Outsourcing Multi-Version Key-Value Stores with Verifiable Data Freshness

Yuzhe Tang
Georgia Institute of Technology
Atlanta, GA, USA
yztang@gatech.edu
Ling Liu
Georgia Institute of Technology
Atlanta, GA, USA
ling.liu@cc.gatech.edu

Ting Wang Xin Hu Reiner Sailer
IBM Research Center,
Yorktown Heights NY, USA
{tingwang, xinhu, sailer}@us.ibm.com
Peter Pietzuch
Imperial College
London, UK
prp@doc.ic.ac.uk

***Abstract*—In the age of big data, key-value data updated by intensive write streams is increasingly common, e.g., in social event streams. To serve such data in a cost-effective manner, a popular new paradigm is to outsource it to the cloud and store it in a scalable key-value store while serving a large user base. Due to the limited trust in third-party cloud infrastructures, data owners have to sign the data stream so that the data users can verify the authenticity of query results from the cloud. In this paper, we address the problem of verifiable freshness for multi-version key-value data. We propose a memory-resident digest structure that utilizes limited memory effectively and can have efficient verification performance. The proposed structure is named INCBM-TREE because it can INCrementally build a Bloom filter-embedded Merkle TREE. We have demonstrated the superior performance of verification under small memory footprints for signing, which is typical in an outsourcing scenario where data owners and users have limited resources.**

I. INTRODUCTION

In the big data era, data sources are generating intensive data streams in terms of both high arrival rates and large volumes. Such streams are widely observed in system logs, network monitoring, social events, among many others. To digest such streams, which is beyond the capability of a normal data owners, a popular new method is to outsource the processing to the cloud. As shown in Figure 1, processing, storage and query serving of the data stream is outsourced to a number of servers inside the cloud, which substantially reduces the effort of the data owner.

A cloud infrastructure, while being capable of handling big data, may not be fully trusted: owners should not assume that a third-party cloud will act as it claims. In an outsourced scenario, owners have to sign the outsourced data in order to protect its authenticity. In other words, the data user who issues the query to a cloud should be able to verify that the query result includes authentic copies of the data, as published by the original data owner.

In our targeted applications, we consider a data model of multi-version object in a key-value format. In particular, an object is of an unique row key k and of multiple overwriting versions, each version associated with a value v and a unique timestamp ts. The query model focuses on the snapshot read, which is based on the Get operation of generic key-value stores. Given a data key k and timestamp ts, it returns the oldest object version before ts and of key k. Its API is defined as below,

$$\mathsf{VGet}(k, ts) \to \{\langle k, v', ts' \rangle\} \cup \pi \tag{1}$$

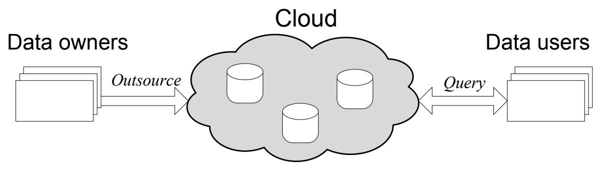


Fig. 1: Outsourced data system in the cloud

Here, π is a proof presented by the cloud to the data user who issued the query. π is used to verify data authenticity in two aspects:

- **Correctness**: The server cannot alter data without being noticed. Each $\langle k, v', ts' \rangle$ in the result is the same as the original data published by owners.
- **Freshness**: The server cannot omit a fresh data version without being noticed. In other words, the result version ts' is indeed the newest version for key k as of time ts.

In this work, we address the problem "how to authenticate snapshot queries on an intensive stream with correctness and freshness." This problem is unique and different from existing streaming authentication work. Proof-infused streams [1] do not consider data freshness while considering range queries on ordered keys. Prior work [2] addresses result freshness of streaming data, but in the context of aggregation queries. CAT [3] addresses a similar query model as our work, but freshness is provided by expensive update-in-place actions, which does not work in an intensive stream scenario.

