DOI: 10.1109/CNS.2014.6997505

Integrity for distributed queries

Sabrina De Capitani di Vimercati*, Sara Foresti*, Sushil Jajodia†,
Giovanni Livraga*, Stefano Paraboschi‡, Pierangela Samarati*
*Università degli Studi di Milano, 26013 Crema - Italy Email: firstname.lastname@unimi.it
†George Mason University, Fairfax, VA 22030-4444 - USA Email: jajodia@gmu.edu
‡Università degli Studi di Bergamo, 24044 Dalmine - Italy Email: parabosc@unibg.it

Abstract—**We propose an approach for allowing users to assess the integrity of distributed queries computed by a computational cloud, which is trusted neither for data confidentiality nor for query integrity. In particular, we consider join queries over multiple data sources, maintained at separate (trusted) storage servers, where join computation is performed by an inexpensive, but potentially untrusted, computational cloud. We illustrate the working of our approach and its application in a MapReduce scenario. We also provide an analysis and experimental results.**

Keywords*—*Cloud; Distributed queries; Integrity; MapReduce

I. Introduction

Users and companies are today more and more resorting to cloud-based solutions for accessing information and services. Such a success has motivated cloud providers to enrich their offers allowing users to enjoy a large variety of configurations by different service providers, offering storage and computational capabilities with different features and at different economical costs. An emerging trend shows a clear distinction between storage providers, with reliability and continuity of access as critical factors, and computational providers, which see the price of the service as the competitive factor.

In this context, we consider a scenario where a *client* wishes to perform joins over multiple relations stored at different *storage servers* by using a *computational cloud* (Figure 1). A specific feature that drives our work is the consideration of the MapReduce paradigm, which supports the processing of a vast amount of data in parallel on a large number of nodes. The join of data originating from multiple sources represents the operation most commonly discussed when analyzing the behavior of MapReduce architectures. We assume that the storage servers are trustworthy while the computational cloud is not. There are many reasons for which a join computation can be delegated to a computational cloud, instead of being performed by the same servers storing the data. First, storage servers - chosen based on reliability and continuity of access - might not be the most economically viable solution for executing computations. They can then perform simple operations (like selection, projection, or encryption of data) but may result too expensive for joins, which are computation intensive operations. This reasoning applies also to the case where storage servers are not external providers but data owners themselves (where each storage server might be under control of a different authority), which can accommodate retrieval on their data but cannot dedicate resources to executing computations. Second, the computation might require a many-to-many join on large relations, with the need to represent internally a large collection of intermediate results, going beyond the ability of a storage server. Third, a computation might require joins among a multiplicity of servers (join chains), and relying on a dedicated computational service is more appropriate than introducing high redundancy in the communication of data. Fourth, the evaluation of joins by storage servers might be problematic when the internal organization of the data requires a dynamic re-organization for the join execution.

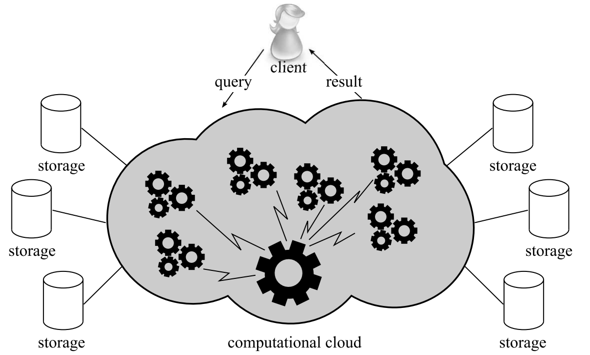


Figure 1. Reference scenario

Since the computational cloud is not trusted, there is the need to protect confidentiality of the data and provide guarantees on the integrity of the join results. Our approach allows a client to protect the confidentiality of data and join operations w.r.t. the computational cloud and to assess, in a probabilistic way, the integrity of the result. It relies on the use of complementary techniques, originally introduced in [1], which are extended to the consideration of a distributed computational cloud, as the MapReduce paradigm. A great advantage of our solution is the support of any kind of join (one-to-one, one-to-many, and many-to-many) as well as of sequences of joins, which are either not supported (many-to-many joins or sequences) or require additional overhead (one-to-many joins) in earlier approaches. The remainder of the paper is organized as follows. Section II introduces the MapReduce paradigm and the basic techniques that we extend and apply in our work. Section III illustrates the working of our approach with reference to joins involving two relations, describing query execution and the application of integrity techniques to ensure control of all components in the MapReduce framework. Section IV extends the treatment to arbitrary sequences of join operations involving multiple storage servers. Section V analyzes the working of our approach and the integrity guarantees offered. Section VI presents experimental results. Section VII discusses related work.

