Proof-Infused Streams: Enabling Authentication of Sliding Window Queries On Streams

Feifei Li†   Ke Yi‡   Marios Hadjieleftheriou‡   George Kollios†

†Computer Science Department
Boston University
{lifeifei, gkollios}@cs.bu.edu

‡AT&T Labs Inc.
{yike, marioh}@research.att.com

ABSTRACT

As computer systems are essential components of many critical commercial services, the need for secure online transactions is now becoming evident. The demand for such applications, as the market grows, exceeds the capacity of individual businesses to provide fast and reliable services, making outsourcing technologies a key player in alleviating issues of scale. Consider a stock broker that needs to provide a real-time stock trading monitoring service to clients. Since the cost of multicasting this information to a large audience might become prohibitive, the broker could outsource the stock feed to third-party providers, who are in turn responsible for forwarding the appropriate sub-feed to clients. Evidently, in critical applications the integrity of the third-party should not be taken for granted. In this work we study a variety of authentication algorithms for selection and aggregation queries over sliding windows. Our algorithms enable the end-users to prove that the results provided by the third-party are correct, i.e., equal to the results that would have been computed by the original provider. Our solutions are based on Merkle hash trees over a forest of space partitioning data structures, and try to leverage key features, like update, query, signing, and authentication costs. We present detailed theoretical analysis for our solutions and empirically evaluate the proposed techniques.

1. INTRODUCTION

Online services, like electronic commerce and stock market applications, are now permeating our modern way of life. Due to the overwhelming volume of data that can be produced by these applications, the amount of required resources, as the market grows, exceeds the capacity of individual businesses to provide fast and reliable services.

Consider the following example. A stock broker needs to provide a real-time stock trading monitoring service to clients. Since the cost of multicasting this information to a large audience might become prohibitive, as many clients could be monitoring a large number of individual stocks, the broker could outsource the stock feed to third-party
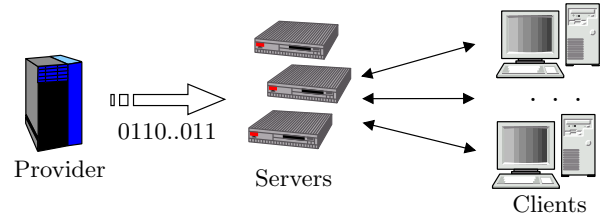
Figure 1: The outsourced stream model.

providers, who would be in turn responsible for forwarding the appropriate sub-feed to clients (see Figure 1). Evidently, especially in critical applications, the integrity of the third-party should not be taken for granted, as the latter might have malicious intent or might be temporarily compromised by other malicious entities. Deceiving clients can be attempted for gaining business advantage over the original service provider, for competition purposes against important clients, or for easing the workload on the third-party servers, among other motives. In that respect, assuming that the original service provider can be trusted, an important consideration is to give clients the ability to prove the service furnished by the third-party (a.k.a. the server).

Consider an end-user that issues a monitoring query for the moving average of a specific stock within a sliding window. This monitor can be viewed as a selection-aggregation query over the original stock feed. A third-party server is assigned to forward the aggregate over all qualifying trades to this client. Given that the server has total access over the feed, it can 1. drop trades; 2. introduce spurious trades; or 3. modify existing trades. Similar threats apply for selection queries, e.g., reporting all bids larger than a threshold within the last hour, and group by queries, e.g., reporting the potential value per market of a given portfolio. It will be the job of the stock broker to proof-infuse the stream such that the clients are guarded against such threats, guaranteeing both *correctness* and *completeness* of the results.

In this work we concentrate on *authenticated one-shot and sliding window queries on data streams.* To the best of our knowledge, no work has studied the authentication of exact selection and aggregation queries over streaming outsourced data. We introduce a variety of authentication algorithms for answering multi-dimensional (i.e., on multiple attributes) selection and aggregation queries. One-shot window queries report answers computed once over a user defined temporal range. Sliding window queries report answers continuously as they change, over user defined window sizes and update intervals. We assume that clients register a multiplicity of

ad-hoc queries with the servers, which in turn compute the results and forward the answers back to the clients along with the necessary signatures and authentication information needed to construct a proof of correctness. For one-shot queries the servers construct answers and proofs, and send the results back to clients. For sliding window queries, the servers update the query results incrementally, communicating only authenticated changes to the clients. By ad-hoc queries we mean that clients can register and cancel queries at any time, using arbitrary window sizes (sliding or not) and update intervals.

It should be emphasized here that one-shot queries are not only interesting in their own right, but also because they are an essential building block for answering sliding window queries: The initial answer of a sliding window query is constructed using a one-shot algorithm. We design one set of authentication structures optimized for one-shot window queries, and another for ad-hoc sliding window queries. Then, we combine both solutions to provide efficient algorithms for all settings. Our solutions are based on Merkle hash trees over a forest of space partitioning data structures, and balance key features, like update, query, signing and authentication cost, from the perspective of the service provider, the server and the client.

The contributions of this paper are: 1. Designing a variety of authentication algorithms for multi-dimensional selection and aggregation queries on data streams, concentrating both on one-shot and variable-sized sliding window queries; 2. Deriving detailed theoretical performance bounds for all algorithms over a variety of cost metrics; 3. Conducting a comprehensive empirical evaluation using real data sets, validating the practicality of the proposed schemes.

This paper is organized as follows. Section 2 formally defines the problem and presents essential cryptographic tools. Sections 3 and 4 present solutions for sliding window queries. Section 5 presents solutions for one-shot window queries. Section 6 summarizes our results. The empirical evaluation is conducted in Section 7. Related work is discussed in Section 8, before concluding the paper.

2. PRELIMINARIES

2.1 Problem formulation

Stream outsourcing. We adopt the traditional data outsourcing model in a streaming environment. Formally, we define three entities, the *service provider* who is the originator of the stream, the *server* who answers queries, and the *client* who registers queries and receives authenticated results (see Figure 1). The service provider constructs special authentication structures that can be updated in real-time over the stream and that are tailored for answering one-shot and sliding window selection and aggregation queries. The provider forwards the original stream to the server along with the necessary information to enable reconstruction of the authentication structures at the server side. The server uses these structures to generate verifiable query answers, and forwards the final results to the clients.

The data stream model. We model a data stream S as an infinite sequence of tuples $S = (a_1, a_2, \dots)$. Tuples arrive one at a time, i.e., in the i-th time unit tuple a_i arrives. A sliding window of size n consists of elements $(a_{t-n+1}, \dots, a_t)$ where a_t is the last tuple received so far, and n is in number

of tuples. This model is typically referred to as *tuple-based sliding windows* [10, 4] (i.e., querying the most recent n tuples). In some applications it might be desirable to use *time-based sliding windows*, where each tuple is associated with a *timestamp* and we are interested in querying all tuples within time interval $[t_{\text{now}} - T, t_{\text{now}}]$ where t_{now} is the current time and T is the window size in timestamps. For ease of exposition we focus on tuple-based sliding windows, and discuss extensions for time-based sliding windows in Section 6.

Queries. Assume that each tuple consists of multiple attributes and clients issue continuous selection queries of the following form:

```
SELECT * FROM Stream WHERE
```
$l_1 \leq A_1 \leq u_1$ `AND ... AND` $l_d \leq A_d \leq u_d$
```
WINDOW SIZE n, SLIDE EVERY σ
```

where (l_i, u_i) are the selection ranges over attributes A_i.[1] We will also consider aggregation queries of a similar form:

```
SELECT AGG(Ax) FROM Stream WHERE ...
```

A_x is any tuple attribute and `AGG` is any distributive aggregate function, like SUM, COUNT, MIN, and MAX.

Assume that a query is issued at time t. The answer to a selection query consist of those tuples that fall within the window $(a_{t-n+1}, \dots, a_t)$ and whose attributes satisfy the selection predicates. For one-shot queries the server constructs the answer once and reports the result. For sliding window queries, the server constructs the initial answer at time t, which is again a one-shot query, and incrementally communicates the necessary changes to the clients, as tuples expire from the window and new tuples become available.

Our authentication algorithms will guarantee that the server does not introduce any spurious tuples, does not drop any tuples, and does not modify any tuples. In other words, our techniques guarantee both *correctness* and *completeness* of the results. Similarly for aggregation queries, we will guarantee that the aggregate is computed over the correct set of tuples, and properly updated over the sliding window.

An important observation for authenticating queries in a streaming setting is that any solution that can provide authenticated responses on a per tuple basis will have to trigger a signing operation at the provider on a per tuple arrival basis, which is very costly (see Section 2.2). The only alternative is to amortize the signing cost by performing signing operations across several tuples. This approach will lower the update overhead at the cost of providing delayed query responses. In many applications, delayed responses can often be tolerated and, given the complexity of this problem, our main interest will be to design algorithms that minimize signing, authentication and querying costs, given a maximum permissible response delay b. For one-shot window queries, clients will receive replies in the worst case b tuple arrivals after the time that the query was issued. For sliding window queries, clients will receive necessary updates with at most a b-tuple delay. Even though we introduce delays, we do not change the ad-hoc window semantics: The answers provided are with respect to the user defined window specifications t, n, σ. In critical applications (e.g., anomaly detection) preserving window semantics is very important.

