Authenticated Join Processing in Outsourced Databases

Yin Yang 1, Dimitris Papadakis 1, Stavros Papadopoulos 1, Panos Kalnis 2

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

2 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 \( Q_0 = \sigma_{\text{quantity}>100} \text{Purchase} \), the correct result set is \( RS_1 = \{ <p_4, c_1, 200>, <p_5, c_2, 500> \} \) and \( RS_2 = \{ <p_4, c_1, 200>, <p_5, c_2, 555> \} \) are not sound because they either contain fake \((p_5, c_1, 200)\), or altered \((p_5, c_5, 200)\) records. \( RS_3 = \{ <p_4, c_1, 200> \} \) 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.

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 $s_3s_2s_1$ of three records. A signature $s(m)$ cannot be used to certify $s_3s_2s_1$, or any order other than $s_3s_2s_1$.

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 fixed order of $s_1|s_2|s_3$ of three records. A signature $s(m)$ cannot be used to certify $s_3s_2s_1$, or any order other than $s_3s_2s_1$.

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 $Q_5 = \sigma_{quantity>100} (Purchase ~ v~ \land ~ Customer)$ can be materialized in a view $V_5$. If an ADS is maintained on $V_5$, the DSP can answer $Q_6 = \sigma_{quantity=100} (Purchase ~ v~ \land ~ Customer)$ by transforming it to $Q_7 = \sigma_{quantity=100} (V_5)$. 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 $T_5$ (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 $R ~ \times_{a=s} S$. Initially, the signature of $R$, the cardinality $|R|$ of $R$ and the signature of $T_5$ 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 $T_5$ (we use the term range to also denote equality conditions). As discussed in Section 2.1 (see Figure 2), the output of this

![Figure 2 Example of Merkle B-tree](image)
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 \( \text{round} \)). The processing terminates when all records of \( R \) are exhausted, and the DSP transmits the \( VO \) to the client.

\[
\text{AINL (Relation } R, \text{ MBTree } T_s, \text{ VO) // DSP}
\]

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 \( VO \)

\[
\text{Figure 3 Algorithm AINL}
\]

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.

\[
\text{Verify AINL (VO) // Client}
\]

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.
7. Generate a join result combining \( r \) and \( s \)

\[
\text{Figure 4 Algorithm Verify_AINL}
\]

For instance, consider \( Q_1 = \text{Purchase} \bowtie_{=1} \text{Customer} \) in Figure 1, with \( \text{Purchase} \) as the outer relation. Initially, the \( VO \) contains the signature of \( \text{Purchase} \), its cardinality \( 5 \) and the signature of \( T_{Customer} \). Let the verifiable order of \( \text{Purchase} \) be \( (p_1, p_2, p_3, p_4, p_5) \). The DSP performs a range query on \( T_{Customer} \) to find the matching customers \( c_i \) of \( p_i \). 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 \( VO \): \( p_2, c_3, c_4, c_5 \) and the necessary digests. In total, there are 5 rounds, each of which corresponds to a tuple in \( \text{Purchase} \). The \( VO \) contains 5 \( \text{Purchase} \) tuples and 13 records from \( \text{Customer} \).