II. Basic concepts

A MapReduce framework supports execution of tasks by multiple nodes in a cloud architecture. Among the nodes, one works as *manager*, distributing tasks and collecting results from other nodes operating as *workers*. Briefly, MapReduce works as follows. A user-defined *map function* translates the input data into a set of pairs of the form ⟨*key*, *value*⟩. The manager then distributes these pairs to a set W=$\{w_1,\ldots,w_l\}$ of workers according to an *assignment function* $f: K \rightarrow W$, with K the domain of the *key* element, so that all pairs with the same key are assigned to the same worker. Each worker performs a user-defined *reduce function* on the input pairs and sends the result of its computation to the manager. Finally, the manager combines the results received from all workers to produce the final result. In our scenario, MapReduce is used to perform joins among relations at different storage servers.

Our approach for ensuring confidentiality and assessing the integrity of joins is based on the combined adoption of three different techniques [1] that we adapt and extend to our scenario. *Encryption on the fly*: storage servers encrypt data before sending them to the computational cloud, which then processes them in encrypted form. *Markers*: storage servers insert fake tuples (not recognizable as such by the computational cloud) in their relations. *Twins*: storage servers duplicate (twin) some of the tuples in their relations before sending them to the computational cloud (with twins not recognizable as such). These basic techniques provide us building blocks for ensuring protection of the information and for enabling our integrity checks. Encryption ensures confidentiality to the information being processed by the computational cloud, also making markers and twins not recognizable as such. It also provides basic integrity guarantees for the individual tuples (in particular to the join attributes, which, as we will see in the next section, are the only attributes processed by the computational cloud in our solution). Markers and twins provide integrity guarantees on the query results by enabling checking its completeness.

III. Query execution and integrity verification

We first present our approach with reference to join queries involving only two relations (i.e., two storage servers). For simplicity, but without loss of generality, we assume storage servers to be fully trusted and to have visibility of the data they store in plaintext. Our approach can however be applied also when the storage servers are considered *honest-but-curious* and should not have visibility on the data, by storing the relations in encrypted form together with indexes used for query execution [2]. In Section IV, we will extend the treatment to queries involving joins over multiple relations at an arbitrary number of storage servers. For ease of presentation, we illustrate the working of our approach incrementally, by first providing an abstract description of the query execution, referring to the MapReduce framework as a computational cloud. We then discuss how the specific working of MapReduce is controlled.

A. Query execution

We employ a semi-join execution strategy. In absence of security considerations, performing a join as a semi-join means that the storage servers communicate to the computational cloud only the join attributes of their relations. After performing the join, the resulting tuples are then extended with the non-join attributes to be returned in the result. The consideration of a semi-join execution strategy, besides enabling a more efficient execution (especially for joins involving large tuples), permits to translate any join to operate as one-to-one joins in the cloud, with notable advantages. The first is the ability to efficiently support one-to-one, one-to-many, many-to-many joins and join sequences. The second is that it enables protecting the join profile (i.e., how many occurrences of each value appear in the base relations and how many of their tuples participate in each match) without introducing any overhead.

