

# Authenticated Join Processing in Outsourced Databases

Yin Yang<sup>1</sup>

Dimitris Papadias<sup>1</sup>

Stavros Papadopoulos<sup>1</sup>

Panos Kalnis<sup>2</sup>

<sup>1</sup>Department of Computer Science and Engineering  
Hong Kong University of Science and Technology  
Clear Water Bay, Hong Kong  
{yini, dimitris, stavros}@cse.ust.hk

<sup>2</sup>Division of Math. and Computer Sciences and Engineering  
King Abdullah University of Science and Technology  
Saudi Arabia  
panos.kalnis@kaust.edu.sa

## ABSTRACT

Database outsourcing requires that a query server constructs a proof of result correctness, which can be verified by the client using the data owner's signature. Previous authentication techniques deal with range queries on a single relation using an *authenticated data structure* (ADS). On the other hand, authenticated join processing is inherently more complex than ranges since only the base relations (but not their combination) are signed by the owner. In this paper, we present three novel join algorithms depending on the ADS availability: (i) *Authenticated Indexed Sort Merge Join* (AISM), which utilizes a single ADS on the join attribute, (ii) *Authenticated Index Merge Join* (AIM) that requires an ADS (on the join attribute) for both relations, and (iii) *Authenticated Sort Merge Join* (ASM), which does not rely on any ADS. We experimentally demonstrate that the proposed methods outperform two benchmark algorithms, often by several orders of magnitude, on all performance metrics, and effectively shift the workload to the outsourcing service. Finally, we extend our techniques to complex queries that combine multi-way joins with selections and projections.

## 1. INTRODUCTION

Database outsourcing [8] has been successfully applied to edge computing [18], peer-to-peer networks [7], database caching [11], etc. In this setting, a *data owner* outsources database functionality to a third-party *database service provider* (DSP) that maintains the data in a DBMS, and answers queries to *clients*. *Authenticated query processing* enables the DSP to prove the correctness of the results. Existing methods are based on the secret/public key framework. Specifically, the DSP indexes the signed data using an *authenticated data structure* (ADS). During query processing, it traverses the ADS and returns a *verification object* (*VO*) that includes the actual result and additional verification information. The *VO* is transmitted to the client, which can establish *soundness* and *completeness* using the public key of the owner. Soundness means that every record in the result set is present in the owner's database and not altered. Completeness means that all valid results are included.

In our examples, we use the database and queries of Figure 1. Given  $Q_0 = \sigma_{quantity>100}Purchase$ , the correct result set is  $RS = \{ \langle p_4, c_1, 200 \rangle, \langle p_5, c_2, 500 \rangle \}$ .  $RS_1 = \{ \langle p_4, c_1, 200 \rangle, \langle p_5, c_2, 500 \rangle \}$  and  $RS_2 = \{ \langle p_4, c_1, 200 \rangle, \langle p'_5, c_2, 555 \rangle \}$  are not sound because they either contain fake ( $p_6$  in  $RS_1$ ), or altered ( $p'_5$  in  $RS_2$ ) records.  $RS_3 = \{ \langle p_4, c_1, 200 \rangle \}$  is not complete because  $p_5$  is missing. Besides achieving soundness and completeness, authenticated query processing methods should minimize (i) the *VO* size, which dominates the communication overhead between the client and the DSP, (ii) the verification cost at the client, and (iii) the query processing time at the DSP. In most applications, (i) and (ii) are more important goals than (iii), since the client usually has less computational power and bandwidth compared to the DSP.

500>,  $\langle p_6, c_2, 600 \rangle$  and  $RS_2 = \{ \langle p_4, c_1, 200 \rangle, \langle p'_5, c_2, 555 \rangle \}$  are not sound because they either contain fake ( $p_6$  in  $RS_1$ ), or altered ( $p'_5$  in  $RS_2$ ) records.  $RS_3 = \{ \langle p_4, c_1, 200 \rangle \}$  is not complete because  $p_5$  is missing. Besides achieving soundness and completeness, authenticated query processing methods should minimize (i) the *VO* size, which dominates the communication overhead between the client and the DSP, (ii) the verification cost at the client, and (iii) the query processing time at the DSP. In most applications, (i) and (ii) are more important goals than (iii), since the client usually has less computational power and bandwidth compared to the DSP.

Purchase			Customer		
pid	cid	quantity	cid	name	city
$p_1$	$c_1$	20	$c_1$	Tom	New York
$p_2$	$c_3$	50	$c_2$	Brian	London
$p_3$	$c_2$	80	$c_3$	Susan	Tokyo
$p_4$	$c_1$	200	$c_4$	Jane	New York
$p_5$	$c_2$	500	$c_5$	Carl	London

$$Q_0 = \sigma_{quantity>100}Purchase$$

$$Q_1 = Purchase \bowtie_{cid} Customer$$

$$Q_2 = (\sigma_{quantity>100}Purchase) \bowtie_{cid} Customer$$

$$Q_3 = (\sigma_{quantity>100}Purchase) \bowtie_{cid} (\sigma_{city='New York'} Customer)$$

Figure 1 Running Example

Existing solutions focus on single-relation ranges. On the other hand, authenticated join processing is inherently more complex than ranges because only the base relations, and not their combinations, are signed by the owner. As it will become clear later, the client must always perform some computations to verify, as well as, generate part of the result locally. Previous work on this intricate problem is scarce and has severe shortcomings, defeating the goal of data outsourcing. Motivated by this, we propose three novel authenticated join algorithms depending on the ADS availability: (i) *Authenticated Indexed Sort Merge Join* (AISM), which utilizes a single ADS in one of the base relations, (ii) *Authenticated Index Merge Join* (AIM) that requires an ADS for both relations, and (iii) *Authenticated Sort Merge Join* (ASM), which does not rely on any ADS. Going one step further, we describe the adaptation of our methods to authentication of complex queries involving joins over multiple tables, possibly combined with selections and projections. In particular, we show that for such queries, the best execution plan may involve not only AIM, but also AISM and ASM, in the presence of all required ADSs.

The rest of the paper is organized as follows. Section 2 surveys related work on authenticated query processing. Sections 3, 4 and 5 describe AISM, AIM and ASM, respectively. Section 6 extends these methods to complex query authentication. Section 7 contains an extensive experimental evaluation, and Section 8 concludes the paper.

To appear in SIGMOD 2009

SIGMOD '09, Providence, RI, USA.  
Copyright 2009 ACM 1-58113-000-0/00/0004...\$5.00.

## 2. RELATED WORK

Section 2.1 overviews authentication techniques for range queries. Section 2.2 discusses authenticated join processing. Before we proceed, we provide some basic cryptographic background. A *one-way, collision-resistant hash function*  $H$  takes as input a message  $m$  of arbitrary length and produces a *digest* of fixed length.  $H$  has two properties: (i) computing  $m$  from  $H(m)$  is intractable, and (ii) the probability of two different messages to have the same digest is very low. A *public key digital signature scheme* involves the generation of a secret ( $sk$ ) and a public ( $pk$ ) key:  $sk$  is known only to the signer, whereas  $pk$  is published. To produce signature  $s$  of a message  $m$ , the signer applies  $sk$  to the digest of  $m$ . Given  $s$ ,  $m$  and  $pk$ , the verifier can certify that  $m$  has not been falsified (*integrity*) and that  $m$  indeed originates from the party that signs it (*authenticity*). Note that since  $H$  is not commutative, a signature  $s$  on a set of records  $S$  can only certify a fixed order of  $S$  called the *verifiable order*. For instance, assume that a message  $m$  contains the concatenation  $s_1|s_2|s_3$  of three records. A signature  $s(m)$  cannot be used to certify  $s_3|s_2|s_1$ , or any order other than  $s_1|s_2|s_3$ .

### 2.1 Authenticated range query processing

Authenticated range query processing was first studied in computer security community. [5] proposes a method that sorts the records on the query attribute and indexes them by a *Merkle Hash Tree (MHT)* [13]. The MHT is a binary tree that provides the foundation for a broad class of ADSs, e.g., [5], [14], [10]. Every leaf node contains the digest of a record. The tree is constructed bottom-up; each internal node stores a hash value computed on the concatenation of the children digests. The data owner signs the root using the secret key. Given a range query, the DSP first expands it to include two *boundary records*, and processes it using the MHT. The client can verify soundness by exploiting the collision-resistance property of the hash function. Furthermore, the boundary records ensure completeness, i.e., that no result is missing at the query endpoints. [14] extends the concepts of the MHT to Directed Acyclic Graphs, including dictionaries, tries, and range search trees. Dynamic versions of the MHT for outsourced data streams are discussed in [12], [19].

The first disk-based ADS for range query processing [18] guarantees soundness, but not completeness. A subsequent *signature chaining* approach [17][16] ensures both soundness and completeness. Currently, the state-of-the-art ADS is the *Merkle B-tree (MB-tree)* [10], which combines the MHT with the B+-tree, i.e., it can be thought of as a MHT where the node fanout is determined by the block size. Figure 2 illustrates query processing using the MB-tree. Given a range query, the DSP traverses the MB-tree top-down until it finds the first record (let  $s_i$ ) in the range. During the traversal, the following items are inserted into the verification object  $VO$ : (i) the digests of the left siblings of  $N_1$  in the root, (ii) the digests of the left siblings of  $N_3$  in  $N_1$ , (iii) the (boundary) record  $s_{i-1}$  preceding  $s_i$ , and (iv) the digests of the left siblings of  $s_{i-1}$  in  $N_3$ . Next, the DSP retrieves the query result  $s_i, s_{i+1}, \dots, s_j$  by following the pointers between leaf nodes. The (boundary) record  $s_{j+1}$  is added to the  $VO$ . Finally, a second traversal from the root to  $s_{j+1}$  inserts all the digests on the right of the path. In Figure 2, the digests contained in the  $VO$  are shaded. Given  $s_{i-1}, s_i, \dots, s_j, s_{j+1}$  and the digests, the client can re-compute the digest of the root and verify it against the owner's signature. The EMB-tree [10] reduces the  $VO$  size by embedding a binary MHT inside each internal node of the MB-tree. The MR-tree [24]

applies the concept of the MHT to R-trees for authentication of multi-dimensional ranges on outsourced spatial data.

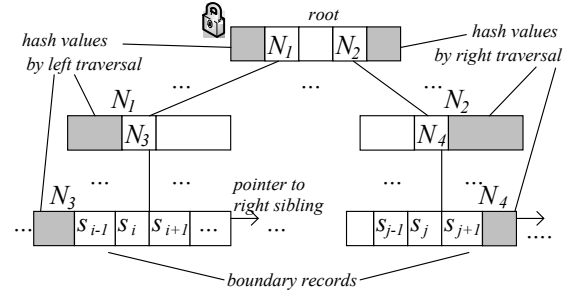


Figure 2 Example of Merkle B-tree

[2] introduces a theoretical approach with improved asymptotic bounds for the  $VO$  size. To eliminate the threat of revealing sensitive information to unauthorized clients, [9] proposes an alternative scheme that avoids boundary records and hash values in the  $VO$ . Several papers investigate outsourcing in applications with semi-trusted DSP (e.g., [22]) or clients (e.g., [23]), in which case authentication can be accomplished without an ADS. Specifically, [22] assumes that the DSP's only motivation to cheat is to save resources, and proposes a solution in which the DSP proves that it has performed the necessary computations to correctly answer the queries. In [23], the owner introduces fake tuples to the outsourced database, which are known to the clients but not the DSP. A client can thus establish soundness and completeness by analyzing the fake records in the result. These solutions are not applicable to our model since we do not rely on any degree of trust for the DSP or the clients. Finally, several papers ([8],[6],[3]) investigate *privacy preservation* of outsourced data. GhostDB [1] answers queries with both an untrusted server and a secure chip embedded in a USB key. These issues are orthogonal to join authentication and the proposed methods.

