations to aid in query cost estimation. Unless changes occur in the system configuration this information need not be updated.

Fig. 6. Database Configuration.

Database Parameters		
DB Page Size	ρ	32KB
Server Cache Size	μ_s	32768
SCPU Cache Size	μ_t	1024
B^+-Tree Order	θ	100

(c) **Database Configuration** (Figure 6).
These parameters are directly set on the database to improve performance. The server DBMS is an off the shelf industrial quality database system (section 4) which provides large range of configuration parameters. With the TrustedDB design this host DBMS can be configured independently.

(d) **Data Statistics** (Table 2).
A data scan is employed to collect statistics about the actual data. This scan is configured to run periodically. The statistics on public attributes are collected server side. However, private attributes are scanned via the SCPU, decrypted and then analyzed. Note that since the collection process involves scan of the database it is a time consuming task and needs to be scheduled accordingly (e.g. nightly or weekends).

TABLE 1
Benchmarked parameters.

Benchmarked Parameters			
Disk Read	ϕ_s	0.02 ms	Avg time to read 32 KB blk from server disk
Server↔SCPU	λ	5.26 ms	Avg time to transfer a 32 KB blk between server and SCPU
Cycles / aggregation	δ_g	3	number of cpu cycles per aggregate operation (e.g. group by)
Cycles / addition	δ_a	1	number of cpu cycles per addition
Cycles / comparison	δ_c	1	number of cpu cycles per comparison between two values
Crypto	ϵ_a	0.012 μs	Time to encrypt/decrypt a single (32 byte) attribute in SCPU

TABLE 2
Collected data statistics.

lineitem			
Attribute	**Values**	**Max**	**Size**
l_shipdate	$\upsilon_{l_{sd}}$ = 2526	$\vartheta_{l_{sd}}$	$\kappa_{l_{sd}}$ = 4
l_shipmode	$\upsilon_{l_{sm}}$ = 7		$\kappa_{l_{sm}}$ = 10
l_linestatus	$\upsilon_{l_{ls}}$ = 2		$\kappa_{l_{ls}}$ = 16
l_quantity			$\kappa_{l_{qt}}$ = 4
l_discount	$\upsilon_{l_{dc}}$ = 11		$\kappa_{l_{dc}}$ = 16
l_orderkey			$\kappa_{l_{ok}}$ = 4
l_linenumber			$\kappa_{l_{ln}}$ = 4
Indexes			
Table	**Attribute(s)**	**Type**	**Organization**
lineitem	l_orderkey, l_linenumber	B^+-Tree	clustered
lineitem	l_shipdate	B^+-Tree	non-clustered

TABLE 3
Collected relation level statistics.

	lineitem		orders		part	
Number of tuples	φ_l	6 M	φ_o	1.5 M	φ_p	200 K
Tuple size	τ_l	120	τ_o	100	τ_p	160
Tuples per page ($\frac{\rho}{\tau}$)	ω_l	273	ω_o	328	ω_p	203

3.1.4 Analysis of Basic Query Operations

The cost of a plan is the aggregate of the cost of the steps that comprise it. In this section we present how execution times for a certain set of basic query plan steps are estimated. These steps are re-used in multiple plans and hence we group their analysis here.

(i) **Index-based lookup.**
Consider the selection query $\mathcal{Q} = \sigma_{l\_shipdate=10/01/1998}$. As per the System Catalog (table 2) there is a B^+-Tree index available on the attribute $l\_shipdate$. Hence the expected execution time to locate the first leaf page containing the result tuples will be

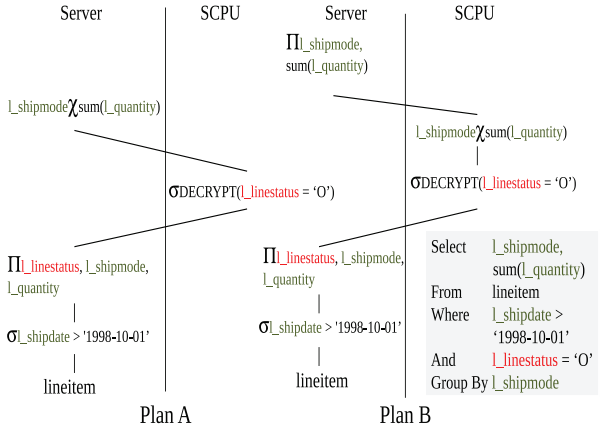
$$\mathcal{ET}(\mathcal{Q}) = \log_\theta \varphi_l \cdot \phi_s \quad (5)$$

Here, $\log_\theta \varphi_l$ is number of index pages read while ϕ_s is the time to read a single page from disk on server.

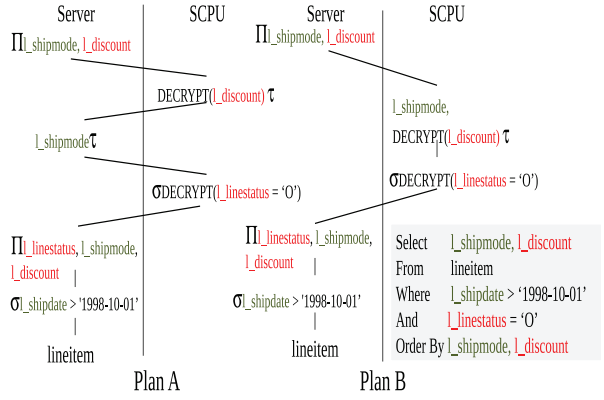
(ii) **Selection.**
Estimating the execution time of selection queries requires estimation of the number of tuples that would comprise the query result. For this, we first define the following from [24].

Definition 1: Values: The *Values* of an attribute A, $Values(A)$ or υ_A is the number of distinct values of A.