---

[1]The selection attributes can be categorical as well, but without loss of generality, we will concentrate on numerical attributes in the paper.
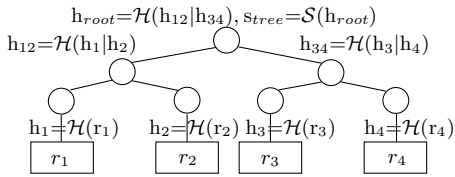
$$h_{root}=\mathcal{H}(h_{12}|h_{34}),\; s_{tree}=\mathcal{S}(h_{root})$$
$$h_{12}=\mathcal{H}(h_1|h_2) \qquad h_{34}=\mathcal{H}(h_3|h_4)$$
$$h_1=\mathcal{H}(r_1) \quad h_2=\mathcal{H}(r_2) \quad h_3=\mathcal{H}(r_3) \quad h_4=\mathcal{H}(r_4)$$

Figure 2: A Merkle hash tree.

A straightforward approach for answering sliding window queries would be to re-compute the exact answer every σ tuple arrivals, which is unacceptable. Alternatively, we could maintain the answers of all registered queries at the server side, update them incrementally as the window evolves, and communicate only the necessary changes to clients. This solution is unacceptable in streaming settings with a large number of queries and clients, putting unnecessary storage burden on the server. Hence, we explore solutions that incrementally compute result updates without explicitly keeping state on a per query/per client basis. In addition, we will assume that a maximum permissible window size N has been determined, and we will require our structures to have storage linear or near-linear in N.

Finally, we will evaluate our solutions, both analytically and experimentally, using the following metrics: 1. The query cost for the server; 2. The communication overhead for the authentication information, i.e., the size of the *verification object*, or the $\mathcal{VO}$. 3. The authentication cost for the client; 4. The update cost for the provider and the server; 5. The storage cost for the provider and the server; 6. Support for multi-dimensional queries.

2.2 Cryptographic essentials

Collision-resistant hash functions. A hash function $\mathcal{H}$ is an efficiently computable function that takes a variable-length input x to a fixed-length output $y = \mathcal{H}(x)$. *Collision resistance* states that it is computationally infeasible to find two inputs $x_1 \neq x_2$ such that $\mathcal{H}(x_1) = \mathcal{H}(x_2)$. An example of a practical hash function is SHA1 [29], which takes variable-length inputs to 160-bit (20-byte) outputs.

Public-key digital signature schemes. A public-key digital signature scheme [16] is a tool for authenticating the integrity and ownership of a signed message. The signer generates a pair of keys (SK, PK), keeps the secret key SK secret, and publishes the public key PK. Subsequently, for any message m sent by the signer, a signature $s_m = \mathcal{S}(SK, m)$ is produced. The recipient of s_m and m can verify s_m via $\mathcal{V}(PK, m, s_m)$. A valid signature assures the recipient that the message was signed by the owner of (SK, PK), and that no tampering has occurred.

Relative cost of cryptographic operations. It is imperative to evaluate the relative cost of cryptographic operations in order to design efficient solutions, since such operations will adversely affect update costs. Based on experiments with two widely used cryptography libraries, OpenSSL [30] and Crypto++ [9], we obtained results for hashing, signing and verifying. Evidently, the cost of one hashing operation on our testbed computer takes approximately 1 to 2 μs. A signing operation is 1,000 times slower, while a verification operation 100 times slower. It is clear that hashing should be chosen over signing wherever possible.

The Merkle hash tree. The straightforward solution for verifying a set of n values is to generate n digital signatures. An improvement on this solution is the *Merkle hash tree* [24], or simply the *Merkle tree*. Its basic idea is exactly to replace signatures with the much cheaper hashes. The Merkle tree is a binary tree where each leaf contains the hash of a data value, and each internal node contains the hash of the concatenation of its two children (see Figure 2). Verification of data values is based on the fact that the hash value of the root of the tree is authentically published using a digital signature s. To prove the authenticity of any value, the prover provides the verifier with the data value itself and the hash values of the siblings of the nodes that lie in the path that connects the root of the tree with this data value. The verifier, by iteratively computing and concatenating the appropriate hashes, can recompute the hash of the root and verify its correctness using s. Correctness is guaranteed due to, in addition to the security of the public-key digital signature for the hash value of the root node, the collision-resistance of the hash functions. By hashing a given node, it becomes computationally infeasible for an adversary to modify the node in a way that ultimately preserves the hash value of the root. The correctness of any data value can be proved at the cost of computing $\log n$ hash values plus verifying one signature. The Merkle tree concept can also be used to authenticate range queries using binary search trees (where data entries are sorted), and it has been shown how to guarantee completeness of the results as well (by including boundary values in the results) [23]. External memory Merkle B-trees have also been proposed [32, 22]. Finally, it has been shown how to apply the Merkle tree concept to more general data structures [23].

3. THE TUMBLING MERKLE TREE

In this section we present a structure based on the Merkle binary search tree for answering one-dimensional sliding window queries. Consider a predefined maximum window size N, a maximum permissible response delay b, and ad-hoc queries with window sizes $n < N$, on selection attribute A. The provider builds one Merkle binary search tree on the values of the attribute A for every b tuples within the window of the most recent $N + b$ tuples $(a_{t-N+1-b}, \ldots, a_t)$. Note that tuples in the tree are sorted by value A and not by order of arrival. An example is shown at the top of Figure 3, where every rectangle represents a Merkle tree on attribute A. Then, the provider signs the $\lceil N/b \rceil + 1$ Merkle trees. Each signature also includes $t_\vdash$ and $t_\dashv$, the indices of the oldest and newest tuple contained in the tree, i.e., $s = \mathcal{S}(SK, h_{root}|t_\vdash|t_\dashv)$. All signatures are forwarded to the server. We call this structure the *Tumbling Merkle Tree* (TM-tree). Updating the TM-tree is easy. On every b new tuple arrivals the provider and the server bulk load a new Merkle tree with the new values and discard the oldest tree. In addition, the provider signs the tree and propagates the signature to the server. An advantage of this solution is that it amortizes the signing cost over every b tuples. The penalty is that authenticated results are provided with a delay up to b tuples. When the server receives a query but has not obtained the up-to-date signature from the provider, it temporarily buffers the query. When the next signature is received, the oldest buffered query refers to a sliding window with a low boundary at most $N + b$ tuples in the past.
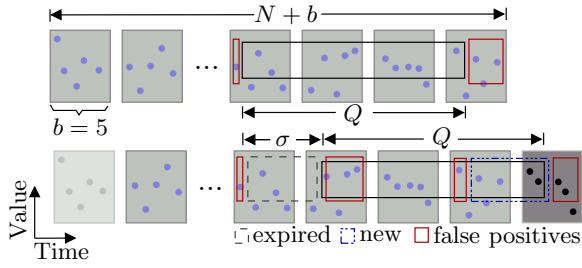
Figure 3: The Tumbling Merkle Tree.

One-Shot Queries. Consider a one-shot window query of size n. To answer the query the server traverses the $\lceil n/b \rceil + 1$ Merkle trees whose corresponding time intervals together cover the query time window, reporting all tuples satisfying the selection predicates (see Figure 3). For every tree, a *verification object* ($\mathcal{VO}$) is constructed using the original Merkle tree algorithm, and it is extended with the indices of the oldest and newest tuple in the tree. For trees fully contained in the query window, all tuples satisfying the selection predicate need to be reported (plus two boundary tuples for proof of completeness [23]). For the (at most) two boundary Merkle trees, up to $2b$ false positive answers might be returned (false positives are introduced in the temporal dimension though they are all satisfying tuples in the value dimension defined by the selection attribute A; see Figure 3). The server includes the false positives in the result, and leaves the filtering task to the client—this is necessary for proper authentication. Now, upon receiving a reply the client verifies correctness by authenticating the result of every tree individually; only then it filters out false positives from tuples in the two boundary Merkle trees whose indices are outside the query window. Correctness stems from the properties of Merkle trees discussed in Section 2. Completeness is guaranteed by proving that all qualifying values have been returned and that the right $\lceil n/b \rceil + 1$ trees have been traversed. Completeness of the values returned from each individual tree is guaranteed from the original Merkle binary search tree verification algorithm by including the left and right boundary values in the result [23, 22]. Verifying that the correct trees have been traversed is possible by examining the indices of the oldest and newest tuples in the trees, whose values have been verified using the signature of each tree. Since trees do not intersect in the temporal dimension, we expect those indices to be consecutive and cover the query window completely.

Assuming that k is the result size of the query, the TM-tree achieves the following.