3. AISM

Similar to AINL, our first algorithm AISM (for Authenticated Indexed Sort-Merge join) utilizes an \( \text{ADS} \) for the inner relation. We demonstrate the basic idea of AISM using \( Q_1 = \text{Purchase} \bowtie_{=1} \text{Customer} \), and assuming that the DSP maintains a MB-tree \( T_{Customer} \) on \( \text{Customer.cid} \). In a pre-processing step, the DSP sorts the outer table \( \text{Purchase} \) on the join attribute \( \text{cid} \), and generates the \( \text{rank list} Q_{\text{Purchase}} \). 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 \( Q_{\text{Purchase}} = (1, 4, 3, 5, 2) \), meaning that \( p_1 \) has the smallest value (\( c_i \)) on \( \text{cid} \), \( p_2 \) 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 (\( \text{Purchase} \)) to the client in their verifiable order, along with the owner's signature and \( Q_{\text{Purchase}} \). Next, the DSP turns to the inner relation \( \text{Customer} \). Observe from Figure 1 that all purchases involve clients \( c_1, c_2, c_3, c_4 \). Therefore, it is possible to find all matching customers for these purchases by a single range \( Q = s_{cid=\text{Customer}} \) on \( T_{Customer} \). Due to the nature of the MB-tree, the results of \( Q \) (i.e., customers with \( \text{cid} = c_1, c_2, \ldots, c_5 \)) are sorted on \( \text{cid} \) and can be authenticated. Meanwhile, the rank list \( Q_{\text{Purchase}} \) explicitly specifies the order of purchases when sorted on \( \text{cid} \). Therefore, if the client obtains \( Q_{\text{Purchase}} \), \( Q_{\text{Purchase}} \), and the results of \( Q = s_{cid=\text{Customer}} \), it can generate and authenticate the result of \( \text{Purchase} \bowtie_{=1} \text{Customer} \), by merging the output of \( Q \) with \( Q_{\text{Purchase}} \).

In general, given a query \( R \bowtie_{a} S \) and a MB-tree \( T_s \) on \( S.a \), the DSP processes a single \( \text{multi-range} \) \( Q \) with one traversal of \( T_s \), and merges its \( VO \), denoted as \( VO(Q) \), with the rank list \( Q \). \( Q_{\text{Purchase}} \) signifies that the \( i \)-th element of \( Q \) corresponds to the \( i \)-th record in the verifiable order. For ease of presentation, we also use \( Q_{\text{Purchase}} \) 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 \( (Q_1[1]_s, s_1), (Q_2[2]_s, s_2) \) and \( (Q_3[3]_s, s_3) \). 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, sorts \( R \) according to the join attribute \( a \), breaking ties by the original (i.e., verifiable) order of \( R \), and generates the rank list \( Q \). For each \( Q \), the DSP first inserts it in the \( VO \), and performs two operations on \( T_s \), which we

\footnote{For simplicity, we refer to a tuple by its id, i.e., \( p_i \) signifies the first record of \( \text{Purchase} \).}
call index-traversal and leaf-scan. Index-traversal traverses \( T_5 \) to the leaf node that corresponds to the left boundary record of \( \Omega_{[i]} \). In the example of Figure 5, the DSP first inserts \( \Omega_{[1]} \) in the \( VO \), and descends \( T_5 \) until node \( A \), whose second entry corresponds to \( s_2 \), the left boundary of \( \Omega_{[1]} \). The digests of all left siblings (\( h_i \)) 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 \( \Omega_{[1]} \)) and a separator ",", signifying the end of the first round. The DSP proceeds to the next round, and inserts \( \Omega_{[2]} \) 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 \( \Omega_{[2]} \). The digest (\( h_3 \), \( h_6 \), \( h_C \)) of each skipped child is inserted into the \( VO \). Leaf-scan adds \( s_{10} \), the matching tuple \( s_{11} \) of \( \Omega_{[2]} \), and \( s_{12} \) (i.e., right boundary) to the \( VO \).

At the beginning of the third round, the DSP appends \( \Omega_{[3]} \) to the \( VO \), which has the same join attribute value as \( \Omega_{[2]} \). 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 \( \Omega_{[4]} \) to the \( VO \). At the fifth round, index-traversal reaches \( s_{15} \), appending \( h_3, h_6, h_7, h_8, h_9 \) 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 \( T_5 \), \( VO \))**  
// DSP
1. Append to the \( VO \) the signature of \( R \), \( |R| \), the root signature of \( T_5 \), and all records of \( R \) in a verifiable order
2. Sort \( R \) to generate the rank list \( \Omega_R \)
3. Initialize \( n = T_5.Root \)
4. For \( i = 1 \) to \( |R| \)
5. Append \( \Omega_{[i]} \) to \( VO \)
6. Call IndexTraversal(\( n, R[\Omega_{[i]}].a, VO \))
7. Call LeafScan(\( n, R[\Omega_{[i]}].a, VO \))
8. Append a separator ";" to \( VO \)
9. Call IndexTraversal(\( n, +\infty, VO \)) to complete the traversal of \( T_5 \)