### 2.2 Authenticated join processing

[18] proposes the pre-computation and storage of all possible join results in materialized views. Each view is treated as a conventional table, meaning that an ADS can be built on it to support more complex queries. For example, the result of  $Q_1 = Purchase \bowtie_{cid} Customer$  can be materialized in a view  $V_1$ . If an ADS is maintained on  $V_1$ ,  $quantity$ , the DSP can answer  $Q_2 = (\sigma_{quantity > 100} Purchase) \bowtie_{cid} Customer$  by transforming it to  $\sigma_{quantity > 100} V_1$ . This approach imposes a significant overhead for the owner to construct and update a large number of materialized views. Moreover, in most practical applications it is infeasible to determine all possible joins in advance. The only existing algorithm for on-line join processing is discussed in [17] and [10] as an extension of range authentication. We refer to this algorithm as *Authenticated Index Nested Loop (AINL)* since it is based on the index nested loop paradigm, and discuss it in detail below.

Let  $R$  and  $S$  be the two relations to be joined on a common attribute  $a$ , and consider that there is an ADS  $T_S$  (i.e., MB-tree) on  $S.a$ .  $R$  constitutes the *outer* and  $S$  the *inner* relation. Figure 3 illustrates the pseudo-code of AINL assuming a join  $R \bowtie_{R.a=S.a} S$ . Initially, the signature of  $R$ , the cardinality  $|R|$  of  $R$  and the signature of  $T_S$  are inserted into the  $VO$ . Then, for each record  $r \in R$  in the *verifiable* order, the DSP appends  $r$  to the  $VO$  and retrieves the matching records in  $S$ , by processing a range query using  $T_S$  (we use the term *range* to also denote equality conditions). As discussed in Section 2.1 (see Figure 2), the output of this

query includes (i) the join matches of  $r$ , (ii) boundary records, and (iii) digests obtained during  $T_S$  traversal. These values are inserted into the  $VO$ , together with a separator ";" that signifies the end of each range query (hereafter, denoted as a *round*). The processing terminates when all records of  $R$  are exhausted, and the DSP transmits the  $VO$  to the client.

---

```
AINL (Relation  $R$ , MBTree  $T_S$ ,  $VO$ )           // DSP
// The join query is  $R \bowtie_{R.a=S.a} S$ 
1. Append to  $VO$  the signature of  $R$ ,  $|R|$ , the root signature of  $T_S$ 
2. For each  $r \in R$                                // in the verifiable order
3.   Append  $r$  to  $VO$ 
4.   Call  $RangeSearch(T_S, r.a, VO)$  // process range query on  $T_S$ 
5.   Append a separator ";" to  $VO$ 
```

---

**Figure 3** Algorithm *AINL*

The client can reconstruct and authenticate the join result using the algorithm of Figure 4. Specifically, it can establish the correctness of  $R$  based on the owners' signature. Furthermore, for each record  $r \in R$ , it can verify  $RangeSearch(T_S, r.a, VO)$  using the mechanisms of the MB-tree. Note that since the  $VO$  of the range query contains some additional (boundary) records, the actual matching tuples of  $r$  are extracted in line 6.

---

```
Verify_AINL ( $VO$ )                               // Client
1. Read the signature of  $R$ ,  $|R|$ , the root signature of  $T_S$  from  $VO$ 
2. For  $i = 1$  To  $|R|$ 
3.   Read tuple  $r$  from  $VO$ 
4.   Read until reaching the separator ";"
5.   Verify that the data read in line 4 are the  $VO$  for
       $RangeSearch(T_S, r.a, VO)$ 
6.   Extract each matching pair of tuples  $s$  of  $r$  read in line 4,
      generate a join result combining  $r$  and  $s$ 
7.   Verify the signature of  $R$ 
```

---

**Figure 4** Algorithm *Verify\_AINL*

For instance, consider  $Q_1 = Purchase \bowtie_{cid} Customer$  in Figure 1, with *Purchase* as the outer relation. Initially, the  $VO$  contains the signature of *Purchase*, its cardinality (5) and the signature of  $T_{Customer}$ . Let the verifiable order of *Purchase* be  $(p_1, p_2, p_3, p_4, p_5)$ <sup>1</sup>. The DSP performs a range query on  $T_{Customer}$  to find the matching customer  $c_1$  of  $p_1$ . Consequently it adds to the  $VO$ :  $p_1, c_1, c_2$  (boundary record for  $c_1$ ) and the digests of  $T_{Customer}$  needed to verify the correctness of the range. The separator ";" denotes the end of the first round. Similarly, the second round appends to the  $VO$ :  $p_2, c_3$ , boundary records  $c_2$  and  $c_4$ , and the necessary digests. In total, there are 5 rounds, each of which corresponds to a tuple in *Purchase*. The  $VO$  contains 5 *Purchase* tuples and 13 records from *Customer*.

Let  $|R|, |S|, |RS|$  be the cardinality of  $R, S$  and the join result, respectively.  $R \bowtie_{R.a=S.a} S$  necessitates the transmission of  $|R|$  records of  $R$  and  $2|R|+|RS|$  tuples of  $S$  (the matching tuples plus 2 boundaries records per  $R$  tuple), in addition to a large number of digests. Furthermore, AINL incurs a high computational overhead for both the DSP (to process 5 range queries in the example) and the client (to verify them). This motivates the naive alternative (referred to as *NAI*) of executing the join exclusively at the client side. Specifically, according to NAI, the DSP simply transmits the base relations along with their signatures to the client, which

verifies them and performs the join locally. The  $VO$  size ( $|R| + |S|$  tuples) of NAI is usually much lower than that of AINL, except for the case where  $|R| \ll |S|$  and the join is highly selective. Furthermore, unless  $|R|$  is very small, the verification of  $|R|$  range queries in AINL burdens the client more than joining the two tables directly. Although it is often better than AINL, NAI is far from an ideal solution. First, the query is processed entirely by the client, which contradicts the purpose of data outsourcing. Second, the DSP transmits all records of the base relations, while the client only needs those with matching partners. Finally, NAI cannot take advantage of the existence of ADS on the data, or selection conditions on the query. Next, we propose algorithms that overcome these shortcomings of AINL and NAI. For ease of presentation, we first focus on binary equi-joins, and defer the discussion on other join conditions and complex queries for later sections.

### 3. AISM

Similar to AINL, our first algorithm AISM (for Authenticated Indexed Sort-Merge join) utilizes an ADS for the inner relation. We demonstrate the basic idea of AISM using  $Q_1 = Purchase \bowtie_{cid} Customer$ , and assuming that the DSP maintains a MB-tree  $T_{Customer}$  on *Customer.cid*. In a pre-processing step, the DSP sorts the outer table *Purchase* on the join attribute *cid*, and generates the rank list  $\Omega_{Purchase}$ . The purpose of the rank list is to inform the client on how to restore the verifiable order of the records (required for signature verification). For instance, assuming the verifiable order  $p_1, p_2, p_3, p_4, p_5$ , we have  $\Omega_{Purchase} = (1, 4, 3, 5, 2)$ , meaning that  $p_1$  has the smallest value ( $c_1$ ) on *cid*,  $p_4$  has the second smallest value, and so on. Note that unlike conventional sort-merge join, in AISM the sole purpose of sorting the outer relation is to generate the corresponding rank list. Thus, it suffices to sort only the join attribute values, which can often be performed in main memory. The DSP transmits all tuples of the outer relation (*Purchase*) to the client in their verifiable order, along with the owner's signature and  $\Omega_{Purchase}$ . Next, the DSP turns to the inner relation *Customer*. Observe from Figure 1 that all purchases involve clients  $c_1, c_2$  and  $c_3$ . Therefore, it is possible to find all matching customers for these purchases by a single range  $Q = \sigma_{c_1 \leq cid \leq c_3} Customer$  on  $T_{Customer}$ . Due to the nature of the MB-tree, the results of  $Q$  (i.e., customers with *cid*  $c_1, c_2$ , and  $c_3$ ) are sorted on *cid* and can be authenticated. Meanwhile, the rank list  $\Omega_{Purchase}$  explicitly specifies the order of purchases when sorted on *cid*. Therefore, if the client obtains *Purchase*,  $\Omega_{Purchase}$  and the results of  $Q = \sigma_{c_1 \leq cid \leq c_3} Customer$ , it can generate and authenticate the result of  $Purchase \bowtie_{cid} Customer$ , by merging the output of  $Q$  with  $\Omega_{Purchase}$ .

In general, given a query  $R \bowtie_{R.a=S.a} S$  and a MB-tree  $T_S$  on  $S.a$ , the DSP processes a single *multi-range*  $Q$  with one traversal of  $T_S$ , and merges its  $VO$ , denoted as  $VO(Q)$ , with the rank list  $\Omega_R$ .  $\Omega_R[i]=j$  signifies that the  $i$ -th element of  $\Omega_R$  corresponds to the  $j$ -th record in the verifiable order. For ease of presentation, we also use  $\Omega_R[i]$  to denote this record. We illustrate the processing of  $Q$  using the relations of Figure 5, where  $|S|=15$  and  $|R|=6$ . The join result contains three pairs  $(\Omega_R[1], s_3)$ ,  $(\Omega_R[2], s_{11})$  and  $(\Omega_R[3], s_{11})$ . Initially, the DSP inserts into the  $VO$ : (i) the signature of  $R$ , (ii) the root signature of  $T_S$ , and (iii) all records of  $R$  in the verifiable order (which can be arbitrary). Then, it sorts  $R$  according to the join attribute  $a$ , breaking ties by the original (i.e., verifiable) order of  $R$ , and generates the rank list  $\Omega_R$ . For each  $\Omega_R[i]$ , the DSP first inserts it in the  $VO$ , and performs two operations on  $T_S$ , which we

<sup>1</sup> For simplicity, we refer to a tuple by its id, i.e.,  $p_1$  signifies the first record of *Purchase*.

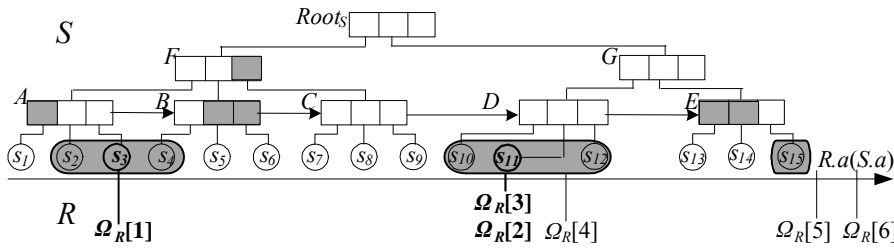


Figure 5 Example of AISM

call *index-traversal* and *leaf-scan*. Index-traversal traverses  $T_S$  to the leaf node that corresponds to the left boundary record of  $\Omega_R[i]$ . In the example of Figure 5, the DSP first inserts  $\Omega_R[1]$  in the  $VO$ , and descends  $T_S$  until node  $A$ , whose second entry corresponds to  $s_2$ , the left boundary of  $\Omega_R[1]$ . The digests of all left siblings ( $h_1$ ) along the path ( $Root-F-A$ ) are appended to the  $VO$ .