LEMMA 1. *Given a maximum permissible response delay b, the TM-tree uses $O(N)$ space, and requires $O(\log b)$ time and $O(1/b)$ signing operations amortized to process a tuple. It takes $O(n/b \cdot \log b + b + k)$ time to answer a one-shot window query, and provides a $\mathcal{VO}$ of size $O(n/b \cdot \log b + b)$. The client takes $O(n/b \cdot \log b + b + k)$ time to authenticate the results.*

Note that we interpret the $\mathcal{VO}$ as all information transmitted to the client besides the query results, so the $\mathcal{VO}$ size above also includes the false positives.

Sliding Window Queries. Consider a sliding window query issued at time t, with window size n and sliding period σ. The server first constructs the initial answer to the

query for the window starting at t as described for one-shot queries, and sends the results to the client with delay at most b. Then, the server has to keep the results up-to-date as follows. Every b tuples, if a sliding period has ended, it constructs a $\mathcal{VO}$ that contains the tuples that have expired from the query window and the new tuples that have been generated since the last update to the client. The expired tuples can be found simply by traversing the (at most) $\lceil \sigma/b \rceil + 1$ Merkle trees at the left-most boundary of the sliding window; the new tuples can be found by querying the (at most) $\lceil \sigma/b \rceil + 1$ new Merkle trees that have been created since the last update. The cost of both operations is $O(\lceil \sigma/b \rceil \log b + b + k)$, where k is the total number of new result tuples in the current query window and expiring tuples in the previous query window. Note that the server may return up to $4b$ false positives (see Figure 3), which need to be filtered out by the client. False positives correspond to values from the boundary Merkle trees (expired and new) and also values that have appeared in the result already and are reported again. Notice that in order to construct an update the server is oblivious to the previous state of the query answer. Hence, no per query/client state needs to be retained. Furthermore, for large n, updating the result is more efficient than reconstructing the answer from scratch, since a large number of intermediate Merkle trees do not have to be traversed. Correctness and completeness is guaranteed given that the answers provided by every Merkle tree can be authenticated individually, verifying that both the expired and the new tuple sets are correct. Clearly, if $\sigma < b$, the server cannot do better than reporting updates only once every b tuples. If $\sigma > b$, the server reports updates every σ tuples with a maximum delay of b.

LEMMA 2. *The TM-tree can support sliding window queries with a per period cost $O(\lceil \sigma/b \rceil \log b + b + k)$ and $\mathcal{VO}$ size $O(\lceil \sigma/b \rceil \log b + b)$. The client spends $O(\lceil \sigma/b \rceil \log b + b + k)$ time to authenticate the results.*

Supporting aggregations. Now we consider aggregate queries. Take SUM as an example. The Merkle binary search tree can support range-sum queries by using standard techniques [20, 21]. Assume that all data values are stored on the leaf nodes. We associate with every internal node u, the sum ($sum(u)$) of the aggregation attribute of all tuples stored in the subtree rooted at u. To enable authentication, we include this sum into the hash value stored in the parent node. More precisely, for an internal node u with children v and w, the hash value of u is computed as $h_u = \mathcal{H}(h_v | h_w | sum(u))$. Now, authenticating a SUM aggregate can be done efficiently by retrieving the covering set of leaf and internal nodes that span the query range, and computing the aggregate without having to traverse all the leaf nodes satisfying the selection predicate. It is easy to show that the covering set has a logarithmic size. The correctness and completeness proof is a technicality, and full details can be found in [21]. Similar techniques can be used for all other distributive aggregates.

The TM-tree can be used for answering aggregation queries as well. For every Merkle tree that is completely contained in the window, we return the sum of all tuples falling in the selection range. However, for the two trees crossing the boundary, the server needs to return all the tuples in the selection range, instead of just the sum, so that the client can compute the sum of the tuples that are both in the selection range and in the query window. The same analysis

applies to sliding window queries and for all other distributive aggregates, except MIN and MAX. For these aggregates, in addition the client has to maintain the MIN/MAX of the (at most) $\lceil n/b \rceil$ trees that do not contain any expiring tuples. These values will be needed for recomputing the total aggregate after an update is received. Without giving the details of the derivation, which is rather straightforward, we conclude that for aggregation queries, all the aforementioned bounds hold by setting $k = 1$.

4. THE TUMBLING MKD-TREE

So far we have an efficient solution for one-shot and sliding window queries that has three drawbacks: it introduces $O(b)$ false positive answers (leading to a large $\mathcal{VO}$ size for large b), supports only one-dimensional queries, and has a high one-shot query cost. In this section we address the first two drawbacks and the third one in the next section. Notice that selection queries on variable window sizes with multiple selection predicates are essentially a range searching problem. This motivates the use of multi-dimensional range searching structures, like kd-trees [5, 11], range trees [11], or R-trees [18]. In this work we will use authenticated *Merkle kd-trees* (Mkd-trees) as a building block for our solutions. The Mkd-tree is of independent interest for general authenticated range searching problems, but also a very good candidate for streaming settings, since it is a main memory data structure that can be bulk-loaded very efficiently and provides *guaranteed worst case performance* (as opposed to R-trees). Nevertheless, our techniques are not designed specifically for kd-trees; any space partitioning structure, such as R-trees, can be authenticated in the same manner.

4.1 The kd-tree

We briefly review the kd-tree in two dimensions; the extension to higher dimensions is straightforward. For simplicity we assume that all coordinates are distinct; if this is not the case, standard techniques such as *symbolic perturbation* [11] can be used to resolve degeneracy. The kd-tree is a balanced binary tree $\mathcal{T}$. To build $\mathcal{T}$ on a set P of n points, we first divide P into two subsets P_l and P_r using a vertical line that divides the points into two sets of approximately equal size. We store the dividing line at the root node of $\mathcal{T}$ and continue with sets P_l and P_r as the left and right children of the root, by dividing each set into two subsets using a horizontal line. We continue recursively for all sets, alternating the direction of the dividing line for every level of the tree. The recursion stops when a set consists of only one point, which is stored as a leaf node of the tree. Each node u of the tree is naturally associated with a *bounding box*, denoted $box(u)$, which encloses all the points stored in the subtree rooted at u (see Figure 4).

To answer an orthogonal range search query Q, we start from the root of $\mathcal{T}$ and traverse all nodes whose associated bounding boxes intersect with or are contained in the query range Q. Please refer to Figure 4 for a query example where the query range is indicated by the small dark rectangle and the nodes accessed in the kd-tree are marked by the gray color. It is known that the kd-tree has excellent performance in practice, and furthermore, the following guarantee:

LEMMA 3   ([5]). *Let $\mathcal{T}$ be the kd-tree built on a set of n points. For any orthogonal query Q, the number of nodes in $\mathcal{T}$ whose bounding boxes intersect Q is at most $O(\sqrt{n} + k)$,*
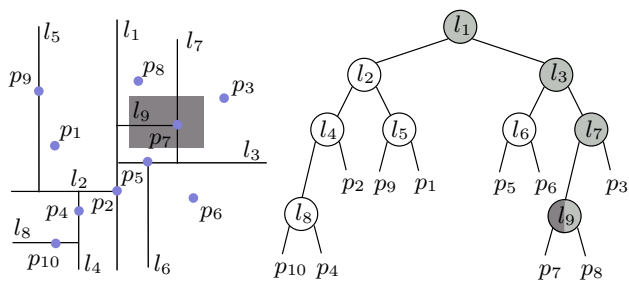

Figure 4: A kd-tree.

where k is the number of points in the result, and the number of nodes whose bounding boxes intersect any side of Q (i.e., the boundary $\partial(Q)$ of Q) is at most $O(\sqrt{n})$.

4.2 The Mkd-tree

We can extend the kd-tree with authentication information similarly to the Merkle tree. The authenticated Mkd-tree stores one hash value for every node. A leaf node v that contains point p stores hash value $h_v = \mathcal{H}(p)$. An internal node u with children v and w and dividing line l_u stores hash value $h_u = \mathcal{H}(h_v|h_w|l_u)$. The hash value of the root of the tree is signed by the data provider, producing signature $s = \mathcal{S}(SK, h_{\text{root}})$.

The query algorithm. To answer a range query Q, we recursively visit all nodes whose bounding boxes intersect with or are contained in Q. In addition to the query results, we also return the following $\mathcal{VO}$ in order for the client to authenticate the results: 1. The hash value h_u for each unvisited node u that has a visited sibling; 2. The dividing line l_u for each u where $box(u) \cap \partial(Q) \neq \varnothing$; 3. The point p of a visited leaf u if $p \notin Q$ (note that all $p \in Q$ are included in the query results); 4. The signature s. We also include a label for each of these nodes, for identifying its level and position in the tree. The label is described by $O(\log n)$ bits, or $O(1)$ words.