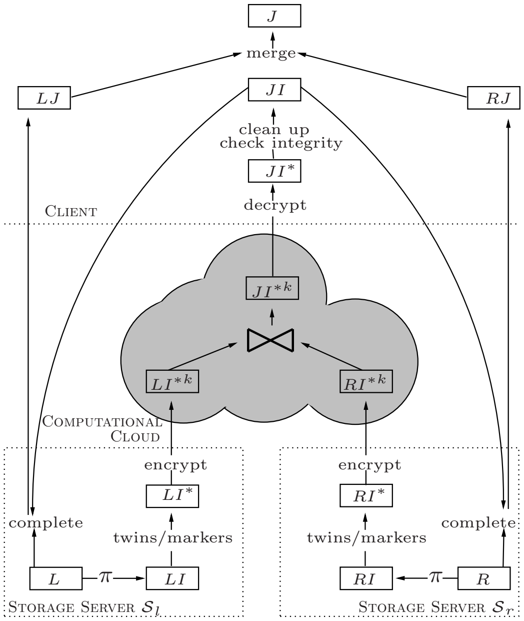


Figure 2. Information flow among the parties

Query execution works as follows. Let us consider a query q of the form "SELECT A FROM B_l JOIN B_r ON $B_l.I = B_r.I$ WHERE C_l AND C_r AND C_{lr}," where B_l and B_r are the relations stored at server $\mathcal{S}_l$ and $\mathcal{S}_r$, respectively; A is a subset of the attributes in $B_l \cup B_r$; I is the set of join attributes; and C_l, C_r, and C_{lr} are Boolean formulas of conditions over attributes in B_l, B_r, and $B_l \cup B_r$, respectively. Applying usual push-selections-down optimization, we assume conditions C_l and C_r to be pushed down for evaluation at the respective storage server. Each storage server then receives in encrypted form its sub-query together with information regulating insertion of markers and twins (see Section III-B). As the execution of these sub-queries does not introduce any challenge, we will simply assume them to be taken care of and refer to L and R as the relations at the storage servers $\mathcal{S}_l$ and $\mathcal{S}_r$, respectively, after the evaluation of selection conditions C_l and C_r, and focus on the derivation at the client of the join J on which C_{lr} can be executed and A projected. The data flow and execution of the operations work as follows (Figure 2). Storage server $\mathcal{S}_l$ ($\mathcal{S}_r$, resp.) computes on relation L (R, resp.) the projection LI (RI, resp.) of the set I of join attributes. It then adds twins and markers obtaining relation LI^* (RI^*, resp.), which is then encrypted, tuple by tuple, producing relation LI^{*k} (RI^{*k}, resp.) with a schema including only attribute I^k, with values corresponding to the encrypted chunks, which are sent to the

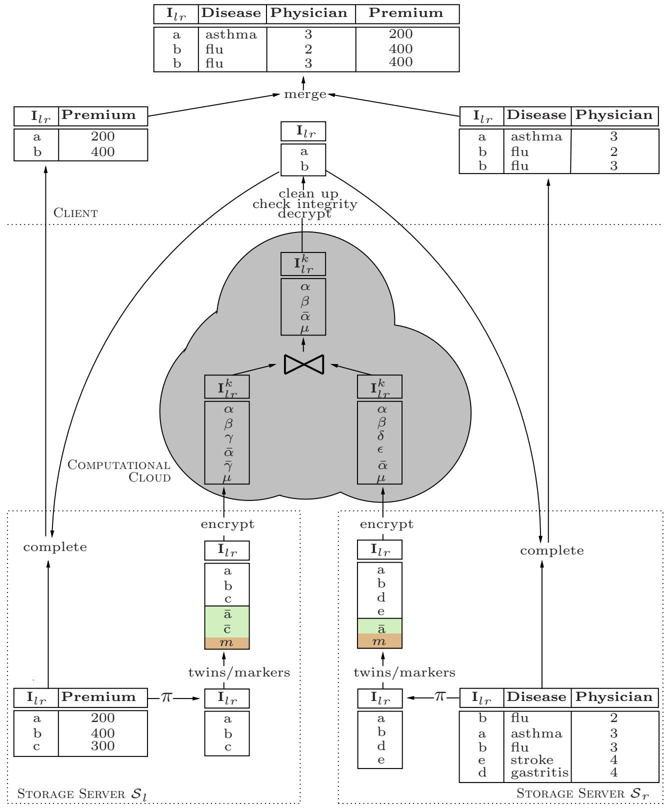


Figure 3. An example of join evaluation

computational cloud. The computational cloud produces the join between LI^{*k} and RI^{*k} and returns the resulting relation JI^{*k} to the client. The client decrypts it, obtaining relation JI^{*}, and checks integrity (i.e., tuples decrypt correctly and all expected markers and twins are present). If no violation is detected, the client removes twins and markers and asks the storage servers to send the remaining attributes of the tuples whose join attribute values are in JI. The client then merges the relations LJ and RJ received from the storage servers, obtaining join J on which it can evaluate condition C_{lr} and project the needed attributes. Note that, while the semi-join strategy requires several data exchanges among the involved parties, it limits the transfer of non-join attributes and the overall amount of data transferred. Furthermore, the operations performed by the client and the storage servers require limited cost. Figure 3 illustrates an example of application of our techniques where: the join attribute is I_{lr}, there is one marker (m), and twins are created for tuples with a value for the join attribute equal to a or c. We use the bar notation to denote twins (e.g., the twin of a tuple t is denoted by $\overline{t}$, and the twin of join attribute value a is denoted by $\overline{a}$). In the following, we use the term twin to refer to either the tuples (original and derived) involved in a relation. We use Greek letters to denote encrypted values and encrypted tuples. Note how twins, markers, and the join profile are protected from the eyes of the computational cloud, which simply observes a set of possibly matching encrypted chunks where mapping is always one-to-one, even if one-to-many in reality (e.g., for value b in Figure 3 there are two occurrences in the original relation at $\mathcal{S}_r$).