**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 \( T_5 \), and match it against the signature, and (iv) verify the results for each \( \Omega_{[i]} \), 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 \( |\Omega_{[i]}| = |R| \), and for each pair of subsequent elements in \( \Omega_R \), \( \Omega_{[i]}[a] \leq \Omega_{[i+1]}[a] \). Moreover, if \( \Omega_{[i]}[a] = \Omega_{[i+1]}[a] \), the client checks that \( \Omega_{[i]}[\{i\}] < \Omega_{[i+1]}[\{i\}] \) (lines 5-7). For operation (iii), the client uses the records and digests of \( S \) to derive the digest \( h_{Root} \) at the root of \( T_5 \) bottom-up (line 11). In the above example, the \( VO \) contains \( h_3, h_6, h_C, h_J, h_{a1}, h_{a2} \). The client computes \( h_3, h_6, h_{a1}, h_{a2}, h_J, h_{a1}, h_{a2}, h_{a2} \) by applying \( H \) on the corresponding records. Then, it obtains \( h_3 \) (using \( h_6 \), \( h_3 \)), \( h_6 \) (using \( h_3, h_6 \)), \( h_J \) (using \( h_3, h_6, h_C \)), \( h_{a1} \) (using \( h_3, h_6, h_{a2} \)), \( h_{a2} \) (using \( h_3, h_6, h_{a2} \)), and \( h_{Root} \) (using \( h_3, h_6 \)), which is matched against the signature of \( T_5 \). 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 \( T_5 \), and all records of \( R \) from \( VO \)
2. Verify the signature of \( R \)
3. Initialize integer \( j = 0 \) and record \( r \) so that \( r.a = -\infty \)
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 \( (r.a > r'.a) \lor (r.a = r'.a \land j > j') \)
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 \( h_{Root} \)
12. Read digests from \( VO \) until it is empty, use them to incrementally compute \( h_{Root} \)
13. Verify \( h_{Root} \) 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/altered. 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 \( T_5 \).

**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 tree, 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 $h_{\text{root}}$, the VO must contain the digest $h$ of $s$ or of a node covering $s$. For instance, if $s_j$ were omitted, then the VO of Figure 5 should include $h_j$. 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 given 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 $r_s$.

AISM 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 $h_{\text{root}}$, 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 $\text{Root}_S$ to the first boundary $s_2$ of $r_1$. The digest ($h_{r_1}$) 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 $r_2.a = s_2.a$, 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 $T_S$ and $T_R$ 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 $r_3.a$, ascending to $\text{Root}_S$ 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., $h_{s_9}, h_{s_{10}}, h_{s_{11}}$, are inserted into the VO. The subsequent leaf-scan appends $s_{11}$, and $s_{12}$. Similarly to Round 1, a match ($s_{12}$) of the target tuple ($r_2$) is identified. Consequently, another leaf-scan is performed on $T_B$ (with target $s_{12}.a = r_2.a$) 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 $s_{12}.D-G-E-s_{13}$ 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 $\text{Root}_S$, adding the digest of $r_5$ 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 $h^S_{\text{root}}, h^R_{\text{root}}$ 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

[Diagram of AIM process]

Figure 8 Example of AIM
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 $T_b$, MBTree $T_s$, VO) // DSP
// The join query is $R \bowtie_{S.a=c} S$.
1. Append the root signatures of $T_b$ and $T_s$.
2. Traverse $T_b$ from $Root_b$ to the leftmost node, and insert the first record $r_1$ to the VO.
3. Initialize $n_b = T_b.Root$, $n_s = leftmost node of T_s$ storing $r_1$.
4. Initialize tuples $t_b = null$, $t_s = r_1$.
5. Loop
6. Call IndexTraversal($n_b$, $t_b.key$).
7. If the traversal of $T_b$ finishes
8. Call IndexTraversal($n_s$, $\infty$).
9. Return.
10. Call LeafScan($n_b$, $t_b.key$).
11. If a matching record of $t_b$ is encountered in line 10
12. Append a separator ";" to $VO$.
13. Call LeafScan($n_s$, $t_s.key$).
14. Set $t_s$ to the right boundary record corresponding to $t_b.key$.
15. Append a separator ";" to $VO$.