LEMMA 4. *For any query Q, the Mkd-tree $\mathcal{T}$ built on n points returns a $\mathcal{VO}$ of size at most $O(\sqrt{n})$.*

PROOF. We only need to bound the total number of nodes in the three categories defined above. According to Lemma 3, the number of nodes in $\mathcal{T}$ whose bounding boxes intersect $\partial(Q)$ is $O(\sqrt{n})$. This naturally bounds the number of type (2) and (3) elements. For a type (1) node u, its parent must have a bounding box that intersects $\partial(Q)$, so the number of type (1) nodes is also $O(\sqrt{n})$.   □

The authentication algorithm. Using the $\mathcal{VO}$ and the query results the client can recompute the hash value of the root and verify the signature s of the tree. Let R be the set of points in the query result. The client computes h_{root} using a recursive function starting from the root node. The recursive call has three inputs: a node u (i.e., the label of u), $box(u)$ and all $p \in R$ s.t. $p \in box(u)$. The algorithm (shown as Algorithm 1) is initialized with ComputeHash(root, $\mathbb{R}^2$, R). When the recursive call completes, the hash value of the root has been computed. The basic idea is to reconstruct the part of the Mkd-tree that was traversed for constructing the query answer at the server.

Since there are $O(\log n)$ levels of recursion, and each level involves $O(k)$ operations, the total running time is $O(k \log n)$.

Algorithm 1: ComputeHash($u, box(u), R_u$)

input: u: a node in $\mathcal{T}$, $box(u)$: u's bounding box, l_u: the dividing line of u, R_u: $p \in R$ s.t. $p \in box(u)$.
return: h_u: the hash value of u.

1 **if** u is a leaf **then**
2   **if** $|R_u| \neq 1$ **then** report error;
3   let p be the only point in R_u;
4   **if** $p \notin box(u)$ **then** report error;
5   **return** $\mathcal{H}(p)$;
6 **else**
7   let v and w be u's children;
8   compute $box(v)$ and $box(w)$ from $box(u)$ and l_u;
9     $R_v := R_u \cap box(v)$;
10     $R_w := R_u \cap box(w)$;
11   **for** $z = v, w$ **do**
12   **if** $box(z) \cap Q = \varnothing$ **then**
13   **if** h_v *not available* **then** report error;
14   **else if** $box(z) \cap \partial(Q) \neq \varnothing$ **then**
15   **if** l_z *not available* **then** report error;
16         $h_z :=$ ComputeHash($z, box(z), R_z$);
17   **else**
18   // $box(z)$ is contained in Q;
19   build the Mkd-tree $\mathcal{T}_z$ of R_z;
20         $h_z :=$ hash value of the root of $\mathcal{T}_z$;
21   **return** $\mathcal{H}(h_v|h_w|l_u)$;

Correctness of the results is guaranteed similarly to the original Merkle tree, due to the collision-resistance of the hash function. Completeness is guaranteed by the following.

LEMMA 5. *Successful $\mathcal{VO}$ authentication of the Mkd-tree implies completeness of query results.*

PROOF. If any node fully contained in Q is missing from the $\mathcal{VO}$, authentication will fail. Consider nodes that intersect with $\partial(Q)$. For any such node, the construction algorithm will have to traverse all the way to the leaves to identify potential query results. At some level of this recursion there will be one path fully disjoint with Q and another fully contained in Q. Clearly, if any fully contained node or its children is omitted from the $\mathcal{VO}$ authentication will fail; a needed hash for constructing the hash of the parent is missing. Similarly, if any node that has a parent that intersects with $\partial(Q)$ is missing, authentication will fail; the hash of this parent will have to be computed due to a child that is fully contained in Q. Hence, if the hash of the root authenticates correctly, no points below nodes that are contained or intersect with Q have been omitted from the result. $\square$

The following holds for the Mkd-tree:

THEOREM 1. *Given a set of n points in the plane, an Mkd-tree takes $O(n)$ space and can be built in $O(n \log n)$ time. Given an orthogonal range search query Q, it takes $O(\sqrt{n} + k)$ time to construct the answer, with a $\mathcal{VO}$ size $O(\sqrt{n})$. The client needs $O(\sqrt{n} + k \log n)$ time to authenticate the results.*

The kd-tree can be extended into an aggregate structure similarly to the discussion in Section 3, by hashing a node with $h_u = \mathcal{H}(h_v|h_w|l_u|sum(u))$. The bounds in Theorem 1 hold by setting $k = 1$ for aggregation queries.
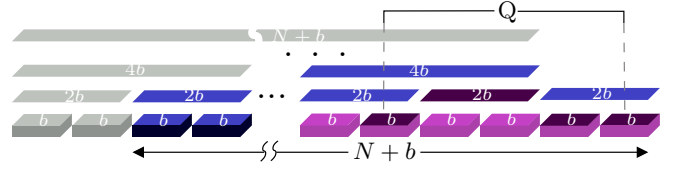

Figure 5: The DMkd-tree. Trees with boundaries outside of the maximum window can be discarded. A one-shot query can be answered by accessing only a logarithmic number of trees.

4.3 The Tumbling Mkd-tree

We can now extend the TM-tree into a Tumbling Mkd-tree (TMkd-tree), by replacing the Merkle trees with Mkd-trees. Clearly, the new structure eliminates false positives, since now for the boundary kd-trees that intersect with the query range we can issue two-dimensional range queries that will report qualifying tuples within both the selection and temporal axes. The TMkd-tree significantly reduces the sliding window query cost, especially the $\mathcal{VO}$ size.

THEOREM 2. *Given a maximum permissible response delay b, the TMkd-tree uses $O(N)$ space, and requires $O(\log b)$ time and $O(1/b)$ amortized signing operations to process a tuple. It supports sliding window queries with per period cost $O(\lceil \sigma/b \rceil \sqrt{b} + k)$, and $\mathcal{VO}$ size $O(\lceil \sigma/b \rceil \sqrt{b})$. For aggregation queries, the same bounds hold by setting $k = 1$.*

In addition, the structure naturally extends to higher dimensions. For queries with d selection attributes, we use $(d+1)$-dimensional kd-trees (time is always one dimension).

5. REDUCING ONE-SHOT QUERY COST

The TM-tree and TMkd-tree are robust solutions for sliding window queries. Nevertheless, they suffer from high one-shot query cost, especially for large n, since a large number (n/b) of trees need to be queried for constructing the answer. In the following we focus on algorithms for reducing one-shot query cost at the expense of either slightly increased storage cost or slightly increased update cost. Clearly, the same solutions can be used for constructing the initial results of sliding window queries (recall that constructing the initial answer to a sliding window query is a one-shot query), which is a very important improvement especially in streaming scenarios where a large number of users pose queries dynamically. In the rest, we do not discuss extensions for aggregate queries, since they are similar to the solutions discussed already for the TMkd-tree.

5.1 The Dyadic Mkd-tree

We present a structure that combines the Merkle tree and the Mkd-tree, yielding a solution that has much faster one-shot query cost than the TM-tree for one-dimensional queries, at the expense of slightly increased maintenance cost at the provider and server, and slightly increased storage cost at the server.

The structure. We call the new structure the *Dyadic Mkd-tree* (DMkd-tree). Assume that $(N + b)/b$ is a power of 2 and let $\ell = \log((N+b)/b) - 1$. A DMkd-tree consists of $\ell+1$ levels of forests of trees arranged in dyadic ranges over the time dimension. On level 0, we build a *Mkd-tree* for every b consecutive tuples in the stream (denoted with boxes in

Figure 5). Levels 1 through ℓ consist of a forest of Merkle binary search trees over the values of attribute A (rectangles in the figure). More precisely, on level i, we build a Merkle binary search tree for every $2^i b$ tuples.

The maintenance algorithm. The DMkd-tree is fairly easy to maintain. The structure is initialized with the first $N+b$ tuples, building all levels in the dyadic hierarchy. After b new tuples have been received we build one new Mkd-tree on level 0 and discard the last kd-tree. At the same time we discard all Merkle trees that contain tuples outside the maximum window $N + b$. In general, after $2^i b$ tuples have been received, we build one new Merkle tree on level $i, 1 \leq i \leq \ell$ and discard all Merkle trees that fall outside the maximum window. For example, in Figure 5, after $2b$ tuples, the two left-most kd-trees and the left-most Merkle tree on all levels can be deleted.

A very important observation is that to build a new Merkle binary search tree $\mathcal{T}$ on level $i, 2 \leq i \leq \ell$, we do not need to re-sort any tuples. We can simply retrieve the tuples by scanning the leaf levels of the level-$(i-1)$ Merkle trees that fully span the range of $\mathcal{T}$. Hence, we can build tree $\mathcal{T}$ in a bottom-up fashion in time $O(2^i b)$.

LEMMA 6. *The amortized maintenance cost per tuple for the DMkd-tree is $O(\log N)$. The amortized number of signing operations is $O(1/b)$.*

PROOF. On level 0, we spend $O(b \log b)$ time to build each Mkd-tree. The amortized cost is $O(\log b)$ per tuple. Similarly, on level 1 we spend $O(b \log b)$ time to build a Merkle tree for every $2b$ tuples, so the amortized cost is also $O(\log b)$. On level $i, 2 \leq i \leq \ell$, we spend $O(2^i b)$ time to build the Merkle tree for every $2^i b$ tuples (as mentioned already the sorting operation can be avoided). This yields an amortized cost of $O(1)$. So the overall cost per tuple is $O(\log b + \ell) = O(\log N)$.

