Authenticating Query Results in Edge Computing

HweeHwa Pang
Institute for Infocomm Research
21 Heng Mui Keng Terrace
Singapore 119613
hhpang@i2r.a-star.edu.sg

Kian-Lee Tan
Department of Computer Science
National University of Singapore
3 Science Drive 2, Singapore 117543
tankl@comp.nus.edu.sg

Abstract

Edge computing pushes application logic and the underlying data to the edge of the network, with the aim of improving availability and scalability. As the edge servers are not necessarily secure, there must be provisions for validating their outputs. This paper proposes a mechanism that creates a verification object (VO) for checking the integrity of each query result produced by an edge server – that values in the result tuples are not tampered with, and that no spurious tuples are introduced. The primary advantages of our proposed mechanism are that the VO is independent of the database size, and that relational operations can still be fulfilled by the edge servers. These advantages reduce transmission load and processing at the clients. We also show how insert and delete transactions can be supported.

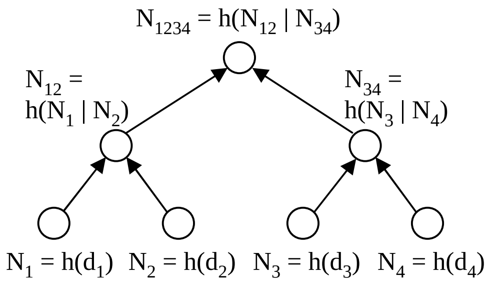
1. Introduction

Edge computing is being promoted as a strategy to achieve scalable and highly available Web services (e.g. [10]). It pushes business logic and data processing from corporate data centers out to proxy servers at the "edge" of the network. There are several potential advantages: Running applications at the edge cuts down network latency and produces faster responses to end-users' applications and partners' Web services. Adding edge servers near user clusters is also likely to be a cheaper way to achieve scalability than fortifying the servers in the corporate data center and provisioning more network bandwidth for every user. Finally, by lowering the dependency on the corporate data center, edge computing removes the single point of failure in the infrastructure, hence reducing its susceptibility to denial of service attacks and improving service availability.

In theory, edge computing is a natural extension of the Content Delivery Network (CDN) architecture. In practice, pushing application logic to edge servers introduces a number of technical challenges, one of which is data security: For applications that run on a database, edge computing entails the distribution of (parts of) the database, to edge servers that perform query processing on behalf of the central DBMS. Since the edge servers are not necessarily as secure as the corporate data center, the query results produced by them must be checked for integrity. Specifically, a recipient must be able to verify that the values in his query result have not been tampered with, and that no spurious tuples are introduced.

In this paper, we propose a mechanism for authenticating the query results produced by edge servers. The central database server maintains on each base table one or more *verifiable* B-trees (VB-tree), which are B-trees extended with concepts from the Merkle hash tree [11]. Each internal node of a VB-tree is associated with a signed digest or hash value, derived from all the tuples in the subtree that is rooted at the node. The VB-tree is distributed to the edge servers along with the base table. In processing a query, an edge server will identify the smallest subtree that envelops the query result, and return a *verification object* (VO) containing the digests for all the attributes and branches in the subtree that are not part of the query result. With this VO, the recipient can verify the query result by computing a digest from them, and checking for a match with the subtree's digest that was signed by the trusted central database server.

One of the primary contributions of this paper over existing work is that the proposed mechanism creates VOs that are linear in the size of the query answers, and independent of the database size. Moreover, all the relational operations can be fulfilled by the edge servers; for example, we do not push projection operations to the clients. Hence, our mechanism lowers the transmission overhead, and also the storage and processing demands on the clients. Finally, we demonstrate how the VB-trees can be updated dynamically while



$N_{1234} = h(N_{12} \mid N_{34})$
$N_{12} = h(N_1 \mid N_2)$
$N_{34} = h(N_3 \mid N_4)$
$N_1 = h(d_1)$
$N_2 = h(d_2)$
$N_3 = h(d_3)$
$N_4 = h(d_4)$

Figure 1. Example of a Merkle Hash Tree

ensuring data consistency.

The remainder of this paper is organized as follows: Section 2 summarizes related work on data authentication, with particular emphasis on authentication in database systems. Our proposed authentication mechanism is introduced in Section 3, while Section 4 analyzes the costs and performance of the mechanism. Finally, Section 5 concludes the paper.

2. Related Work

This paper builds upon the work by Merkle in [11]. We shall explain the Merkle hash tree with the example in Figure 1, which is intended for authenticating data values d_1, .., d_4. Each leaf node N_i is assigned a digest $h(d_i)$, where h is a one-way hash function. The value of each internal node is derived from its child nodes, e.g. $N_{12} = h(N_1 \mid N_2)$ where $\mid$ denotes concatenation. In addition, the value of the root node is signed. The tree can be used to authenticate any subset of the data values, in conjunction with a verification object (VO). For example, to authenticate d_1, the VO contains N_2, N_{34} and the signed N_{1234}. The recipient first computes $h(d_1)$ and $h(h(h(d_1) \mid N_2) \mid N_{34})$, then checks if the latter is the same as the signed N_{1234}. If so, d_1 is accepted; otherwise, the recipient concludes that d_1 has been tampered with.

Merkle hash trees have inspired many proposals on data authentication, including certifying answers to queries over XML documents [4], for proving the presence or absence of public key certificates on revocation lists [7, 12], and for authenticating JPEG2000 images [13]. The proposal that is most closely related to our work is [5], which describes a scheme for verifying query results produced by untrusted third-party publishers. The scheme calls for the data owner to periodically distribute signed digests directly to users. The digests are hashes computed recursively over tree indices on the owner's database. To prove that the answer to a query is correct, the publisher constructs a VO using the same tree index that the owner used to compute the signed digest. The VO provides a hard-to-forge proof that links the answer to the signed digest.

This work by Devenbu et al [5] is among the first few papers to address the authentication of query results in database systems. However, when applied directly in our context of edge computing, their scheme poses a number of limitations that we aim to overcome: First, a Merkle tree is needed for every sort-order on a table; this incurs large storage overheads, and makes updates very expensive. Second, a VO needs to contain links all the way to the digest for the root of the tree index. This means that the VO grows linearly to the query result and logarithmically to the base table, which can be quite sizeable for large databases. Another potential problem is that projections have to be performed by the clients, which leads to wasteful data transfers especially if the filtered attributes are BLOBs. There is also no provision for dynamic updates on the database and the associated Merkle trees. Finally, the approach is weak in terms of access control – even attributes that are supposed to be filtered out through projection must be returned to users for verification. Moreover, to check for completeness, tuples beyond the left and right boundaries of the query result must be exposed to the user; this would undermine any tuple-based access control in the system.