The leaf-scan starts from a leaf entry and follows the successor pointers, until reaching the right boundary record. During this step, all encountered tuples are inserted into the  $VO$ . Continuing the example, the DSP appends to the  $VO$   $s_2, s_3, s_4$  (right boundary of  $\Omega_R[1]$ ) and a separator “;” signifying the end of the first round. The DSP proceeds to the next round, and inserts  $\Omega_R[2]$  into the  $VO$ . Index-traversal starts from the *current position* (first entry of  $B$ ), ascends the tree until the root, and then descends to node  $D$ , which contains the left boundary  $s_{10}$  of  $\Omega_R[2]$ . The digest ( $h_5, h_6, h_C$ ) of each skipped child is inserted into the  $VO$ . Leaf-scan adds  $s_{10}$ , the matching tuple  $s_{11}$  of  $\Omega_R[2]$ , and  $s_{12}$  (i.e., right boundary) to the  $VO$ .

At the beginning of the third round, the DSP appends  $\Omega_R[3]$  to the  $VO$ , which has the same join attribute value as  $\Omega_R[2]$ . Index-traversal discovers that the left boundary  $s_{10}$  is before the current position  $s_{12}$ . An important principle of AISM is that the DSP never traverses the tree *backwards*, and index-traversal is skipped. Similarly, because the right boundary  $s_{12}$  has also been found, leaf-scan is also omitted, and the third round terminates. For the same reason, the fourth round simply appends  $\Omega_R[4]$  to the  $VO$ . At the fifth round, index-traversal reaches  $s_{15}$ , appending  $h_{13}, h_{14}$  to the  $VO$ . Since  $s_{15}$  is already the last record in  $S$ , leaf-scan inserts  $s_{15}$ , and the sixth round is skipped. Figure 6 illustrates AISM at the DSP side. We omit the pseudo-code for index-traversal and leaf-scan since their functionality is clear from the examples.

```

AISM (Relation R, MBTree TS, VO) // DSP
// The join query is R ⋈R.a=S.a S
1. Append to the VO the signature of R, |R|, the root signature of TS,
   and all records of R in a verifiable order
2. Sort R to generate the rank list ΩR
3. Initialize n = TS.Root
4. For i = 1 To |R|
5.   Append ΩR[i] to VO
6.   Call IndexTraversal(n, R[ΩR[i]].a, VO)
7.   Call LeafScan(n, R[ΩR[i]].a, VO)
8.   Append a separator “;” to VO
9. Call IndexTraversal(n, +∞, VO) // to complete the traversal of TS

```

Figure 6 Algorithm AISM

Figure 7 describes the verification process, which includes the actual result extraction from the  $VO$ . Specifically, the client performs a single scan of the  $VO$  to (i) verify the signature of  $R$ , (ii) establish the correctness of  $\Omega_R$ , (iii) reconstruct the root hash of  $T_S$  and match it against the signature, (iv) verify the results for

	$VO$ : signature of $R$ , root signature of $T_S$ , $r_1-r_6$ in their verifiable order
Round 1	$\Omega_R[1], h_1, s_2, s_3, s_4;$
Round 2	$\Omega_R[2], h_5, h_6, h_C, s_{10}, s_{11}, s_{12};$
Round 3	$\Omega_R[3];$
Round 4	$\Omega_R[4];$
Round 5	$\Omega_R[5], h_{13}, h_{14}, s_{15};$
Round 6	$\Omega_R[6];$

each  $\Omega_R[i]$ , and generate join output. Operation (i) is trivial since  $R$  is received in the verifiable order (line 2). In operation (ii), the client checks that  $|\Omega_R| = |R|$ , and for each pair of subsequent elements in  $\Omega_R$ ,  $\Omega_R[i].a \leq \Omega_R[i+1].a$ . Moreover, if  $\Omega_R[i].a = \Omega_R[i+1].a$ , the client checks that  $\Omega_R[i] < \Omega_R[i+1]$  (lines 5-7). For operation (iii), the client uses the records and digests of  $S$  to derive<sup>2</sup> the digest  $h_{Root}$  at the root of  $T_S$  bottom-up (line 11). In the above example, the  $VO$  contains  $h_1, h_5, h_6, h_C, h_{13}, h_{14}$ . The client computes  $h_2, h_3, h_4, h_{10}, h_{11}, h_{12}, h_{15}$  by applying  $H$  on the corresponding records. Then, it obtains  $h_A$  (using  $h_1-h_3$ ),  $h_B$  (using  $h_4-h_6$ ),  $h_D$  (using  $h_{10}-h_{12}$ ),  $h_E$  (using  $h_{13}-h_{15}$ ),  $h_F$  (using  $h_A-h_C$ ),  $h_G$  (using  $h_D-h_E$ ), and finally  $h_{Root}$  (using  $h_F-h_G$ ), which is matched against the signature of  $T_S$ . Operation (iv) corresponds to range query verification; i.e., the client ensures, for each  $r \in R$ , that the boundary records enclose the matching tuples and only matching tuples, which are extracted to generate join results (line 10).

```

Verify_AISM (VO) // Client
1. Read the signature of R, |R|, the root signature of TS, and all
   records of R from VO
2. Verify the signature of R
3. Initialize integer j=0 and record r so that r.a = -∞
4. For i = 1 To |R|
5.   Set j' = j and r' = r
6.   Read integer j from VO, set r = R[j]
7.   Check the condition (r.a > r'.a) ∨ (r.a = r'.a ∧ j > j')
8.   Read from VO until reaching the separator “;”
9.   Verify that (i) the previous step only reads S tuples and digests,
   (ii) the S tuples either match R or are boundary records, and
   (iii) no digest is enclosed by boundary records
10.  Generate join results of r and its matching S records
11.  Use the values read in line 8 to incrementally compute hRoot
12.  Read digests from VO until it is empty, use them to incrementally
   compute hRoot
13.  Verify hRoot against the root signature

```

Figure 7 Algorithm Verify\_AISM

*Proof of soundness:* Let  $rs$  be an incorrect answer. Then, either (i)  $r$  does not match  $s$ , or (ii)  $r$  or  $s$  are bogus/alterred. The first case cannot happen because the client generates matching pairs by itself. For the second case, a fake  $r$  tuple is detected by the authentication information of  $R$ . An incorrect  $s$  tuple leads to the wrong  $h_{Root}$ , failing the verification against the signature of  $T_S$ . □

*Proof of completeness:* Let  $rs$  be a valid result of the query missed by the client. Then either (i) the client does not receive  $r$  or  $s$ , or (ii) the client does not identify  $r$  and  $s$  as a matching pair. For case (i), if  $r$  is missing, the verification against the authentication information of  $R$  fails. On the other hand, if  $s$  is absent from the

<sup>2</sup> When the tree is not full, the DSP must put additional boundary tokens in the  $VO$  to inform the client about the tree structure [10].

$VO$ , for the client to correctly construct  $h_{Root_S}$ , the  $VO$  must contain the digest  $h$  of  $s$  or of a node covering  $s$ . For instance, if  $s_3$  were omitted, then the  $VO$  of Figure 5 should include  $h_3$ . The client, however, will detect either that a digest is enclosed by two boundary records, or that a boundary record is missing (line 9 of *Verify\_AISM*). For case (ii), note that the client has all relevant  $R$  (and  $S$ ) tuples sorted on the join attribute  $a$ . Specifically, the order of  $R$  records is established by  $\Omega_R$  (verified by the client), while the order of  $S$  records is given by ADS  $T_S$  and the *no-go-back* policy during the tree traversal. Since the client performs the merging by itself, it finds all matching pairs of  $R$  and  $S$ , eliminating the possibility of missing  $rs$ .  $\square$

AIMS eliminates the repeated computations and redundant transmissions incurred by AINL. Specifically, the DSP visits a node in  $T_S$  at most once, rather than up to  $|R|$  times in the case of AINL. Meanwhile, using AISM, the client never repeats the computation of any hash value, and each element of  $T_S$  (e.g., digest or  $S$  tuple) is included in the  $VO$  at most once. Comparing AISM with NAI, the former avoids the transmission of  $S$  records that do not have join partners. In addition, whereas the client performs the entire join processing in NAI, AISM shifts most of the workload to the DSP (e.g., sorting  $R$ , traversing  $T_S$ ), leaving only inexpensive operations (e.g., rebuilding  $h_{Root_S}$ , merging the sorted relations) to the client. An interesting observation is that AISM is notably more efficient for the indexed relation ( $S$ ) than the non-indexed one ( $R$ ), suggesting that the performance can be improved by utilizing a second ADS on  $R$ . This motivates the next algorithm.

#### 4. AIM

Authenticated Index Merge join (AIM) utilizes ADSs on the join attribute in both input relations. Figure 8 illustrates two MB-trees  $T_S$  on  $S.a$  and  $T_R$  on  $R.a$  for the datasets of Figure 5. Initially, the DSP inserts the root signatures of  $T_S$  and  $T_R$  into the  $VO$ . It then chooses one tree, say  $T_R$ , reaches its first leaf node ( $H$ ), finds the first record ( $r_1$ ), and inserts it into the  $VO$ . With  $r_1.a$  as target, the DSP performs index-traversal and leaf-scan on  $T_S$ , to retrieve matching and boundary records. Index-traversal visits the path from  $Root_S$  to the first boundary  $s_2$  of  $r_1$ . The digest ( $h_{s_1}$ ) of the left sibling entry is appended to the  $VO$ . In the subsequent leaf-scan,  $s_2, s_3$  and  $s_4$  are also added. Note that a match (i.e.,  $s_3$ ) for  $r_1$  is found. Every time AIM identifies a result, it performs another leaf-scan on  $T_R$ . Continuing the example, the DSP appends a separator “;” to the  $VO$ , performs the leaf-scan on  $T_R$ , with target  $r_1.a = s_3.a$ , leading to the insertion of  $r_2$  (right boundary) to the  $VO$ . This additional operation retrieves all  $R$  tuples with identical join attribute as  $r_1$ , and is vital to the correctness of AIM.

The second round starts at the current positions at  $T_S$  ( $s_4$ ) and  $T_R$  ( $r_2$ ). The DSP swaps the roles of  $T_R$  and  $T_S$ , and performs index-traversal and leaf-scan on  $T_R$  with  $s_4.a$  as target value. Since both the left (i.e.,  $r_1$ ) and right ( $r_2$ ) boundary records are already present in the  $VO$ , neither operation has any effect. There is no extra leaf-scan on the opposite relation (as in Round 1) because  $s_4$  has no matching partner. In the third round, the DSP switches back to  $T_S$ , and performs index-traversal with target value  $r_2.a$ , ascending to  $Root_S$  and then descending to  $s_{10}$ , the left boundary for  $r_2$ . The digests of the skipped entries in nodes  $B$  and  $F$ , i.e.,  $h_{s_5}, h_{s_6}, h_C$ , are inserted into the  $VO$ . The subsequent leaf-scan appends  $s_{11}$ , and  $s_{12}$ . Similarly to Round 1, a match ( $s_{11}$ ) of the target tuple ( $r_2$ ) is identified. Consequently, another leaf-scan is performed on  $T_R$  (with target  $s_{11}.a=r_2.a$ ) adding  $r_3$  (match) and  $r_4$  (right boundary) to the  $VO$ . Note that if this operation were omitted, the hash value of  $r_3$ , rather than the full record, would be inserted to the  $VO$  during the next round; consequently, the client would miss the result ( $r_3, s_{10}$ ).