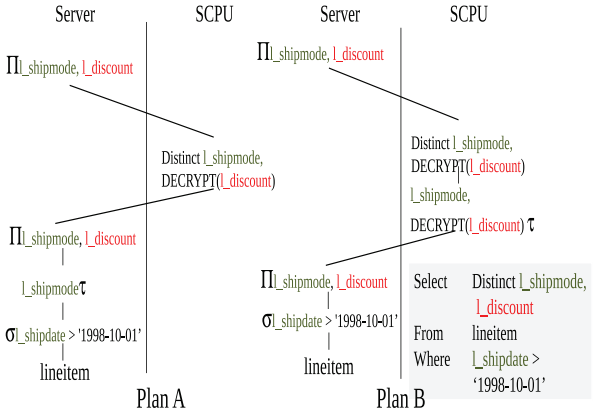
Definition 2: Reduction Factor: The *Reduction Factor* of a condition C, $Reduction(C)$ or Θ_C is the reduction



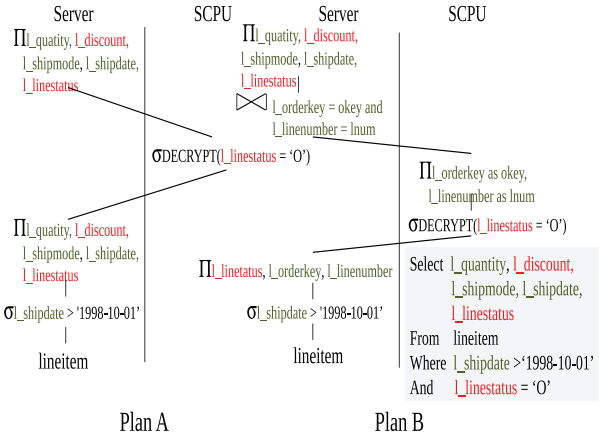
(a) Case I: Group By on public attribute.



(b) Case II: Order By on public and private attributes.



(c) Case III: Distinct clause on public and private attribute.



(d) Case IV: Projections.

Fig. 7. Query optimization plans for cases I - IV of section 3.1.5. Green and Red indicate public & private attributes respectively.

in size of the relation caused by the execution of a selection clause with that condition.

E.g.

$$Reduction(l\_shipdate = 10/01/1998) = \frac{1}{Values(l\_shipdate)} \quad OR \quad (6)$$

$$\Theta_{l\_shipdate=10/01/1998} = \frac{1}{\upsilon_{l_{sd}}}$$

Further,

$$\Theta_{l\_shipdate>10/01/1998} = \frac{\vartheta_{l_{sd}} -' 10/01/1998'}{\upsilon_{l_{sd}}} \quad (7)$$

Note, that the above definitions assume a uniform distribution of attribute values i.e. each distinct attribute value is equally likely to occur in a relation.

Now, consider the selection query $\sigma_{l\_shipdate>10/01/1998}$. In its execution, the index on $l\_shipdate$ is used to locate the first leaf page containing the result. Then, subsequent leaf pages are scanned to gather all tuples comprising the result. Hence the estimated execution time is $\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$. Here the term $\frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l}$ estimates the number of leaf pages containing all tuples that satisfy the query.

(iii) **Server↔SCPU data transfer.**

The intermediate results from query plan execution are often transferred between the server and the SCPU. This data transfer occurs in fixed sized pages in a synchronous fashion. If the data to be transferred is B bytes then the total transfer time is $\lceil \frac{B}{\rho * 1024} \rceil \cdot \lambda$. Here, ρ is the page size and hence $\lceil \frac{B}{\rho * 1024} \rceil$ gives the number of pages needed to transfer B bytes. λ is the time required to transfer a single page of size ρ (see table 1 and figure 6 for values). Suppose that we need to transfer the results of the query $\Pi_{sum(l\_quantity)}(\sigma_{l\_shipdate>10/01/1998 \; and \; l\_linestatus='O'})$. Then we can estimate the intermediate query result size by multiplying the number of tuples in the query result with the total size of the projection operation. The size of the projection is simply the sum of the sizes of the individual attributes $l\_linestatus$ and $l\_quantity$. Hence, the total data transfer time for this query is estimated as $\frac{(\kappa_{l_{ls}} + \kappa_{l_{qt}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \cdot \lambda$.

(iv) **External Sorting.**

Since database relations can require large amount of storage space external sorting is employed whenever the relation(s) to be sorted cannot fit in memory. External sorting has been studied extensively [24] and the I/O cost for an external merge sort is given as $2 \cdot F \cdot log_{M-1} F$, where F is the total number of relation pages and M is the number of pages that can be stored in memory ($M << F$). Using this we can estimate the execution time for sorting a relation r on the server as

$$2 \cdot \frac{\varphi_r \cdot \tau_r}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_s \cdot 1024}{\rho} - 1} \frac{\varphi_r \cdot \tau_r}{1024 \cdot \rho} \right) \cdot \phi_s \quad (8)$$

TABLE 4
Cost computations for query plans shown in figures 7(a), 7(b) corresponding to cases I-II.

