CorrectDB: SQL Engine with Practical Query Authentication

Sumeet Bajaj
Stony Brook University
New York, USA
sbajaj@cs.stonybrook.edu

Radu Sion
Stony Brook University
New York, USA
sion@cs.stonybrook.edu

ABSTRACT

Clients of outsourced databases need *Query Authentication* (QA) guaranteeing the integrity (correctness and completeness), and authenticity of the query results returned by potentially compromised providers.

Existing results provide QA assurances for a limited class of queries by deploying several software cryptographic constructs. Here, we show that, to achieve QA, however, it is significantly cheaper and more practical to deploy server-hosted, tamper-proof co-processors, despite their higher acquisition costs. Further, this provides the ability to handle arbitrary queries.

To reach this insight, we extensively survey existing QA work and identify interdependencies and efficiency relationships. We then introduce CorrectDB, a new DBMS with full QA assurances, leveraging server-hosted, tamper-proof, trusted hardware in close proximity to the outsourced data.

1. INTRODUCTION

Today, all major cloud providers offer a database service of some kind as part of their overall solution. However, such services require their customers to fully trust the provider. This is often unacceptable and technology-backed security assurances are key to clients' adoption decisions.

The essential issues of data confidentiality and privacy *for semi-trusted, yet curious* providers have been tackled extensively [30, 4]. Of equal importance is the case of actively malicious or faulty service providers, the query results of which cannot be trusted without proper *Query Authentication* (QA) mechanisms [28].

Existing research tackles the QA problem by designing mechanisms that allow clients to verify remote query execution results by placing additional authentication data structures at the provider's site. At query execution, the provider sends relevant authentication information derived from these structures to the client. A set of such approaches have been proposed for individual query types such as projections and range queries (section 3).

Unfortunately, the performance of solutions based on client-side checking is necessarily limited by inherent network bandwidth and latency characteristics. Further, unifying the set of existing individual query-type-specific solutions into a universal mechanism for arbitrary queries has proven extremely challenging.

In this paper we propose to look at this problem from a different angle. Specifically, we posit that server-hosted close-to-data trusted hardware acting on behalf of clients can bring about a fundamental paradigm shift in this domain and result in a general purpose QA solution that is also significantly cheaper and more efficient than client-server software protocols. We are encouraged by previous results determining that trusted hardware is significantly cheaper than cryptographic alternatives [5] and subsequent work on-trusted hardware based DBMS for data privacy [30].

The result of this pursuit is CorrectDB, an SQL DBMS that deploys tamper proof secure co-processors at the server's side to provide full QA guarantees cheaply and efficiently, despite higher hardware acquisition costs. It achieves this through its close proximity to the outsourced data, by minimizing the authentication data, and by reducing the client-server communication overheads.

The contributions of this paper are threefold:
(i) A comparative survey of existing QA research that explores both theoretical and empirical dimensions based on published results.
(ii) A cost analysis and associated insights showing that using trusted hardware for QA is significantly more efficient, both in cost and performance as compared to existing work.
(iii) The design, development, and evaluation of CorrectDB, a trusted hardware-based DBMS providing QA assurances.

2. MODEL

2.1 Threats and Deployment

Data Owner. Data is placed by the data owner with a remote, untrusted *service provider*. For authenticity and integrity of query results the data owner computes and places additional authentication data with the provider. After the initial upload, the server-side, trusted hardware manages the authentication data on behalf of the data owner. The data owner also issues search and update queries to the provider.
Clients. Client's authorized by the data owner query the outsourced datasets through an interface exposed by the provider. A client query can either perform a search or request a data-only-update operation. Client's have no access to the server-side authentication data.

Adversary. The service provider is not trusted. Due to compromise or malicious intent, if given the possibility to get away undetected, it may attempt to compromise one or more of the QA security requirements, described next.

2.2 Query Authentication (QA) Requirements

Correctness. Correctness has two components.

First, each tuple in the query result should be authentic i.e., originate from the original database that was uploaded to the provider's site. It must be impossible for the provider to introduce any spurious tuples in the result.

Second, each tuple in the result must satisfy the query predicates exactly, thus ensuring that query execution adhered to all predicates specified in clients' queries.

Completeness. Completeness states that all tuples that are supposed to be part of the query result must be present in the result, i.e., query execution should not exclude any valid tuples from the result.

2.3 Security Components

Trusted Hardware. CorrectDB leverages trusted hardware such as the IBM 4764 [13] co-processor (SCPU) in close proximity to the outsourced data. The SCPU features tamper-resistant design [14], thus providing a secure execution environment. In the event of illicit physical handling, the device destroys its internal state and shuts down.

Cryptography. The deployed SCPUs offer several cryptographic primitives including digital signatures (RSA, DSA), and crypto hash functions (MD5, SHA).

We will denote by PK_{ALICE} a public key that belongs to Alice while SK_{ALICE} represents her private key. A signature of message M with private key K is written as $S(M, K)$. The hash of message M is denoted by $H(M)$. Concatenation of two messages M_1 and M_2 is denoted by $M_1||M_2$.

A *Merkle Hash Tree (MHT)* [19] authenticates a set relationship between multiple items such as tuples in a database. If t_1, t_2 ... t_n denote individual tuples in the dataset, the leaves of the MHT are composed of the hash of each individual tuple i.e., $H(t_1)$, $H(t_2)$ ... $H(t_n)$. Then, each internal parent node at higher levels up the tree is constructed as the hash of its two child nodes e.g., the parent node of $H(t_1)$ and $H(t_2)$ is $H(H(t_1)||H(t_2))$. To subsequently prove that a tuple belongs to the dataset associated with the MHT, it suffices to re-compute and verify the root of the tree from the tuple's value and the siblings of all the nodes in the traversal path up to the root of the MHT.

3. EXISTING WORK SURVEY

Here, we provide an overview of the important existing QA works which are summarized in Table 1. This survey serves to understand the factors that make trusted hardware more suitable for QA, and to identify the best QA solutions for range and join processing which can then be compared with CorrectDB.

Overview. Existing QA solutions follow the deployment model of section 2 with the following key difference. All database updates, including the modifications to the authentication data are performed by the data owner. Client queries are read only.

At a high level, existing solutions work similarly using the following steps. (1) The data owner uploads the database to the service provider. She then computes and uploads additional data structures known as *Authentication Data Structures* (ADS). Also, she may distribute certain information to the clients. (2) A client submits a query request to the service provider. (3) The provider executes the query to get the desired results. With the aid of the ADS, he also computes the data necessary for the client to verify both correctness and completeness of the results. This additional information is referred to as the *Verification Object* (VO). (4) The service provider delivers both the query results and the VO to the client. (5) Using the information from the owner, and the query results and VO from the service provider, the client determines whether QA assurances are met.

Name	Year	QCT	QCP	Ops	Updates
Tree Based					
MHT [8]	2003	✓	✓	S,R	×
VBT [24]	2004	✓	✓	S,R	✓
EMBT [15]	2006	✓	✓	S,R,J	✓
Singh et al [28]	2008	✓	✓	S	✓
XBT [26]	2009	✓	✓	R	✓
AIM [32]	2009	✓	✓	S,R,J	×
MRT [33]	2009	✓	✓	S_p	✓
AABT [16]	2010	✓	✓	A	✓
MR-SKY [17]	2011	✓	✓	S_p	×
Yang et al [34]	2011	✓	✓	R	×
Signature Based					
AGS [21, 20]	2004	✓	×	S,R	×
DSAC [22]	2005	✓	✓	S,R	✓
Pang et al [23]	2005	✓	✓	S,R,J	✓
VKDT,VRT [7]	2006	✓	✓	S_p	×
Pang et al [25]	2009	✓	✓	S,R,J	✓
VNA [12]	2010	✓	✓	S_p	✓
Others					
DICT [10]	2001	✓	✓	K	✓
AUDIT [31]	2007	✓	×	S,J	✓
HLT [35]	2012	✓	✓	S,R,J,A	✓

Table 1: Summary of existing approaches (QCT - Query Correctness, QCP - Query Completeness, S - Select, R - Range, J - Join (Equi,<,>), A - Aggregation, S_p - Spatial, K - Key lookup).

The properties of the ADS and the VO prevent the provider from compromising integrity of the query results.

Classification. Existing solutions can primarily be classified as either *tree-based* or *signature-based*. The two categories differ in the data structures used for the ADS and the VO, and hence in the query execution and verification.