Applying similar arguments, the amortized number of signing operations on level i is $O(1/(2^i b))$, which yields a total of $O(1/b)$ signing operations per tuple. □

Since each level occupies $O(N)$ space the entire structure at the server side takes $O(N \log(N/b))$ space. On the other hand, the service provider uses linear space for storing the DMkd-tree. Once a kd-tree or a Merkle tree at any level of the hierarchy has been covered by a higher level Merkle tree and its signature has been propagated to the server, it can be discarded (at the provider side the trees are only useful for producing signatures and not for answering queries).

Query and authentication. The main idea behind using dyadic ranges is that constructing a query answer requires accessing only a logarithmic number of trees. Given a one-shot window query of size n, we decompose its time range into a series of sub-ranges, and answer each of them individually. First, we query the two level-0 Mkd-trees at the endpoints of the query range. Then by a standard argument, the remaining portion of the range can be decomposed into $O(\log(n/b))$ sub-ranges, each of which is exactly covered by one of the Merkle trees. For example, in Figure 5 only three kd-trees and one Merkle tree need to be traversed. Trees at level 0 might be contained fully in the query time range or not. Trees at higher levels are always fully contained in the query. Hence, for higher levels in the hierarchy we only need to maintain one-dimensional structures on the selection at-

tribute. For level 0 we need to maintain two-dimensional kd-trees to be able to filter out false positives.

To authenticate the results the client first individually authenticates the $\mathcal{VO}$ of each tree as in the original Merkle tree and Mkd-tree. This verifies correctness. To authenticate completeness, the client needs to verify that the appropriate trees have been traversed. First, the provider signs each tree individually creating a signature that contains the hash value of the root node and the indices of the oldest and newest tuple contained therein. Second, the server includes in the $\mathcal{VO}$s the indices of the newest and oldest tuple of the traversed trees. The client verifies that the indices returned by the server are consecutive and also cover the query window. Since no trees overlap in the temporal dimension this invariant has to hold. The authenticity of the timestamps returned is established by the signature of each tree. The following holds for the DMkd-tree:

THEOREM 3. *For a maximum response delay b, the DMkd-tree uses $O(N \log(N/b))$ space, and requires $O(\log N)$ time and $O(1/b)$ signing operations amortized to process a tuple. It takes $O(\log n \log(n/b) + \sqrt{b} + k)$ time to answer a one-shot query, and provides a $\mathcal{VO}$ of size $O(\log n \log(n/b) + \sqrt{b})$. The client takes $O(\log n \log(n/b) + \sqrt{b} + k \log b)$ time to authenticate the results.*

Note that level-0 of DMkd-tree is essentially the same as TMkd-tree. Hence for sliding window queries, once the initial answers have been reported, subsequent updates can be handled by using the level-0 TMkd-tree, as in Section 4.3.

5.2 The Exponential Mkd-tree

So far, the DMkd-tree can be used for one-dimensional queries only. In this section, we present an algorithm that arranges a forest of Mkd-trees in an exponential hierarchy and can answer multi-dimensional queries. We call this structure the *Exponential Mkd-tree* (EMkd-tree). This approach can construct initial query answers much faster than the TMkd-tree, with a slight increase in the amortized per tuple update cost.

The structure. For simplicity assume that N/b is a power of 2 and let $\ell = \log(N/b) - 1$. The EMkd-tree consists of up to 2ℓ Mkd-trees: $\mathcal{T}_0, \mathcal{T}_0', \mathcal{T}_1, \mathcal{T}_1', \ldots, \mathcal{T}_\ell$. Each $\mathcal{T}_i$ is always present, but a $\mathcal{T}_i'$ may be present or absent, as will become clear shortly. The tree $\mathcal{T}_i$ or $\mathcal{T}_i'$ (if present) stores $2^i b$ consecutive tuples, hence $\sum_{i=0}^{\ell} |\mathcal{T}_i| = b(2^{\log N/b} - 1) = N - b$. For any $i < j$, all tuples stored in $\mathcal{T}_i$ are newer than any tuple stored in $\mathcal{T}_j$; and for any i, if $\mathcal{T}_i'$ is present, all tuples in $\mathcal{T}_i'$ are newer than any tuple in $\mathcal{T}_i$, such that no trees overlap in the time dimension (see Figure 6). The EMkd-tree structure is initialized with the first N tuples in the stream as follows. Tree $\mathcal{T}_\ell$ receives the first $N/2$ tuples, $\mathcal{T}_{\ell-1}$ the following $N/4$ tuples, until tree $\mathcal{T}_0$ which receives b tuples, for a total of $N - b$ tuples. The remaining b tuples are assigned to tree $\mathcal{T}_0'$, for a total of $\ell + 2$ Mkd-trees. No other $\mathcal{T}_i'$ exists yet (see Figure 7).

The update algorithm. After initializing the EMkd-tree we update it every b new arrivals. The update procedure first combines trees $\mathcal{T}_0$ and $\mathcal{T}_0'$ into tree $\mathcal{T}_1'$, then creates a new tree $\mathcal{T}_0$ using the latest b tuples and stops (see Figure 7). The next update proceeds similarly by creating a new tree $\mathcal{T}_0'$ using the latest b tuples and stops. This procedure continues by merging and propagating trees on consecutive levels of
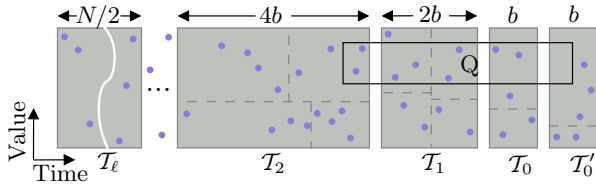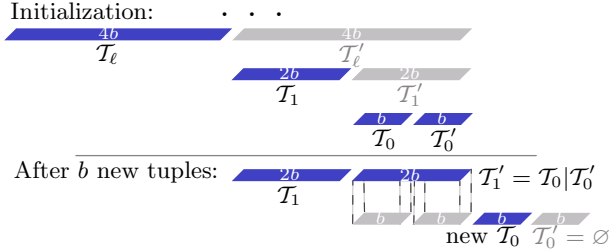
Figure 6: Querying the EMkd-tree.

Figure 7: Initializing and updating the EMkd-tree.

the hierarchy, until a level with a non-existent $\mathcal{T}_i'$ is found, at which point the propagation stops. Notice that we merge two trees $\mathcal{T}_i, \mathcal{T}_i'$ only if a third tree has been propagated from level $i - 1$. Otherwise, we leave the trees intact. It is easy to see that after any update, the invariant that no trees overlap in the time dimension is preserved. A special case occurs at level ℓ. When tree $\mathcal{T}_\ell'$ is created the update procedure immediately discards tree $\mathcal{T}_\ell$ containing the oldest $N/2$ tuples, and sets $\mathcal{T}_\ell = \mathcal{T}_\ell'$. This iterative procedure is detailed in Algorithm 2.

Algorithm 2: EMkd-tree-Maintain

input: The b newest tuples in the stream.

1 build Mkd-tree R_0 on the b new tuples;
2 $i := 0$;
3 **while** *true* **do**
4   **if** $\mathcal{T}_i'$ present **then**
5   merge $\mathcal{T}_i$ and $\mathcal{T}_i'$ into R_{i+1};
6   remove $\mathcal{T}_i$ and $\mathcal{T}_i'$;
7     $\mathcal{T}_i = R_i$;
8     $i := i + 1$;
9   **else**
10   **if** $i = \ell$ **then** $\mathcal{T}_i = R_i$;
11   **else** $\mathcal{T}_i' = R_i$;
12   **return**;

Although the maintenance algorithm in the worst case may affect all the trees, its amortized cost can be effectively bounded by the following lemma:

LEMMA 7. *The amortized maintenance cost per tuple for the EMkd-tree is $O(\log(N/b)\log N)$. The amortized number of signing operations is $O(1/b)$.*

PROOF. We use a charging argument. According to Theorem 1, the construction cost of a Mkd-tree built on n tuples is $O(n \log n)$, so we can charge an $O(\log n)$ cost to each tuple. Consider each tuple in the stream. It is involved in the construction of each $\mathcal{T}_i'$ exactly once, so it is charged by a

total cost at most (asymptotically)

$$
\sum_{i=0}^{\ell} \log(2^i b) = \sum_{i=0}^{\ell} (i + \log b) = O\left(\log \frac{N}{b}\left(\log \frac{N}{b} + \log b\right)\right)
$$
$$
= O(\log(N/b)\log N).
$$

For the signing cost, we use a similar charging scheme. Since building a Mkd-tree on n tuples requires only one signing operation, we can charge $O(1/n)$ to each tuple. Every tuple is involved in the construction of each $\mathcal{T}_i'$ exactly once, so it is charged with a total of $\sum_{i=1}^{\ell} 1/(2^i b) = O(1/b)$ signing operations. □

It is easy to verify that in the worst case this algorithm will maintain $3N/2$ tuples, hence the storage cost is $O(N)$. In addition, there is an easy optimization for merging two Mkd-trees efficiently, without having to bulk load the structures from scratch. For all trees $\mathcal{T}_i, \mathcal{T}_i'$, and R_i, we first divide the data along the attribute dimension if i is even, and along the time dimension if i is odd (see Figure 6). Now, for even i, given that $\mathcal{T}_i$ and $\mathcal{T}_i'$ have no overlap in the temporal dimension and their roots are split on the attribute dimension, we can merge them by creating a new kd-tree root node and attaching $\mathcal{T}_i$ and $\mathcal{T}_i'$ as its two subtrees. On even levels the merging operation has constant cost. On odd levels (since the attribute domains of the two trees might overlap) bulk loading from scratch is unavoidable.