Case	Plan	Step	Execution Time ($\mathcal{ET}$)	Total $\mathcal{ET}$ (s)
(I) Group By on public attribute	A	(i). $\sigma_{l\_shipdate>\text{'1998-10-01'}}$ (ii). Server $\rightarrow$ SCPU transfer (iii). $\sigma_{DECRYPT(l\_linestatus=\text{'O'})}$ (iv). Server $\leftarrow$ SCPU transfer (v). $_{l\_shipmode}\chi_{sum(l\_quantity)}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$ $\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$ $\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$ $\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$ $(\Theta_{l_{sd}} \cdot \varphi_l \cdot \delta_g + \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l \cdot \delta_a) \cdot \frac{1000}{\eta_s}$	0.57
	B	(i). $\sigma_{l\_shipdate>\text{'1998-10-01'}}$ (ii). Server $\rightarrow$ SCPU transfer (iii). $\sigma_{DECRYPT(l\_linestatus=\text{'O'})}$ (iv). $_{l\_shipmode}\chi_{sum(l\_quantity)}$ (iv). Server $\leftarrow$ SCPU transfer	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$ $\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$ $\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$ $2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda$, $F = \left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil$ $\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{sm}}) \cdot v_{l_{sm}}}{\rho \cdot 1024} \right\rceil \cdot \lambda$	0.78
(II) Order By on Public and Private attributes	A	(i). $\sigma_{l\_shipdate>\text{'1998-10-01'}}$ (ii). Server $\rightarrow$ SCPU transfer (iii). $\sigma_{DECRYPT(l\_linestatus)=\text{'O'})}$ (iv). Server $\leftarrow$ SCPU transfer (v). $_{l\_shipmode}\tau$ (vi). $_{DECRYPT(l\_discount)}\tau$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$ $\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$ $\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$ $\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$ $\frac{\Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l \cdot 1000}{\eta_s} \cdot \delta_c$ $2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda$, $F = \left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$	0.68
	B	(i). $\sigma_{l\_shipdate>\text{'1998-10-01'}}$ (ii). Server $\rightarrow$ SCPU transfer (iii). $\sigma_{DECRYPT(l\_linestatus)=\text{'O'})}$ (iv). Server $\leftarrow$ SCPU transfer (v). $_{l\_shipmode, DECRYPT(l\_discount)}\tau$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$ $\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$ $\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$ $\left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil \cdot \lambda$ $2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda$, $F = \left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$	1.29

and the time for the same sort from within the SCPU as

$$2 \cdot \frac{\varphi_r \cdot \tau_r}{1024 \cdot \rho} \cdot \left(\log_{\frac{\Upsilon_t \cdot 1024}{\rho} - 1} \frac{\varphi_r \cdot \tau_r}{1024 \cdot \rho} \right) \cdot (\phi_s + \lambda) \quad (9)$$

Here, φ_r is the total number of tuples in r and τ_r is the size of an individual tuple.

3.1.5 Plan Evaluations

Case I: Group-By on Public Attribute.
Figure 7(a) shows two alternative plans for a Group-By operation on the public attribute $l\_shipmode$. The difference between the two plans (A & B) is whether the grouping is performed by the server or the SCPU. If it is performed by the server, then the cheap server cycles are utilized. However, if done within the SCPU the SCPU$\rightarrow$Server data transfer is reduced, the reduction depending upon the number of distinct values of $l\_shipmode$. Table 4 shows the computation of the execution times of both plans A & B and the actual estimation. It is observed that under the parameters and System catalog data from figures 5-6 and tables 1-3 it is more efficient to perform the Group By on the server. This is because the selection on $l\_shipdate$ has high selectivity and minimizes the data transfer cost. If this selection operation had low selectivity then aggregation within the SCPU would be less expensive.

Case II: Order-By on Public and Private Attributes.
If an Order-By clause has a public attribute followed by a private attribute the server can first order the intermediate results on the public attribute leaving the private ordering to the SCPU (figure 7(b) - Plan A). Or, the SCPU can process the entire Order-By clause (figure 7(b) - Plan B). Under the specific data statistics Plan A is preferred. The reason for this is that in Plan A at one time the SCPU has to order tuples having the same value for the attribute $l\_shipmode$. The size of this intermediate result being small the sort operation is more efficient. Note that the SCPU employs external sorting. Hence, any reduction in the size of intermediate results directly lowers the Server$\leftrightarrow$SCPU transfer cost.

Case III: Distinct clause on Public and Private Attributes.
A distinct clause can be processed either by using a hash table or by first sorting the input [24]. Here we analyze the later approach. Similar to the Order-By case the first option (figure 7(c) - Plan A) here is for the server to sort the intermediate results on the public attribute $l\_shipmode$ and then have the SCPU process the distinct clause. The second option (figure 7(c) - Plan B) is for the SCPU to sort and process the distinct entirely. As seen from table 5 the optimizer prefers Plan A. The number of distinct values of $l\_shipmode$ play a critical role in plan selection here. In the dataset $l\_shipmode$ has a high number of distinct values which means that after the sort on server, the SCPU only operates on a small portion of data at a time. A small number of unique $l\_shipmode$ values instead, would favor plan B since the advantage of sorting on the server will be reduced.

Case IV: Projections.
When the number of projected attributes in a query is high the server need not transfer all these attributes to the SCPU for evaluation of private selection operations (figure 7(d) - Plan B). The alternative (figure 7(d) - Plan

TABLE 5
Cost computations for query plans shown in figures 7(c) and 7(d) corresponding to cases III-IV.