The fourth round searches  $T_R$  with target  $s_{12}.a$ . Index-traversal has no effect, since the  $VO$  already contains the left boundary  $r_4$ ; leaf-scan finds no matches, and appends the right boundary  $r_5$ . In the fifth round, the DSP first performs index-traversal on  $T_S$ , with target value  $r_5.a$ , following path  $s_{12}-D-G-E-s_{15}$  and appending  $h_{s_{13}}, h_{s_{14}}$  (digests of skipped tuples) as well as  $s_{15}$  (the left boundary record) to the  $VO$ . Because  $s_{15}$  is the last tuple of  $T_S$ , leaf-scan is skipped. Finally (sixth round), since  $s_{15}.a$  is still smaller than the target value  $r_5.a$ , the DSP simply traverses  $T_R$  back to  $Root_R$ , adding the digest of  $r_6$  to the  $VO$ . This is implemented by an index-traversal with a target value of  $+\infty$ . Figure 9 illustrates AIM. The loop invariance of the algorithm is that, given a target record from one relation, its left and right boundaries (and possibly matching records) are determined in the opposite tree; the right boundary is used as a new target in the next round. Lines 11-13 correspond to the extra leaf-scan operation, performed whenever a matching pair is identified.

The verification process is summarized in Figure 10. The client (i) verifies that the boundary records only enclose matching tuples, (ii) checks whether the additional leaf-scan operations are performed properly by the DSP, (iii) generates join results from the data records contained in the  $VO$ , and (iv) reconstructs  $h_{Root_S}^S$ ,  $h_{Root_R}^R$  of  $T_S$  and  $T_R$ , and verifies them against their respective signatures. Operations (iii) and (iv) jointly ensure *soundness*, while the combination of (i), (ii), (iii) and (iv) guarantees *completeness* of the result. The proof is similar to that of AISM, except that in AIM the client must also verify condition (ii). Specifically, if the DSP cheats by not executing this operation, or

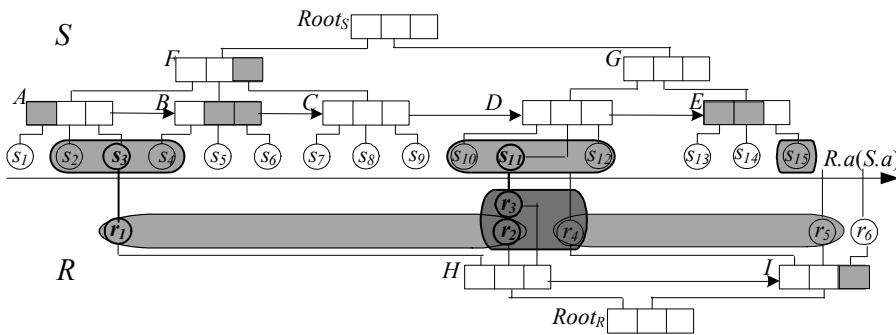


Figure 8 Example of AIM

	$VO$ : root signature of $T_S$ , root signature of $T_R, r_1$
Round 1	$h_{s_1}, s_2, s_3, s_4, r_2;$
Round 2	;
Round 3	$h_{s_5}, h_{s_6}, h_C, s_{10}, s_{11}, s_{12}, r_3, r_4;$
Round 4	$r_5;$
Round 5	$h_{s_{13}}, h_{s_{14}}, s_{15};$
Round 6	$h_{r_6};$

doing it improperly (e.g., inserting a falsified tuple, or a hash value rather than the actual record), then either the check at line 9 fails, or the reconstructed root hashes do not match their corresponding signatures, alarming the client.

---

```

AIM (MBTree  $T_R$ , MBTree  $T_S$ , VO) // DSP
// The join query is  $R \bowtie_{R.a=S.a} S$ 
1. Append the root signatures of  $T_S$  and  $T_R$ 
2. Traverse  $T_R$  from  $Root_R$  to the leftmost node, and insert the first
   record  $r_l$  to the VO
3. Initialize  $n_S = T_S.Root$ ,  $n_R =$  leftmost node of  $T_R$  storing  $r_l$ 
4. Initialize tuples  $t_S = \text{null}$ ,  $t_R = r_l$ 
5. Loop
6.   Call  $IndexTraversal(n_S, t_R.key)$ 
7.   If the traversal of  $T_S$  finishes
8.     Call  $IndexTraversal(n_R, +\infty)$ 
9.     Return
10.  Call  $LeafScan(n_S, t_R.key)$ 
11.  If a matching record of  $t_R$  is encountered in line 10
12.    Append a separator “;” to VO
13.    Call  $LeafScan(n_R, t_R.key)$ 
14.  Set  $t_S$  to the right boundary record corresponding to  $t_R.key$ 
15.  Append a separator “;” to VO
16. Repeat lines 6-15, reversing  $R$  and  $S$ 

```

---

**Figure 9** Algorithm AIM

---

```

Verify_AIM (VO) // Client
1. Read the root signatures of  $T_S$  and  $T_R$  from VO
2. Read tuple  $r$  from VO
3. While VO is not empty
4.   Read from VO until reaching separator “;”
5.   Verify that in the previous step (i) only  $S$  tuples and digests are
     read, (ii) the  $S$  tuples either match  $R$  or are boundary records
     and (iii) no digest is enclosed by boundary records
6.   Use the values read in line 4 to incrementally compute  $h^S_{Root}$ 
7.   If matching records of  $r$  are identified during line 4
8.     Read from VO until reaching separator “;”
9.     Verify that in the previous step (i) only  $R$  tuples are read, (ii)
        these  $R$  tuples only include those with identical join
        attribute values as  $r$ , and one right boundary record
10.    Use the values read in line 8 to incrementally compute  $h^R_{Root}$ 
11.    Generate join results with  $r$ ,  $S$  tuples read in line 4, and  $R$ 
        tuples read in line 8
12.    Set  $s$  to the right boundary record of  $r$ 
13. Repeat lines 4-12, reversing  $R$  and  $S$  (also  $r$  and  $s$ )
14. Verify  $h^S_{Root}$  and  $h^R_{Root}$  against their respective signatures

```

---

**Figure 10** Algorithm Verify\_AIM

AIM improves the performance of AISM. For relation  $S$ , the two algorithms perform similar operations at the DSP as well as the client, and place the same amount of data in the VO. However, for relation  $R$ , AISM requires the DSP to sort all tuples, and the client to verify and re-order them, whereas AIM only incurs one traversal of  $T_R$  for the DSP, and one  $h^R_{Root}$  computation for the client. Regarding the VO size, AISM inserts all  $R$  records, while AIM only adds those with join partners, and boundary records / digests for the remaining ones.

## 5. ASM

If there are no ADSs on the join attribute, the DSP has to return at least the join inputs to the client, so that the latter can establish their correctness. Furthermore, the client has to generate the join output itself, as there is no way to authenticate a join result

received by the DSP. However, instead of the client executing the entire join locally as in NAI, the proposed ASM (for Authenticated Sort-Merge join) alleviates the burden of the client as follows: (i) the DSP performs a sort-merge join, and (ii) generates a VO such that the client can efficiently reconstruct the join output. Specifically, the transmitted VO includes  $VO(R)$ ,  $B_R$ ,  $VO(S)$ ,  $B_S$ , and  $\Omega_{RS}$ .  $B_R$  and  $B_S$  are two bitmaps, and  $\Omega_{RS}$  is a list of integers. The meaning of  $B_R$ ,  $B_S$  and  $\Omega_{RS}$  will be explained soon. For our examples we use  $Q_I = \text{Purchase} \bowtie_{cid} \text{Customer}$  in the database of Figure 1, assuming that the verifiable orders are  $(p_1, p_2, p_3, p_4, p_5)$  and  $(c_1, c_2, c_3, c_4, c_5)$  for *Purchase* and *Customer*, respectively.

Figure 11 illustrates ASM at the DSP side. Initially (lines 1-4), the DSP sorts  $R$  and  $S$  on the join attribute  $a$ , and inserts the result in  $R'$  and  $S'$ . The use of temporary tables is necessary as the original (i.e., verifiable) order is needed when  $R$  is included in the VO. Then, it generates the rank lists  $\Omega_R$  and  $\Omega_S$ . Line 5 corresponds to the merge phase. This process marks every tuple that has matching records in the other relation, and generates  $\Omega_{RS}$ .  $\Omega_{RS}$  combines  $\Omega_R$  and  $\Omega_S$  in a single sorted list on  $a$ . In order to distinguish the two relations, we negate each element of  $\Omega_S$ . If for two records ( $r \in R, s \in S$ ),  $r.a = s.a$ , then the element of  $r$  in  $\Omega_{RS}$  precedes that of  $s$ . Continuing the example, given  $\Omega_{Purchase} = (1, 4, 3, 5, 2)$  and  $\Omega_{Customer} = (1, 2, 3, 4, 5)$ ,  $\Omega_{Pur-Cus} = (1, 4, -1, 3, 5, -2, 2, -3, -4, -5)$ .

---

```

ASM (Relation R, Relation S, VO) // DSP
// The join query is  $R \bowtie_{R.a=S.a} S$ 
1. Create a temporary table  $R'$  with a single column  $a$ 
2. For each record  $r \in R$ , append  $r.a$  to  $R'$ 
3. Repeat lines 1-2 for  $S$ , creating a temporary table  $S'$ 
4. Sort  $R'$  and  $S'$ , and generate rank lists  $\Omega_R$  and  $\Omega_S$ 
5. Merge the sorted  $R'$  and  $S'$ , mark tuples with join partners, and
   generate a combined rank list  $\Omega_{RS}$ 
6. Create a bitmap  $B_R$  of size  $|R|$ 
7.  $\forall 1 \leq i \leq |R|$ , set  $B_R[\Omega_R[i]]$  to 1 if  $R[i]$  has a join partner, and 0
   otherwise
8. Append the signature of  $R$  and  $|R|$  to VO
9. For  $i = 1$  To  $|R|$ , Append  $R[i]$  and  $B_R[i]$  to VO
10. Repeat lines 6-9 for  $S$ 
11. Append  $\Omega_{RS}$  to VO

```

---

**Figure 11** Algorithm ASM

Next, the DSP generates the bitmap  $B_R$  of  $R$ . Recall that the merging phase marks each tuple that can be joined. Let  $j = \Omega_R[i]$  be the rank of a record  $r \in R$  in the verifiable order. If  $r$  is marked,  $B_R[j]$  is set to 1; otherwise ( $r$  has no join partners in  $S$ ),  $B_R[j]$  is set to 0. In the running example:  $B_{Purchase} = (1, 1, 1, 1, 1)$  and  $B_{Customer} = (1, 1, 1, 0, 0)$ , since customers  $c_4$  and  $c_5$  do not appear in *Purchase*.  $B_R$  is inserted into the VO, together with  $VO(R)$ . Specifically,  $VO(R)$  includes the records of  $R$  in the verifiable order, the cardinality of  $|R|$  and the owner's signature. The bitmap  $B_S$  is generated in the same way and appended to the VO, together with  $VO(S)$ . Finally, the DSP adds  $\Omega_{RS}$ , and the entire VO is transmitted to the client.