In tree-based approaches, the ADS is constructed as a tree (MHT, MB-Tree, VB-Tree etc). As part of query execution, the service provider traverses the tree and gathers the respective nodes that form the VO, which is sent to the client along with the query results. The client can then reconstruct the traversal path used in query execution and verify that it is indeed authentic.

Signature-based approaches provide a mechanism to verify the ordering between tuples, when using specific search attributes. To this end, an authenticated chain of unforgeably signed tuples is constructed by the data owner. At query time, the service provider gathers the signatures of all the tuples that comprise the contiguous range query result. This set of signatures comprises the VO. Since each tuple is now linked to its predecessor and successor in an unforgeable manner the client can verify that no tuple is either illicitly inserted or omitted from the query result.

3.1 Tree-based Solutions

The approach designed in [8] forms the basis for most tree-based *range query* approaches. It utilizes a MHT for query processing on the provider's site. The MHT root node is signed by the owner and distributed to the client before uploading the MHT to the provider. In response to a client query, the provider delivers the actual query results and relevant nodes from the MHT such that the client can reconstruct the root node. The client then verifies the signature on the root node and is thereby assured that the query was processed correctly.

The MHT approach can be extended and applied to B^+-Trees which are then referred to as MB-Trees (MBT). The digest for a leaf node is constructed by a hash of the concatenation of the hashes of the k tuples pointed to by it i.e., H_{li} $= H(H(t_1)||H(t_2)||...||H(t_k))$. Each non-leaf node's digest is the hash of the concatenation of the digests of each of its child nodes. Then the VO consists of all the additional node digests required for the client to reconstruct and verify the digest of the root node of the tree.

The first approach to deploy such a B^+-Tree is known as Verifiable B-Tree or VBT [24] and considers an edge computing model, wherein all digests are computed, signed, and later updated by a trusted central server and then distributed to the edge servers. The VO then consists of all the authenticated nodes up to the root node of the subtree that do not envelope the tuples present in the query result. The client can construct this sub-tree and verify the signature on its root. The optimization here lies in the fact that the VO need not contain the path up to the root of the entire B^+-Tree, like in MHT.

An MB-Tree based ADS is also used in [15]. Here, each individual node of the B^+-Tree in turn stores an embedded MB-Tree and this is thus known as an Embedded Merkle B-Tree (EMBT). The embedded tree aids in quickly constructing the composite hash of the node being traversed as part of query execution thereby reducing the number of B^+-Tree read operations in constructing the VO.

A standard MHT is used in [28] in a slightly different context. Here, the provider periodically commits the state of the database to the client and QA is performed only against the last committed version. This provides weaker security than other approaches but allows more frequent updates. The ADS in this case is a MHT and the hash for a leaf node is computed as $H(id||H(A_{i1}||S(A_{i1}, SK_{DO})$ ... $H(A_{ik}||S(A_{ik}, SK_{DO}))$ where, id is a unique identifier for the tuple, $A_{i1}, ...A_{ik}$ are the tuple attribute values, and SK_{DO} is the secret signing key of the data owner. Support for projections is added by including the tuple attribute values in the composite hash.

For *join processing*, straight-forward approaches have been proposed that materialize the entire cross product and then build the ADS on it. This is inefficient both in terms of storage and update operations. [23] extends range query authentication to joins as follows. Consider the relations R_1, R_2, and the join query $R_1 \bowtie_{R_1.A=R_2.B} R_2$. Then the data owner constructs an ADS on both $R_1.A$ and $R_2.B$. Also, assume that R_1 is smaller. First the provider sends R_1 to the client along with the authentication data for R_1. Then, for each tuple in R_1 the provider performs a range query on R_2 to find the matching tuples. The VO resulting from each such range query is appended to construct the VO for the entire join query results.

The first comprehensive work on QA for joins can be found in [32]. The most efficient solution proposed here is Authenticated Index Merge Join (AIM). Again, consider the join query $R_1 \bowtie_{R_1.A=R_2.B} R_2$, and that ADSs exists on both $R_1.A$ and $R_2.B$. Each ADS used is an MBT (discussed earlier). Now, for the first tuple in R_1, the provider performs an index traversal and leaf scan on the ADS for R_2 to find the boundary tuples which are included in the VO. Then, at each step the roles of R_1 and R_2 are reversed and additional boundary tuples are included. The detailed algorithm is involved and we refer the reader to [32].

A solution for authenticating *aggregation queries* (SUM, COUNT etc) using a tree-based ADS is proposed in [16]. This ADS is referred to as the Authenticated Aggregation B-Tree (AABT). In a AABT each node stores the aggregated sum α of its child nodes on the value of the search attribute i.e., $\alpha = \alpha_1 + \alpha_2 + ... + \alpha_k$ where α_1, α_2 ... α_k are the aggregated values of the individual child nodes (η_1, η_2 ... η_k). In addition a node stores the hash $H(\eta_1||\alpha_1||...||\eta_k||\alpha_k)$ which is included in the VO. All other solutions do not authenticate aggregate operations, but transfer the relevant data to the client for client-side aggregation.

Key Insights. Transferring additional ADS nodes to the clients for verification means that the VO sizes in tree-based approaches can become quite large. This in turn increases both query latencies and the cost of data transmission. As shown in [6] cloud to client transfer can cost upwards of 3500 picocents/bit, 2-3 orders of magnitude higher than processing costs (1 US picocent = $\$1 \times 10^{-14}$).

3.2 Signature-based Solutions

A straight-forward approach [24, 22] using signatures for *query correctness*, but not for *completeness*, is for the data owner to sign individual tuples before uploading to the provider. The client can then verify the signatures on the tuples in the query result. We refer to this as NBS.

The first approach to use signatures [21] (appeared in 2004, published in 2006) for *range* QA addresses only query correctness. Here signature aggregation [21, 20] is used to combine multiple tuple signatures into a single signed message thereby resulting in a small, constant sized VO .

[22] extends [21] to provide completeness for *range* QA. While in [21] the ADS consisted of the signature of the hash of each individual tuple i.e., $S(H(t_i), SK_{DO})$, in [22] the ADS consists of the signature of each individual tuple along with its immediate predecessor i.e., $S(H(t_i)||H(t_{i-1}), SK_{DO})$, where t_{i-1} is the predecessor of t_i when sorted on the search attribute. By including the predecessor in the signature, a chain of all tuples is formed, ordered on the search attribute. In effect the client verifies that the set of tuples received in the result do form a valid chain. The scheme is also applicable to multiple search attributes.

Simultaneously, [23] devised a signature-based scheme for *range* and *join* QA, to overcome the limitation of [8], where two additional range boundary tuples are revealed to the client, potentially causing a violation of access control mechanisms. Here, the ADS consists of the signature of each individual tuple along with its immediate predecessor and successor i.e., $S(H(t_{i-1})||H(t_i)||H(t_{i+1})), SK_{DO})$. Signature aggregation can also be applied here to reduce VO size.

Signature-based schemes are inefficient for *join processing* as they result in large VO sizes. Again, consider the join query $R_1 \bowtie_{R_1.A=R_2.B} R_2$. For each tuple in R_1 present in the result, the VO contains the boundary tuples for the matching tuple in R_2. Also, for each tuple in R_1 that is not part of the result we need a proof that no matching tuple exists in R_2. Again, the VO will contain two boundary tuples showing that no such tuple in R_2 exists. The resulting VO thus becomes large.

Key Insights. The VO for signature-based solutions that employ signature aggregation is a single signed message. Hence, they do not incur the high transmission costs of tree-based approaches, at least not for range queries. However, the operations needed to construct such small VO*s* are expensive in computation on the server side, since computing even a single cryptographic trapdoor requires a high number of CPU cycles costing up to 30,000 picocents [6, 30].

3.3 Update Operations

Providing QA complicates update operations on the outsourced database, since changes made to the database tuples involve modifications in the related ADS. Hence, some of the works from previous sections, either do not address update operations [20, 28] or assume fairly static/infrequently updated databases [8, 18, 22, 21]. Moreover, only the data owner is permitted to perform all update operations.

For tree-based approaches, the most straight-forward way to update database tuples is for the data owner to re-upload the new, modified tuples along with the entire ADS path from the leaf node to the root. Then, the data owner can compute a new signature for the root node and re-distribute it to the clients. This approach clearly becomes inefficient in case of a large number of clients querying the database.

Also, in this case, it is important to avoid replay attacks wherein a service provider by using old signatures can provide stale results to clients. To handle this, the data owner can lock the database and then make the relevant changes. This however means that the database can only be updated intermittently as in [24].