Case	Plan	Step	Execution Time ($\mathcal{ET}$)	Total $\mathcal{ET}$ (s)
(III) Distinct on Public and Private attributes	A	(i). $\sigma_{l\_shipdate>\text{'1998-10-01'}}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$	0.23
		(i). $_{l\_shipmode}\tau$	$2 \cdot \frac{\varphi_l \cdot \Theta_{l_{sd}} \cdot \tau_l}{1024 \cdot \rho} \cdot \left(\log \frac{\Upsilon_s \cdot 1024}{\rho} - 1 \frac{\varphi_l \cdot \Theta_{l_{sd}} \cdot \tau_l}{1024 \cdot \rho} \right) \cdot \phi_s$	
		(iii). $\text{Distinct}_{l\_shipmode, DECRYPT(l\_discount)}$	$\upsilon_{l_{sm}} \cdot 2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda$, $F = \left\lceil \frac{\kappa_{l_{dc}} \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$	
		(iv). Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{sm}} + \kappa_{l_{dc}}) \cdot \upsilon_{l_{sm}} \cdot \upsilon_{l_{dc}}}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
	B	(i). $_{l\_shipmode, DECRYPT(l\_discount)}\tau$	$2 \cdot \frac{\varphi_l \cdot \tau_l}{1024 \cdot \rho} \cdot \left(\log \frac{\Upsilon_t \cdot 1024}{\rho} - 1 \frac{\varphi_l \cdot \tau_l}{1024 \cdot \rho} \right) \cdot (\phi_s + \lambda)$	0.20
		(iii). $\text{Distinct}_{l\_shipmode, DECRYPT(l\_discount)}$	$2 \cdot F \cdot log_{\mu_t - 1} F \cdot \lambda$, $F = \left\lceil \frac{(\kappa_{l_{dc}} + \kappa_{l_{sm}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho * 1024} \right\rceil$	
		(iii). Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{sm}} + \kappa_{l_{dc}}) \cdot \upsilon_{l_{sm}} \cdot \upsilon_{l_{dc}}}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
(IV) Projections	A	(i). $\sigma_{l\_shipdate>\text{'1998-10-01'}}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$	0.81
		(ii). Server $\rightarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{sd}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
		(iii). $\sigma_{DECRYPT(l_{linestatus}=\text{'O'})}$	$\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$	
		(iv). Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{qt}} + \kappa_{l_{dc}} + \kappa_{l_{sm}} + \kappa_{l_{sd}} + \kappa_{l_{ls}}) \cdot \Theta_{l_{sd}} \cdot \Theta_{l_{ls}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
	B	(i). $\sigma_{l\_shipdate>\text{'1998-10-01'}}$	$\log_\theta \varphi_l \cdot \phi_s + \frac{\Theta_{l_{sd}} \cdot \varphi_l}{\omega_l} \cdot \phi_s$	0.32
		(ii). Server $\rightarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{ls}} + \kappa_{l_{ok}} + \kappa_{l_{ln}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
		(iii). $\sigma_{DECRYPT(l\_linestatus=\text{'O'})}$	$\Theta_{l_{sd}} \cdot \varphi_l \cdot \epsilon_a$	
		(iv). Server $\leftarrow$ SCPU transfer	$\left\lceil \frac{(\kappa_{l_{ok}} + \kappa_{l_{ln}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil \cdot \lambda$	
		(v). $\bowtie_{l\_orderkey=okey, l\_linenumber=lnum}$	$\left((2 \cdot F \cdot log_{\mu_t - 1} F) + \frac{\Theta_{l_{ls}} \cdot \varphi_l}{\omega_l} \right) \cdot \phi_s$, $F = \left\lceil \frac{(\kappa_{l_{ok}} + \kappa_{l_{ln}}) \cdot \Theta_{l_{sd}} \cdot \varphi_l}{\rho \cdot 1024} \right\rceil$	

A) is to pass all attributes to the SCPU and then back to the server after the selection operation within the SCPU is performed. However, in Plan B the server needs to perform additional lookups to locate the corresponding projected attributes for each tuple in the result set. To optimize the final lookup join, the server first sorts the intermediate results received from the SCPU on the tuple primary key. This greatly reduces the disk operations thereby making plan B more efficient.

4 Experiments

Setup. The SCPU of choice is the IBM 4764-001 PCI-X with the 3.30.05 release toolkit featuring 32MB of RAM and a PowerPC 405GPr at 233 MHz. The SCPU sits on the PCI-X bus of an Intel Xeon 3.4 GHz, 4GB RAM Linux box (kernel 2.6.18). The server DBMS is a standard MySQL 14.12 Distrib 5.0.45 engine. The SCPU DBMS is a heavily modified SQLite custom port to the PowerPC. The entire TrustedDB stack (figure 4(a)) is written in C.

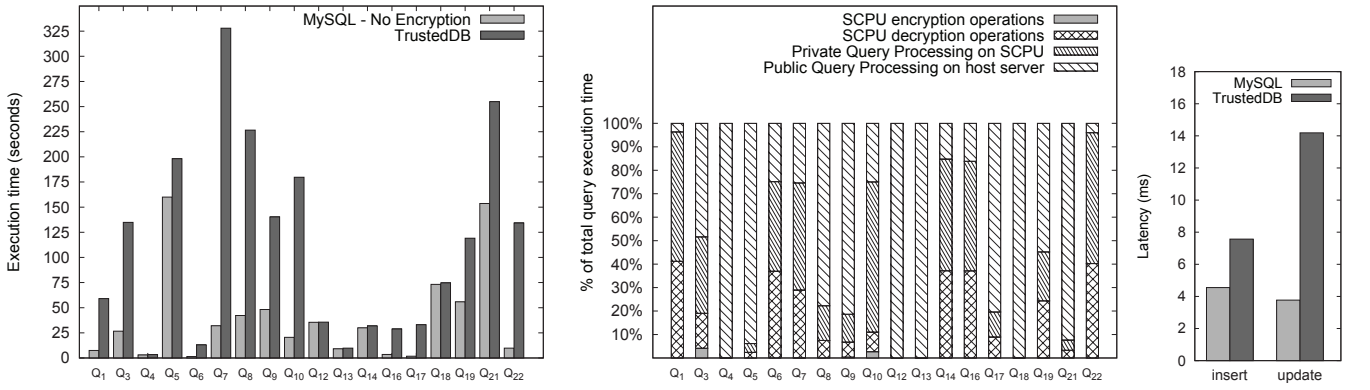
TPC-H Query Load. To evaluate the runtime of generalized queries, we chose several queries from the TPC-H set [2] of varying degrees of difficulty and privacy. The TPC-H scale factor is 1 i.e, the database size is $1GB$.

Figure 8(a) shows the execution times compared to a simple un-encrypted MySQL setup. Figure 8(b) also depicts the breakdown of times spent in execution of the public and private sub-queries. The execution times of private queries include the time required for encryption and decryption operations inside the SCPU. The public queries executed on the host server also include the processing times to interface the TrustedDB stack with the server database engine and output the final results.