A more recent work by Roos et al [16] also employs the Merkle hash tree to authenticate range queries. However, the focus of that paper is on encoding the VO in a more compact form to minimize communication overhead; it has the same limitations as the scheme in [5].

3. Authenticated Query Processing at Edge Servers

This section presents our proposed mechanism for verifying the query results produced by proxy servers in edge computing. The idea is for the trusted central DBMS to maintain and distribute verifiable B-trees (VB-tree) on the base tables, that the edge servers use to create a verification object (VO) for each query. This VO enables the recipient to check the integrity of the query result – that the values of the result tuples have not been tampered with, and that no superfluous tuples are introduced. It is computationally infeasible for an edge server to compromise the integrity of the result, without invalidating the VO and being detected.

We start with an overview of the system setup, then present the structure of the verifiable B-tree. Following that, we show how an edge server constructs a VO from the VB-tree for the selection, projection and join operations in a query. Finally, we address how the central DBMS carries out insertions and deletions on the VB-tree while preserving the consistency requirement of other queries and update operations.

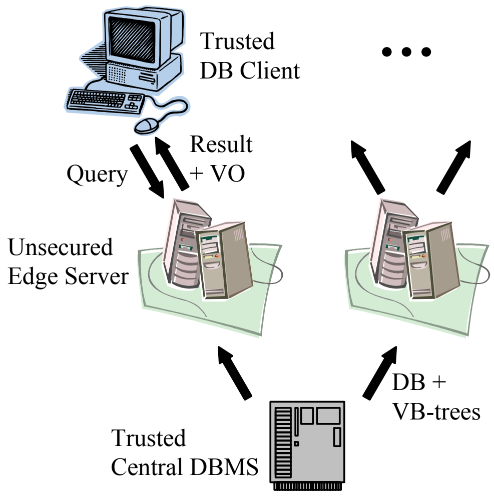


Figure 2. Edge Computing Set-Up

3.1. System Overview

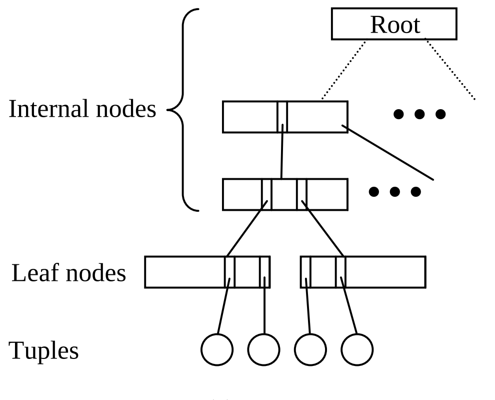
Figure 2 shows the set-up of a generic edge computing environment. The central server hosts the master database, and is responsible for creating and maintaining the VB-trees. The database (or parts of it) and VB-trees are distributed to servers situated at the edge of the network, i.e., edge servers, near the users and their application clients. The database queries issued by the applications are processed at the edge servers; the result for each query is returned together with a VO that the client can use to verify the query result. Unlike the proposal in [5], all the information that the client needs for verifying the query result are returned by the edge server on-demand; the central server does not need to periodically broadcast any information to the clients. Finally, the edge servers are assumed to be unsecured, meaning it is possible for a hacker to tamper with the data there, but the servers themselves do not act maliciously, e.g. they do not intentionally drop qualifying tuples from the query results.

3.2. Verifiable B-Tree

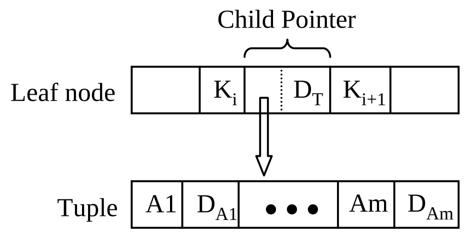
Figure 3 depicts the structure of the verifiable B-tree (VB-tree). It is constructed by adding signed digests to the B+-tree as follows:

- Within each tuple, a signed digest D_i is added for every attribute Ai, i.e.,

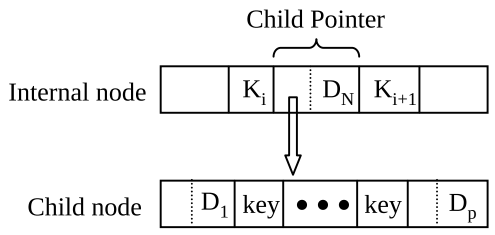
$$D_{Ai} = s(k(DBname \mid tablename \mid attribname \mid tuplekey \mid Ai)) \quad (1)$$



(a) Overview



(b) Structure of Leaf Node



(c) Structure of Internal Node

Figure 3. Verifiable B-Tree

where k is a one-way hash function on the concatenation of the database name, the table name, the attribute name, the key of the tuple, and the attribute value; and s encrypts its argument with the private key of the DBMS. A one-way hash function is one that is easy to compute but effectively impossible to invert [6]; i.e., if $b = k(a)$, then it is easy to compute b given a and k, but difficult to recover a given b and k. Popular one-way hash functions include MD5 [14] and SHA [1]. We assume that the central server has a pair of public and private keys in a digital signature scheme, and the public key can be disseminated to users through an authenticated channel using, for example, a public key infrastructure [8].

The (pre-signed version of the) attribute digests are

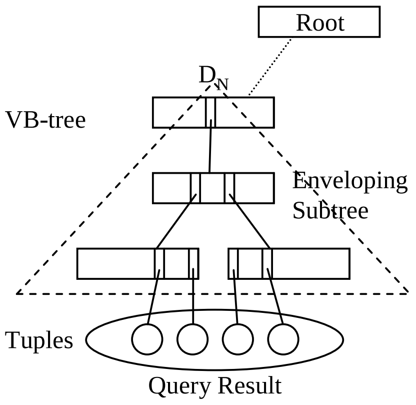


Figure 4. Enveloping Subtree of Result Tuples

used to compute the signed tuple digest D_T:

$$D_T = s(h(\sum s^{-1}(D_{Ai}))) \ \forall \text{ attributes } Ai \quad (2)$$

where h is another one-way hash function and s^{-1} decrypts its argument with the public key of the DBMS. D_T is then stored with the corresponding tuple pointer in the leaf node of the VB-tree. With the DBMS' public key, any user can subsequently confirm the integrity of the tuple, by recalculating the tuple digest from the retrieved attribute values or their digests using formulas (1) and (2), and matching it with $s^{-1}(D_T)$.