II. APPROACH: INCBM-TREE

To securely publish an intensive data stream, we follow the common wisdom to partition the stream into multiple windows or data batches, and for each batch to build a digest and apply a digital signature. In this framework, we propose a new digest structure called an INCBM-TREE. The motivation is based on the observation that data owners with limited memory can only build a key-ordered Merkle tree with constrained size. In key-ordered Merkle trees (e.g., as used in [1]), the data in a batch is first sorted based on a key, after which the Merkle tree digest is constructed. Since all data has to be kept in local memory before signing, the batch size is constrained by the local memory size. On the other hand, batch size is crucial to the verification overhead on the user side: the bigger the batch size, the fewer signatures need to be verified, thus reducing the latency of authenticated query processing. Therefore, we propose the INCBM-TREE based on a time-ordered design, i.e., we construct the Merkle tree on the raw data stream with records in their arrival order. This way, the digest or Merkle tree can be constructed incrementally, significantly reducing the memory footprint.

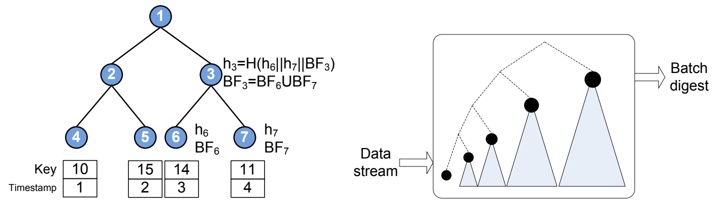
Time-ordered Merkle tree is not enough: Our starting point is a time-ordered Merkle tree. Consider an example where a stream has 4 key-value records, $\langle ts = 1, k = 10\rangle, \langle 2, 15\rangle, \langle 3, 14\rangle, \langle 4, 11\rangle$, and a 7-node Merkle tree (with 4 leaf nodes) can be built on top of this key-unordered list. Given a snapshot query on key 10 as of now, the Merkle tree can prepare a proof of authentication path from leaf node $\langle 1, 10\rangle$ to verify its correctness. On the other hand, the freshness is not easy to prove: verifying that record $\langle 1, 10\rangle$ is the newest version of key 10 is equivalent to presenting a proof for "non-membership" of key 10 in the data list after time 1, i.e., to prove that 10 is not in the list of $15, 14, 11$. To prove this, a Merkle tree can do nothing more than exhaustively returning the whole list. This approach is inefficient and when the queried data refers to a stable version that has not updated in a while.

The Structure of INCBM-TREE*:* Based on the observation that a BLOOM FILTER is a summary for (non)-membership, we propose to combine a BLOOM FILTER with a Merkle tree for efficient verification of both correctness and freshness. The structure of an INCBM-TREE is illustrated in Figure 2a. Compared to the traditional Merkle tree, each tree node in an INCBM-TREE maintains not only a hash digest but also a BLOOM FILTER that summarizes the key set in the subtree rooted at this node. For instance, a leaf node 6 maintains a BLOOM FILTER BF_6 summarizing key 14 on node 6 and a hash digest h_6. Given node 3 which is the parent of two leaf nodes (node 6 and node 7), its digest is a BLOOM FILTER of the union of its child nodes' BLOOM FILTER, namely $BF_3 = BF_6 \cup BF_7$. In addition, the hash digest is the hash value of the concatenation of all children's' hash and bloom filters, i.e., $h_3 = H(h_6 \| h_7 \| BF_3)$. Generally, an INCBM-TREE uses the following construction:

$$\begin{aligned} BF(\text{parent}) &= BF(\text{left\_child}) \cup BF(\text{right\_child}) \\ h(\text{parent}) &= H(h(\text{left\_child}) \| h(\text{right\_child}) \| BF(\text{parent})) \end{aligned} \quad (2)$$

Here, in different levels in the tree hierarchy, we consider a BLOOM FILTER of the same length.

Proof construction and verification: To enable verification of a query result of VGet, the INCBM-TREE needs to be consulted to construct a proof. To describe the proof construction, we start with the ideal case where the BLOOM FILTER is without error. Following the previous example, in order to provide a proof for result freshness, it suffices to return only two nodes, i.e., node 5 and node 3 in Figure 2a. Since the BLOOM FILTER BF_3 can answer non-membership queries of a given key in the subtree, and digest h_3 can be used to verify authenticity of BF_3. Note that the BLOOM FILTER may contain errors—in the case that non-membership cannot be provided at certain levels, we may need to descend into lower levels so that non-membership can be guaranteed.