Upon obtaining the VO, the client computes and verifies the result by applying the algorithm of Figure 12. When a tuple  $r \in R$  is received, the client uses it to incrementally compute the verification information (e.g., digest) required for matching the signature. Then, it checks the bitmap value of  $r$ . If it is 1 (i.e.,  $r$  has join partners in  $S$ ), the entire tuple is stored on the disk. Otherwise, only the join attribute is kept. These bitmaps are

verified later in the subsequent merging step. The same process is repeated for  $S$ . In our example, the *name* and *city* of  $c_4$  and  $c_5$  are deleted, as these customers will not appear in the join result. At this point the client can verify the individual relations. Next, it remains to compute their join result. This is achieved by a merge operation (lines 7-18) based on  $\Omega_{RS}$ , which constitutes the last part of the  $VO$ . Specifically, matching records from the two relations appear *sequentially* in  $\Omega_{RS}$ ; hence, merging reduces to a scan of  $\Omega_{RS}$  and retrieval of the corresponding tuples from the stored files. When multiple records have identical join attributes, they are temporarily stored in a buffer *buf* (line 13), and later examined to produce join results (lines 15-18). Meanwhile, the client verifies the correctness of the bitmaps, i.e., records marked “0” and stored partially must not participate in any join results (line 16). Note that the usage of the bitmap reduces the I/O operations because, for tuples without join partners, only the join attribute is written to, and then read from the disk. This optimization also applies to AISM for handling the non-indexed relation.

---

```

Verify_ASM (VO)                                     // Client
1. Read the signature for R and |R|
2. For i = 1 To |R|
3.   Read next record r ∈ R and a bit mark from VO
4.   Use r to incrementally verify against the signature of R
5.   If mark is 0, store only r.a, otherwise store the entire r
6. Repeat lines 1-5 for S
7. Read an integer j from VO, set t=R[j] if j>0, and t=S[-j] otherwise
8. Initialize buffer buf with only one record t
9. While VO is not empty
10.  Set j'=j and t'=t
11.  Read j from VO, and set t in the same way as in line 7
12.  Verify that t'.a ≤ t.a
13.  If t'.a = t.a, verify that j' < j, and insert t into buf
14.  Else // t'.a < t.a
15.    If buf contains records from both R and S
16.      Verify that all tuples in buf are stored as full tuples
17.      Join tuples in buf to generate results
18.    Remove all records from buf, and insert t into buf

```

---

**Figure 12** Algorithm *Verify\_ASM*

*Proof of soundness:* Suppose that the DSP deceives the client into generating a wrong result  $rs$ . Then either (i)  $r$  does not match  $s$ , or (ii)  $r$  or  $s$  are bogus/alterred. The first case is impossible as the client generates matching pairs locally. Case (ii) is detected by the authenticated information of  $R$  and  $S$ . □

*Proof of completeness:* Let  $rs$  be a valid result of the query. The DSP must transmit the unaltered  $r$  ( $s$ ) to the client, otherwise the checking against the authenticated information of  $R$  ( $S$ ) will fail. Therefore the only possibility for the client to miss  $rs$  is that the DSP provides the wrong rank list  $\Omega_{RS}$ , which is detected in the same way as in AISM. Furthermore, if the DSP cheats in the marking step (i.e., sets the bitmap to 0, although the tuple can be joined), the client will detect it during the joining step, since it keeps the join-attribute values for all records. □

Compared with AISM and AIM, ASM is naturally less efficient as it does not utilize any ADS. This loss of efficiency, however, is compensated by its flexibility, which, as we clarify in the next section, is an important property for authenticating complex queries. Furthermore, as demonstrated in the experiments, ASM significantly outperforms AINL on all aspects, and exhibits clear performance advantages over NAI in terms of the workload of the client.

So far we have focused on equi-joins. Since all proposed algorithms are based on the sort-merge join paradigm, they can be easily applied to *band joins*, whose join predicates are of the form  $|R.a - S.a| \leq b$ . Note that an equi-join is a special case of the band join where  $b = 0$ . For AISM and AIM the main change is that the target of index-traversal for  $\Omega_R[i]$  is the first record  $s$  such that  $\Omega_R[i].a - b \leq s.a \leq \Omega_R[i].a + b$  (instead of  $s.a = \Omega_R[i].a$ ); meanwhile, leaf-scan stops at the last such record. Similar extensions can be applied to ASM. AIM involves one more complication; when matching records are identified during leaf-scan (say, on  $T_S$ ), an additional leaf-scan is performed on  $T_R$  with the largest join attribute value among the matching  $S$  records. If there are  $R$  tuples that match this value, another leaf-scan is executed on  $T_S$ , with the largest  $a$  value among such records, causing a chain of leaf-scans. This chain terminates when no match is found. For *arbitrary* join predicates, the sort-merge join paradigm may no longer work, but other optimizations in the proposed algorithms still apply. Specifically, using ASM, the DSP transmits all records in both relations, identifying those without join partners, so that the client can discard unnecessary attributes. The utilization of MB-trees by AISM and AIM can reduce communication overhead by transmitting all attributes only for tuples with join partners. When it is possible to verify by checking a key range that a sequence of records in a relation do not have join partners, AISM and AIM can further decrease the transmission cost by sending boundary records and digests for them.

## 6. COMPLEX QUERY AUTHENTICATION

In practice, users may pose complicated and descriptive queries involving joins over multiple relations, as well as other operators such as selections and projections. Based on the proposed binary algorithms, Section 6.1 develops optimized solutions for authenticating multi-way joins, and Section 6.2 discusses general selection-projection-join processing.

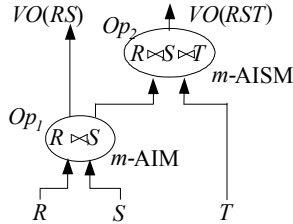
### 6.1 Multi-way join

For ease of presentation we focus on authenticating the join results of 3 relations, and discuss the extension to  $m$ -way ( $m > 3$ ) joins whenever necessary. Consider a query  $Q_{RST} = R \bowtie_{R.a=S.a} S \bowtie_{S.b=T.b} T$ , where  $a$  ( $b$ ) is the join attribute between  $R$  and  $S$  ( $S$  and  $T$ ) respectively<sup>3</sup>. As in traditional query processing, the DSP answers  $Q_{RST}$  through a *plan* of binary join operators. Without loss of generality we assume that all join attributes ( $R.a$ ,  $S.a$ ,  $S.b$ ,  $T.b$ ) are indexed by MB-trees. Figure 13 depicts an example left-deep plan for  $Q_{RST}$  using two binary operators  $Op_1$ ,  $Op_2$ . Given the ADSs on all join attributes,  $Op_1$  employs a variant  $m$ -AIM of AIM, optimized for multi-way joins to be discussed shortly. Since the output of  $Op_1$  is not indexed, a variant  $m$ -AISM of AISM is used for  $Op_2$ . The DSP sends two  $VO$ s to the client:  $VO(RS)$  generated by  $Op_1$  to be used for verification of  $R \bowtie_{R.a=S.a} S$ , and  $VO(RST)$  to authenticate the output of  $Op_2$ . Alternatively, the DSP can answer  $Q_{RST}$  through a right-deep plan that first joins  $S$  with  $T$ , and then  $R$  with  $S \bowtie T$ . Similar to conventional multi-way join processing, the best choice is computed by dynamic programming, or any query optimization method.

The algorithms  $m$ -AISM,  $m$ -ASM and  $m$ -AIM used in multi-way join plans differ from their binary counterparts in two aspects.

<sup>3</sup> If there is also a condition between  $R$  and  $T$ , we model this condition as an additional selection on top of the join, and use the techniques of Section 6.2 to process the query.

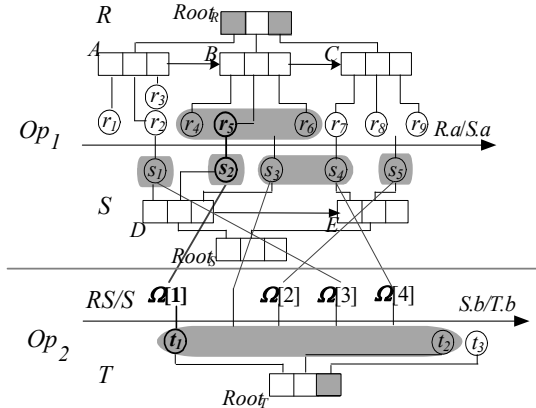
First, recall that AISM and ASM transmit all tuples of the un-indexed relations to the client. In  $m$ -AISM and  $m$ -ASM, when the un-indexed input is the output of another operator, these records are not replicated in the  $VO$ . In Figure 13, since  $Op_1$  sends a separate  $VO$  to the client (based on which the client can derive  $R \bowtie S$ ), it is not necessary for  $Op_2$  to include a redundant copy. Second, a  $VO$  produced by an intermediate operator (e.g.,  $VO(RS)$ ) contains only the information for verifying and reconstructing those partial results that have matching partners in all other relations. For example, in Figure 13,  $VO(RS)$  authenticates an intermediate result  $rs$  consisting of components  $r \in R$  and  $s \in S$ , if and only if there is a tuple  $t \in T$  that matches  $s$ . In other words, intermediate joins are evaluated *incompletely*, and in fact *minimally*, based on the demand of subsequent operators, leading to considerable reduction of the  $VO$  size.



**Figure 13** Authenticated join tree

Figure 14 illustrates an instance of multi-way join using the plan of Figure 13. Before executing the plan, the DSP computes the set  $S_N$  of  $S$  tuples that do not have a join partner in  $T$ . In our example, we assume  $s_2$  matches  $t_1$ , and  $S_N = \{s_1, s_3, s_4, s_5\}$ . Given this information, the DSP starts processing  $R \bowtie S$  using  $m$ -AIM (i.e.,  $Op_1$ ). After inserting the root signature of  $T_R$  and  $T_S$  to  $VO(RS)$ , the DSP follows the left-most path of  $T_S$  (i.e.,  $Root_S-D-s_1$ ) to the first record  $s_1$ , and adds  $s_1$  to  $VO(RS)$ . At this point, the binary AIM algorithm would use  $s_1$  to probe  $T_R$  and retrieve matching tuples  $r_2, r_3$  as well as boundary records  $r_1, r_4$ . This, however, is unnecessary since  $S_N$  suggests that  $s_1$  does not match any  $T$  tuple, meaning that all intermediate results generated by joining  $s_1$  and  $R$  records are useless. Therefore,  $m$ -AIM (similarly,  $m$ -AISM) requires the DSP to skip traversing  $T_R$  whenever it encounters a record in  $S_N$ , eliminating intermediate results not demanded by subsequent operators.

$VO(RS)$ : {root signature of  $T_R$  and  $T_S$ ,  $s_1, s_2$ ;  $h_A, r_4, r_5, r_6, s_3, s_4, s_5$ ;  $h_C$ }  
 $VO(RST)$ : {root signature of  $T_T$ ,  $\mathcal{A}[1], t_1, t_2$ ;  $\mathcal{A}[2]; \mathcal{A}[3]; \mathcal{A}[4]$ ;  $h_3$ }



**Figure 14** Example of  $m$ -AIM and  $m$ -AISM

Continuing the example, the DSP skips traversing  $T_R$  for  $s_1$ , and retrieves the second record  $s_2 \in T_S$ , appending it to  $VO(RS)$ . Since  $s_2 \notin S_N$ , the DSP switches to  $T_R$ , inserting matching tuple  $r_5$ , boundary record  $r_4, r_6$ , and digest  $h_A$  of node  $A$  to  $VO(RS)$ . Then, it traverses  $T_S$  with new target  $r_6, a$ , adding boundary records  $s_3$  and  $s_4$  to  $VO(RS)$ . Because  $s_4$  is in  $S_N$ , the DSP suspends operations on  $T_R$  and arrives at  $s_5$ , which is also in  $S_N$  and does not initiate  $T_R$  traversal. Since  $s_5$  is the last record of  $S$ , the DSP terminates  $m$ -AIM by adding digests of all right siblings in  $T_R$  (i.e.,  $h_C$ ) to  $VO(RS)$ . In total,  $VO(RS)$  contains only 3  $R$  records, compared to all 9 records if the DSP were applying AIM. Next the DSP executes  $Op_2$ . In addition to authenticating the join results,  $VO(RST)$  must establish that the records in  $S_N$  for which  $Op_1$  skipped traversing  $T_R$  ( $s_1, s_4$  and  $s_5$ ) indeed have no matching  $T$  tuples. For this purpose, the input of  $Op_2$  contains both the join results of  $Op_1$  (i.e., tuple  $r_5s_2$ ) and  $s_1, s_4, s_5$ .  $Op_2$  sorts the four tuples and traverses  $T_T$  to produce  $VO(RST)$ , shown in Figure 14.