Figure 9 Algorithm AIM

Verify AIM (VO) // Client
1. Read the root signatures of $T_b$ and $T_s$ 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 $h^R_{Root}$.
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 $h^S_{Root}$.
12. Set $s$ to the right boundary record of $r$.
13. Repeat lines 4-12, reversing $R$ and $S$ (also $r_s$ and $s$).
14. Verify $h^R_{Root}$ and $h^S_{Root}$ 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 $T_b$ 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 $VO(R), B_S, VO(S), B_S$, and $\Omega_{gs}, B_S$ and $B_S$ are two bitmaps, and $\Omega_{gs}$ is a list of integers. The meaning of $B_S, B_S$ and $\Omega_{gs}$ will be explained soon. For our examples we use $Q_1 = Purchase \bowtie_{S.custid= Customer}$ in the database of Figure 1, assuming that the verifiable orders are $(p_1, p_2, p_3, p_4)$ 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 $\Omega_{gs}$. $\Omega_{gs}$ 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 $\Omega_R$.

If for two records $(r \in R, s \in S)$, $r.a = s.a$, then the element of $r$ in $\Omega_{gs}$ precedes that of $s$. Continuing the example, given $\Omega_{Purchase} = (1, 4, 3, 5, 2)$ and $\Omega_{Customer} = (1, 2, 3, 4, 5)$, $\Omega_{Purchase-Customer} = (1, 4, -1, 3, 5, -2, 2, -3, -4, -5)$.

ASM (Relation R, Relation S, VO) // DSP
// The join query is $R \bowtie_{S.a=c} S$.
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 $\Omega_{gs}$.
6. Create a bitmap $B_R$ of size $|R|$.
7. $1 \leq i \leq |R|$, 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 $\Omega_{gs}$ 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 $\Omega_{gs}$, 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
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 hand joins, whose join predicates are of the form \( |r.a−s.a| ≤ b \). 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 \( \Omega_{b} \) is the first record \( s \) such that \( \Omega_{b}(s)−a−b ≤ s.a ≤ \Omega_{b}(s)−a+b \) (instead of \( s.a = \Omega_{b}(s).a \)); 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 \( T_{3} \)), an additional leaf-scan is performed on \( T_{7} \) 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 \( T_{9} \), 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 \( Q_{\text{BST}} \) over two binary operators \( Op_{1}, Op_{2} \). Given the \( S \) on all join attributes, \( Op_{1} \) employs a variant \( m \)-AISM 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 AIM is used for \( Op_{2} \). The DSP sends two \( V_{O} \)s to the clients: \( V_{O}(RS) \) generated by \( Op_{1} \) to be used for verification of \( R \) to \( S \), and \( V_{O}(RST) \) to authenticate the output of \( Op_{2} \). Alternatively, the DSP can answer \( Q_{\text{BST}} \) through a right-deep plan that first joins \( S \) with \( T \), and then \( R \) with \( S \). 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.

\[ 3 \] 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 Op3 sends a separate VO to the client (based on which the client can derive R→ΔS), it is not necessary for Op2 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 ∈ R and s ∈ S, if and only if there is a tuple t ∈ 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.

\[
\text{VO(RS)}: \{\text{root signature of } T_R, \text{T}_S, s_1, s_2, h_k, r_1, r_2, r_3, s_4, s_5, h_c\}
\]
\[
\text{VO(RST)}: \{\text{root signature of } T_R, \text{A}[1], t_1, t_2, A[2]: A[3]; A[4]: h_c\}
\]