B. Integrity checks and work distribution

We now describe the coordination, insertion, and verification of integrity controls enabling the client, with the cooperation of the storage servers, to assess the integrity of the join result and detect possible misbehavior of the computational cloud and of a specific worker in particular.

In addition to *encryption* (which, as already noted, provides basic confidentiality, integrity of individual tuples and makes our integrity controls non-recognizable) our solution relies on the use of fake check tuples (*markers*), and data replication (*twins*). Encryption, as well as insertion of twins and markers, is done autonomously by each storage server whose operations must however be ensured to be coordinated with the others, to guarantee that the integrity checks work properly. In particular, encryption is performed by each storage server independently, by using the same symmetric key provided by the client together with the query (the key changes for every query that, like the control information, is protected in communication since the client encrypts them with a key shared with the storage server). The client communicates to each storage server also: *i)* the number N of markers and *ii)* a twinning condition C_{twin} regulating tuple duplication to use for the query. Condition C_{twin} is defined on the result of a keyed hash function operating on the join attribute, which produces a uniform distribution of values and permits to regulate the percentage of twins in the relations.

A complicating factor in our MapReduce scenario is that computation is distributed among different workers. This introduces the problem of distributing the controls (twins and markers) over the different workers so to ensure a complete overseeing of the system as well as enabling accountability for misbehaviors and failures (see Section V). In presence of many servers (workers, in our MapReduce scenario), if no condition is enforced on the distribution of twins and markers to workers, a worker might end up receiving none of them, making a complete failure of the worker not detectable. In a MapReduce scenario, work is distributed by the manager to the workers based on a (deterministic) assignment function that maps each pair $\langle$*key,value*$\rangle$ to a worker. For our query computations, pair $\langle$*key,value*$\rangle$ is of the form $\langle \tau[I^k], LI^{*k} \rangle$ ($\langle \tau[I^k], RI^{*k} \rangle$, resp.), meaning that the key is the encrypted join attribute and the value is the name of the relation where it comes from. The assignment can be any function (e.g., a hash function with output in the range $1, \ldots, |\, W\,|$) performed by the manager for allocating pairs to workers. The assignment function trivially guarantees that tuples with the same values for the join attribute (i.e., matching in the join) are assigned to the same worker and therefore that no tuple can be missed from the join due to an improper allocation. While the assignment is performed by the manager (which is part of the non-trusted computational cloud) the assignment function is known to the client and the storage servers.

Markers. Let N be the number of markers that the client decides to use for verifying the completeness of a join result, and l be the number of workers. As markers are randomly generated, there is no guarantee that they will be evenly distributed among the different workers. We can imagine different strategies that could be applied for the distribution of markers among the workers:

- *Random*: no condition is required on the distribution of markers to workers, which is then completely random.
- *At-least-n*: no worker can go completely uncontrolled; every worker should receive at least n markers (with $n \leq \lfloor N/l \rfloor$).
- *Perfect balance*: markers should be distributed evenly among workers. Each worker should receive a number of markers from $\lfloor N/l \rfloor$ to $\lceil N/l \rceil$.

A random distribution strategy provides guarantees on the behavior of the system overall but can leave some workers completely uncontrolled as noted above. The other two strategies ensure instead that a certain amount of control is enforced at every worker. We can capture all these three strategies with the following definition.

Definition 3.1 (Marker distribution strategy): A *marker distribution strategy* is a triple $\langle N, N_{min}, N_{max} \rangle$ where N is the total number of markers to be assigned, and N_{min} and N_{max} are the minimum and maximum, respectively, number of markers to be assigned to every worker.

Definition 3.1 specifies that a distribution strategy dictates the total number of markers that should be applied imposing, in addition, constraints on the number of markers that each worker should receive. Trivially, Definition 3.1 covers the three strategies above specifying, respectively: $\langle N, 0, N \rangle$, $\langle N, n, n+(N-n*l) \rangle$, $\langle N, \lfloor N/l \rfloor, \lceil N/l \rceil \rangle$. Note that both the total number of markers and the number of markers at each worker should satisfy the strategy. Hence, although in principle with an at-least-n strategy, a worker can receive all the spare $(N - n * l)$ markers, its bound will be dynamically adjusted depending on whether other workers have already received some markers.