As can be seen, when compared with the completely unsecured baseline scenario, security does not come cheap with execution times being higher by factors between 1.03 and 10. These factors benefit from TrustedDB's leveraging of the untrusted server's CPU for non-sensitive query portions. However, recall from section 2 that the actual costs are orders of magnitude lower than any solution based on software-only cryptography on legacy server hardware.

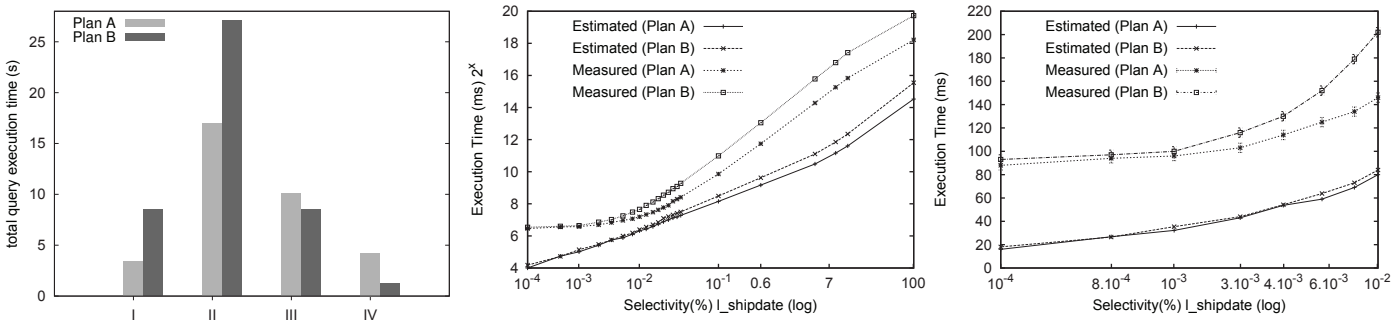
Updates. Figure 8(c) shows the latencies for insert and update statements. The reported times are for a random insert/update of a single tuple in the lineitems relation averaged over ten runs.

Query Optimization. In section 3.1 we presented different query plans, analyzed their execution and showed how the optimizer computed their execution times. Tables 4 and 5 summarized the theoretical costs and estimated execution times. To verify whether the plans selected in each of the cases is indeed the best plan we executed each of the plans on the TPC-H dataset and measured their execution times. The results are in figure 9(a). We find that in each of the cases (I-IV) the following holds. If $\mathcal{ET}_{est}(P_A) > \mathcal{ET}_{est}(P_B)$ in table 4 or 5 then $\mathcal{ET}_{real}(P_A) > \mathcal{ET}_{real}(P_B)$ in figure 9(a). For a more detailed evaluation, we compare the estimated and measured times for varying selectivity of the public attribute $l\_shipdate$ in case I. Note that this selectivity directly influences the amount of Server$\leftrightarrow$SCPU data transfer and thus the overall processing costs. As seen in figure 9(b) the optimizer correctly estimates which plan would have lower execution time for most of the cases. Figure 9(c) shows the results for very low selectivity of $l\_shipdate$. At low selectivity the accuracy of estimation lowers. There are two reasons for this (a) The measured



(a) TPC-H query execution times. (b) Query time profiles. (c) DML ops.

Fig. 8. TPC-H query execution times, time profiles and latencies for DML operations. Q_i = i^{th} query from TPC-H [2].



(a) Optimization Cases I - IV. (b) Varying selectivity for case I. (c) Low selectivity for case I.

Fig. 9. Measured execution times for optimization cases I-IV from section 3.1 and with varied selectivity for Case I.

times vary by $\pm$3.5ms between runs. Thus when the estimated times for two plans differ by $<$3.5ms they are practically equivalent. (b) The optimizer assumes a uniform distribution of attribute values. For the TPC-H data this does not hold especially at low selectivity. The accuracy of estimation in this case can be increased by simply populating the System Catalog with more accurate information.

5 Discussion

Security. Data Encryption is only one of the links in a chain of trust that ensures the security of TrustedDB. The other aspects of encryption granularity and custom-cipher design are discussed in prior work [39]. In addition, several other assurances are necessary. Clients need to be confident that (i) the remote SCPU was not tampered with, (ii) the SCPU runs the correct TrustedDB code stack, OS and firmware, and (iii) the client-SCPU communication is secure.

(i) is assured by the tamper-resistant construction of the SCPU which meets the FIPS 140-2 level 4 [1] physical security requirements. In the event of SCPU tamper detection, sensitive memory areas containing critical secrets are automatically erased. (ii) is ensured by deploying the SCPU *Outbound Authentication* (OA) [37] mechanisms. (iii) is achieved by deploying public-private key cryptography in key messaging stages. Both, client and the SCPU possess a public-private key pair (Figure 4(a)). Not unlike HTTPS/SSL communication, messages sent between the client and the SCPU are encrypted. Thus, despite acting as a communication conduit between the client and the SCPU, the server cannot perform man-in-the-middle attacks and gain access to sensitive data. Since full details are significantly more complex and out of scope, we refer the reader to [37], [39] for the detailed OA concepts.

Scalability. In an outsourced environment it is often desired that multiple clients access the database simultaneously. We note that a single SCPU is not sufficient to handle the workload in such a scenario. However, the extension of TrustedDB to utilize multiple SCPUs is straightforward using the following steps. (a) Physical installation of additional SCPUs along with the TrustedDB codebase, and (b) Importing the database schema and encryption keys securely in to the new SCPU from an existing SCPU or client.