- For each leaf node N, a signed node digest D_N is derived from the tuple digests within it, i.e.,

$$D_N = s(h(\sum s^{-1}(D_i))) \quad \forall \text{ tuples } 1 \leq i \leq p \text{ in node } N \quad (3)$$

  The node digest is stored with the corresponding child pointer in the parent node. This process is performed recursively up the VB-tree.

- Finally, a signed digest is computed for the root node and stored as part of the metadata of the VB-tree.

Definition: The **enveloping subtree** is the smallest subtree within the VB-tree that covers all the result tuples of a query (involving selection, projection or join operations), or all the tuples affected by an update. Figure 4 illustrates the enveloping subtree.

Before we explain how the VB-tree is used to create VOs for query answers, let us first consider the choice of the one-way hash function h. h plays a key role in ensuring that it is infeasible for an edge server to tamper with the tuples or introduce superfluous tuples, in such a way that the query results still match the signed digests. While any one-way hash function can be employed, a judicious choice can lead to improved performance. For this work, we use the function $h(x) = g^x \bmod q$, which combines its input digests with a commutative operator:

$$\begin{aligned} h(x+y) &= g^{(x+y)} \bmod q \\ &= ((g^x \bmod q)g^y) \bmod q \\ &= (h(x) \cdot h(y)) \bmod q \\ &= h(y+x) \end{aligned}$$

As we shall discuss shortly in the following subsections, this property has three advantages. First, it allows digests to be combined in arbitrary order, i.e., a set of digests $x_1, x_2, \ldots, x_n$ can be combined in any order without affecting the final digest $h(\sum_{i=1}^{n} x_i)$. Second, it enables projection operations to be performed at the edge servers. Third, it facilitates insertion of new tuples with minimal effect on other digests. Hence, even though the hash function itself is computationally more expensive than alternatives with only polynomial complexity, the resulting savings in transmission load and processing at the clients would justify the choice.

Nevertheless, to minimize processing overhead, we can implement h by picking $q = 2^r$ for some r in order to optimize the modulo operation, and by computing exponentiation through repeated squaring coupled with modulo reductions. For example, instead of 15 multiplications followed by a large modulo reduction at the end, we perform only 4 multiplications and 4 modulo reductions:

$$\begin{aligned} g^{16} \bmod q &= (((g^2 \bmod q)^2 \bmod q)^2 \\ &\qquad \bmod q)^2 \bmod q \end{aligned}$$

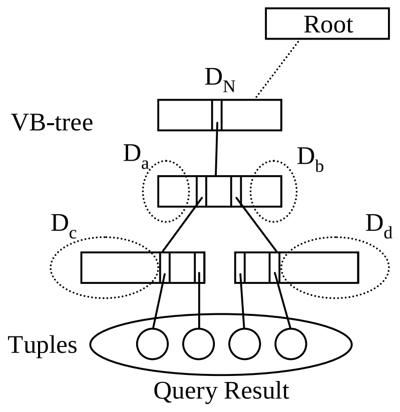
3.3. Query Verification

In this section, we shall present the VOs for selection, projection and join operations, and show that they are sufficient to verify the correctness of the answers for the respective operations.

Selection: $\sigma_C(R) = \{t \mid t \in R \text{ and } C(t)\}$ where R is a relation, C is a condition of the form $A_i \ \Theta \ c$, A_i is an attribute of R, and $\Theta \in \{=, \neq, <, \leq, >, \geq\}$.

When an edge server receives a query that performs only selection(s) on the primary key of a table, the result is a range of contiguous tuples as illustrated in Figure 5. In this case, the edge server needs to compose a verification object that includes:

- D_N, the signed digest for the node N at the top of the enveloping subtree;
- for each node in the enveloping subtree, the signed digests representing those branches that do not overlap the result.



Verification object = D_N + D_S
where D_S = {D_a, D_b, D_c, D_d}

Figure 5. Verification Object for Selection

Verification object = D_N + D_S

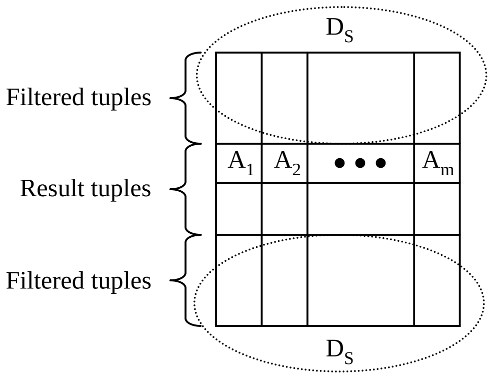


Figure 6. Example Query Result for Selection Operation

The VO and the query result are sufficient for the recipient to reconstruct the hierarchy of digests to check for a match with the signed digest for node N. As the attribute digests within the result tuples are not needed here, they are filtered out by the edge server.

In the case of a query that performs selection on non-key attribute(s), the result is likely to contain non-contiguous tuples from the table; in other words, there are "gaps" within the result range. The above procedure still applies, but the VO is expected to be bigger here as it needs to include additional digests to account for the gaps.

As we maintain a signed digest for every VB-tree node, our VO does not contain digests for branches all the way up to the root node of the index as in [5]. Hence the VO grows only linearly with the size of the query result, but is independent of the table size. Although our decision to sign every node digest imposes processing overhead on the central server in creating and maintaining the VB-tree, the expected I/O savings at the edge servers during runtime more than justify the overhead. In addition, as input digests to the hash function h are coalesced with a commutative operator, the VO does not need to preserve the order in which the digests are merged into node N's digest in the VB-tree. Hence our VO simply contains a *set* of signed digests, D_S; it is simpler than the scheme in [5], which requires its VOs to contain the structure among the digests. These advantages reduce the transmission overhead, and also the amount of processing that the client must perform in order to verify the query result.

Figure 6 shows an example of the query result and the associated VO after a selection operation.

Lemma 1: The VO is sufficient for verifying the correctness of the result for selection operations.

Proof: We consider different cases based on the height of the enveloping subtree.

Case 1. The enveloping subtree consists of only a leaf node N.