Batch updates are suggested in [15]. Updates to tuples which are close together i.e., lie on same/adjacent leaf nodes, can be performed in a single batch. Since these adjacent leaf nodes would share nodes from the path to the root node the number of ADS tree nodes modified would be small.

[8, 24, 22] suggest using materialization of the entire cross product to compute join queries. This further complicate update operations, since a minor change to a single tuple can potentially affect a large number of join tuples.

For signature-based approaches, any update to a tuple requires re-computing the signed digests for its neighboring tuples. A concrete update mechanism to achieve this is given in [22]. Here, a multi-round protocol is employed between the owner and the provider to perform data manipulation operations on individual tuples. This can also be used for [23]. However, this is vulnerable to replay attacks.

3.4 Empirical Evaluation

QA performance entails the following key metrics.

VO size (VOS). The VO is transferred over the network and thus determines query latency and bandwidth usage. Hence, all solutions target to minimize the VO size.

Query Execution Time (QET). The provider builds VO by computing on the ADS. This computation adds to the overall execution time.

Verification Time (VT). Some client-side computation is required to verify the authenticity of query results. Since client(s) may be limited in computing abilities it is imperative that this processing is minimal.

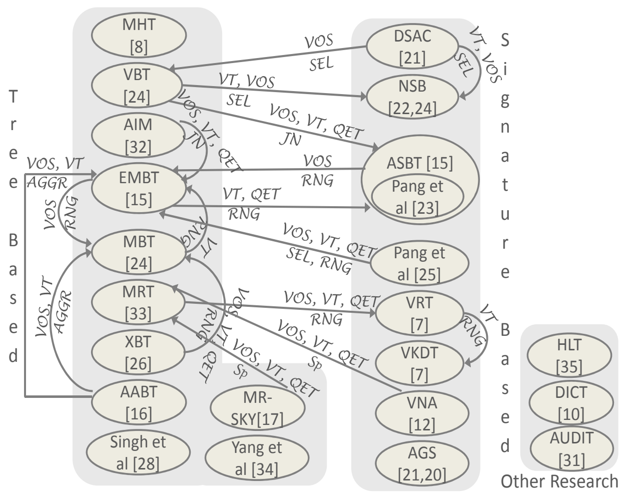


Figure 1: Comparison of published results. Metrics: VOS = Verification Object (VO) Size, VT = Client-side Verification Time, QET = Server-side Query Execution Time. Queries: SEL = Selection, AGGR = Aggregation, RNG = Range, JN = Join. A $\longrightarrow$ B = A outperforms B for the metrics denoted above the arrow and queries denoted below. Nodes with no edges = no experimental comparisons available.

Over time, newer solutions have shown their benefits over existing prior work by experimental comparisons. Looking at all of the published experimental results we can draw a map as to which solutions perform better than others and thus identify the most efficient QA solutions. Figure 1 shows a comparative summary of existing research.

3.5 Summary

For selection and range queries, signature-based schemes using aggregation [22] provide the smallest VO and thus have the least client-server network overhead. However, tree-based solutions [15] in 2006 have been shown to have an advantage over signature-based schemes when evaluated on both QET and VT. This changes again in 2009 with the speedup in crypto operations [25]. Note that crypto operations such as signature verification and aggregation are CPU but not I/O intensive. Faster processors in 2009 reverse the 2006 result and signature-based schemes now perform better on the metrics QET and VT as well. Signature-based schemes are still expensive for join processing due to the large VO sizes. AIM [32] is the only comprehensive approach for join processing and is shown to perform better than other tree-based approaches.

4. THE CASE FOR TRUSTED HARDWARE

We now discuss the benefits of SCPUs for QA. In section 5 we describe how they are achieved in CorrectDB.

Data Proximity. One of the major performance and cost considerations for QA has to do with VO sizes that need to be transmitted from the provider to the client with each query execution. This is applicable specifically to tree-based solutions. Since SCPUs communicate with the host server locally over the PCI bus they virtually eliminate these high transmission costs. Later, in section 5.5 we show that despite the higher acquisition and processing costs of SCPU the data proximity factor leads to significant savings in overall processing costs.

Query Expressiveness. The general-purpose SCPUs can be programmed to execute arbitrary queries. Thus, limitations on the query expressiveness can now be removed and a single solution utilizing SCPUs can be used for authenticating range, join and aggregation queries even with complex predicates.

Database Updates. In existing QA solutions, since there is no trusted component server-side the data owner cannot simply issue an update query to the server. Instead, the data owner is relied on entirely to perform all update operations. Hence, if the data owner does not retain a local copy of the database it needs to fetch the relevant tuples from the server, modify them locally, construct new ADS, and then upload the new tuples and ADS back to the server. This adds significant data transfer overheads. In addition, tree-based approaches require re-distribution of the ADS root hash to all clients. In the scenario using SCPUs, the data owner can issue an update query directly to the SCPU. All updates are then performed by the SCPU incurring no additional data transfer overheads (section 5.7). Also, the ADS can now completely be stored server-side. Hence, if the query result verification is also performed using the SCPU, the re-distribution to clients is avoided.

Access Control. If the server environment is untrusted to provide correct and complete query results, it may be argued that it should also not be trusted to enforce access control policies. Access control can be efficiently executed and enforced within the SCPU.

Untrusted Clients. Each update operation involves both modifying the actual data and the ADS. In existing QA solutions updates are limited to the data owner only. If clients were permitted to perform data updates, it would be necessary to give the clients access to the ADS as well. Hence, a compromised client could alter the ADS causing incorrect results to be computed for other clients' queries.

In a trusted hardware-based solution, the SCPU acts as a trusted entity on behalf of the owner and performs all updates server-side. Clients can now issue update queries, but the underlying data and ADS are modified only by the server-side SCPU. Further, client update queries can be filtered by SCPU enforced access control mechanisms, thereby avoiding malicious updates.

Client Synchronization. If clients are storing any authentication information such as the root hash in tree-based approaches, any change to the authentication data involves updates on all clients in synchronization, lest some clients end up with stale data. If the number of clients is large this problem becomes acute. Using a SCPU-based design avoids such synchronization issues.

Replay Attacks. To prevent replay attacks, where the server sends old authentication data to clients, tree-based approaches require some way of locking the database while the data owner recomputes the root hash of the ADS. Further, the owner is required to securely distribute the new root to all clients. Signature-based schemes are inherently vulnerable to replay attacks. By computing the ADS locally during updates, SCPU-based designs can avoid replay attacks entirely in an efficient manner (section 5.7).

Querying without ADS. In existing work queries with predicates on attributes that do not have any associated ADS require the intermediate results to be transmitted for client-side evaluation. This incurs significant data transmission costs. By comparison, server-side SCPUs can leverage

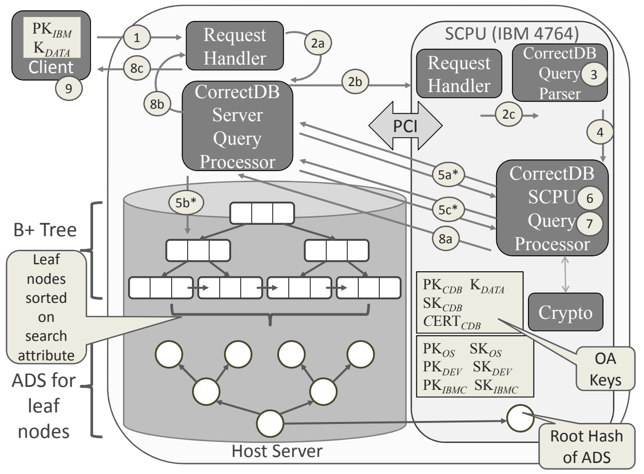


Figure 2: CorrectDB Architecture

Step	Description	Details in
1	Client submits query	
2,3	Forward client query	
4	Parse into server & scpu-side sub-queries	section 5.1
5	Forward parsed queries	
6*	Request leaf nodes from server	sections 5.2 to 5.5
7*	Find leaf + MHT nodes	
8*	Request MHT nodes from server	
9*	Verify leaf nodes + process query	
10	Sign digest of query results (VO)	
11,12	Forward results + signed VO	
13	Client-side verification	section 5.6

Table 2: Legend for figure 2, * = multi-round steps.

ADS on attribute(s) other than the search attribute(s). This is specifically applicable for processing joins (section 5.4).

Data Privacy. For privacy data can be encrypted before deployment to the provider. However, encryption greatly limits the query predicates to very simple conditions [9, 11]. Within a SCPU however, data can be processed in plaintext and complex predicates can be evaluated over it (section 7).