Upon receiving  $VO(RS)$ , the client rebuilds the root digests of  $T_R, T_S$  and verifies them against their respective signatures. It also checks that boundary records enclose matching tuples. From  $VO(RS)$ , the client extracts  $S$  records whose corresponding  $T_R$  traversals are skipped (i.e.,  $s_1, s_4, s_5$ ), and generates the (partial) join result  $r_5s_2$  of  $R \bowtie S$ . When it later receives  $VO(RST)$ , it prefixes the tuples computed in the previous step (i.e.,  $s_1, r_5s_2, s_4, s_5$ ) and verifies  $VO(RST)$  as in AISM. The proofs for soundness and completeness of the multi-way algorithms are analogous to those of the binary methods.

The above solution incurs some computational overhead on the DSP and the client. Specifically, the DSP has to generate  $S_N$ , and the client must produce some intermediate results. Considering that network transmission is usually the bottleneck, this tradeoff is desirable. Nevertheless, the additional cost can be eliminated when the join condition satisfies the property that  $a=b$ , i.e., all tables join on the same attribute. This situation is common in data warehousing applications, where multiple dimension tables are joined with the fact table on the record ID attribute [21] rather than through foreign keys. For such queries, we propose AST (for *authenticated synchronous traversal*), which traverses the ADSs of indexed relations in a synchronous fashion, avoiding unnecessary intermediate results.

Figure 15 illustrates AST for  $R \bowtie_{R.a=S.a} S \bowtie_{S.a=T.a} T$ , assuming two MB-trees  $T_R, T_T$  on  $R.a$  and  $T.a$ , respectively ( $S.a$  does not have an ADS). Initially, the owner's signatures for  $T_R, T_T$  and  $S$ , as well as the tuples  $s_1-s_4 \in S$  are appended to the  $VO$ . The DSP computes (i) a bitmap  $B_S$  specifying the  $S$  tuples with join partners in both  $R$  and  $T$ , which reduces the client's storage consumption, and (ii) a rank list  $\Omega_S$  signifying the sort order of  $S$  tuples on attribute  $a$ . Assuming the *join order*<sup>4</sup> is  $(R \bowtie S) \bowtie T$ , AST interleaves the computation of  $R \bowtie S$  with that of  $(R \bowtie S) \bowtie T$  to eliminate the generation of unnecessary  $R \bowtie S$  results. Specifically, the DSP starts by retrieving the first element  $\Omega_S[1]$  from  $\Omega_S$ , appending it to the  $VO$ , and traversing  $T_R$  with  $\Omega_S[1]$  as target value. During the traversal, the digest of  $r_1$ , the join partner  $r_3$ , as well as the boundary records  $r_2$  and  $r_4$  are inserted into the  $VO$ . Because a matching pair  $(\Omega_S[1], r_3)$  is identified, the DSP switches to  $T_T$ , adding matching tuple  $t_1$  and boundary record  $t_2$  to the  $VO$ . After that, the DSP continues joining  $R$  and  $S$ . Note that

<sup>4</sup> More sophisticated optimizations such as round-robin [21] and dynamic re-ordering [4] are also applicable to AST.



at this point the DSP is aware that there are no  $T$  records with  $a$  values between  $t_1.a$  and  $t_2.a$ . Based on this, the DSP scans  $\Omega_S$ , and appends to the  $VO$  each tuple  $s \in S$  such that  $s.a \leq t_2.a$  (without trying to find a join partner of  $s$  in  $T_R$ ). For instance, although  $\Omega_S[2]$ ,  $\Omega_S[3]$  and  $\Omega_S[3]$  match  $r_4$ ,  $r_8$  and  $r_9$ , respectively, none of these results is inserted to the  $VO$ . Finally, when  $\Omega_S$  is depleted, the DSP completes the traversal of  $T_R$  and  $T_T$  by adding to the  $VO$  the digests of all right sibling nodes, i.e., those of  $r_5$ ,  $r_6$ , node  $C$  (for  $T_R$ ) and  $t_3$  (for  $T_T$ ). The verification algorithm and soundness/completeness proofs are similar to those of AISM and AIM and omitted for brevity.

$VO$ : {root signature of  $T_R$  and  $T_T$ , signature of relation  $S$ ,  
 bitmap  $B_S = \text{"1000"}$ ,  $s_1$ - $s_j$  in a verifiable order,  
 $\Omega_S[1], h_{r_1}, r_2, r_3, r_4, t_1, t_2; \Omega_S[2]; \Omega_S[3]; \Omega_S[4]; h_{r_5}, h_{r_6}, h_C, h_{t_3}$ }

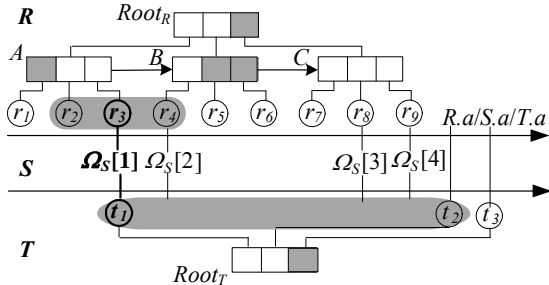


Figure 15 Example of AST

## 6.2 Selection-projection-join query

In addition to joins, complex queries often contain selection and projection components. For projections, we follow the general methodology of [17], i.e., we build a Merkle hash tree  $MHT(t)$  on each tuple  $t$  that indexes all its attributes. The root hash of  $MHT(t)$  serves as the digest of  $t$  in the ADSs. Consider that in Figure 1, there is an MHT on each of  $p_1$ - $p_3$  ( $c_1$ - $c_3$ ) indexing their  $pid$ ,  $cid$  and  $quantity$  ( $cid$ ,  $name$  and  $city$ ) attributes. Assuming MB-trees on both  $Purchase.cid$  and  $Customer.cid$ , the query  $Q_5 = \pi_{name, quantity} (Purchase \bowtie_{cid} Customer)$  is processed as follows. The DSP uses AIM to compute the join, with two modifications: (i) for each  $Purchase$  ( $Customer$ ) record with matching  $Customer$  ( $Purchase$ ) tuples, its  $cid$ ,  $quantity$  ( $name$ ) attributes, as well as the hash of the remaining  $pid$  ( $city$ ) attribute are inserted to the  $VO$  (instead of the entire tuple); (ii) for each boundary record, its  $cid$  value and the hash of other attributes are added to the  $VO$ . Using these values, the client rebuilds the root digest of the MHT of each tuple, and subsequently the root hash of the MB-trees.

Next, we address queries involving both selection and join operators. Similar to multi-way joins, we distinguish two cases: (i) the selection applies on the join attribute, and (ii) the selection is on a different column. For case (i), when ADSs are available on the selection/join attribute, the DSP answers the query with a single traversal of the ADS, combining AISM (or AIM, AST) with the range selection. For example, consider that in the join of Figure 5, the user imposes a further selection  $\sigma_{a < C}$  where  $C$  is a constant between  $r_4.a$  and  $r_5.a$ . After round 1 of AISM execution, the DSP finds that  $\Omega_R[2]$  already violates the range selection, and thus it simply inserts  $\Omega_R[2]$  to  $\Omega_R[6]$  directly into the  $VO$  without traversing  $T_S$  to identify their matching  $S$  tuples. Instead, it simply appends digests of the right siblings of the root-to-leaf path of  $s_4$  to complete the traversal of  $T_S$ . For case (ii), the selection and join operators compete for the use of ADSs. An example is the

complex query  $Q_4$  in Figure 1, which involves a join and two selections  $\sigma_1 = \sigma_{quantity > 100} Purchase$ ,  $\sigma_2 = \sigma_{city = \text{"New York"}} Customer$ . Assuming an ADS is available on each of the 4 attributes ( $cid$ ,  $quantity$  in  $Purchase$  and  $cid$ ,  $city$  in  $Customer$ ), an important observation is that for a particular relation (e.g.,  $Purchase$ ), only one ADS (e.g., either the one on  $cid$  or the other on  $quantity$ ) can be utilized to answer the query. Figure 16 illustrates three different plans for  $Q_4$ . In the first plan (Figure 16a), the join is able to utilize the ADSs on both  $Purchase.cid$  and  $Customer.cid$  with algorithm AIM, achieving maximal performance. However, since its output is not indexed, the DSP must transmit all join results to the client, and the latter performs the selections (i.e.,  $\sigma_1$  and  $\sigma_2$ ) locally. On the other hand, the plan in Figure 16c processes  $\sigma_1$  and  $\sigma_2$  with the corresponding ADSs, and feeds their (non-indexed) outputs to the join, which must be performed with the less efficient ASM algorithm. The plan in Figure 16b lies between the other two, in which the join is able to use only one ADS. In such situations, the optimal plan depends on the relative selectivity and the cost of the operators. Finally, when there are selections on top of the join (e.g., plans in Figure 16a and Figure 16b), a further optimization is to use a variant of AIM (AISM) similar to  $m$ -AISM ( $m$ -AIM), which only evaluates the join partially for records satisfying the subsequent selections.

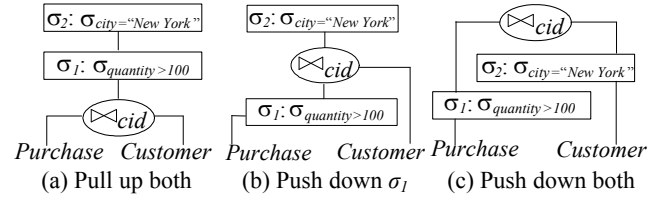


Figure 16 Three query plans for  $Q_4$

## 7. EXPERIMENTAL EVALUATION

We implemented all algorithms using the OpenSSL library [20], and executed all experiments on an Intel Core2 Duo 2.13GHz CPU. The MB-tree implementation is based on B+-trees with 4Kbytes page size. Section 7.1 compares the performance of the proposed algorithms against AINL and NAI. Section 7.2 evaluates multi-way join queries. Our prototype employs SHA1 with a digest of 20 bytes as the hash function, and RSA+SHA1 with a signature of 128 bytes as the digital signature scheme, both of which are widely used in practice [15].