(a) The INCBM-TREE structure (b) Incrementally construction

Fig. 2: INCBM-TREE internals

Incremental construction: As mentioned earlier, the time-ordered digest allows for incremental construction. We describe the incremental process of constructing a INCBM-TREE. It is done by maintaining a "frontier" of the INCBM-TREE, a sequence of roots of subtrees. As illustrated in Figure 2b, when new data in the stream arrives, it tries to merge the data with the existing node at the same level. Initially the new data record is at leaf level 0. Merging two INCBM-TREE nodes involves the calculation in Equation 2. If it succeeds in merging, the merged root is promoted one level up and replaces the existing node. A similar merging attempt is tried iteratively until it does not find other existing roots at the same level or the root is reached. The constructing entity only needs to maintain the frontier roots, but not their descendants. The batch digest is produced by exporting the highest subtree with its root hash. Notice that the subtree is always complete and balanced.

III. SYSTEM: MATERIALIZING INCBM-TREE

We implement the proposed INCBM-TREE in a client-server system for outsourced stream serving. The system has two client libraries, which are deployed at owners and users, respectively, allowing them to sign and verify the outsourced stream. The server hosts the outsourced stream and runs on a number of machines in the cloud.

Owner client: We show the system architecture of the owner in Figure 3. Given a stream of data writes, the client invokes a RPC call Write that writes the data to the remote server. At the same time, it locally constructs the digest. It first buffers a batch of data writes and then (more specifically, when the buffer overflows) flushes all the pending data, building a batch digest that includes the root hash of the key-ordered Merkle tree and the bloom filter of this data batch. Recall that the digest based on the key-order Merkle tree cannot work with large batch sizes. This digest of small batches is then sent to the INCBM-TREE construction component, which can build a digest of larger batches. When the INCBM-TREE construction component completes, it will export a digest of a logic tree in which the upper part is an INCBM-TREE and the lower part

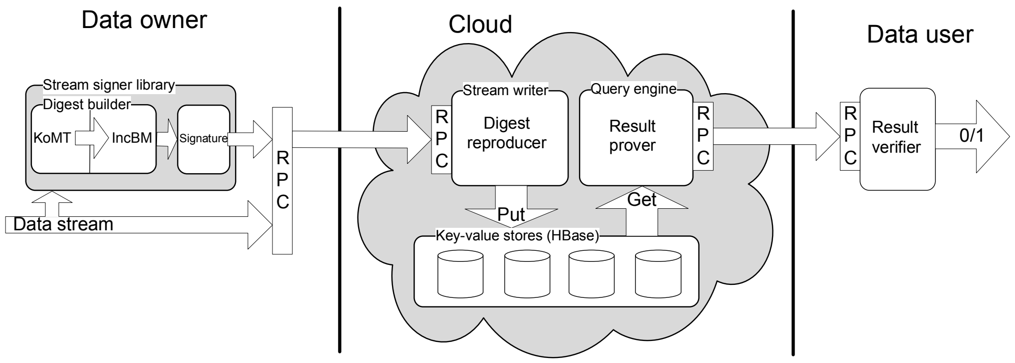


Fig. 3: Outsourced system architecture based on HBase

consists of multiple regular key-ordered Merkle trees. After that, the client invokes the Sign RPC call, which first signs the exported digest using a digital signature and then writes the signed data to the remote server. The two RPC calls used by the owner client have the following function signature in our system:

- $\mathsf{Write}(k, v, ts)$: Given data key k, it inserts or updates this row to be value v.
- $\mathsf{Sign}(d, ts_{\text{last}}, ts_{\text{now}})$: Given a batch of data in time period $\langle ts_{\text{last}}, ts_{\text{now}}\rangle$, the owner client would generate certain digest d, on which the Sign operation signs.

The owner-side client system is designed to be a generic platform which a user or system administrator can configure regarding which stream-signing approach to use. By changing the buffer size, one can use the pure key-ordered Merkle tree or the INCBM-TREE, or their combinations.