By definition (3), $D_N = s(h(\sum \text{unsigned digest for tuple } Ti))$. For our choice of hash function $h(x) = g^x \bmod q$, the formula can be translated to:

$$\begin{aligned} s^{-1}(D_N) &= (h(\sum \text{unsigned digest for} \\ &\qquad \text{result tuple } Ti) \times \\ &\qquad h(\sum s^{-1}(D_{\text{filtered}} \\ &\qquad\qquad \text{tuple }_{Ti}))) \bmod q \end{aligned}$$

Case 2. The enveloping subtree has a height of 2, with N being the top node.

By definition, $D_N = s(h(\sum \text{unsigned digest for child node } Ni))$. If a child node Ni contains at least one result tuple, it is represented by $(h(\sum \text{unsigned digest for result tuple } Tj) \times h(\sum s^{-1}(D_{\text{filtered tuple }Tj}))) \bmod q$ as in case 1. If a child node Ni does not contain any result tuple, it can be represented by its digest $s^{-1}(D_{Ni})$.

For our choice of $h(x) = g^x \bmod q$, the formula can be translated to:

$$\begin{aligned} s^{-1}(D_N) &= (h(\sum \text{unsigned digest for} \\ &\qquad \text{result tuple } Tj) \times \\ &\qquad h(\sum s^{-1}(D_{\text{filtered}} \\ &\qquad\qquad \text{tuple }_{Tj})) \times \\ &\qquad h(\sum s^{-1}(D_{Ni}))) \bmod q \end{aligned} \tag{4}$$

Case 3. The enveloping subtree has a height > 2.

Apply Case 2 recursively up the subtree.

Therefore, the recipient can decrypt D_N with the public key of the central server, compute $h(\sum$ unsigned digest for result tuple $Tj)$ from the result tuples using formulas (1) and (2), combine the signed digests for the filtered tuples and nodes in D_S, and finally check that they satisfy the above equation. Since D_N and the digests in D_S are signed, and the digests for the result tuples are derived with a one-way function [6], it is practically impossible to manipulate the values of the result tuples without violating the above equation. Thus the VO is sufficient for verifying the correctness of the result for selection operations.

As the edge server is not necessarily secure, it cannot combine the signed digests in D_S for the client. Rather, the client needs the original signed digests to check their validity, before combining them to verify the query result. ¶

Projection: $\pi_{A1,..,Ak}(R) = \{< t.A1,..,t.Ak >|\ t \in R\}$ where Ai's are attributes of relation R.

Conceptually, constructing a VO for a projection operation is similar to handling selection on non-key attributes – the gaps within the result tuples left by the filtered attributes are accounted for by their respective attribute digests. Again, we can exploit the commutative property of the hash function h to consolidate those attribute digests into a set D_P without recording the attribute orders, and which attribute digest belongs to which result tuple. Thus, the VO remains linear in the size of the query results, containing only the signed digest for the node at the top of the enveloping subtree, D_S for the digests from the selection operation(s) as described above, and D_P for the projection operation(s). As with D_S, D_P contains the signed version of the digests for the filtered attributes, so they cannot be tampered with at the edge servers.

Figure 7 shows how the query result and the associated VO in Figure 6 change after a further projection operation.

Lemma 2: The VO is sufficient for verifying the correctness of the result for projection operations.

Proof: By definition (2), $D_T = s(h(\sum$ unsigned digest for attribute $Ai))$. For our choice of $h(x) = g^x \bmod q$, the formula can be translated to: unsigned digest for result tuple j = $(h(\sum$ unsigned digest for result attribute Ajk$) \times$ $h(\sum s^{-1}(D_{\text{filtered attribute } Ajk}))) \bmod q$. Plugging into formula (4), we have:

$$s^{-1}(D_N) = (h(\sum \text{unsigned digest for projected tuple } Tj) \times h(\sum s^{-1}(D_{\text{filtered attribute } Ajk})) \times$$

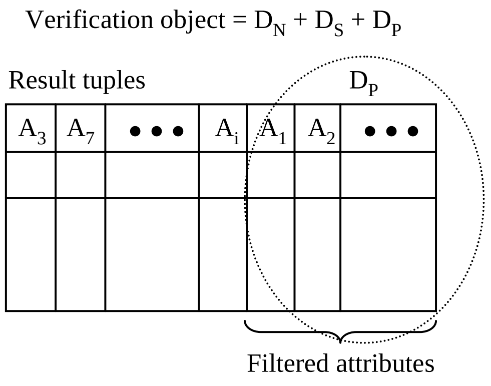


Figure 7. Example Query Result for Projection Operation

$$h(\sum s^{-1}(D_{\text{filtered tuple } Tj})) \times h(\sum s^{-1}(D_{Ni}))) \bmod q \qquad (5)$$

Since the VO contains the signed digest D_N, the signed digests for the filtered attributes in D_P, and the signed digests for the filtered tuples/nodes in D_S, the client can verify the results by checking whether the above equation holds. ¶

Join: $R \bowtie_C S$ where C is a condition of the form $A_i\ \Theta\ A_j$, A_i and A_j are attributes of relations R and S respectively, and $\Theta \in \{=, \neq, <, \leq, >, \geq\}$.

Unlike selection and projection that operate on single tables, a join operation could conceivably involve any two base tables, or even tables of intermediate results from other relational operations. It is thus infeasible for the central database server to construct and distribute VB-trees for all possible ad-hoc join operations.

Fortunately, in edge computing most of the database queries are not likely to be ad-hoc, but are embedded in application programs and hence known in advance. It is thus possible to materialize each join operation, and construct a VB-tree on the materialized view for authenticating the query results. Thus, join operations can be verified in a similar manner as selection-projection operations.

3.4. Update Operation

While queries can be processed by any edge server, update operations have to be channeled back to the central database server as they change the digests in the VB-trees, and only the central server possesses the private key for signing new digests. Depending on the consistency requirement, the DBMS can either lock and update the VB-trees at all the edge servers concurrently, or propagate the changes periodically. In the former case, a distributed concurrency control mechanism like basic 2PL [3], with the central server hosting the master copy, can be applied. For delayed broadcast of the updates to the edge servers, the central server can include the timestamp or version number in its public key, and make available to users the validity period of each public key at a well-known location. This would ensure that edge servers cannot masquerade out-of-date data, signed with an old private key, as the latest data without being detected by the recipients.