The query and authentication algorithms. To answer the query the server traverses the kd-trees starting from $\mathcal{T}_0$, until the query window is entirely covered. For kd-trees that are entirely contained in the query window, the server poses a 2-sided range query. For the first and last kd-trees the server poses a 3-sided query, to filter out false positives. The server returns an individual $\mathcal{VO}$ for each tree using the original construction algorithm of the Mkd-tree. In addition, it includes the indices of the oldest and newest tuple in each tree. Authentication at the client side proceeds in exactly the same way as for the DMkd-tree.

The following can be stated for the EMkd-tree:

LEMMA 8. *The EMkd-tree spends $O(\sqrt{n+b}+k)$ to construct an answer and returns a $\mathcal{VO}$ of size $O(\sqrt{n+b})$. It takes the client $O(\sqrt{n+b}+k\log(n+b))$ time to authenticate the results.*

PROOF. Since the size of the Mkd-trees doubles every level, given a query with window size $n \geq b$, it is not difficult to see that the biggest tree that needs to be traversed to construct the answer has size at most n. For each level of the exponential hierarchy, at most two trees are traversed. So the total $\mathcal{VO}$ size is at most

$$
O(\sqrt{n}) + O(\sqrt{n/2}) + \cdots + O(\sqrt{b}) = O(\sqrt{n}).
$$

When $n < b$, at most two Mkd-trees of size b need to be traversed, so the overall $\mathcal{VO}$ size is $O(\sqrt{n+b})$. The same analysis applies for the total query time, plus an additional term linear to the size of the query results.

The bound on the authentication cost can be easily obtained by observing that the maximum height of the kd-trees that need to be traversed is $O(\max\{\log n, \log b\})$. □

In order to answer sliding window queries, the provider and the server also need to maintain a TMkd-tree, concurrently with the EMkd-tree. The EMkd-tree is used for constructing the initial answers, while the TMkd-tree is used

	TM-tree	TMkd-tree	DMkd-tree	EMkd-tree
Space	N	N	$N \log \frac{N}{b}$	N
Update cost	$\log b$	$\log b$	$\log N$	$\log \frac{N}{b} \log N$
Signing operations	$1/b$	$1/b$	$1/b$	$1/b$
Sliding query cost	$\lceil \sigma/b \rceil \log b + b + k$	$\lceil \sigma/b \rceil \sqrt{b} + k$	-	-
Sliding $\mathcal{VO}$ size	$\lceil \sigma/b \rceil \log b + b$	$\lceil \sigma/b \rceil \sqrt{b}$	-	-
Sliding authen. cost	$\lceil \sigma/b \rceil \log b + b + k$	$\lceil \sigma/b \rceil \sqrt{b} + k \log b$	-	-
One-shot query cost	$\frac{n}{b} \log b + b + k$	$\frac{n}{b} \sqrt{b} + k$	$\log n \log \frac{n}{b} + \sqrt{b} + k$	$\sqrt{n+b} + k$
One-shot $\mathcal{VO}$ size	$\frac{n}{b} \log b + b$	$\frac{n}{b} \sqrt{b}$	$\log n \log \frac{n}{b} + \sqrt{b}$	$\sqrt{n+b}$
One-shot authen. cost	$\frac{n}{b} \log b + b + k$	$\frac{n}{b} \sqrt{b} + k$	$\log n \log \frac{n}{b} + \sqrt{b} + k \log b$	$\sqrt{n+b} + k \log (n+b)$
Dimensions (*)	One	Multiple	One	Multiple
Aggregation (**)	Yes	Yes	Yes	Yes

Table 1: Summary of the (asymptotic) results for various solutions. (*) For d dimensions, all the $\sqrt{b}$ terms become $b^{1-1/(d+1)}$; all the $\sqrt{n+b}$ terms become $(n+b)^{1-1/(d+1)}$. (**) For aggregation queries, all the bounds hold by setting $k = 1$.

for constructing subsequent updates. Notice here that even if the initial answer from the EMkd-tree is delayed for some reason, the TMkd-tree can still provide updates unhindered (deltas can be computed even without knowing the answer of the previous window). The updates will have to be buffered by the client until the initial answer arrives. The following summarizes the performance of the EMkd-tree:

THEOREM 4. *For a maximum response delay b, the EMkd-tree uses $O(N)$ space, takes $O(\log(N/b) \log N)$ time and $O(1/b)$ signing operations amortized to process a tuple. It takes $O(\sqrt{n+b} + k)$ time to answer a one-shot query, with a $\mathcal{VO}$ size $O(\sqrt{n+b})$. The client takes $O(\sqrt{n+b} + k \log(n+b))$ time to authenticate the results.*

6. DISCUSSIONS

Supporting time-based windows. Supporting time-based windows is also possible. First, we use standard symbolic perturbation techniques to uniquely associate each tuple with a timestamp, if there are multiple tuples per time instant. Then, we generate dummy tuples for time instants without activity (a time instant can be defined as the smallest permissible window slide). The rest poses only technical difficulties that are not hard to overcome. Dummy tuples in general can be ignored and are used only for triggering signing operations. The theoretical bounds of the solutions hold, where N now expresses the maximum number of tuples within a maximum window.

Summary of various solutions. Table 1 summarizes the performance of various solutions. We can see that for sliding window queries, the TMkd-tree is better than the TM-tree for typical values of σ and b. The difference in the $\mathcal{VO}$ size could be significant when the sliding period σ is smaller than or comparable to b, which is common in real scenarios. However, for one-shot queries (or equivalently the initialization cost for sliding window queries), the two tumbling approaches both perform badly. The proposed DMkd-tree and EMkd-tree structures complement nicely in this respect. For one-dimensional queries, the DMkd-tree is the structure of preference, as it has excellent query performance, for a small penalty in the server's storage cost. When the server's memory is limited or for multi-dimensional queries, the EMkd-tree can be used. Notice that for sliding window queries, the server needs to maintain both an EMkd-tree and a TMkd-tree, which doubles the storage cost in the

worst case. For most practical cases though, the storage cost will still be smaller than the DMkd-tree. Finally, all structures can be extended easily to support aggregates like SUM, COUNT, AVG, MIN, and MAX.

7. EXPERIMENTS

We implemented the proposed techniques and evaluated their performance over two real data streams [3, 1]. The following cost metrics are considered: 1. the amortized update cost per tuple for the data owner; 2. the storage cost of the authentication structure for the server; 3. the query cost; 4. the $\mathcal{VO}$ size; and 5. the verification cost for the client.

All algorithms are implemented using GNU C++. Cryptographic functions are provided by [9, 30]. Two real data streams have been tested. The *World Cup (WC)* data stream [3] consists of web server request traces for the 1998 Soccer World Cup. Each request contains attributes such as a timestamp, a client id, a requested object id, a response size, etc. We used the request streams of days 46 and 47 that have about 100 millions records. The *IP traces (IPs)* data stream [1] is collected over the AT&T backbone; each tuple is a TCP/IP packet header. Since similar patterns have been observed for all cost metrics for both data streams, we present results from the WC data only. We use tuples consisting of attributes response size, object id, and client id, and a unique timestamp. Experiments were run on a Linux box with an Intel Pentium 2.8GHz CPU. The SHA1 hash function takes about $1 \sim 2\mu s$ (for input size up to 500 bytes); 128-byte RSA has a signing cost of about $2ms$ and verifying cost of $120\mu s$. Each hash value produced by SHA1 is 20 bytes and a signature from RSA is 128 bytes.

7.1 Sliding window queries.

The TM-tree and the TMkd-tree are the two candidates for authenticating sliding window queries. We compare them using one-dimensional queries on the "response size" attribute (as the TM-tree does not support multi-dimensional queries). Since kd-tree requires its indexed data having distinct values (see Section 4.1), we perturb the attribute "response size" so that all tuples have unique values. In addition, in order to easily generate a set of random queries with a fixed query selectivity, tuples are perturbed so that they have uniformly distributed values in "response size". Our performance study is not affected by this, as the query cost is solely determined by the query selectivity.
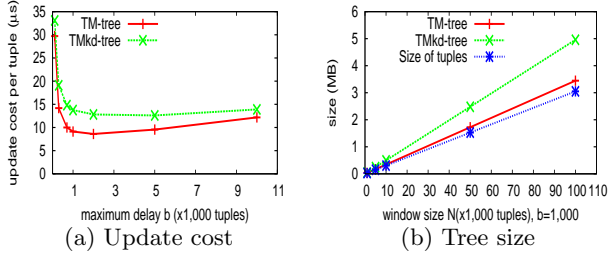
(a) Update cost   (b) Tree size

Figure 8: Tumbling trees: update cost and size.