Cloud server system: As shown in Figure 3, the cloud server is structured into a two-tier system architecture. An application server is responsible for data coordination, while a storage server persists the streaming data and metadata (i.e., the signatures and digests). On the write path, the application server digests the write stream of data and metadata and sends them to the storage servers. In addition, the digest structures (i.e., INCBM-TREEs and Merkle trees) are not transmitted, as they can be reproduced from the base data stream.

The storage tier uses a write-optimized key-value store (e.g., HBase, Cassandra or BigTable). These stores adopt a log-structured design [4], which lends themselves to persisting the intensive data stream. A key-value store typically exposes a Put/Get interface. There are multiple tables for storing different kinds of data and metadata. The original data is stored in the base table, which is sorted by the data key. To store the signatures, we use the generation time as a signature's primary key and store it in the signature table. To store the reproduced digest, we explore two design choices:

- *Storing digests in the signature table*: By default, the digests, after being reproduced by the application server, are stored in the signature table by application servers issuing the Put call to key-value stores.
- *Caching digests in the base table*: When the application server receives a VGet call from clients, it retrieves the digest and signatures from the signature table to construct the proof for verification. In this process, we allow the retrieved digest to be cached on the base table, which facilitates the processing of repeated VGet requests in the (near) future.

User client: The user system, designed as in Figure 3, processes a query by sending a RPC request, whose function signature is described in Equation 1, to the remote server, which returns both the data result and the verification proof. Given such a proof, the client applies the verification process (e.g., verifying the authentication path in Merkle trees, or verifying the Bloom filters and hash digests in the INCBM-TREE) to verify the authenticity of the data result returned from the outsource server.

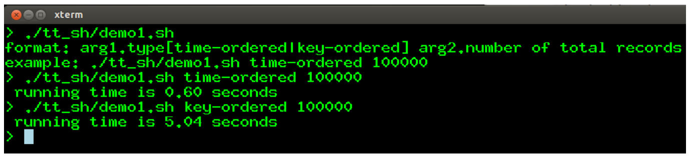
IV. DEMONSTRATION SCENARIOS

In the demonstration, we show the efficiency of the stream signature and verification operations using our INCBM-TREE. The demonstration is based on a key-value dataset generated by YCSB [5], an industry standard cloud benchmark tool. In particular, we choose an update-heavy workload (i.e., the workload-A in YCSB dataset) to simulate the real-world cloud streaming application used in Yahoo! (e.g., session store). In this workload, data keys are generated by a Zipf distribution, and there are 50% write operations and 50% read operations. The cloud server used in this demonstration is a 11-node cluster in Emulab [6]. Each node is equipped with a 2.4 GHz 64-bit Quad Core Xeon processor with hyper-threading support and 12 GB RAM. To deploy our 2-tier server in this cluster, we use one node for application server and the rest for storage servers. The storage servers use HBase [7][1], an industrial strength NoSQL server that is optimized for writes and is modeled after Google BigTable [8]. The clients are two regular servers at our university, whose location is geographically remote to the servers in Emulab.

A. Scenario 1: Owner's write throughput

This demonstration shows INCBM-TREE's capability to ingest the data stream. We load a dataset into the system and measure the total running time. To be specific, it starts with replaying a data stream from a pre-materialized YCSB dataset

[1]We use version $0.94.2$.



```
> ./tt_sh/demo1.sh
format: arg1.type[time-ordered|key-ordered] arg2.number of total records
example: ./tt_sh/demo1.sh time-ordered 100000
> ./tt_sh/demo1.sh time-ordered 100000
 running time is 0.60 seconds
> ./tt_sh/demo1.sh key-ordered 100000
 running time is 5.04 seconds
> 
```

Fig. 4: The command-line interaction in Demo 1

and shipping it to the client machine, which runs an owner client to absorb and sign the stream before publishing it to the servers. In the demonstration, the audience observes how long it takes to complete the stream ingestion. For comparison, we demonstrate the conventional approach of using key-ordered Merkle tree, and the audience can see the different running time.