### 7.1 Evaluation of Binary Joins

We use two synthetic relations  $R$  and  $S$ , each containing two independent attributes  $a_1$  and  $a_2$ . The values of  $S.a_1$  are uniformly distributed in the range  $[1, 10^7]$ , whereas  $R.a_1$ ,  $R.a_2$  and  $S.a_2$  follow Gaussian distribution (with mean  $5 \cdot 10^6$ ,  $3 \cdot 10^6$  and  $7 \cdot 10^6$  respectively), and share the same variance  $\sigma = 10^6$ . In addition,  $R.a_1$  is a foreign key that references  $S.a_1$ , which is the primary key of  $S$ . We construct an MB-tree on each of the four attributes ( $R.a_1$ ,  $R.a_2$ ,  $S.a_1$ ,  $S.a_2$ ), before query processing. We investigate two queries: (i) a foreign-key to primary-key join  $R \bowtie_{a_1} S$  denoted as FK, and (ii) an equi-join  $R \bowtie_{a_2} S$  denoted as EQ. In all experiments,  $R$  is the outer relation and  $S$  the inner one.

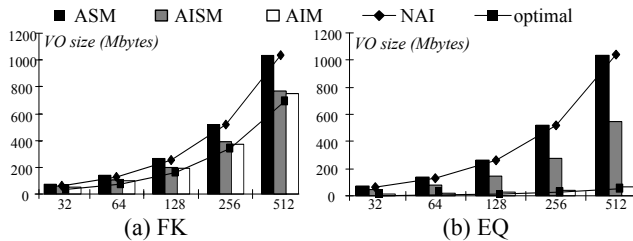
We compare the authenticated join algorithms on the following metrics: (i)  $VO$  size  $C_{VO}$ , (ii) total computation overhead  $C_{Client}$  of the client and (iii) query processing cost  $C_{DSP}$  of the DSP. In the first set of experiments, we fix the cardinality of both  $R$  and  $S$  to  $10^6$  records, and study the impact of the record

size, which is equal in  $R$  and  $S$ . In this setting, AINL requires the DSP to answer  $10^6$  range queries. Table 1 summarizes the AINL results for the FK query because its cost is too high to be included in the diagrams (the numbers for the EQ query are similar and omitted). The most serious drawback of AINL is the enormous  $VO$  size. In particular, when the tuple size is 32 bytes, AINL requires the DSP to transmit 8.9Gbytes to the client, while the entire database consumes merely 64Mbytes. This is due to the huge number (several hundreds) of hash values that are sent with each range query result, which dominate the  $VO$  size (note that the  $VO$  size is almost insensitive to the record size). In addition, the computation costs for both the DSP and the client are in the order of hundreds of seconds, which are significantly higher than those of the proposed methods.

**Table 1** Cost of AINL vs. tuple size

Tuple size (bytes)	32	64	128	256	512
$C_{VO}$ (Gbytes)	8.9	9.0	9.2	9.6	10.3
$C_{Client}$ (seconds)	205	207	210	214	219
$C_{DSP}$ (seconds)	262	271	429	1728	4603

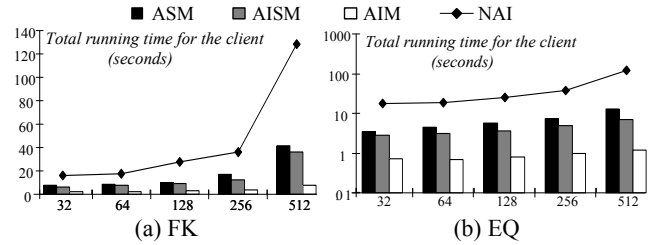
Figure 17 shows  $VO$  size as a function of the tuple length for NAI, ASM, AISM and AIM, as well as the theoretically optimal  $VO$  size, which is the total size of tuples from  $R$  and  $S$  that are part of the join results (i.e., those with matching partners in the opposite relation). The  $VO$  of all methods increases linearly with the tuple length, and is always far below that of AINL. NAI and ASM incur similar overhead because they both transmit the entire relations to the client. The  $VO$  of AISM and AIM depends on the selectivity of the join. For the FK dataset, because of the foreign-key / primary-key constraint, every tuple in  $R$  has a matching partner in  $S$ . Therefore, any distributed join algorithm (i.e., with or without authentication) has to transmit the entire relation of  $R$  to the client. On the other hand, some tuples of  $S$  do not participate in the join, and their transmission is avoided by AISM and AIM. For the EQ dataset, both  $R$  and  $S$  contain unmatched records. AIM achieves the lowest  $VO$  size by excluding such tuples from both tables, followed by AISM which eliminates unnecessary transmissions in  $S$ . The  $VO$  of ASM and NAI includes all tuples. Note that in terms of absolute values, the network overhead for large tuples is quite high, especially for the FK dataset. However, this is unavoidable because of the join cardinality. As shown in the figure, the  $VO$  size of AIM is close to optimal. In practice, joining large tables with foreign-key relationship rarely occurs without projections or selections, which in effect reduces the tuple size, or the cardinality of the relations respectively.



**Figure 17**  $VO$  size vs. tuple size

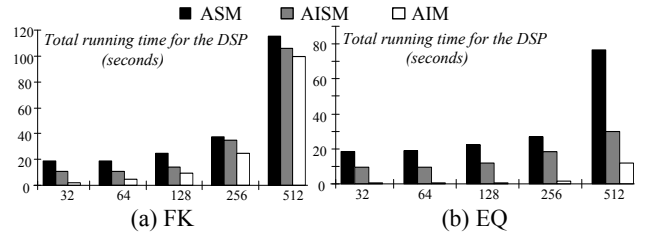
Figure 18 compares the methods in terms of the client's workload. Our methods outperform NAI by more than an order of magnitude (note the logarithmic scale for the EQ dataset). Comparing the proposed algorithms, AIM is the clear winner because the client

computes the digests of the two MB-trees' root nodes, and verifies them against their respective signatures in main memory. ASM and AISM, on the other hand, require the client to buffer records on the disk, and retrieve them in a random order. Subsequently, their cost grows much faster than AIM with the tuple size. Note that ASM has significantly better performance on EQ than FK. Recall that for tuples of  $R$  without join partners, the client stores and accesses only the join attribute. Since the join result of EQ contains only  $31 \cdot 10^3$  tuples, a large portion of  $R$  is pruned reducing the clients' overhead.



**Figure 18** Query cost for the client vs. tuple size

Figure 19 studies the effect of tuple length on the query processing time of the DSP. Since in NAI the DSP does not perform any computations, it is excluded from the diagrams. Comparing Figure 18 and Figure 19, most of the workload is performed by the DSP, which is desirable given that in practice the DSP has much more powerful hardware than the client. Again, the costs of all methods are well below that of AINL. ASM always requires the DSP to sort all tuples of the two relations on the join attributes, and its overhead is similar in both charts. AIM incurs the lowest cost since it does not perform expensive sorting operations. It is cheaper for the EQ dataset because, due to the high selectivity, the ADSs prune the majority of records. The overhead of AISM always lies between the other two.



**Figure 19** Query cost for the DSP vs. tuple size

Next we fix the record size to 128 bytes,  $|R|$  to  $10^6$ , and vary the cardinality of  $S$  from  $|R|/10$  to  $5 \cdot |R|$ . This has a similar effect as applying an additional selection (with variable selectivity) on the inner table. The results of AINL on FK are summarized in Table 2. Because AINL answers  $|R|$  range queries, each of which traverses the MB-tree in  $S$ , its cost grows logarithmically as  $|S|$  increases. In fact, as we discuss in Section 2.2, the situation where the outer relation ( $R$ ) is much smaller than the inner one ( $S$ ) is most favorable for AINL. Nevertheless, in all settings, AINL incurs significantly higher overhead than other algorithms on all metrics, and is excluded from further discussion.

**Table 2** Cost of AINL vs.  $|S|/|R|$

$ R  /  S $	0.1	0.5	1	2	5
$C_{VO}$ (Gbytes)	7.8	8.9	9.2	9.5	9.7
$C_{Client}$ (seconds)	196	205	210	218	223
$C_{DSP}$ (seconds)	296	311	429	540	647

Figure 20 displays the  $VO$  sizes for NAI, ASM, AISM and AIM. In case of FK, the entire relation of  $R$  has to be included in the  $VO$  due to the foreign-key constraint, which contributes a constant overhead to all methods. For relation  $S$ , NAI and ASM transmit all its tuples to the client, while AISM and AIM utilize the ADS to prune most unmatched ones. Accordingly, the  $VO$  size of the former two increases linearly with  $|S|$ , while that of the latter two scales much better. For the EQ dataset, the performance gap between AISM/AIM and NAI/ASM is wider (up to an order of magnitude) due to higher selectivity of the join. Similar to Figure 17, the  $VO$  size of AIM is close to the optimal one.

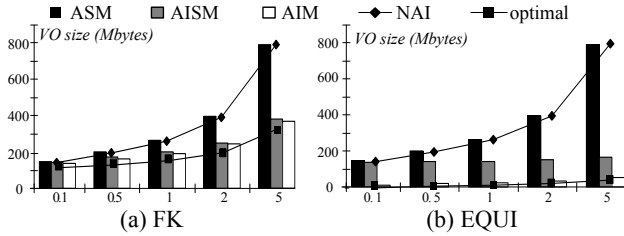


Figure 20  $VO$  size vs.  $|S|/|R|$

Figure 21 evaluates the effect of  $|S|/|R|$  on the running time of the client. The cost of both ASM and NAI increases linearly with  $|S|$ , with the former always well below the latter. AISM and AIM incur even lower workload for the client, and exhibit better scalability with  $|S|$ . AIM is always the best choice since it only involves main memory (hash and verification) operations. Finally, Figure 22 illustrates the processing cost at the DSP as a function of  $|S|/|R|$  (excluding NAI since it does not involve DSP computations). The proposed algorithms successfully shift most of the computation to the DSP. The ratio  $|S|/|R|$  has similar impact on the overhead for both the DSP and the client.

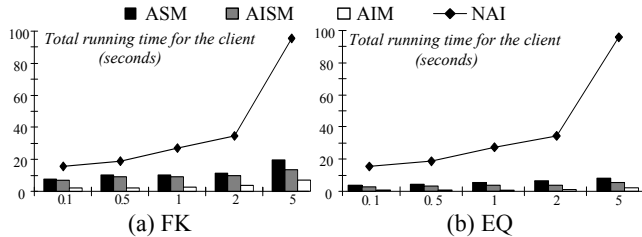


Figure 21 Query cost the client vs.  $|S|/|R|$

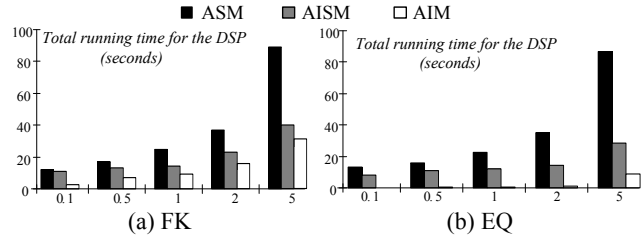


Figure 22 Query cost for the DSP vs.  $|S|/|R|$

Summarizing, AISM, AIM and ASM outperform AINL by orders of magnitude on every performance metric. Compared to NAI, the proposed methods incur significantly lower cost for the client. In addition, AISM and AIM utilize ADSs to considerably reduce the  $VO$  size. Overall, AIM is the most efficient solution on all aspects, followed by AISM, and then ASM. The performance gap between different algorithms widens with the join selectivity (as in the EQ dataset).