Regulatory Compliance. Secure code execution within SCPUs can be easily augmented with a "Compliance Module" to support regulatory compliance. Policies that regulate data can be communicated to and enforced by the SCPU securely at runtime without additional costs. E.g., enforcement of tuple-level retention times.

5. CORRECTDB ARCHITECTURE

CorrectDB is built around a set of core components (figure 2) including a *request handler*, a *query parser*, server and SCPU side *query processors*, and a *crypto library*. Following are some of the key details.

Overview. The outsourced data is stored at the provider's site. For each relation R in the database a B^+-Tree is built on the search attribute(s)/key(s) of R. In addition, a separate Merkle-Hash tree (MHT) based ADS is built on the leaf nodes of each of the B^+-Trees. Each leaf of the ADS is a hash of the entire contents of a single leaf node of the corresponding B^+-Tree. Figure 3 shows how the ADS is constructed from the corresponding B^+-Tree. The root hash of each ADS is stored inside the SCPU and is never accessible from the outside. This avoids having to digitally sign each root node hash, thereby saving a signature operation on each update, and multiple signature verifications on each query.

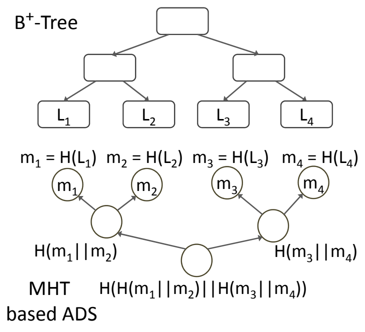


Figure 3: MHT based ADS.

Note that for static data sets both the B^{+}-Trees and their ADS can be constructed by the data owner prior to uploading the database. However, since CorrectDB supports insert and update queries these structures are continuously updated by the server-side SCPU in response to client update queries.

Query processing proceeds as follows (figure 2): (1) A client queries the server through a standard SQL interface. (2) The server forwards the query to the Request Handler inside the SCPU which then forwards it to the CorrectDB Query Parser. (3) The client query is parsed and re-written into two sub-queries: a server query and a SCPU query. (4) The parsed queries are forwarded to the CorrectDB query processor. (5) The server sub-query is executed on the untrusted server host and its results are validated by the SCPU by using the MHT ADS. (6) The SCPU sub-query is processed on the results of the server sub-query, within the SCPU, to get the final results. (7) The SCPU signs the final query result. (8) The signed result is then sent to the client. (9) The client verifies the signature.

5.1 Query Parsing and Execution

A query can be composed of various operations such as selections, ranges, projections, aggregations, group-by etc. The Query Parser's job is then to re-write the original client query into sub-queries, ensuring the following. (a) Processing within the SCPU is minimized. (b) Any intermediate results generated by server-side query processing can be validated by the SCPU using the ADS built on the leaf nodes of the B^{+}-Tree indices of the relevant relations. (c) Any operations that cannot be authenticated if executed server-side are processed on the intermediate results inside the SCPU. (d) The net result of the sub-queries is the same as if the original client query was executed without any re-writes.

To see how queries are re-written and processed consider the following query derived from TPC-H [2]:

```
SELECT sum(l_extendedprice*l_discount), o_priority
FROM   lineitem, orders
WHERE  l_shipdate >= '1993-01-01'
AND    l_shipdate < '1994-01-01'
AND    o_orderdate between '1992-01-01' AND '1993-01-01'
AND    l_discount between 0.05 AND 0.07
AND    l_orderkey = o_orderkey
AND    o_priority in ('W', 'R', 'Q')
```

Suppose we have B^{+}-Tree indices and MHT based ADS on the attributes *l_shipdate* and *o_orderdate*. Then the server-side sub-query searches for all leaf nodes from the relations *lineitem* and *orders* that satisfy the conditions

```
    l_shipdate >= '1993-01-01'
AND l_shipdate < '1994-01-01'
AND o_orderdate between '1992-01-01' AND '1993-01-01'
```

For this, the server uses the B^{+}-tree indices on the attributes *l_shipdate* and *o_orderdate*. Note that the server does not actually identify the individual tuples that satisfy these predicates but only the leaf nodes that contain the tuples which may potentially satisfy them. The server and the SCPU-side query processors then engage in a interactive protocol in which, for each round, a set of leaf nodes is first read into the SCPU, then verified for correctness and completeness using the corresponding MHT based ADS, and finally evaluated for the remainder of the query predicates:

```
SELECT sum(l_extendedprice*l_discount), o_priority
FROM   lineitem, orders
AND    l_discount between 0.05 AND 0.07
AND    l_orderkey = o_orderkey
AND    o_priority in ('W', 'R', 'Q')
```

The SCPU-side verification step ensures the authentication of the results of the server sub-query (section 5.2). Since the server sub-query filtered out unwanted tuples from the relations *lineitem* and *orders*, the SCPU sub-query only processes a subset of the data. This is essential, else the join condition *l_orderkey* = *o_orderkey* would become an expensive operation to perform entirely within the SCPU.

5.2 Range Queries

Consider the execution of a range query for all tuples with keys in the range (L, U), $L \leq U$. Let R denote the set of tuples in the query result. Let L_1, L_2,..., L_n be the leaf nodes at the lowest level of the B^{+}-Tree. Since the B^{+}-Tree stores data sorted on the search keys, the same sort order applies to the leaf nodes as well, including the ordering of tuples within a single leaf i.e., for two leaf nodes L_i and L_j where $i < j$, we have $\forall\ t \in \mathsf{L}_i$, $\forall\ t^{'} \in \mathsf{L}_j$, $t.key \leq t^{'}.key$. The server performs the search for all tuples in the range and identifies the leaf nodes L_l, L_{l+1},...L_m, where $l \geq 1$, $m \leq$ n and $l \leq m$. It then sends these leaf nodes to the SCPU query processor. The SCPU query processor computes the hash of the leaf nodes $H(\mathsf{L}_l)$, $H(\mathsf{L}_{l+1})$,...$H(\mathsf{L}_m)$. Using these hash values and by requesting additional ADS nodes from the server, the SCPU query processor constructs and verifies the root hash of the MHT. Finally, the SCPU scans the leaf nodes to find all tuples t such that $t.key \in (L, U)$, which comprise the result set R.

The following properties hold:

Correctness. (1) $\forall$ tuples $t \in \mathsf{R}$, $t.key \in (L, U)$.

Proof (sketch): The SCPU verifies evaluation of the range predicate. Correctness then reduces to the security/collision resistance of the MHT. Since the MHT based ADS is used to verify the integrity of each leaf node the server cannot alter any tuples to subvert query correctness.

Completeness. (2) $\forall\ t$, if $t.key \in (L, U)$ then $t \in \mathsf{R}$.

Proof (sketch): Completeness is violated iff, $\exists$ tuple t such that $t.key \in (L, U)$, but $t \notin \mathsf{R}$. Note that the leaf nodes L_l, L_{l+1}, ... L_m are consecutive nodes at the lowest level of the B^{+}-tree. This is easily verified by the SCPU using the MHT, since the same leaf nodes correspond to the leaves at the lowest level of the MHT. Thus, when the root hash of the MHT is constructed, this chain linking between the B^{+}-tree leaf nodes is automatically verified. Now, in addition the SCPU also checks the following. (1) $min\{t.key, t \in \mathsf{L}_l\} < L$, and (2) $max\{t.key, t \in \mathsf{L}_m\} > U$. (1) and (2) together with chain linking of consecutive leaf nodes ensure completeness. Proof then reduces again to collision resistance of the MHT.

5.3 Projections

Projection operations are performed by the SCPU query processor. Thus, no additional ADS are needed to support them. For each leaf node being processed the SCPU query processor simply discards any attributes not required for current query processing. Supporting projections in this manner enables CorrectDB to build the MHT based ADS on the contents of the entire leaf nodes of the B^+-Trees, rather than on individual tuples. This saves considerably on hash operations in intermediate query result verification In effect, for the verification of L_n leaf nodes of average size L_s KB with an average tuple size of T_s bytes, the number of hash operations required during verification (by the SCPU) are reduced from $\frac{L_n * L_s * 1024}{T_s}$ to L_n. Also, hashing entire leaf nodes enables us to utilize the SCPU's crypto-hardware engine which has high throughput for bulk operations, thereby circumventing the high setup latencies involved in hashing small data items such as individual tuples.

5.4 Joins

Join processing is a relatively straight-forward extension of range processing. CorrectDB essentially uses two separate methods for evaluating join queries, a sort-merge join for predicates with ordering-based operators such as $=$,$<$ and $>$, and a full nested join for any other arbitrary join predicates.