Figure 4 shows a command-line interface that is used to specify in the demonstration script (1) which signing approach to use, be it either INCBM-TREE or key-ordered approach; and (2) the size of the data stream, which affects the duration of the demonstration. For example, in the figure, the first execution with time-ordered INCBM-TREE runs in 0.6 second, while the key-ordered Merkle tree approach takes 5.04 seconds.

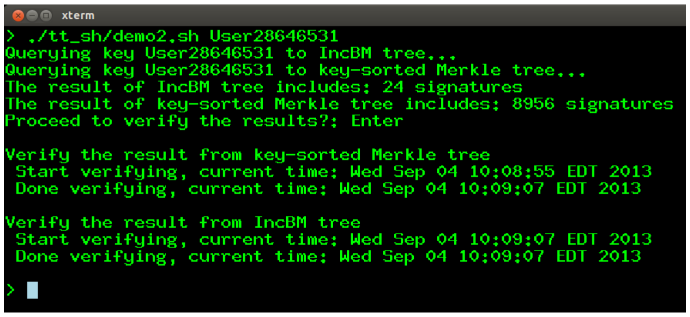
B. Scenario 2: User's verification cost

Given a VGet query, the verification costs are dominated by the number of signatures returned from the outsource server. In this demonstration, we show the efficiency of INCBM-TREE in terms of verification costs. Initially, the server is preloaded with both the INCBM-TREE and the key-ordered Merkle tree. In the demo, the user can issue a VGet query to the server and then inspect the number of signatures in the raw result from the server.

For the key-ordered Merkle tree, the number of signatures equals the number of signed Merkle root hashes that are newer than the result version. For example, the query in Figure 5 specifies the key with k = User28646531 and the demo issues two queries against the server, one to the INCBM-TREE for constructing a verification proof and one to the key-sorted Merkle tree. The result returned from INCBM-TREE includes 24 signatures while the result from the Merkle tree has 8956 signatures to verify. After the user's confirmation, the demo then continues to actually execute the verification process. This process is implemented by using the DSA signatures with SHA1 padding in the standard Java.security library. With this implementation, verifying 8956 signatures is a process with human perceivable delay, while running 24 verifications can finish without human notice, as shown in Figure 5. In our live demo, the user can freely specify the queried key and inspect different query results. Lower number of signatures of INCBM-TREE is expected regardless of which key is used in the query.

ACKNOWLEDGMENTS

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defense and was accomplished under Agreement Number W911NF-06-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defense or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. Yuzhe Tang and Ling Liu were partially funded by NSF grants under NetSE, SaTC and I/UCRC.



```
> ./tt_sh/demo2.sh User28646531
Querying key User28646531 to IncBM tree...
Querying key User28646531 to key-sorted Merkle tree...
The result of IncBM tree includes: 24 signatures
The result of key-sorted Merkle tree includes: 8956 signatures
Proceed to verify the results?: Enter

Verify the result from key-sorted Merkle tree
 Start verifying, current time: Wed Sep 04 10:08:55 EDT 2013
 Done verifying, current time: Wed Sep 04 10:09:07 EDT 2013

Verify the result from IncBM tree
 Start verifying, current time: Wed Sep 04 10:09:07 EDT 2013
 Done verifying, current time: Wed Sep 04 10:09:07 EDT 2013

> 
```

Fig. 5: The command-line interaction in Demo 2

REFERENCES

[1] F. Li, K. Yi, M. Hadjieleftheriou, and G. Kollios, "Proof-infused streams: Enabling authentication of sliding window queries on streams," in *VLDB*, 2007, pp. 147–158.

[2] G. Cormode, A. Deligiannakis, M. Garofalakis, and S. Papadopoulos, "Lightweight authentication of linear algebraic queries on data streams," in *SIGMOD*, 2013.

[3] D. Schröder and H. Schröder, "Verifiable data streaming," in *ACM Conference on Computer and Communications Security*, 2012, pp. 953–964.

[4] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (lsm-tree)," *Acta Inf.*, vol. 33, no. 4, pp. 351–385, 1996.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *SoCC*, 2010, pp. 143–154.

[6] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *OSDI*, 2002.

[7] "http://hbase.apache.org/."

[8] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data (awarded best paper!)," in *OSDI*, 2006, pp. 205–218.