Property 3.1 (Correct markers distribution): A set M of markers is correct w.r.t. a distribution strategy $\langle N, N_{min}, N_{max} \rangle$ if: *i)* $\forall w_i \in W$: $N_{min} \leq n_{w_i} \leq N_{max}$, with n_{w_i} the number of markers assigned to worker w_i, and *ii)* $\sum_{i=1}^{l} n_{w_i} = N$.

Each storage server generates markers via a function μ (e.g., a pseudorandom function or a progressive counter) communicated by the client together with the query. Applying such a function, each storage server will produce the same sequence of markers. When the computational cloud is composed of a worker only, simply taking the first N markers generated suffices. The complication in our scenario is that the first N markers generated might not guarantee the distribution requested by the client. Fortunately, the implementation of a correct marker distribution strategy is pretty straightforward and can be enforced independently by every storage server. Figure 4 illustrates function Generate_Markers executed by each storage server for generating markers. Basically, each storage server generates markers in sequence and checks every marker generated to see to which worker it would be assigned. If the worker to which the marker would be assigned has not reached the minimum number N_{min} of markers, or has passed it but has not reached N_{max} and there are spare markers still to be allocated, the marker is retained. Else, the marker is discarded and a new marker is generated. The function terminates when N markers have been assigned to workers. Since each storage server uses the same function, they produce the same set of markers, allocating them to the same workers as formally stated in the following theorem. (The proofs of theorems are omitted for space constraints).

```
Function Generate_Markers($\langle N, N_{min}, N_{max} \rangle$)
$M$ := $\emptyset$ /* set of generated markers */
$spare$ := $N - (N_{min} * l)$ /* spare markers */
for $i$:=1 to $l$ do $num\_markers[w_i]$ := 0 /* n. of markers assigned to $w_i$ */
repeat
   generate new marker $m$ via function $\mu$
   $w$ := $f(E_k(m))$ /* $f$ is the assignment function */
   if ($num\_markers[w] < N_{min}$) $\vee$
      ($num\_markers[w] < N_{max} \wedge spare > 0$)
   then
      $num\_markers[w]$ := $num\_markers[w] + 1$
      if $num\_markers[w] > N_{min}$ then $spare$ := $spare - 1$
      $M$ := $M \cup \{m\}$
until $|M| = N$
return $M$
```

```
Function Generate_Twins($B$, $C_{\text{twin}}$)
Let $T$ be the set of tuples in $B$ satisfying $C_{\text{twin}}$
$\overline{T}$ := $\emptyset$ /* set of generated twins*/
for each $t \in T$ do
   $w$ := $f(E_k(t[I]))$
   repeat
      generate salt $s$ via function $\sigma$
      $\overline{t}$ := $t$
      $\overline{t}[salt]$ := $s$
      $\bar{w}$ := $f(E_k(\overline{t}[I] \oplus \overline{t}[salt]))$
   until $\bar{w} \neq w$
   $\overline{T}$ := $\overline{T} \cup \{\overline{t}\}$
return $\overline{T}$
```

Figure 4. Functions generating markers and twins

Theorem 3.1: Let M_l and M_r be the sets of markers for relations L and R computed by storage servers $\mathcal{S}_l$ and $\mathcal{S}_r$, respectively, with function Generate_Markers (Figure 4). The following conditions hold: *i)* M_l and M_r satisfy Property 3.1; *ii)* M_l=M_r; and *iii)* $\forall m_l, m_r$ s.t. $m_l \in M_l$, $m_r \in M_r$, m_l=m_r $\Rightarrow$ $f(E_k(m_l))$=$f(E_k(m_r))$, with E_k a symmetric encryption function with key k.