Insert:

When a tuple is to be inserted, the DBMS first calculates the attribute digests and the tuple digest D_T using formulas (1) and (2). D_T is then used to update the digests on the path from the root of the VB-tree to the leaf node that holds the new tuple. As the digest for each node is a combination of all the tuple digests under it, and as input digests to the hash function h are coalesced with a commutative operator, each node digest D_N on the path is simply updated using the formula:

$$D_N^{new} = s(h(s^{-1}(D_N^{old}) + s^{-1}(D_T)))$$

To ensure consistency, the DBMS needs to exclusively lock (X-lock) each digest in turn only as it is being modified. Finally, if the insertion necessitates a node to be split, the DBMS would have to X-lock the parent node, so the digests for the split nodes can be safely recorded in the parent. The locking protocol for VB-tree is the same as B-tree locking (e.g. see [2]).

Delete:

Unlike insertion, a tuple deletion transaction is more expensive to process. Since the SQL statement specifies only criteria on the tuples to be deleted rather than the entire tuple content, the tuples' contribution to each node digest cannot be reversed out immediately while the transaction traverses down the VB-tree. Instead, the transaction has to X-lock all of the digests on the path from the root of the VB-tree to the affected leaf nodes, delete the selected tuples, then re-calculate the digests back up to the root.

In the course of the delete transaction, other queries on the same table can still proceed as long as their enveloping subtrees do not overlap the tuples affected by the transaction. This is because a query requires only the digests in its own enveloping subtree, not the root of the entire index as in [5]. To ensure that there is no overlap with the transaction, each query should shared-lock the digests within its subtree.

4. Analysis and Evaluation

Having presented our authentication mechanism, we now analyze the overheads that it introduces. We begin by quantifying the storage overhead on the base tables and the index trees. Next, we look at the communication cost for transmitting the answer and the VO to the user. Following that, the costs incurred by the edge servers in constructing verification objects for queries, and the clients in verifying the query results are presented. Finally, we investigate the processing demand on the central server in updating the VB-trees for insertion and deletion transactions. The parameters used in the analysis are summarized in Table 1.

Parameter	Meaning	Default
$\mid D \mid$	Length of signed node/tuple/attribute digest (Bytes)	16
$\mid K \mid$	Length of search key (Bytes)	16
$\mid ptr \mid$	Length of node pointer (Bytes)	4
$\mid B \mid$	Size of block/node (KBytes)	4
T_R	Number of tuples/rows in table (million)	1
T_C	Number of attributes/columns in table	10
T_h	Height of the VB-tree	-
Q_R	Number of tuples/rows in query result	-
Q_C	Number of attributes/columns in query result	10
Q_h	Height of enveloping subtree for query result	-
$\mid A_i \mid$	Size of attribute A_i (Bytes)	-
$Cost_k$	Average cost for deriving an attribute digest with the 1-way hash function k	-
$Cost_h$	Cost for combining two digests with the 1-way hash function h	-
$Cost_s$	Cost for decrypting a signature, s^{-1}	-
X	The ratio between of $Cost_s$ and $Cost_h$	10

Table 1. Parameters

As a baseline for comparison, we use the naive strategy that transmits, for each answer tuple, the signed digest for verifying the tuple. We shall denote the VB-tree scheme as VB-tree, and the naive strategy as Naive. The cost formulas

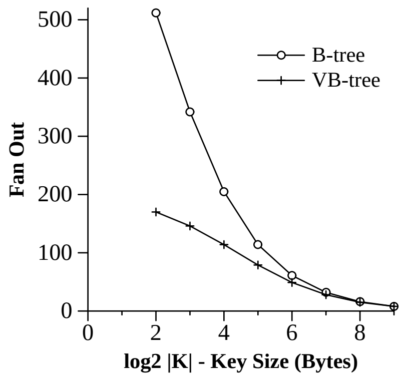


Figure 8. Index Tree Fan-Out versus Key Length

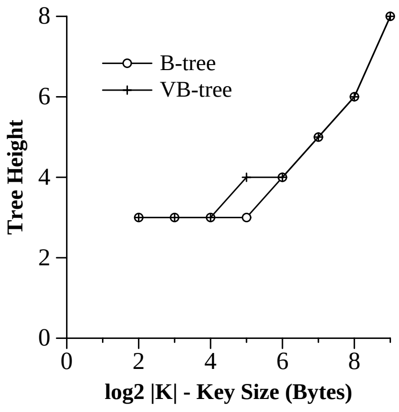


Figure 9. Index Tree Height versus Key Length

for the naive strategy is shown in the Appendix.

4.1. Storage Costs

Base Table:

Since each attribute in a database table is associated with a signed digest, the space overhead for the table is $T_R \times T_C \times \mid D \mid$ bytes.

VB-tree:

The fan-out factor of the original B-tree is:

$$f_{Btree} \;=\; \left\lfloor \frac{\mid B \mid - \mid ptr \mid}{\mid K \mid + \mid ptr \mid} \right\rfloor + 1$$

and the height of a fully packed B-tree is $\left\lceil \log_{(f_{Btree}-1)} T_R \right\rceil$.

In comparison, the fan-out factor of the VB-tree is:

$$f_{VBtree} \;=\; \left\lfloor \frac{\mid B \mid - \mid ptr \mid - \mid D \mid}{\mid K \mid + \mid ptr \mid + \mid D \mid} \right\rfloor + 1 \qquad (6)$$

This means that there is a space overhead of $f_{VBtree} \times \mid D \mid$ bytes per node. Moreover, the height of a fully packed VB-tree is:

$$T_h \;=\; \left\lceil \log_{(f_{VBtree}-1)} T_R \right\rceil \qquad (7)$$

To visualize the height differences, Figures 8 and 9 show the fan-out and height, respectively, of the B-tree versus the VB-tree for different key lengths. As expected, the VB-tree has a significantly smaller fan-out than B-tree when the key length is short relative to the size of the digest. However, as this does not translate to a material difference in height, the performance penalty for index traversal is minimal.

4.2. Query – Communication Cost

For each query, the edge server has to construct two sets of digests – D_S and D_P:

- D_S: Assuming that the query result is a range of contiguous tuples in the base table and that the VB-tree is fully packed, the height of the enveloping subtree is:

$$Q_h \;=\; \left\lceil \log_{(f_{VBtree}-1)} Q_R \right\rceil \qquad (8)$$

  At the most, there are $(f_{VBtree} - 1)$ digests each in the top node of the subtree, and the leftmost and rightmost nodes at each level of the subtree, to be copied to D_S. Hence the maximum number of digests that D_S contains is $(2Q_h - 1)(f_{VBtree} - 1)$.