(a) $\sigma = 1,000$ (b) $\gamma = 0.1$

Figure 9: Query cost per period, $b = 1,000$.

(a) $\sigma = 1,000$ (b) $\gamma = 0.1$

Figure 10: $\mathcal{VO}$ size per period, $b = 1,000$.

Update cost per tuple. The data owner has to maintain the authenticated data structures as tuples are being produced. The data streaming setting mandates this cost to be small. Figure 8(a) shows the amortized update cost per tuple for both trees over different values of b (the maximum delay, which is determined by the owner). We notice that for small b both trees have an excessive update cost; as b increases, the cost drops quickly. After some point both start to grow slowly, due to the fact that the cost consists of an $O(1/b)$ signing and an $O(\log b)$ update cost (see Table 1). For small b, signing is the dominant cost; when b exceeds a certain threshold, the latter begins to dominate. Figure 8(a) reveals that for both trees $b = 1,000$ is a sweet point. For smaller b's the update cost is too high to cope with bursty streams, while larger b's introduce longer response delays for the clients without further reducing the data owner's cost significantly. With $b = 1,000$ the amortized update cost per tuple is only $10 \sim 15\mu s$, i.e., both the owner and the server could handle 10^5 tuples per second.

Structure size. Our analysis has pointed out that both TM-tree and TMkd-tree use linear space given a window size N. It is still interesting to investigate the constant factors associated with this cost. Figure 8(b) plots the results. It should be noted that the storage cost does not depend on b. The size of the raw data (32 bytes per tuple) is also provided in Figure 8(b) as a baseline for comparison. Both trees have very good scalability and introduce very small overhead in size (only 5 MB for 100,000 tuples). Another interesting fact to highlight is that at any time instance, the provider only needs space large enough to store one Merkle tree (or Mkd-tree), built for the latest b tuples. This has size roughly equal to 40 KB for $b = 1,000$.

Query cost. We turn our attention to the per period update cost for sliding window queries. The performance of one-shot window queries (or equivalently the initialization cost for a new sliding window query) will be studied in Section 7.2. The query cost is measured for two different workloads of $1,000$ randomly generated queries: 1. fixed sliding period σ but varying query selectivity γ (Figure 9(a)); and 2. fixed query selectivity γ but varying sliding period σ (Figure 9(b)). We set b to $1,000$ as suggested before and report the average cost for one query. Note that γ is essentially equal to the selectivity on the query attribute dimension, since most tuples in a window, except those in the boundary trees when window slides, are indexed by trees that are fully covered in the time dimension by the query window.

With the sliding period $\sigma = b = 1,000$, four boundary trees will be queried to report the new and expiring tuples. From the results we observe that both TM-tree and TMkd-tree have roughly linearly increasing costs w.r.t query selectivity. TM-tree does have a lower query cost even though
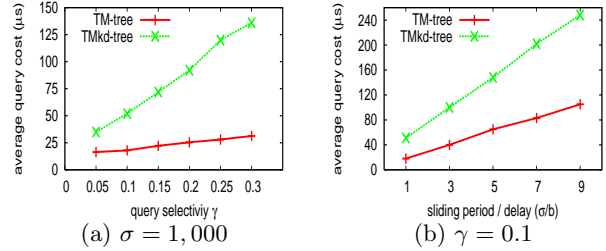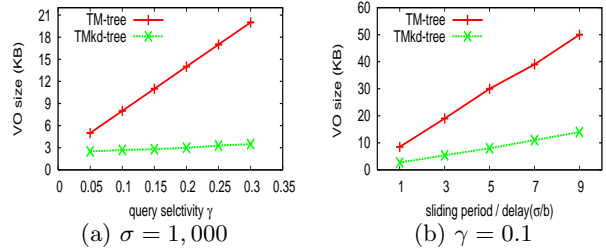
theoretically it may access more nodes than TM-kd tree due to the fact that it incurs false positives. This is explained by the fact that a balanced binary search tree is used as the underlying structure for a single tree in TM-tree, which means that all tuples are retrieved in a sequential scan in the leaf level. In addition, since trees are bulk-loaded there will be strong locality among these leaf nodes in the main memory. Adjacent nodes are loaded into the cache in one memory access and this dramatically reduces the number of memory access. On the other hand, though TMkd-tree accesses less number of nodes theoretically by avoiding any false positives, but they are all random access which in practice leads to more nodes access and slower performance. When the sliding period σ is larger than b, around $2\frac{\sigma}{b} + 2$ trees will be queried. Hence, the query cost is roughly linear in σ as we have observed in Figure 9(b). Finally, since we only retrieve the new and expiring tuples as the window slides, the sliding window query cost does not depend on the window size n.

$\mathcal{VO}$ size. The $\mathcal{VO}$ size is the determining factor for the communication overhead between the server and the client. In Figure 10 we plot the $\mathcal{VO}$ size using the same queries as in Figure 9(a) and 9(b). Figure 10(a) reveals that the TM-tree has a much higher $\mathcal{VO}$ size than the TMkd-tree, as it will incur roughly $4\gamma b$ false positives in this case (see Figure 3). Recall that when $\sigma = b$, four boundary trees will be queried. On the other hand, the TMkd-tree can avoid false positives as it indexes and stores authentication information for both the selection attribute and the time axis. The difference can be order of magnitude as the query selectivity increases. This is due to the fact that the TM-tree generates false positives, which are part of the $\mathcal{VO}$ and each false positive is a tuple (32 bytes in our experiment). Similarly, the linear trend w.r.t σ in Figure 10(b) for both trees is explained by the same reason as in Figure 9(b). Again, the sliding window size n does not affect the results.

Verification cost. The verification cost at the client is a mirror of the query process performed by the server, except for the additional hashing operations to reconstruct the hashes of the roots, and the verification of the digital signatures. Since these costs are common to both the TM-tree
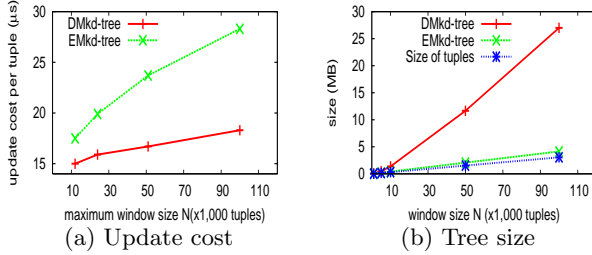
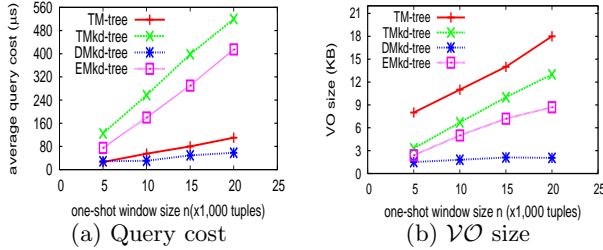**Figure 11: Update cost and size for DMkd-tree and EMkd-tree, $b = 1,000$.**

Figure 12: One-shot window query $b = 1,000$, $\gamma = 0.1$ and $N = 20,000$.

and the TMkd-tree, similar trends as in Figure 9 have been observed for the verification cost. We omit the details.

Experimental conclusion. For sliding window queries, TM-tree has better query cost and TMkd-tree outperforms TM-tree with respect to the $\mathcal{VO}$ size. Nevertheless, TMkd-tree naturally supports multi-dimensional queries while TM-tree can not handle those cases.

7.2 One-shot queries

Both TM-tree and TMkd-tree have linear cost in n (Table 1) for one-shot queries. In this section, we study the performance of the DMkd-tree and the EMkd-tree.

Amortized update cost and structure size. First we study the amortized per-tuple update cost. The result is shown in Figure 11(a) with varying maximum window sizes N. Both trees have higher update cost compared to the TM-tree and TMkd-tree (see Figure 8(a)) due to the $O(\log N)$ dependence (comparing to the $O(\log b)$ cost for tumbling trees). In addition, the EMkd-tree is more expensive than the DMkd-tree as it has an additional factor of $O(\log \frac{N}{b})$. Nevertheless, for fairly large N (up to $100,000$ in our experiments), both DMkd-tree and EMkd-tree achieve update costs of less than $30\mu s$ per tuple. The smaller update cost of DMkd-tree is not for free. The tree occupies more space, as shown in Figure 11(b), by an $O(\log \frac{N}{b})$ factor compared to the EMkd-tree. EMkd-tree utilizes linear space that is almost equal to the raw data size. It should be highlighted though that the DMkd-tree still has a reasonable main memory size. As Figure 11(b) has suggested, it takes less than 30 MB of memory for the maximum window of $100,000$ tuples.