Equi($=$), $<$, $>$ etc joins. The same method is used for both Equi($=$) joins and $<$,$>$ join predicates. To compute and authenticate the results of the join query $\sigma_{P_r}(R) \bowtie_{R.a=S.b} \sigma_{P_s}(S)$ between the relations R and S the server uses the B^+-Trees on $R.a$ and $S.b$ to identify all the leaf pages of R and S that contain the query results. If the predicates P_r and P_s contain conditions on attributes other than $R.a$ and $S.b$ or if they are empty then this means that all leaf pages of both relations potentially contain result tuples. If P_r and P_s contain predicates only on $R.a$ and $S.b$ then the server can identify the subset of the leaf nodes of R and S by traversing the respective trees. Once the server identifies the leaf nodes, the server and SCPU-side query processors engage in a protocol where, in each round, the server-side query processor sends a set of pages from the identified leaf nodes of R and S, in order, to the SCPU. The SCPU performs a sort-merge scan of these pages and includes any identified result tuples in the final result set. Just as in range query processing the SCPU verifies both the integrity and the consecutive linking of all the leaf pages from both R and S by using the respective MHT based ADS, thereby ensuring both correctness and completeness. Projections and any additional predicates are processed by the SCPU query processor.

Arbitrary Joins. Nested loop joins are required when join operations are complex and are not computable using a sort-merge mechanism or when there are no indices available on the join attributes. Nested loop joins can also be the preferred choice when the outer relation is small. Unfortunately nested loop joins can be expensive. Suppose the two relations participating in the join have n_r and n_s number of leaf nodes respectively, each of size P. This could result in reading $n_r * n_s$ leaf nodes into the SCPU to perform the nested join operation.

This cost can be reduced by utilizing the available memory inside the SCPU (e.g., M = 32 MB for the 4764) as follows. M/2 space inside the SCPU is dedicated to hold the leaf nodes for each relation. Then the SCPU will perform a number of $n_r + (\frac{2*P*n_s}{M})^2$ node fetches to do the join. The verification procedure of section 5.2 is repeated for the leaf nodes of both relations to guarantee correctness and completeness. If an ADS is not available for a particular join attribute, we can use the ADS of another attribute of the same relation to perform the authentication within the SCPU, if the entire relation is participating in the join.

5.5 Aggregations, Grouping and Ordering

Existing tree and signature-based mechanisms require the client to perform any aggregation operations. Thus additional data, which is not part of the final query results is transferred to the client incurring both query latency and data-transfer-cost overheads. CorrectDB however, performs all aggregation operations inside the SCPU and only the final result is sent to the client. This saves traffic for all and costs for most but not all queries. Later, in section 6 we compare the performance and cost of aggregate operations for CorrectDB with the data transfer in other QA solutions.

The same argument as above applies to processing GROUP BY clauses within the SCPU rather than transferring additional tuples to the client, and having the client do the grouping. If on average, a GROUP BY clause aggregates n_g values and the total number of tuples satisfying the query predicates is n_r, then this reduces the number of tuples transferred from n_r to $\frac{n_r}{n_g}$.

ORDER BY clauses however are not subject to this argument. If an ORDER BY clause is processed as the last step in query execution then there is no reduction in the number of tuples that need to be transferred to the client. In this case if the client has higher processing capacity than the SCPU, such as desktop clients, then client-side ordering will perform better at least in execution time.

5.6 Client Side Verification

Client-side verification is identical for all query types since most query-specific verifications are already performed by the server-side SCPU.

The tuples comprising the result set R are identified by the SCPU query processor while processing the second subquery on the verified intermediate results of the first subquery executed by the server. Let $\mathsf{R} = \{t_1, t_2, ..., t_r\}$. The SCPU then computes the digest of R, $D(\mathsf{R}) = H(C_{id}||Q_c|| Nonce||H(t_1)||H(t_2)||...||H(t_r))$, where C_{id} is the client identifier, and Q_c is the client query. $Nonce$ is a per query fresh random value sent by the client within the query request. Its purpose is to uniquely associate the result R with the query Q_c. This prevents the server from matching stale results with a recent query thus thwarting replay attacks.

The SCPU then signs $D(\mathsf{R})$ using its private key. The signed message $S(D(\mathsf{R}), SK_{CDB})$, is then sent to the client, along with the query results $\{t_1, t_2, ..., t_r\}$. The client then recomputes $D(\mathsf{R})$ and verifies the signature using the public key of the SCPU.

Standardized *Outbound Authentication* mechanisms exist for key setup and for communication of public keys (PK_{CDB}) to clients. For details refer to [30, 29].

5.7 Database Updates

Clients issue update statements to request modifications to the server-hosted data. Any client update query requires secure modifications to the B^+-Tree storing the tuples, and the MHT based ADS used by the SCPU for verification. Both these data structures are modified only by the SCPU via local interaction with the server without involving the data owner. This avoids the need of inter-client synchronization and increases efficiency. .

Consider the following update query:

```
UPDATE lineitem SET l_discount = l_discount + 0.01
WHERE  l_shipdate >= '1993-01-01'
AND    l_shipdate < '1994-01-01'
AND    l_discount between 0.05 AND 0.07
```

It is processed as follows. (1) The query is parsed by the SCPU query parser and re-written into a server sub-query

```
SELECT * FROM lineitem
WHERE  l_shipdate >= '1993-01-01'
AND    l_shipdate < '1994-01-01'
```

and a SCPU sub-query

```
UPDATE lineitem SET l_discount = l_discount + 0.01
WHERE  l_discount between 0.05 AND 0.07
```

similar to the case of range queries. (2) The server then executes the first sub-query and finds all the leaf nodes that require modifications. Additional leaf nodes may be identified if the server requires re-balancing of the index. (3) The server transfers these leaf nodes to the SCPU query processor, which are then verified using the MHT ADS. Verification ensures correctness and completeness. (4) The SCPU query processor then modifies the nodes as per the second sub-query. It also modifies the MHT ADS, re-computing and updating the root hash stored within the SCPU. (5) Only after making changes to the ADS, the modified leaf nodes are transferred back to the server query processor and the final changes applied to the B^+-tree.

The above is repeated for each B^+-Tree with an ADS. Since on any update operation the root hash of the ADS is immediately re-computed and refreshed within the SCPU, replay attacks are thwarted.

6. EXPERIMENTS

We experimentally evaluate CorrectDB and compare it with [22, 25], the most efficient existing range query mechanisms, and with AIM [32], the most efficient mechanism for join queries (section 3).

Setup. The SCPU used is the IBM 4764 with 32 MB 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 client is an Ubuntu VM with 1 GB RAM and 2 vCPUs. The CorrectDB stack is written in C.

Measurements are made for the three key metrics, Verification Object (VO) size, Query Execution Time (QET), and Verification Time (VT).

For comparative experiments, we combine both Query Execution Time (QET) and Verification Time (VT) into a single Total Query Execution time and later in figure 9(a) we measure each time component separately. CorrectDB has a constant VO size, which is a single signed digest of all tuples that comprise the query result. This VO size is the same for all types of queries, hence we only mention it once here.

Range Queries. For evaluating the performance of CorrectDB and signature aggregation we set up multiple relations with varying tuple sizes and 10^6 tuples each, indexed using a B^+-Tree. Keys are random integers between 1 and 10^7. We evaluate the performance by varying the tuple size, and the number of tuples in the query result. As we shall see, both these parameters have varying effects on performance and it is hence, important to consider both. The test queries are thus of the form

```
SELECT * FROM R where R.key > 'LB' AND R.key < 'UB'
```

'LB' and 'UB' are varied to get query results of different sizes. Note that, although CorrectDB supports a wide range of queries, we only use simple queries here for consistent comparative results, since other approaches [22, 25, 32] support and evaluate only such basic queries.

Figure 4(a) shows the total query execution times for CorrectDB and signature aggregation varying the two parameters. For small tuple sizes (32 bytes - 128 bytes) CorrectDB performs 2-6x better than signature aggregation. Note that the performance of signature aggregation is linear in terms of the number of tuples in the result set, but does not depend on the tuple size, since the aggregation is performed only on the signed hash of the tuples and not on the tuples themselves. This explains the similar times for signature aggregation even when tuple sizes are varied. CorrectDB's performance however, varies with both tuple size, and number of tuples in the result.