- D_P: Since the query requires only Q_C attributes, there are $(T_C - Q_C)$ digests for the filtered attributes per result tuple. The total number of digests to copy into D_P, from across all the result tuples, is thus $Q_R \times (T_C - Q_C)$.

Therefore, the communication cost is determined by the size of the query answers, the digests in D_P and the digests in D_S. Assuming that the Q_C attributes are the first attributes $A_1 \ldots A_{Q_C}$, we have

$$\begin{aligned} Cost_{comm} \;=\; & Q_R \times \sum_{i=1}^{Q_C} |A_i| + \\ & Q_R \times \sum_{i=Q_C+1}^{T_C} |D| + \\ & (2Q_h - 1) \\ & \quad \times (f_{VBtree} - 1) \times |D| \end{aligned} \qquad (9)$$

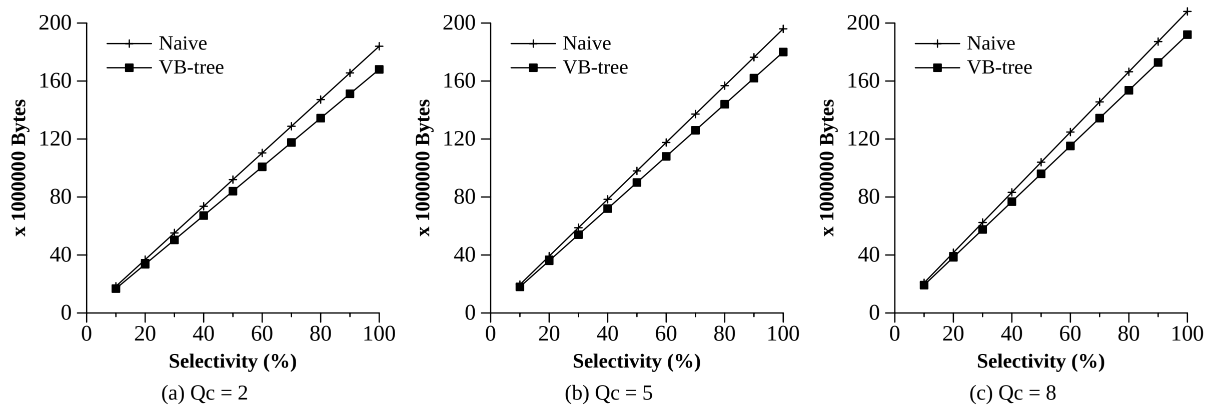


Figure 10. Query – Communication Cost

Figure 10 shows the communication cost of the proposed VB-tree and the Naive schemes for three scenarios – $Q_C = 2$, 5 and 8. Here, we vary the number of answer tuples by changing the selectivity factor, defined as Q_R/T_R. We also fix the size of tuples at 200 bytes with an average of 20 bytes per attribute. As expected, VB-tree incurs smaller transmission costs than Naive, because sending a signed digest per result tuple is expensive especially with high selectivity factors. Moreover, as Q_C increases, the communication cost also increases. This is the case because more attribute values are returned instead of their digests, and the attribute size (in our settings) is larger than the size of the digests.

In Figure 11, we profile the effect of the ratio between the attribute size and the digest size (by setting the attribute size to $2^{attrFactor+1}$, $attrFactor = 1, 2, .., 6$) for two selectivity factors, 20% and 80%, while keeping the default values for the other parameters. As shown in the figure, the communication cost of the two schemes converge gradually as the attribute size increases. This is reasonable because the cost of transmitting the query results dominates the overhead due to the digests. However, the cost differential is still significant in absolute terms. For example, compared to VB-tree, Naive incurs at least 3 MB more communication cost for selectivity factor of 20% and 12 MB more for selectivity factor of 80%.

4.3. Query – Computation Cost

After the query result and VO (i.e., D_S and D_P) are received, the client computer needs to compute the digests for the $Q_R \times Q_C$ attribute values in the result tuples, at a cost of $Q_R \times Q_C \times Cost_k$. Next, the client needs to decrypt the signed digests in D_S and D_P. This incurs a cost of $(2Q_h - 1)(f_{VBtree} - 1) \times Cost_s$ and $Q_R \times (T_C - Q_C) \times Cost_s$ respectively. Following that, the attribute digests are combined with D_P and D_S to de-

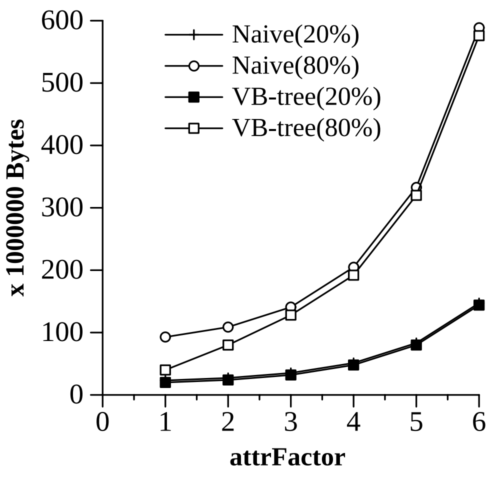


Figure 11. Attribute size = $2^{attrFactor+1}$

rive a final digest for comparison with the signed digest D_N in the VO. The total cost incurred by the client is thus:

$$\begin{aligned} Cost_{Query} = \; & Q_R \times Q_C \times Cost_k + \\ & (2Q_h - 1)(f_{VBtree} - 1) \\ & \quad \times Cost_h + \\ & Q_R \times (T_C - 1) \\ & \quad \times Cost_h + \\ & (2Q_h - 1)(f_{VBtree} - 1) \\ & \quad \times Cost_s + \\ & Q_R \times (T_C - Q_C) \\ & \quad \times Cost_s \end{aligned} \quad (10)$$

For large queries, we expect $Q_R \gg T_C$, Q_C, f_{VBtree} and Q_h, so $Cost_{Query}$ increases roughly linearly with the size of the query result, i.e., $Cost_{Query} = \mathbf{O}(Q_R)$. The reason is that most of the overheads are incurred in computing the digests for the attributes within the result tuples,