Query cost and $\mathcal{VO}$ size. Our theoretical analysis shows that the DMkd-tree and EMkd-tree should perform better than the TM-tree and TMkd-tree respectively, especially for large window sizes. This is confirmed by our findings from Figure 12(a) (where N is set to $20,000$ and $\gamma = 0.1$). The results are obtained by averaging over a workload of $1,000$ randomly generated queries. Clearly, the DMkd-tree and EMkd-tree outperforms their counterparts TM-tree and

TMkd-tree. The gaps between them are increasing with the larger value for n. This saving is critical when there are multiple clients registering many queries in the system. Lastly, we study the $\mathcal{VO}$ size for various trees under one-shot window queries and present the results in Figure 12(b). The TM-tree has the worst performance since it has to include roughly $2\gamma b$ tuples as false positives in the two boundary trees. Both the DMkd-tree and EMkd-tree have less $\mathcal{VO}$ size than the TM-kd tree.

Verification cost. Following the discussion in Section 7.1 for the verification cost, similar trends as those reflected in the query cost for one-shot window queries have been observed. The results are omitted for the brevity.

Experimental conclusion. Our results reveal that the DMkd-tree and EMkd-tree are good candidates for answering one-shot window queries. For one dimensional query, the data owner and the server could combine either the DMkd-tree, if reducing update and query cost is a higher priority, or the EMkd-tree, if reducing the space usage is a higher priority, together with the TM-tree, if the efficient query cost for maintaining the updates to sliding window queries is a higher priority, or the TMkd-tree if low $\mathcal{VO}$ size for maintaining the updates to sliding window queries is a higher priority. This gives the data owner and the server flexible choices to answer all queries efficiently in different settings.

7.3 Aggregation and multi-dimensional queries

All of the proposed structures in this paper can support authentication of aggregation queries as we have discussed. The detailed evaluation of the performance under different cost metrics is rather involved and they are not discussed here. However, as Table 1 has suggested, the cost analysis remains almost the same for aggregation queries. We would like to highlight that all of our index structures could be made to support the authentication of selection and aggregation queries at the same time, as the trees for authenticating aggregation queries trivially support answering and authenticating selection queries.

Finally, the TMkd-tree and EMkd-tree support multi-dimensional queries. The combination of these two structures provide a nice treatment for a highly dynamic environment where multiple clients register various sliding window queries at any time instance, possibly with different dimensionality, window sizes and sliding periods.

8. RELATED WORK

General issues for outsourced databases first appeared in [19]. A large corpus of related work has appeared on authenticating queries for outsourced databases ever since [12, 23, 6, 32, 31, 28, 27, 17, 25, 22, 33, 7]. Previous work focuses on authenticating selection, projection and join queries for relational databases. To the best of our knowledge, this work is the first to address query authentication issues on sliding window queries for outsourced streams. Nevertheless, previous work utilizes similar cryptographic primitives. Signature based approaches [28, 31] produce a signature for each consecutive pair of tuples in the database, ensuring both correctness and completeness of query results. These techniques could be generalized to support multi-dimensional queries, as shown in [28, 7], but clearly are very inefficient when applied to data streams, due to the high signing cost. Index based approaches apply the Merkle tree ideas over

binary search trees [12, 27] or B$^+$ trees [32, 22]. Index based approaches have not been shown to work for multi-dimensional queries. Nevertheless, applying the Merkle tree principles to more generic index structures has been studied in the past [2, 23, 17, 35]. Our extensions for the kd-tree (and similarly for other multi-dimensional structures like the R-tree) are straightforward but new. Most importantly, arranging these structures into hierarchies, making them suitable for sliding windows over data streams, and studying their performance over a variety of cost metrics has not been addressed before.

Other related work includes the work of [6] that uses Merkle trees to authenticate XML documents, and techniques for integrity in data exchange [26] that are also relevant in an outsourced data setting (we refer readers to an excellent thesis [25] for more details). Query execution assurance [33] makes the assumption that the client has a copy of the database, and hence does not apply to streams. Finally, our problem is orthogonal to watermarking techniques for proving ownership of streams [34] and work on sketches in sensor streams that can provide verifiable probabilistic guarantees for distributed aggregation queries [13].

Sliding window queries over data streams have been studied extensively recently [4]. A number of algorithms have been proposed that utilize exponential hierarchies [10] and dyadic ranges [8, 14] to solve sliding window problems. Readers are referred to an excellent thesis [15] for more details. Our improved versions of the Mkd-trees use the similar spirit.

9. CONCLUSION

To conclude, we propose structures for authenticating multi-dimensional selection and aggregation queries over sliding windows on data streams. Our solutions combine concepts from the Merkle tree, kd-tree, dyadic ranges, and exponential hierarchies. We provide theoretical bounds for all techniques considered across a number of cost metrics. Our experimental evaluation shows that the proposed structures exhibit excellent performance in practice. In the future we plan to extend our results to joins and other types of queries.

10. REFERENCES

[1] AT&T network traffic streams. AT&T Labs.
[2] A. Anagnostopoulos, M. Goodrich, and R. Tamassia. Persistent authenticated dictionaries and their applications. In *ISC*, pages 379–393, 2001.
[3] M. Arlitt and T. Jin. http:// www.acm.org/ sigcomm/ITA/. ITA, 1998 World Cup Web Site Logs.
[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.
[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.
[6] E. Bertino, B. Carminati, E. Ferrari, B. Thuraisingham, and A. Gupta. Selective and authentic third-party distribution of XML documents. *TKDE*, 16(10), 2004.
[7] W. Cheng, H. Pang, and K. Tan. Authenticating multi-dimensional query results in data publishing. In *DBSec*, 2006.
[8] G. Cormode and S. Muthukrishnan. What's hot and what's not: tracking most frequent items dynamically. In *PODS*,

[9] pages 296–306, 2003.
Crypto++ Library. http:// www.crytopp.com/.
[10] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *SODA*, pages 635–644, 2002.
[11] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer, 2000.
[12] P. Devanbu, M. Gertz, C. Martel, and S. Stubblebine. Authentic data publication over the internet. *Journal of Computer Security*, 11(3):291–314, 2003.
[13] M. Garofalakis, J. M. Hellerstein, and P. Maniatis. Proof sketches: Verifiable in-network aggregation. In *ICDE*, 2007.
[14] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *VLDB*, pages 454–465, 2002.
[15] L. Golab. *Sliding Window Query Processing over Data Streams*. PhD thesis, University of Waterloo, Canada, 2006.
[16] S. Goldwasser, S. Micali, and R. L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal on Computing*, 17(2):96–99, 1988.
[17] M. T. Goodrich, R. Tamassia, N. Triandopoulos, and R. Cohen. Authenticated data structures for graph and geometric searching. In *CT-RSA*, pages 295–313, 2003.
[18] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.
[19] H. Hacigumus, B. R. Iyer, and S. Mehrotra. Providing database as a service. In *ICDE*, pages 29–40, 2002.
[20] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *SIGMOD*, pages 401–412, 2001.
[21] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Authenticated index sturctures for aggregation queries in outsourced databases. Technical report, Boston University, 2006.
[22] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin. Dynamic authenticated index structures for outsourced databases. In *SIGMOD*, pages 121–132, 2006.
[23] C. Martel, G. Nuckolls, P. Devanbu, M. Gertz, A. Kwong, and S. Stubblebine. A general model for authenticated data structures. *Algorithmica*, 39(1):21–41, 2004.
[24] R. C. Merkle. A certified digital signature. In *CRYPTO*, pages 218–238, 1989.
[25] G. Miklau. *Confidentiality and Integrity in Distributed Data Exchange*. PhD thesis, University of Washington, 2005.
[26] G. Miklau and D. Suciu. Implementing a tamper-evident database system. In *ASIAN*, pages 28–48, 2005.
[27] E. Mykletun, M. Narasimha, and G. Tsudik. Authentication and integrity in outsourced databases. In *NDSS*, 2004.
[28] M. Narasimha and G. Tsudik. DSAC: Integrity of outsourced databases with signature aggregation and chaining. In *CIKM*, pages 235–236, 2005.
[29] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute of Standards and Technology, 1995.
[30] OpenSSL. http:// www.openssl.org.
[31] H. Pang, A. Jain, K. Ramamritham, and K.-L. Tan. Verifying completeness of relational query results in data publishing. In *SIGMOD*, pages 407–418, 2005.
[32] H. Pang and K.-L. Tan. Authenticating query results in edge computing. In *ICDE*, pages 560–571, 2004.
[33] R. Sion. Query execution assurance for outsourced databases. In *VLDB*, pages 601–612, 2005.
[34] R. Sion, M. Atallah, and S. Prabhakar. Rights protection for discrete numeric streams. *TKDE*, 18(5):699–714, 2006.
[35] R. Tamassia and N. Triandopoulos. Computational bounds on hierarchical data processing with applications to information security. In *ICALP*, pages 153–165, 2005.