Continuing the example, the DSP skips traversing T_R for s_1, and retrieves the second record t_2 ∈ T_R, appending it to VO(RS). Since s_2 ∈ S_N, the DSP switches to T_R, inserting matching tuple r_3, boundary record r_4, r_5, and digest h_4 of node A to VO(RS). Then, it traverses T_S with new target r_4.A, adding boundary records s_3 and s_4 to VO(RS). Because s_2 is in S_N, the DSP suspends operations on T_R and arrives at s_3, which is also in S_N and does not initiate T_R traversal. Since s_1 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 Op2. In addition to authenticating the join results, VO(RST) must establish that the records in S_N for which Op2 skipped traversing T_R (s_1 and s_2) indeed have no matching T tuples. For this purpose, the input of Op2 contains both the join results of Op1 (i.e., tuple r_3s_2) and s_1, s_4, s_5. Op2 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 digest of T_R 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_2, s_3, s_4, s_5), and generates the (partial) join result r_s:2 of R→ΔS. When it later receives VO(RST), it prefixes the tuples computed in the previous step (i.e., s_2, r_3s_2, s_3, s_4) 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 t→ΔS and S t→ΔS to T, assuming two MB-trees T_R, T_S on R.a and T.a, respectively (S.a does not have an ADS). Initially, the owner’s signatures for T_R, T_S and S, as well as the tuples t_r ≡ s_3 ∈ S are appended to the VO. The DSP computes (i) a bitmap B_R specifying the S tuples with join partners in both R and T, which reduces the client’s storage consumption, and (ii) a rank list Q_R, signifying the sort order of S tuples on attribute a. Assuming the join order is (R→ΔS)→T, AST interleaves the computation of R→ΔS with that of (R→ΔS)→T to eliminate the generation of unnecessary R→ΔS results. Specifically, the DSP starts by retrieving the first element Q_R[1] from Q_R, appending it to the VO, and traversing T_R with Q_R[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 (Q_R[1], r_1) is identified, the DSP switches to T_T, adding matching tuple r_2 and boundary record t_2 to the VO. After that, the DSP continues joining R and T. Note that

\footnote{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 $t_{5,a}$. Based on this, the DSP scans $\Omega_3$, and appends to the VO each tuple $s \in S$ such that $s.a \leq t_{5,a}$ (without trying to find a join partner of $s$ in $T_3$). For instance, although $\Omega_2[2]$, $\Omega_3[3]$ and $\Omega_3[3]$ match $r_a$, $r_b$ and $r_c$, respectively, none of these results is inserted to the VO. Finally, when $\Omega_3$ 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 $r_s$, $r_t$, node $C$ (for $T_R$) and $t_1$ (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 $S$],

$\Omega_2[1]$, $h_{r_1}$, $r_2$, $r_3$, $r_5$, $t_1$;

$\Omega_3[3]$; $h_{r_4}$, $h_{r_5}$, $h_{j_1}$;

$T_R$

$\Omega_2[2]$

$\Omega_3[3]$

$\Omega_3[4]$

$S$

$R$

$A$

$T$

$Root_a$

$Root_T$

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 $p_{1}, p_{2}, c_{1}, c_{2}, c_{3}$ indexing their $pid$, $cid$ and $quantity$ ($cid$, $name$ and $city$) attributes. Assuming MB-trees on both $Purchase.cid$ and $Customer.cid$, the query $Q_s = \pi_{\text{name, quantity}}(\text{Purchase} \cup_{\text{cid}} \text{Customer})$ 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 $cid$, $quantity$ ($name$, $city$) 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 $s.a < C$ where $C$ is a constant between $s.a$ and $r_c.a$. After round 1 of AISM execution, the DSP finds that $\Omega_2[2]$ already contains the range selection, and thus simply inserts $\Omega_2[2]$ to $\Omega_6[6]$ directly into the VO without traversing $T_R$ to identify their matching $S$ tuples. Instead, it simply appends digests of the right siblings of the root-to-leaf path of $s_a$ to complete the traversal of $T_R$. For case (ii), the selection and join operators compete for the use of ADSs. An example is the complex query $Q_s$ in Figure 1, which involves a join and two selections $\sigma_1 = \sigma_{\text{quantity} \geq 100}(\text{Purchase}, \text{Customer})$, $\sigma_2 = \sigma_{\text{city} = \text{New York}}(\text{Customer})$. Assuming an ADS is available on each of the 4 attributes ($\text{cid}$, $\text{quantity}$ in $\text{Purchase}$ and $\text{cid}$, $\text{city}$ in $\text{Customer}$), an important observation is that for a particular relation (e.g., $\text{Purchase}$), only one ADS (e.g., either the one on $\text{cid}$ or the other on $\text{quantity}$) can be utilized to answer the query. Figure 16 illustrates three different plans for $Q_s$. In the first plan (Figure 16a), the join is able to utilize the ADSs on both $\text{Purchase.cid}$ and $\text{Customer.cid}$ 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.