For larger tuple sizes (512 bytes) and large result sets ($\geq 10^3$) we observe a shift and signature mechanisms perform better by factors of 1.1-2x (figure 4(b)). This is because increasing tuple size also increases the size of the leaf nodes that contain the tuple data. Since these leaf nodes are transferred from the server to the SCPU, the larger the leaf node the higher is the transfer latency. Hence, CorrectDB has higher execution times for large tuples. This is apparent from figure 9(a) which depicts the breakdown of execution times of different query processing stages of CorrectDB. As seen, for result sizes $\geq 10^3$, the data transfer of leaf nodes from the server to the SCPU is significant. Paying this penalty for transferring entire leaf nodes is acceptable since it enables CorrectDB to support projections and complex selection predicates. However, note that in the experiments above we have not considered any projection operations i.e., the entire tuple is part of the result. Once projections are introduced CorrectDB regains the performance advantage even for large tuple sizes, as described next.

Projections. Next, we add projections into the mix. We fix the tuple size to 512 bytes and vary the number of projected attributes in the select condition of the test query for each run. Each tuple is divided into 16 equal sized attributes. Figure 4(c) illustrates that CorrectDB performs better for all cases by 1.5-7x. If p tuples are projected out then the number of signatures to be aggregated increases by p per tuple in the query result [25]. This causes the higher execution times for signature aggregation as compared to CorrectDB even for large tuple sizes.

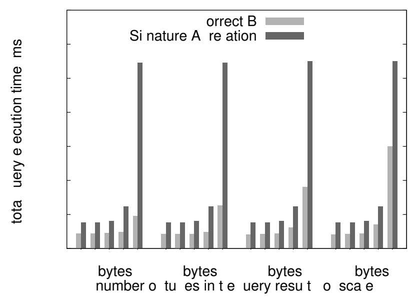
Both Signature Aggregation and CorrectDB have the same VO size and hence, are equivalent on this metric. Figure 7(a) summarizes which solution (CorrectDB or signature aggregation) performs better for each combination of the parameters, tuple size, result size, and number of projected attributes.

Join Queries. For evaluating join queries (CorrectDB and AIM [32]) we use two sets of relations R and S with 10^6 tuples each and varying tuple sizes. Each relation has a B^+-Tree on its key attribute and an MHT based ADS. The test queries are of the form

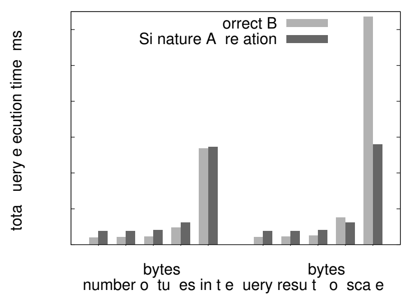
```
SELECT * FROM R, S WHERE R.key = S.key AND R.key > 'LB'
AND R.key < 'UB'
```

We note that in case of AIM the VO size is not constant but varies with the query result size. Since non-trivial VO sizes result in large server-client data transfers we include the data transfer time (network latency) as a measurement parameter. We evaluate the performance on both Foreign Key (FK) and Equi (EQ) joins, as in [32], to keep the comparisons consistent.

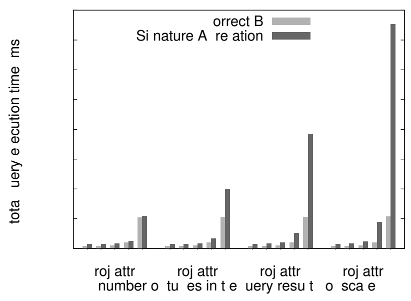
Foreign Key (FK) Join. In a FK joins each tuple in relation R matches at least one tuple in the other relation



(a) Small tuple sizes (no projections).

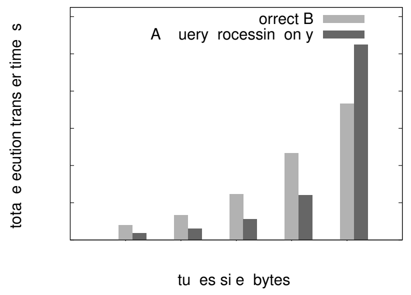


(b) Large tuple sizes (no projections).

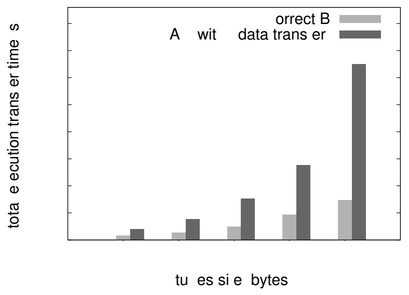


(c) Varying # projected attributes.

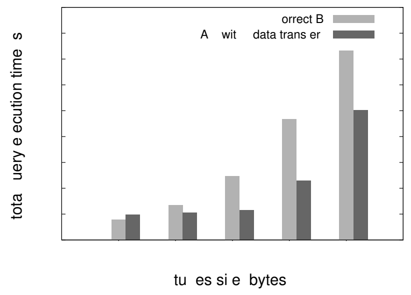
Figure 4: Comparison of CorrectDB and Signature Aggregation [22, 25] for Range queries (with/without projections)



(a) FK join without VO data transfer times.



(b) FK join with VO transfer (10 Mbps link).



(c) Equi join - 10 Mbps link.

Figure 5: Comparison of CorrectDB and AIM [32] for Foreign Key and Equi-Join queries.

S. This results in large query result sizes. Figure 5(a) shows the total query execution times for CorrectDB and AIM for various tuple sizes. Note that the data transfer times are not included here. We see that when compared on processing times alone, AIM outperforms CorrectDB for tuple sizes up to 256 bytes. However, note that the VO size for AIM ranges from 19 MB to 260 MB. Hence, once we consider the data transfer times as well it is observed that the performance relationship inverses. This is depicted in figure 5(b) which also includes the server to client VO transfer times. Figure 5(b) considers a link capacity of 10 Mbps. As seen now CorrectDB features significantly lower overall execution times by factors up to 5x.

It may be possible that in certain settings such as private clouds, the client-server link capacities are larger thereby favoring AIM. Hence, in figure 6(a) we re-compare for different link capacities. In conclusion, up to link capacities of 50 Mbps CorrectDB performs better.

We note that in most commercial settings today, available link capacities for home and businesses range from 1 Mbps to 30 Mbps, increased capacity being available with increased costs [6]. For commercial cloud services such as Amazon EC2 the available TCP bandwidth from external clouds to EC2, has been benchmarked in the 7-27 Mbps range [27].

Equi (EQ) Join. Unlike FK join where each tuple in relation *R* matches at least one tuple in the other relation *S* equi join has a small result set, which we fix at 31000 as in [32]. This reduces the processing times for AIM to construct the VO and makes the VO size small. CorrectDB uses the same processing mechanisms for both FK and EQ joins thereby having similar performance for both queries.

Figure 5(c) shows the execution times for EQ join queries for a link capacity of 10 Mbps, while Figure 6(b) compares the times for varying data link capacities. For capacities $>$ 5 Mbps AIM performs similarly or better.

Figure 7(b) summarizes which solution (CorrectDB or AIM) performs better for each combination of the parameters, join type (FK or EQ), tuple size, and link capacity.

As seen in figures 7(a) and 7(b) using trusted hardware makes CorrectDB the preferred QA solution for a wide range of the parameter space. Note that, in addition, this greatly increases the functionality that can be provided, e.g., support for arbitrary joins, aggregation queries, compliance, access control etc (section 4).

Updates. Update operations are evaluated using the range query data sets and test queries of the form

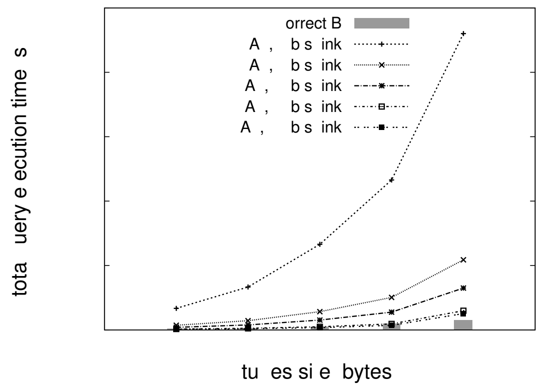
```
UPDATE key=key+1 FROM R where R.key>'LB' AND R.key<'UB'
```

Updates are a highly favorable scenario for CorrectDB. As discussed in section 4 existing QA solutions cannot perform updates server-side. Instead, the data owner obtains the relevant tuples from the server, modifies them locally along with the ADS, and re-uploads to the server. Hence the data transfer overheads are significant. Therefore, as shown in figure 8(a) CorrectDB outperforms [22, 25] even for higher tuple sizes, no projections, and high bandwidth links (50 Mbps). In fact, for update operations CorrectDB outperforms in the entire parameter space.