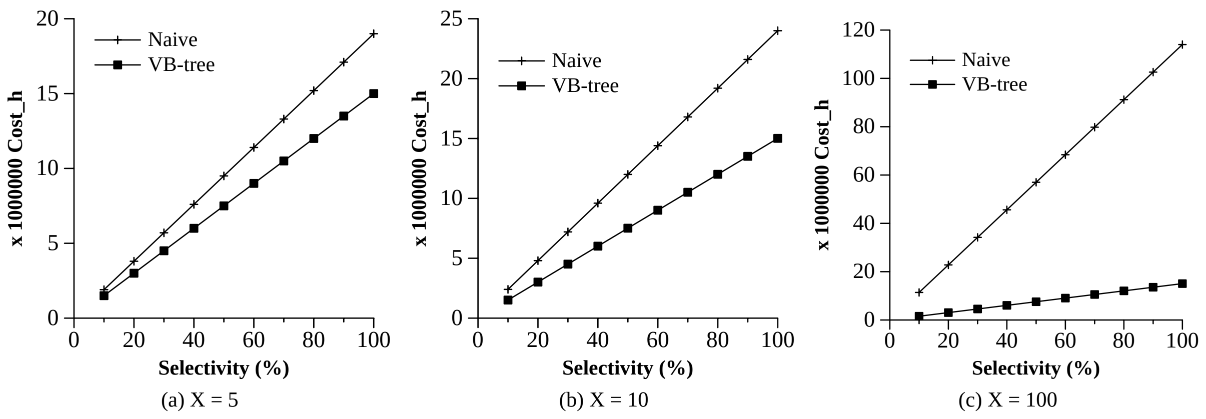


Figure 12. Query – Computation Cost for different X

and then verifying the tuples. The observation is confirmed in Figure 12, which plots $Cost_{Query}$ for various selectivity factors for the default values in Table 1.

According to [15], hash functions are about 100 times faster than signature verification and 10000 times faster than signature generation. This is because public key signature operations involve large prime modulus and are computationally expensive. Clearly, Naive incurs more processing cost as it needs to decrypt the signature for each result tuple. Figure 12 compares the computation cost of VB-tree and Naive for $X = 5$, 10 and 100, where X is the ratio of $Cost_s$ to $Cost_h$. The results clearly show the superiority of VB-tree over Naive since it essentially reduces the number of decryptions needed for verifying the answers. Moreover, the performance difference between the two schemes widens as X increases.

Figure 13 shows more sensitivity analysis of the query cost. In Figure 13(a), we study the effect of $\frac{Cost_k}{Cost_h}$ for selectivity factors of 20% and 80% while keeping the other parameters at their default settings. As the figure shows, VB-tree still outperforms Naive. Moreover, the difference between VB-tree and Naive remains almost constant. This is because the difference in the cost components comes largely from the cost of decrypting the signatures which is independent of $Cost_k$ and $Cost_h$.

In Figure 13(b), we study the effect of Q_C for selectivity factors of 20% and 80%. The results show that Q_C has little effect on the relative performance of the two schemes either, for the same reasons that the main determinant of the performance difference between the two schemes is the decryption cost of the signatures.

4.4. Update Costs

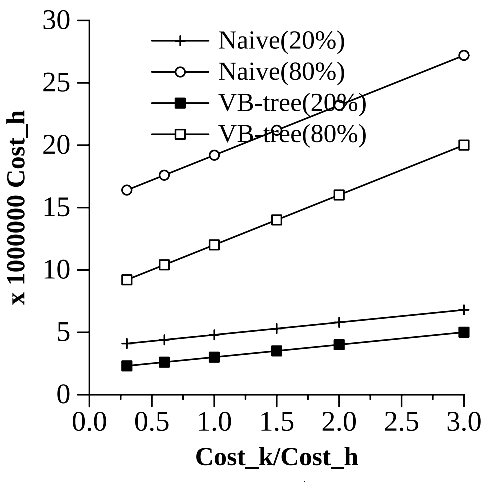
When a new tuple is inserted, the central server computes the digests for the T_C attribute values, and combines them to derive the tuple digest D_T. Next, the digest for each node on the path from the root of the VB-tree to the leaf must be combined with D_T. Assuming that the VB-tree is fully packed, the number of nodes on this path is $\lceil \log_{(f_{VBtree}-1)} T_R \rceil$. The insertion cost is thus:

$$\begin{aligned} Cost_{Insert} &= T_C \times Cost_k + \\ &\quad (T_C - 1) \times Cost_h + \\ &\quad \lceil \log_{(f_{VBtree}-1)} T_R \rceil \\ &\quad \times Cost_h \end{aligned} \tag{11}$$

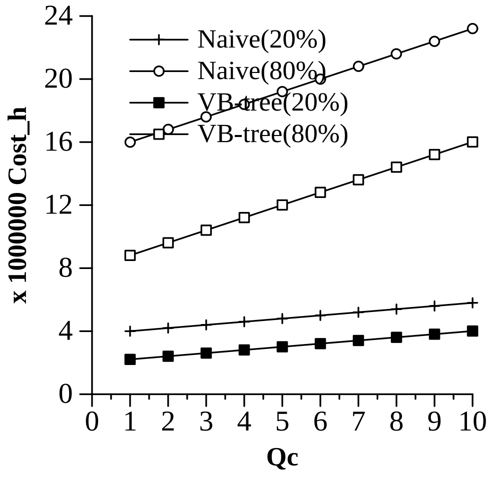
For the deletion of a range of contiguous tuples from the base table, the operation removes some entries from the nodes at the top, left and right boundaries of the enveloping subtree, and empties out the other nodes within the subtree. Denoting the number of deleted tuples with Q_R, the height of the enveloping subtree is $Q_h = \lceil \log_{(f_{VBtree}-1)} Q_R \rceil$ and the number of nodes at the boundaries is $2Q_h - 1$. With at most $f_{VBtree} - 1$ child pointers remaining in each of those nodes, the cost for recomputing the node digests is $(2Q_h - 1)(f_{VBtree} - 1)Cost_h$. Finally, the digests for the nodes on the path from the enveloping subtree up to the root of the VB-tree must also be updated. Each of these nodes can have at most f_{VBtree} child pointers, giving rise to a cost of $(T_h - Q_h) \times f_{VBtree} \times Cost_h$. The total cost for the deletion is, therefore:

$$\begin{aligned} Cost_{Delete} &= ((2Q_h - 1)(f_{VBtree} - 1) + \\ &\quad (T_h - Q_h)f_{VBtree}) \\ &\quad \times Cost_h \\ &= [(T_h + Q_h - 1)f_{VBtree} \\ &\quad - 2Q_h + 1] \times Cost_h \end{aligned} \tag{12}$$

The formula does not account for node merges that are expected to occur only rarely. The reason is that real database



(a) Effect of $cost_k/cost_h$



(b) Effect of Q_C

Figure 13. Query – Computation Cost for X = 10

systems usually do not require their B-tree nodes to actually contain at least d entries, so nodes are merged only when they become empty [9].