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 AIML 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 $[1,10^5]$, whereas $R.a_1$, $R.a_2$ and $S.a_2$ follow Gaussian distribution (with mean $5 \cdot 10^4$, $3 \cdot 10^4$ and $7 \cdot 10^4$ respectively), and share the same variance $\sigma=10^6$. In addition, $R.a_2$ 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 $R \bowtie S$, denoted as FK, and (ii) an equi-join $R \bowtie S$, 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 $10^4$ 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

<table>
<thead>
<tr>
<th>Tuple size (bytes)</th>
<th>32</th>
<th>64</th>
<th>128</th>
<th>256</th>
<th>512</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_{100} ) (Gbytes)</td>
<td>8.9</td>
<td>9.0</td>
<td>9.2</td>
<td>9.6</td>
<td>10.3</td>
</tr>
<tr>
<td>( C_{\text{Client}} ) (seconds)</td>
<td>205</td>
<td>207</td>
<td>210</td>
<td>214</td>
<td>219</td>
</tr>
<tr>
<td>( C_{\text{DSP}} ) (seconds)</td>
<td>262</td>
<td>271</td>
<td>429</td>
<td>1728</td>
<td>4603</td>
</tr>
</tbody>
</table>

Figure 17 shows \( VO \) size as a function of the tuple length for NAI, ASM, AISMA and AIM, as well as the theoretically optimal \( VO \) size, which is equal in \( R \) and \( S \). This has a similar effect as applying an additional selection (with variable selectivity) on the inner 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 relation to the client. The \( VO \) of AIM and AISM 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.

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   |
|-------------|-----|-----|-----|-----|-----|
| \( C_{100} \) (Gbytes) | 7.8 | 8.9 | 9.2 | 9.5 | 9.7 |
| \( C_{\text{Client}} \) (seconds) | 196 | 205 | 210 | 218 | 223 |
| \( C_{\text{DSP}} \) (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](image_url)

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](image_url)

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

![Figure 22](image_url)

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.a1, R.a2, S.a1, S.a2, S.b1, S.b2, T.b1, T.b2. Attributes R.a1 and T.b1 are uniformly distributed in [1,10^7]. S.a1 and S.b1 follow Gaussian distribution with mean value 5*10^6 and variance 10^6. R.a2, S.a2, S.b2 and T.b2 follow Gaussian distribution with variance 10^6 and mean values 4*10^6 (for R.a2, S.b2) and 7*10^6 (S.a2, T.b2). In addition, R.a1 and T.b1 are the primary keys of tables R and T, and S.a1, S.b1 are foreign key referencing R.a1 and T.b1, respectively.

An MB-tree is constructed for each attribute. We evaluate two datasets, the first one is uniformly distributed in [1,10^7], while that of the second one uses Y. Note that “m-AIM+m-AIM” is not applicable since the results of R-R-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 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-AIM+m-AISM, achieve considerable savings in terms of VO size. m-AIM+m-AISM has the best overall performance, followed by m-AIM+m-AISM, and finally m-AIM+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-S-S. Consequently, in the FK dataset, the advantage of m-AIM+m-AISM over m-AIM+m-AISM is limited since the number of these intermediate results is large due to the foreign-key constraint.

![Figure 23](image_url)

Figure 23 VO size vs. tuple size

![Figure 24](image_url)

Figure 24 Query cost for the client vs. tuple size
Finally, we fix the record size to 128 bytes, $|S|$ to $5 \cdot 10^2$, 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 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.

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