Secure SCPU$\leftrightarrow$SCPU or client$\leftrightarrow$SCPU channels needed by step (b) can be easily setup using the Outbound Authentication mechanisms illustrated in [39]. Steps (a) and (b) are sufficient since they cover the entire information required by an SCPU to provide the desired functionality. Note that no changes are required to the data stored on the host server. Also, a single SCPU serves multiple clients without compromising security since the entire code stack within the SCPU (firmware to TrustedDB application) is verifiable by clients (at any time) using the OA mechanism [39].

Key Management. So far we have considered a single

data encryption key shared between the SCPU and client(s). In a multi-client scenario it may be desired to have multiple distinct client-SCPU keys for either access control or increased security in case one or more clients are compromised. The extensions to handle such a scenario are also simple. The data stored on host server disk can be encrypted using a single master encryption key known only to the SCPU. Since all update/insert operations are performed by the SCPU the master key is stored within the SCPU and is never communicated to the outside. Now, all decryptions required as part of query processing use the master key. Only when a sensitive attribute value is to be communicated to the client the SCPU encrypts it using the specific client-SCPU encryption key. This way only the authorized client can access the data. This design is possible since the SCPU has specialized battery backed memory dedicated for the purpose of key storage. The available space (128 KB for the 4764) enables the storage of up to 8K keys assuming a 128 bit key size. For larger key space client keys can be generated from the master encryption key using techniques similar to the key construction in [39].

Data Compression. Although at first glance, using compression along with encryption may seem promising, in the case of TrustedDB compression does not offer any advantages. This is because TrustedDB uses very fine grained attribute level encryption. Since individual attributes (ids, names, etc) are inherently small in size the compressed data will likely be the same size as uncompressed data thereby undermining the advantages of using compression. Instead, if record or page level compression is used then processing over public attributes can no longer be done server-side thereby degrading performance.

Limitations. The TrustedDB query parser does not yet support parsing of multi-level nested sub-queries and user defined views.

6 Related Work

Queries on Encrypted Data. Hacigumus et al. [20] propose division of data into secret partitions and rewriting of range queries over the original data in terms of the resulting partition identifiers. This balances a trade-off between client and server-side processing, as a function of the data segment size. In [21] the authors explore optimal bucket sizes for range queries.

[12] proposes using tuple-level encryption and indexes on the encrypted tuples to support equality predicates. The main contribution here is the analysis of attribute exposure caused by query processing leading to two insights. (a) the attribute exposure increases with the number of attributes used in an index, and (b) the exposure decreases with the increase in database size. Range queries are processed by encrypting individual $B^+ - Tree$ nodes and having the client, in each query processing step, retrieve a desired encrypted $B^+ - Tree$ node from the server, decrypt and process it. However, this leads to minimal utilization of server resources thereby undermining the benefits of outsourcing. Moreover, transfer of entire $B^+ - Tree$ nodes to the client results in significant network costs.

[44] employs *Order Preserving* encryption for querying encrypted xml databases. In addition, a technique referred to as splitting and scaling is used to differ the frequency distribution of encrypted data from that of the plain-text data. Here, each plain-text value is encrypted using multiple distinct keys. Then, corresponding values are replicated to ensure that all encrypted values occur with the same frequency thereby thwarting any frequency-based attacks.

[45] uses a salted version of IDA scheme to split encrypted tuple data amongst multiple servers. In addition, a secure $B^+ - Tree$ is built on the key attribute. The client utilizes the $B^+ - Tree$ index to determine the IDA matrix columns that need to be accessed for data retrieval. To speed up client-side processing and reduce network overheads it is suggested to cache parts of the $B^+ - Tree$ index client-side.

Vertical partitioning of relations amongst multiple untrusted servers is employed in [15]. Here, the privacy goal is to prevent access of a subset of attributes by any single server. E.g., {Name, Address} can be a privacy sensitive access-pair and query processing needs to ensure that they are not jointly visible to any single server. The client query is split into multiple queries wherein each sub-query fetches the relevant data from a server and the client combines results from multiple servers. [4] also uses vertical partitioning in a similar manner and for the same privacy goal, but differs in partitioning and optimization algorithms. TrustedDB is equivalent to both [15], [4] when the size of the privacy subset is one and hence a single server suffices. In this case each attribute column needs encryption to ensure privacy [10]. Hence [15], [4] can utilize TrustedDB to optimize for querying encrypted columns since otherwise they rely on client-side decryption and processing.

[10] introduces the concept of logical fragments to achieve the same partitioning effect as in [15], [4] on a single server. A fragment here is simply a relation wherein attributes not desired to be visible in that fragment are encrypted. TrustedDB (and other solutions) are in effect concrete mechanisms to efficiently query any individual fragment from [10]. [10] on the other hand can be used to determine the set of attributes that should be encrypted in TrustedDB.

Ge et al. [16] propose an encryption scheme in a trusted-server model to ensure privacy of data residing on disk. The FCE scheme designed here is equivalently secure as a block cipher, however, with increased efficiency. [30], like [16] only ensures privacy of data residing on disk. In order to increase query functionality a layered encryption scheme is used and then dynamically adjusted (by revealing key to the server) according to client queries. TrustedDB on the other hand operates

in an un-trusted server model, where sensitive data is protected, both on disk and during processing.

Data that is encrypted on disk but processed in clear (in server memory) as in [16], [30] compromises privacy during the processing interval. In [8] the disclosure risks in such solutions are analyzed. [8] also proposes a new query optimizer that takes into account both performance and disclosure risk for sensitive data. Individual data pages are encrypted by secret keys that are managed by a trusted hardware module. The decryption of the data pages and subsequent processing is done in server memory. Hence the goal is to minimize the lifetime of sensitive data and keys in server memory after decryption. In TrustedDB there is no such disclosure risk since decryptions are performed only within the SCPU.