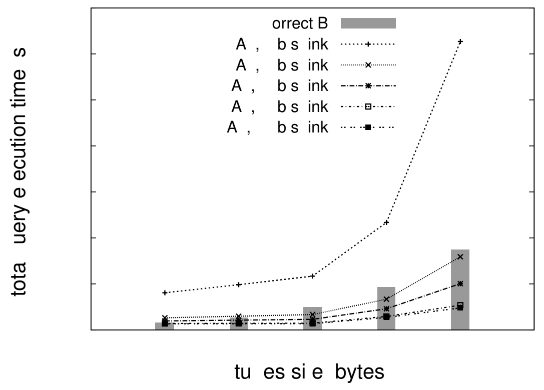
Aggregations. To evaluate aggregate operations we use the same data setup as for range queries. The test queries are of the form.

```
SELECT SUM(key) FROM R where R.key > 'LB' AND R.key < 'UB'
```

Figure 8(b) shows data transfer time in other approaches vs the time required to do the entire computation within the SCPU and only send the result to the client. The link capacity considered is 10 Mbps. As seen, total query execution time in CorrectDB is lower for result sizes $> 10^2$. Note that, here we only compare the data transfer time in other solutions and not the total query execution time, which would include the server processing and client verification. We do

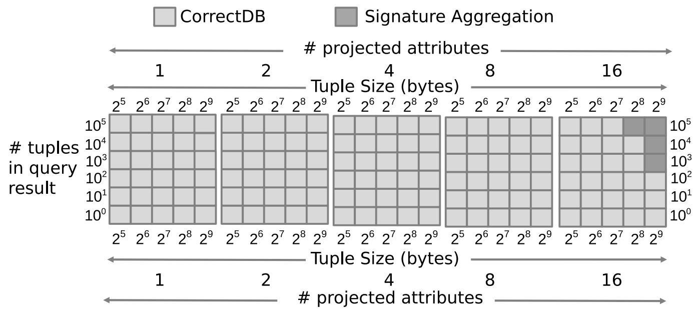


(a) FK join query

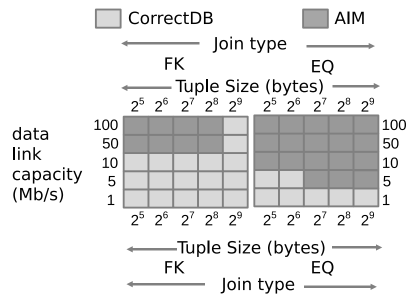


(b) EQ join query

Figure 6: Effect of data link capacity on performance of AIM compared to CorrectDB.



(a) CorrectDB and Signature Aggregation (Range queries)



(b) CorrectDB and AIM (Join queries)

Figure 7: Performance maps for CorrectDB, Signature Aggregation [22, 25] and AIM [32]. A box shaded with the color of a specific solution indicates that this solution has better performance (lower total query execution time) for the set of parameters corresponding to that box.

this specifically to demonstrate the significant data transfer overhead for aggregations in existing QA solutions. If overall query execution times are added to results of figure 8(b), then CorrectDB outperforms for all result sizes.

Further, we also compare the costs of performing aggregation operations in CorrectDB with just the data transfer costs in other approaches. For this, we use the derivation of costs of data transmission and the SCPU cycles based on the work from [6, 30] which also take into consideration the acquisition and operating costs of SCPUs and of traditional server hardware. The cost of a single SCPU CPU cycle is 56 picocents [30] while a single bit transfer from cloud to client environment costs up to 3500 picocents [6].

As a result, it can be seen from figure 8(c), that CorrectDB also performs significantly better in terms of cost above a minimal result set size of around 100. This is because a fixed lower bound cost is incurred in transferring a single data page from the server to the SCPU.

7. DISCUSSION

Data Privacy and QA. An efficient SCPU-based solution for data privacy and confidentiality in the presence of a curious but otherwise trusted server is TrustedDB [30]. TrustedDB partitions relational data into private (encrypted) and public parts. A client query is then split in such a way that private data is decrypted only inside the SCPU while processing on public attributes is done by the host server. TrustedDB supports full SQL and leverages the use of very fine grained encryption (at the attribute level) to off-load significant query operations to the server. TrustedDB does not provide any QA guarantees and although, it may be feasible to endow TrustedDB with an additional layer of QA, we chose not to do so here for several reasons.

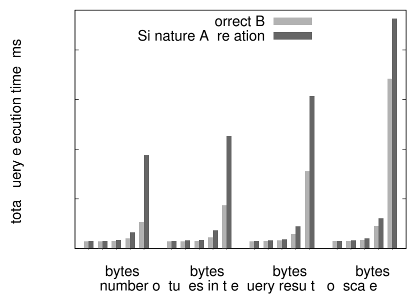
Firstly, since TrustedDB employs fine-grained attribute level encryption, it can keep the SCPU-side processing minimal by offloading range, projection and aggregation operations (on public attributes) to the server. However, to date, we do not have a single authentication data structure (ADS) that can verify the integrity of all these query operations. Hence, if QA were to be added here, a separate ADS is needed for each operation. This increases server-side storage, the SCPU-server data transfers and SCPU-side processing. In short, the overheads of adding QA far exceed the original cost of privacy resulting in a solution that is efficient neither for privacy nor for QA.

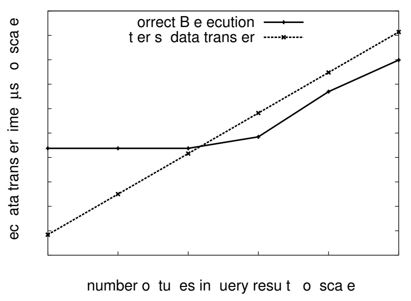
Secondly, for illustration purposes, we felt it was important to clearly outline the cost and performance benefits of trusted hardware over existing QA work without the additional overheads of privacy.

However, CorrectDB allows for a certain degree of privacy at minimal cost by employing tuple level encryption in update/insert operations.

All tuple attributes are encrypted, except for the search attribute(s) on which the B^+-Tree is built. The search attribute is not protected to aid the server in query processing. Also, since projections, aggregations, and the final processing of queries are done inside the SCPU, the tuple data is only decrypted within, on demand.

We measure the overhead of providing data privacy in


(a) Update operations (50 Mbps link).


(b) Aggregations execution time.

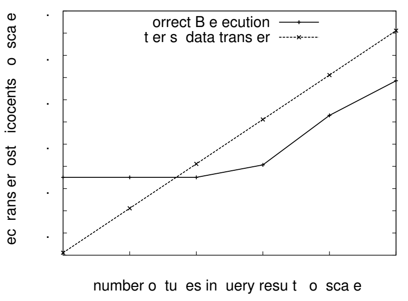
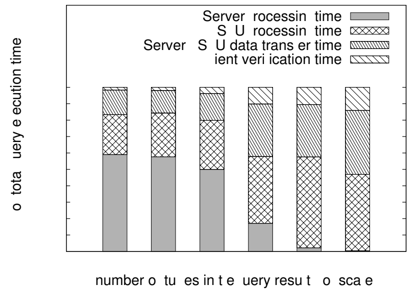
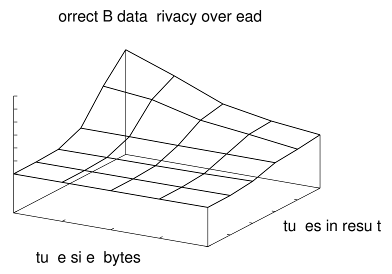

(c) Aggregations execution cost.

Figure 8: CorrectDB aggregate and update operations.


(a) Time profile (512 byte tuples).


(b) Range queries.

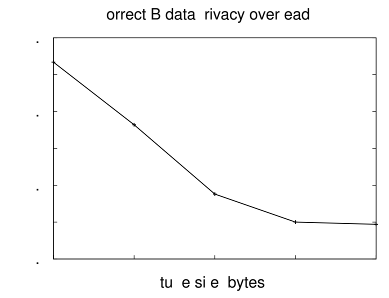

(c) Equi join.

Figure 9: CorrectDB query profile and data privacy overheads (as percentage of total query execution time).

CorrectDB. Figure 9(b) shows the overhead for the set of range queries while figure 9(c) shows the data privacy overhead for the EQ join query tests. We observe that the overhead decreases with tuple size. For small result set sizes as in the case of EQ join queries data privacy is added with very little overhead.