5. Conclusion

In this paper, we propose a mechanism for verifying the query results produced by proxy servers in edge computing. The idea is for the trusted central DBMS to maintain and distribute verifiable B-trees (VB-tree) on the base tables, that the edge servers use to create a verification object (VO) for each query. This VO enables the recipient to check the integrity of the query result – that the values of the result tuples have not been tampered with, and that no spurious tuples are introduced. It is computationally infeasible for an edge server to compromise the integrity of the result, without invalidating the VO and being detected.

One of the primary contributions of this paper over existing work is that the proposed mechanism creates VOs that are linear in the size of the query answers, and independent of the database size. Moreover, all the relational operations can be performed by the edge servers; for example, we do not push projection operations to the clients. Hence, our mechanism lowers the transmission overhead, and also the storage and processing demands on the clients. Finally, we address how the VB-trees can be updated dynamically while ensuring data consistency.

6. *

Acknowledgements

We are grateful to Robert H. Deng, Feng Bao and Chee-Yong Chan for helpful discussions.

References

[1] Secure hash standard (shs). *National Institute of Standards and Technology, FIPS Publication*, 180-1, April 1995.

[2] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. In *Acta Informatica*, volume 9(1), pages 173–189, 1977.

[3] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. In *ACM Computing Surveys*, volume 13(2), pages 185–221, June 1981.

[4] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, and G. Stubblebine. Flexible authentication of xml documents. In *Proceedings of the 8th ACM Conference on Computer and Communications Security*, pages 136–145, 2001.

[5] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic data publication over the internet. In *14th IFIP 11.3 Working Conference in Database Security*, pages 102–112, 2000.

[6] A. Evans, W. Kantrowitz, and E. Weiss. A user authentication system not requiring secrecy in the computer. In *Communications of the ACM*, volume 17(8), pages 437–442, August 1974.

[7] M. Goodrich, R. Tamassia, and A. Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *DARPA Information Survivability Conference and Exposition (DISCEX II)*, volume 2, pages 1068–1084, 2001.

[8] R. Housley, W. Ford, W. Polk, and D. Solo. Internet x.509 public key infrastructure certificate and crl profile. In *RFC 2459*, 1999.

[9] T. Johnson and D. Shasha. Utilization of b-trees with inserts, deletes and searches. In *Proceedings of the Eighth ACM Symposium on Principles of Database Systems*, pages 235–246, 1989.

[10] D. Margulius. Apps on the edge. *InfoWorld*, 24(21), May 2002. http://www.infoworld.com/article/02/05/23/020527feedgetci_1.html.

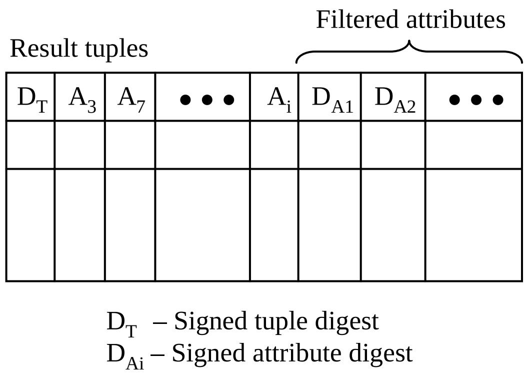


Figure 14. The Naive Strategy of Verifying Result Tuples

[11] R. Merkle. A certified digital signature. In *Proceedings of Advances in Cryptology-Crypto '89, Lecture Notes in Computer Science*, volume 0435, pages 218–238. Springer-Verlag, 1989.

[12] M. Naor and K. Nissim. Certificate revocation and certificate update. In *Proceedings of the 7th USENIX Security Symposium*, pages 217–228, 1998.

[13] C. Peng, R. Deng, Y. Wu, and W. Shao. A flexible and scalable authentication scheme for jpeg2000 codestreams. In *Proceedings of the ACM Multimedia*, November 2003.

[14] R. Rivest. Rfc 1321: The md5 message-digest algorithm. *Internet Activities Board*, April 1992.

[15] R. Rivest and A. Shamir. PayWord and MicroMint: Two simple micropayment schemes. In *http://theory.lcs.mit.edu/ rivest/RivestShamir-mpay.pdf (This version is dated 2001). An earlier version appears in Security Protocols, Lecture Notes in Computer Science, LNCS 1189, pp. 69-87*, 2001.

[16] M. Roos, A. Buldas, and J. Willemson. Undeniable replies for database queries. In *Proceedings of the Baltic Conference, BalticDB&IS*, pages 215–226, 2002.

Appendix

In this appendix, we present the communication and computation costs of the naive strategy. The naive strategy maintains for each attribute a signed digest, and for each tuple a signed digest obtained from the attribute digests. It transmits the result tuples together with their attribute and tuple digests for the client to verify the correctness of the result tuples. Figure 14 illustrates the naive strategy.

We shall show only the formulas that are used in our comparative study in Section 4. Let Q_R be the number of tuples in the result of a selection query. Let T_C be the number of attributes in the queried relation, and Q_C be the number of attributes in the query result. Moreover, suppose that the answers are consecutive tuples in the queried relation.

Communication Cost

The transmission cost includes the signed tuple digest for each result tuple, the actual value of each required attribute, and the signed digest for every filtered attribute.

$$\begin{aligned} Naive_{comm} &= Q_R \times \sum_{i=1}^{Q_C} |A_i| + \\ & \quad Q_R \times |D| + \\ & \quad Q_R \times \sum_{i=Q_C+1}^{T_C} |D| \end{aligned}$$

Computation Cost

For each attribute in the answer, we need to compute the digest. For each result tuple, we need to compute the tuple digest by combining the computed attribute digests and the decrypted digests for the filtered attributes. Finally, we need to compare it with the signed digest (after decrypting it) that the edge server returns for that result tuple. Therefore, the computation cost is:

$$\begin{aligned} Naive_{query} &= Q_R \times Q_C \times Cost_k + \\ & \quad Q_R \times (T_C - 1) \times Cost_h + \\ & \quad Q_R \times Cost_s + \\ & \quad Q_R \times (T_C - Q_C) \times Cost_s \end{aligned}$$