Aggregation queries over relational databases is provided in [19] by making use of homomorphic encryption based on Privacy Homomorphism [33]. The authors in [13] have suggested that this scheme is vulnerable to a cipher text only attack. Instead [13] proposes an alternative scheme to perform aggregation queries based on bucketization [20]. Here the data owner precomputes aggregate values such as SUM and COUNT for partitions and stores them encrypted at the server. Although this makes processing of certain queries faster it does not significantly reduce client side processing.

Ge et al. [42] discuss executing aggregation queries with confidentiality on an untrusted server. Due to the use of extremely expensive homomorphisms [28], [29] this scheme leads to impractically large costs by comparison, for any reasonable security parameter choices. This is discussed in more detail in section 2.

Above solutions are specialized for certain types of query operations on encrypted data. [12] for equality predicates, [20], [45], [44] for range predicates and [19], [42] for aggregation. In TrustedDB, all decryptions are performed within the secure confinements of the SCPU, thereby processing is done on plain-text data. This removes any limitation on the nature of predicates that can now be employed on encrypted attributes including arbitrary user defined functions. We note that certain solutions designed for a very specific set of predicates can be more efficient albeit at the loss of functionality.

Trusted Hardware. In [5] SCPUs are used to retrieve X509 certificates from a database. However, this only supports key based lookup. Each record has a unique key and a client can query for a record by specifying the key. [34] uses multiple SCPUs to provide key based search. The entire database is scanned by the SCPUs to return matching records.

[32] implements arbitrary joins by reading the entire database through the SCPU. Such as approach is clearly not practical for real implementations since it is lower bounded by the Server↔SCPU bandwidth (10 MBps in our setup).

Chip-Secured Data Access [25] uses a smart card for query processing and for enforcing access rights. The client query is split such that the server performs majority of the computation. The solution is limited by the fact that the client query executing within the smart card cannot generate any intermediate results since there is no storage available on the card. In follow-up work, GhostDB [27] proposes to embed a database inside a USB key equipped with a CPU. It allows linking of private data carried on the USB Key and public data available on a server. GhostDB ensures that the only information revealed to a potential spy is the query issued and the public data accessed.

Both [25] and [27] are subject to the storage limitations of trusted hardware which in turn limits the size of the database and the queries that can processed. In contrast TrustedDB uses external storage to store the entire database and reads information into the trusted hardware as needed which enables it to be used with large databases. Moreover, database pages can be swapped out of the trusted hardware to external storage during query processing.

In [7] a database engine is proposed inside a SCPU for data sharing and mining. The SCPU fetches data from external sources using secure jdbc connections. The entire data is treated as private with queries completely executed inside the coprocessor. We find that using the IBM 4764 for processing queries entirely within the trusted hardware module, without utilizing server cpu cycles, is up to 40x slower than traditional server query processing. This is so even when the trusted hardware has access to the local server file system using our *Paging Module* (section 3). Hence using jdbc connections as in [7] can only have higher processing overheads.

7 Conclusions

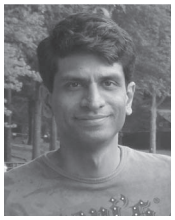
This paper's contributions are threefold: (i) the introduction of new cost models and insights that explain and quantify the advantages of deploying trusted hardware for data processing, (ii) the design and development of TrustedDB, a trusted hardware based relational database with full data confidentiality and no limitations on query expressiveness, and (iii) detailed query optimization techniques in a trusted hardware-based query execution model.

This work's inherent thesis is that, at scale, in outsourced contexts, computation inside secure hardware processors is orders of magnitude cheaper than equivalent cryptography performed on provider's unsecured server hardware, despite the overall greater acquisition cost of secure hardware. We thus propose to make trusted hardware a first-class citizen in the secure data management arena. Moreover, we hope that cost-centric insights and architectural paradigms will fundamentally change the way systems and algorithms are designed.

References

[1] FIPS PUB 140-2, Security Requirements for Cryptographic Modules. Online at http://csrc.nist.gov/groups/STM/cmvp/standards.html#02.