Optimized Solutions. Note that, instead of a general-purpose processor, at the expense of functionality, it is possible to design specific optimized solutions targeted at particular query characteristics. E.g., for EQ join queries we can modify the leaf nodes to store the hash of individual tuples instead of entire tuple contents. This reduces the size of the leaf nodes and thereby the data transfer times from the server to SCPU. To illustrate, for tuple size of 512 bytes from above experiments the transfer times are reduced by up to *95%*. Such optimizations greatly improve the performance of CorrectDB. However, we chose to opt out of such targeted solutions that limit functionality, since we posit that the benefits of utilizing trusted hardware are seen in the increased functionality offered (section 4) at better/comparable performance.

Security. Although the SCPU resides server-side in physical possession of the service provider who also performs management tasks such as software updates, the overall security of the solution remains intact. This is because the tamper-resistant design and *Outbound Authentication* mechanisms of trusted hardware at any point in time provide the following. (i) Guarantee that the remote trusted module was not tampered with. (ii) On-demand proof to clients that the trusted hardware module runs the correct software code stack, including the correct user-land modules as well as the underlying OS and SCPU hardware logic. (iii) Enable setup of secure communication channels between clients and the remote trusted modules.

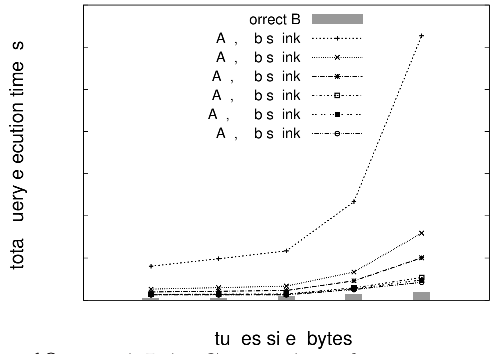


Figure 10: Equi-Join Comparison for new 4765 SCPU which yields faster results for all cases.

Due to space constraints we refer the reader to [30, 29] for details of *Outbound Authentication* and its applications.

New SCPU. Older SCPU technology is now being replaced by new and improved SCPUs such as the recently announced IBM 4765 [1], which features more RAM (128MB+), two faster 400MHz CPUs, and a significantly faster PCIe bus for increased 100MB/s+ throughputs.

Initial intuitions suggest that the overall cost and efficiency proposition of the new 4765 will increase the current CorrectDB advantages by a factor of 4-6x. E.g., the higher PCIe throughput alone decreases overall query execution time by a factor of 3.5x, since we know from figure 9(a) that the server-SCPU conduit bandwidth often dominates. Figure 10 projects the expected performance of CorrectDB on the new platform indicating that CorrectDB will likely out-perform in the entire parameter space.

Query Optimization. [3] details query optimization techniques in a trusted hardware model.

Trusted Hardware in Data Management. Due to space constraints we direct the reader to existing work [30] that summarizes the work on this topic.

8. CONCLUSIONS

This paper's contributions are threefold. (i) a comparative survey of existing QA research, (ii) the insight that trusted hardware significantly reduces overall costs (despite its higher price), increases performance and provides enhanced QA functionality, and (iii) the design and development of CorrectDB, a trusted hardware based DBMS providing QA for a wide range of query types on both read-only and dynamic data sets.

9. REFERENCES

[1] IBM 4765 PCIe Cryptographic Coprocessor. Online at `http://www-03.ibm.com/security/cryptocards/`.

[2] TPC-H. Online at `http://www.tpc.org/tpch/`.

[3] S. Bajaj and R. Sion. Trusteddb: A trusted hardware based database with privacy and data confidentiality. *IEEE Transactions on Knowledge and Data Engineering*, 99(PrePrints):1, 2013.

[4] B. Carminati. Secure data outsourcing. In L. Liu and M. T. Özsu, editors, *Encyclopedia of Database Systems*. Springer US, 2009.

[5] Y. Chen and R. Sion. On securing untrusted clouds with cryptography. In E. Al-Shaer and K. B. Frikken, editors, *WPES*, pages 109–114. ACM, 2010.

[6] Y. Chen and R. Sion. To cloud or not to cloud? musings on costs and viability. In *SOCC*. ACM, 2011.

[7] W. Cheng, H. Pang, and K. lee Tan. Authenticating multi-dimensional query results in data publishing. In *In DBSec*, 2006.

[8] P. Devanbu, M. Gertz, C. Martel, and S. G. Stubblebine. Authentic data publication over the internet. pages 291 – 314, 2003.

[9] Einar Mykletun and Gene Tsudik. Aggregation Queries in the Database-As-a-Service Model. *Data and Applications Security*, 4127, 2006.

[10] M. T. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DISCEX*, pages 68 – 82. IEEE Computer Society Press, 2001.

[11] H. Hacigumus, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *DASFAA*, volume 2973, pages 633–650, 2004.

[12] L. Hu, W.-S. Ku, S. Bakiras, and C. Shahabi. Verifying spatial queries using voronoi neighbors. In *Proceedings of GIS*, pages 350–359. ACM, 2010.

[13] IBM 4764 PCI-X, 4765 PCIe Cryptographic Coprocessors. Online at `http://www-03.ibm.com/security/cryptocards/`.

[14] Joan G. Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer and Leendert van Doorn. Building the IBM 4758 Secure Coprocessor. *IEEE*, 34(10), 2001.

[15] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *Proceedings of SIGMOD*, pages 121 – 132. ACM, 2006.

[16] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index structures for aggregation queries. *ACM Trans. Inf. Syst. Secur.*, 13(4):32:1–32:35, Dec. 2010.

[17] X. Lin, J. Xu, and H. Hu. Authentication of location-based skyline queries. In *Proceedings of CIKM*, pages 1583–1588. ACM, 2011.

[18] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. *Algorithmica, Volume 39 Issue 1*, pages 21 – 41, 2004.

[19] R. C. Merkle. A certified digital signature. In *Proceedings on Advances in cryptology*, pages 218–238. Springer-Verlag New York, Inc., 1989.

[20] E. Mykletun, M. Narasimha, and G. Tsudik. Signature bouquets: Immutability for aggregated/condensed signatures. In *ESORICS*, 2004.

[21] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. *ACM TOS, Volume 2 Issue 2*, pages 107 – 138, 2006.

[22] M. Narasimha and G. Tsudik. Dsac: integrity for outsourced databases with signature aggregation and chaining. In *Proceedings of CIKM*. ACM, 2005.

[23] H. Pang, A. Jain, K. Ramamritham, and K. Tan. Verifying completeness of relational query results in data publishing. In *Proceedings of SIGMOD*, pages 407 – 418. ACM, 2005.

[24] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *Proceedings of ICDE*, page 560. IEEE Computer Society, 2004.

[25] H. Pang, J. Zhang, and K. Mouratidi. Scalable verification for outsourced dynamic databases. *Proceedings of the VLDB Endowment, Volume 2 Issue 1*, pages 802 – 813, 2009.

[26] S. Papadopoulos, D. Papadias, W. Cheng, and K.-L. Tan. Separating authentication from query execution in outsourced databases. In *Proceedings of ICDE*, pages 1148 – 1151. IEEE Computer Society, 2009.

[27] S. Sanghrajka, N. Mahajan, and R. Sion. Cloud performance benchmark series, network performance: Amazon ec2. Online at `www.cloudcommons.org`.

[28] S. Singh and S. Prabhakar. Ensuring correctness over untrusted private database. In *Proceedings of EDBT*, pages 476 – 486. ACM, 2008.

[29] S. W. Smith. Outbound authentication for programmable secure coprocessors. In *Proceedings of ESORICS*, pages 72–89. Springer-Verlag, 2002.

[30] Sumeet Bajaj and Radu Sion. TrustedDB: A Trusted Hardware based Database with Privacy and Data Confidentiality. In *Proceedings of SIGMOD*, pages 205–216. ACM, 2011.

[31] M. Xie, H. Wang, J. Yin, and X. Meng. Integrity auditing of outsourced data. In *Proceedings of VLDB*, pages 782 – 793. VLDB Endowment, 2007.

[32] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis. Authenticated join processing in outsourced databases. In *Proceedings of SIGMOD*, pages 5 – 18. ACM, 2009.

[33] Y. Yang, S. Papadopoulos, D. Papadias, and G. Kollios. Authenticated indexing for outsourced spatial databases. pages 631 – 648, 2009.

[34] Z. Yang, S. Gao, J. Xu, and B. Choi. Authentication of range query results in mapreduce environments. In *Proceedings of CloudDB*, pages 25–32. ACM, 2011.

[35] Q. Zheng, S. Xu, and G. Ateniese. Efficient query integrity for outsourced dynamic databases. 2012.