## 7.2 Evaluation of Multi-way Joins

We generate three relations  $R$ ,  $S$  and  $T$ , involving attributes  $R.a_1$ ,  $R.a_2$ ,  $S.a_1$ ,  $S.a_2$ ,  $S.b_1$ ,  $S.b_2$ ,  $T.b_1$ ,  $T.b_2$ . Attributes  $R.a_1$  and  $T.b_1$  are uniformly distributed in  $[1, 10^7]$ .  $S.a_1$  and  $S.b_1$  follow Gaussian distribution with mean value  $5 \cdot 10^6$  and variance  $10^6$ .  $R.a_2$ ,  $S.a_2$ ,  $S.b_2$  and  $T.b_2$  follow Gaussian distribution with variance  $10^6$  and mean values  $4 \cdot 10^6$  (for  $R.a_2$ ,  $S.b_2$ ) and  $7 \cdot 10^6$  ( $S.a_2$ ,  $T.b_2$ ). In addition,  $R.a_1$  and  $T.b_1$  are the primary keys of tables  $R$  and  $T$ , and  $S.a_1$ ,  $S.b_1$  are foreign key referencing  $R.a_1$  and  $T.b_1$ , respectively. An MB-tree is constructed for each attribute. We evaluate two queries: (i)  $R \bowtie_{a_1} S \bowtie_{b_1} T$ , which is a primary-key to foreign-key join, denoted as FK, and (ii)  $R \bowtie_{a_2} S \bowtie_{b_2} T$ , denoted as EQ. For all settings, the join plan is always left-deep, i.e.,  $(R \bowtie S) \bowtie T$ . We compare NAI against three different combinations of fully optimized join algorithms, namely  $m$ -ASM+ $m$ -ASM,  $m$ -AISM+ $m$ -AISM, and  $m$ -AIM+ $m$ -AISM, where  $X+Y$  means that the first operator joining  $R$  with  $S$  adopts algorithm  $X$  and the second one uses  $Y$ . Note that “ $m$ -AIM+ $m$ -AIM” is not applicable since the results of  $R \bowtie S$ , which is not indexed, feed to the second join operator. AINL is excluded due to its prohibitive cost.

We first fix the cardinality of  $R$ ,  $S$  and  $T$  to  $5 \cdot 10^5$ , and vary the record length. Figures 23-25 demonstrate the  $VO$  size, verification overhead at the client and the processing cost at the DSP, respectively. In general, the efficiency of a multi-way join method depends on its underlying binary join algorithms, which means that (i) all methods outperform NAI in terms of the client’s workload, and (ii) the two solutions utilizing ADSs, i.e.,  $m$ -AIM+ $m$ -AISM and  $m$ -AISM+ $m$ -AISM achieve considerable savings in terms of  $VO$  size.  $m$ -AIM+ $m$ -AISM has the best overall performance, followed by  $m$ -AISM+ $m$ -AISM, and finally  $m$ -ASM+ $m$ -ASM. A major difference between the results for multi-way and binary joins regards the processing cost at the client. Specifically, in binary AIM client verification occurs entirely in main memory and thus it is very fast in both datasets. For multi-way joins, the client has to use disk accesses for reading and writing the intermediate results of  $R \bowtie S$ . Consequently, in the FK dataset, the advantage of  $m$ -AIM+ $m$ -AISM over  $m$ -AISM+ $m$ -AISM is limited since the number of these intermediate results is large due to the foreign-key constraint.

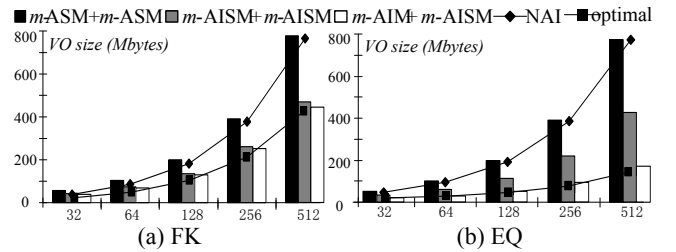


Figure 23  $VO$  size vs. tuple size

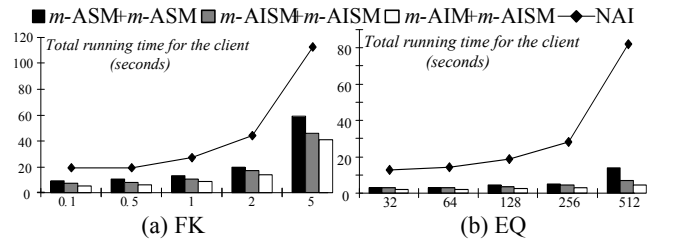


Figure 24 Query cost for the client vs. tuple size

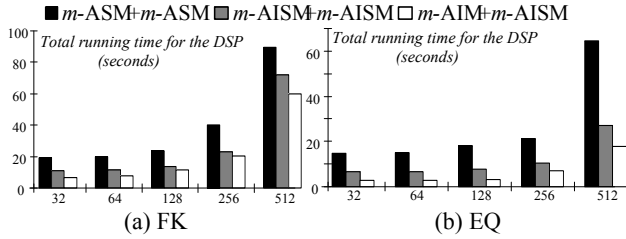


Figure 25 Query cost for the DSP vs. tuple size

Finally, we fix the record size to 128 bytes,  $|S|$  to  $5 \cdot 10^5$ , and vary the cardinality of  $|R|$  and  $|T|$  maintaining  $|T|=|R|$ . Figures 26-28 display the results for the  $VO$  size, query processing cost and verification overhead, respectively. Once again, the proposed methods significantly alleviate the burden of the client compared to NAI, and the use of ADSs leads to considerable  $VO$  reduction. As the sizes of  $|R|$  and  $|T|$  grow, the impact of their MB-trees becomes more pronounced, widening the performance gap among different solutions.

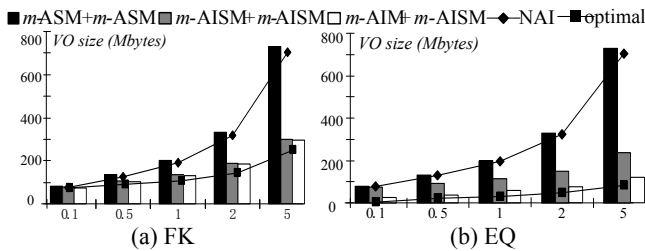


Figure 26  $VO$  size vs.  $|R|/|S|$

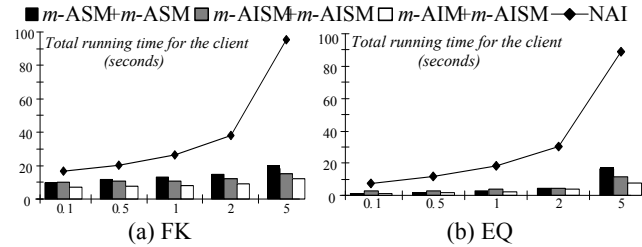


Figure 27 Query cost for the client vs.  $|R|/|S|$

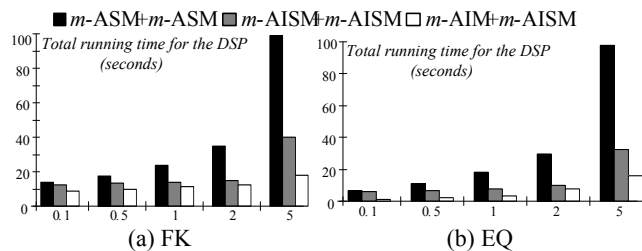


Figure 28 Query cost for the DSP vs.  $|R|/|S|$

## 8. CONCLUSION

This paper constitutes the first comprehensive work on authenticated join processing in outsourced databases. Compared to range queries, authenticated joins are inherently more complex and expensive. We propose three algorithms based on the sort-merge paradigm, which cover the entire spectrum of index availability and possible plans. We show that our techniques outperform two benchmark algorithms, and are truly effective in terms of minimizing the transmission cost as well as the client's

workload. Finally, we deal with complex queries involving joins over multiple tables and, possibly, selections and projections.

## ACKNOWLEDGEMENTS

This work was supported from grant 6181/08 from Hong Kong RGC.

## REFERENCES

- [1] Anciaux, N., Benzine, M., Bouganim, L., Pucheral, P., Shasha, D. GhostDB: Querying Visible and Hidden Data Without Leaks. *SIGMOD*, 2007.
- [2] Atallah, M., Cho, Y., Kundu, A. Efficient Data Authentication in an Environment of Untrusted Third-Party Distributors. *ICDE*, 2008.
- [3] Agrawal, R., Kiernan, J., Srikant, R., Xu, Y. Order Preserving Encryption for Numeric Data. *SIGMOD*, 2004.
- [4] Babu, S., Bizarro, P., DeWitt, D. Proactive Re-Optimization. *SIGMOD*, 2005.
- [5] Devanbu, P., Gertz, M., Martel, C., Stubblebine, S. Authentic Third-party Data Publication. *DBSec*, 2000.
- [6] Damiani, E., Vimercati, C., Jajodia, S., Paraboschi, S., Samarati, P. Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. *CCS*, 2003.
- [7] Huebsch, R., Hellerstein, J., Lanham, N., Loo, B., Shenker, S., Stoica, I. Querying the Internet with PIER. *VLDB*, 2003.
- [8] Hacigümüş, H., Iyer, B., Mehrotra, S. Providing Databases as a Service. *ICDE*, 2002.
- [9] Kundu, A., Bertino, E. Structural Signatures for Tree Data Structures. *VLDB*, 2008.
- [10] Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L. Dynamic Authenticated Index Structures for Outsourced Databases. *SIGMOD*, 2006.
- [11] Luo, Q., Krishnamurthy, S., Mohan, C., Pirahesh, H., Woo, H., Lindsay, B., Naughton, J. F. Middle-Tier Database Caching for e-Business. *SIGMOD*, 2002.
- [12] Li, F., Yi, K., Hadjieleftheriou, M., Kollios, G. Proof-Infused Streams: Enabling Authentication of Sliding Window Queries on Streams. *VLDB*, 2007.
- [13] Merkle, R. A Certified Digital Signature. *CRYPTO*, 1989.
- [14] Martel, C., Nuckolls, G., Devanbu, P., Gertz, M., Kwong, A., Stubblebine, S. A General Model for Authenticated Data Structures. *Algorithmica*, 39(1): 21-41, 2004.
- [15] Menezes, A., van Oorschot, P., Vanstone, S. *Handbook of Applied Cryptography*. CRC Press, 1996.
- [16] Narasimha M., Tsudik G. Authentication of Outsourced Databases Using Signature Aggregation and Chaining. *DASFAA*, 2006.
- [17] Pang, H., Jain, A., Ramamritham, K., Tan, K.-L. Verifying Completeness of Relational Query Results in Data Publishing. *SIGMOD*, 2005.
- [18] Pang, H., Tan, K.-L. Authenticating Query Results in Edge Computing. *ICDE*, 2004.
- [19] Papadopoulos, S., Yang, Y., Papadias, D. CADs: Continuous Authentication on Data Streams. *VLDB*, 2007.
- [20] <http://www.openssl.org>
- [21] Raman, V., Qiao, L., Han, W., Narang, I. S., Chen, Y. L., Yang, K. H., Ling, F. L. Lazy, Adaptive RID-List Intersection, and Its Application to Index Anding. *SIGMOD*, 2007.
- [22] Sion, R. Query Execution Assurance for Outsourced Databases. *VLDB*, 2005.
- [23] Xie, M., Wang, H., Yin, J., Meng, X. Integrity Auditing of Outsourced Data. *VLDB*, 2007.
- [24] Yang, Y., Papadopoulos, S., Papadias, D., Kollios, G. Spatial Outsourcing for Location-based Services. *ICDE*, 2008.