[2] TPC-H Benchmark. Online at http://www.tpc.org/tpch/.
[3] IBM 4764 PCI-X Cryptographic Coprocessor. Online at http://www-03.ibm.com/security/cryptocards/pcixcc/overview.shtml, 2007.
[4] Gagan Aggarwal, Mayank Bawa, Prasanna Ganesan, Hector Garcia-Molina, Krishnaram Kenthapadi, Rajeev Motwani, Utkarsh Srivastava, Dilys Thomas, and Ying Xu 0002. Two can keep a secret: A distributed architecture for secure database services. In *CIDR*, pages 186–199, 2005.
[5] Alexander Iliev and Sean W Smith. Protecting Client Privacy with Trusted Computing at the Server. *IEEE, Security and Privacy*, 3(2), Apr 2005.
[6] Mihir Bellare. New proofs for nmac and hmac: Security without collision-resistance. pages 602–619. Springer-Verlag, 2006.
[7] Bishwaranjan Bhattacharjee, Naoki Abe, Kenneth Goldman, Bianca Zadrozny, Chid Apte, Vamsavardhana R. Chillakuru and Marysabel del Carpio. Using secure coprocessors for privacy preserving collaborative data mining and analysis. In *Proceedings of DaMoN*, 2006.
[8] Mustafa Canim, Murat Kantarcioglu, Bijit Hore, and Sharad Mehrotra. Building disclosure risk aware query optimizers for relational databases. *Proc. VLDB Endow.*, 3(1-2):13–24, September 2010.
[9] Yao Chen and Radu Sion. To cloud or not to cloud?: musings on costs and viability. In *Proceedings of SOCC*, pages 29:1–29:7. ACM, 2011.
[10] Valentina Ciriani, Sabrina De Capitani Di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, and Pierangela Samarati. Combining fragmentation and encryption to protect privacy in data storage. *ACM Trans. Inf. Syst. Secur.*, 13(3):22:1–22:33, July 2010.
[11] Tom Denis. *Cryptography for Developers*. Syngress.
[12] Damiani E., Vimercati C., Jajodia S., Paraboschi S., and Samarati P. Balancing confidentiality and efficiency in untrusted relational dbmss. In *Proceedings of ACM CCS*, 2003.
[13] Einar Mykletun and Gene Tsudik. Aggregation Queries in the Database-As-a-Service Model. *Data and Applications Security*, 4127, 2006.
[14] Foto N. Afrati and Vinayak Borkar and Michael Carey and Neoklis Polyzotis and Jeffrey D. Ullman. Map-reduce extensions and recursive queries. In *Proceedings of EDBT*, pages 1–8. ACM, 2011.
[15] Vignesh Ganapathy, Dilys Thomas, Tomas Feder, Hector Garcia-Molina, and Rajeev Motwani. Distributing data for secure database services. In *Proceedings of PAIS*, pages 8:1–8:10, New York, NY, USA, 2011. ACM.
[16] Tingjian Ge and Stan Zdonik. Fast, secure encryption for indexing in a column-oriented dbms. In *ICDE*, 2007.
[17] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2010.
[18] O. Goldreich. *Foundations of Cryptography I*. Cambridge University Press, 2001.
[19] Bala Iyer Hakan Hacigumus and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *Database Systems for Advanced Applications*, volume 2973, pages 633–650, 2004.
[20] Hakan Hacigumus, Bala Iyer, Chen Li and Sharad Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. In *Proceedings of SIGMOD*, pages 216–227, 2002.
[21] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proceedings of ACM SIGMOD*, 2004.
[22] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2008.
[23] Murat Kantarcioglu and Chris Clifton. Security issues in querying encrypted data. In Sushil Jajodia and Duminda Wijesekera, editors, *DBSec*, volume 3654 of *Lecture Notes in Computer Science*, pages 325–337. Springer, 2005.
[24] Philip Lewis, Arthur Bernstein, and Michael Kifer. *Databases and Transaction Processing*. Addison-wesley, 2002.
[25] Luc Bouganim and Philippe Pucheral. Chip-secured data access: confidential data on untrusted server. In *Proceedings of VLDB*, pages 131–141. VLDB Endowment, 2002.
[26] Einar Mykletun and Gene Tsudik. Incorporating a secure coprocessor in the database-as-a-service model. In *Proceedings of IWIA*, pages 38–44, Washington, DC, USA, 2005. IEEE Computer Society.
[27] Nicolas Anciaux, Mehdi Benzine, Luc Bouganim, Philippe Pucheral and Dennis Shasha. GhostDB: Querying Visible and Hidden Data Without Leaks. In *Proceedings of SIGMOD*, 2007.
[28] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Proceedings of EuroCrypt*, 1999.
[29] Pascal Paillier. A trapdoor permutation equivalent to factoring. In *Proceedings of PKC*, pages 219–222. Springer-Verlag, 1999.
[30] Raluca Ada Popa, Catherine Redfield, and Nickolai Zeldovich. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of SOSP*, 2011.
[31] M. O. Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical report, 1979.
[32] Rakesh Agrawal, Dmitri Asonov, Murat Kantarcioglu, Yaping Li. Sovereign Joins. In *Proceedings of the 22nd International Conference on Data Engineering*, page 26. IEEE Computer Society, 2006.
[33] Ronald Rivest, Len Adleman and Michael Dertouzos. On data banks and privacy homomorphisms. *Foundations of Secure Computation*, 1978.
[34] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM SYSTEMS JOURNAL*, 40(3), 2001.
[35] Sai Wu and Feng Li and Sharad Mehrotra and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proceedings of CCS* , page Article 12. ACM, 2011.
[36] Sanjay Agrawal and Vivek Narasayya and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of SIGMOD*, pages 359 – 370. ACM, 2004.
[37] Sean W. Smith. Outbound authentication for programmable secure coprocessors. Online at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.58.4066.
[38] Shahram Ghandeharizadeh and David J. DeWitt. Hybrid-Range Partitioning Strategy: A New Declustering Strategy for Multiprocessor Database Machines. In *Proceedings of VLDB*, pages 481 – 492. Morgan Kaufmann Publishers Inc., 1990.
[39] Sumeet Bajaj and Radu Sion. TrustedDB: A Trusted Hardware based Database with Privacy and Data Confidentiality. In *Proceedings SIGMOD*, pages 205–216. ACM, 2011.
[40] Sumeet Bajaj and Radu Sion. TrustedDB: A Trusted Hardware based Outsourced Database Engine. *VLDB, DEMO*, 2011.
[41] Alexander Thomson and Daniel J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.
[42] Tingjian Ge and Stan Zdonik. Answering Aggregation Queries in a Secure System Model. In *Proceedings of VLDB*, pages 519–530. VLDB Endowment, 2007.
[43] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *EUROCRYPT*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.
[44] Hui Wang and Laks V.S. Lakshmanan. Efficient secure query evaluation over encrypted xml databases. In *VLDB*, 2006.
[45] Shiyuan Wang, Divyakant Agrawal, and Amr El Abbadi. A comprehensive framework for secure query processing on relational data in the cloud. In *Secure Data Management*, pages 52–69, 2011.



Sumeet Bajaj is a PhD student at Stony Brook University. He received his Masters from Stony Brook University in 2006 and his Bachelors from Pune Institute of Computer Technology in 2003. His industry experience involves building cloud services, financial trading & ERP systems. His research interests include Network Security, Databases, Distributed and Concurrent Systems.

Radu Sion is an Associate Professor in Computer Science in Stony Brook University. His research interests include Cyber Security and Efficient Computing. He builds systems mainly, but enjoys elegance and foundations, especially if of the very rare practical variety. Sponsors and collaborators include NSF, US Army, Northrop Grumman, IBM, Microsoft, Motorola, NOKIA, and Xerox.