Twins. Twins are duplicate copies of actual tuples made distinct by the addition of a salt (which is null for original tuples). Twinning is controlled by the client, which defines a twinning condition C_{twin} communicated to the storage servers together with the query (all tuples whose join attribute values satisfy the twinning condition are duplicated). Controlling distribution of twins over different workers of the computational cloud is more complex than for markers. In fact, while markers are simply generated by a deterministic function (and all storage servers can operate independently while guaranteeing the same behavior), the generation of twins depends on the specific join attribute values of the tuples stored at each storage server. For instance, with reference to the example in Figure 3, $\mathcal{S}_l$ generates two twins (as its relation stores values a and c matching the twinning condition, while only value a is present in the relation at $\mathcal{S}_r$). Each storage server can then observe a different number of twins assigned to different workers. Requiring a possible adjustment in distribution is then not possible without requiring (impractical, if at all doable) explicit coordination among the storage servers. In fact, a storage server cannot simply discard the twin of a tuple satisfying the twinning condition (like it is done for markers) and generate another one with a different salt without informing of this all the other storage servers, as matching twins would be assigned to different workers by different storage servers, and would then not belong to the join result. For instance, with reference to our example, suppose $\overline{a}$=$a \oplus s_1$ would be mapped to a worker w_i and that $\mathcal{S}_l$ considers w_i already complete w.r.t. twins (as $\overline{c}$ was assigned to it). Suppose then that $\mathcal{S}_l$ recomputes $\overline{a}$=$a \oplus s_2$ (with a different salt s_2) to map it to a different worker w_j. Server $\mathcal{S}_r$, having twinned only one tuple, would not observe the same problem and would proceed with the first $\overline{a}$ computed, which maps to worker w_i. As a result, the two twins of a would be assigned to two different workers, and therefore would not belong to the join result. This observation clarifies that, although an adjustment in the production of markers/twins can be enforced to provide their distribution at different workers, such adjustment can depend only on properties on which all storage servers have the same observations. As this is not possible for twins, for them we impose a basic property requiring that a twin be not assigned to the same worker that has the original tuple from which the twin originated, formally expressed as follows.

Property 3.2 (Twin separation): Let T be the set of tuples of relation B at storage server $\mathcal{S}$ satisfying twinning condition $C_{\rm twin}$, $\overline{T}$ be the corresponding twins, and $f : K \rightarrow W$ be the assignment function. The set $T \cup \overline{T}$ is said to satisfy *twin separation* if $\forall t, \overline{t}$, with $t \in T$, $\overline{t} \in \overline{T}$, and $t[I]$=$\overline{t}[I]$: $f(E_k(t[I])) \neq f(E_k(\overline{t}[I] \oplus \overline{t}[salt]))$.

The property nicely enforces in a simple way a sort of two-man-rule since a worker missing a twin tuple t ($\overline{t}$, resp.) would be exposed by the presence of the other twin $\overline{t}$ (t, resp.) in the computation of a different worker. Also, collusion is not possible since workers neither know which tuples are twin nor can determine which other worker has the corresponding twin. Also, we can expect twins of different tuples at a given worker to be distributed to several other workers, then effectively providing a network of distributed control over every single worker (greatly increasing, in practice, the effectiveness of twinning, as splitting a twin pair over two different workers essentially makes each twin work as a marker for the worker to which it was assigned).

Satisfaction of Property 3.2 is rather simple and can be enforced independently at each storage server, to which the client simply communicates the twinning condition $C_{\rm twin}$ and the pseudorandom function to be used for generating the salts for twinning. Figure 4 illustrates function Generate_Twins generating twins. Basically, each storage server determines its join attribute values to be twinned. For each value $t[I]$ to be twinned assigned to worker w, the storage server computes a salt and checks the worker $\overline{w}$ to which the resulting twin (the combination of the original value with the salt) would be assigned and if $\overline{w}$=w, it recomputes the twin with a new salt, until $\overline{w} \neq w$. The following theorem states that the set of twins generated by function Generate_Twins satisfies Property 3.2 and that twin pairs are correctly allocated to workers.

Theorem 3.2: Let $\overline{T}_l$ and $\overline{T}_r$ be the sets of twins for relations L and R computed by storage servers $\mathcal{S}_l$ and $\mathcal{S}_r$, respectively, with function Generate_Twins (Figure 4). The following conditions hold: *i)* $\overline{T}_l$ and $\overline{T}_r$ satisfy Property 3.2; and *ii)* $\forall t_l, t_r$ s.t. $t_l \in L$, $t_r \in R$, and $t_l[I] = t_r[I]$ satisfies twinning condition $C_{\rm twin} \Rightarrow \exists \overline{t}_l \in \overline{T}_l$ and $\exists \overline{t}_r \in \overline{T}_r$ s.t. $f(E_k(\overline{t}_l[I] \oplus \overline{t}_l[salt])) = f(E_k(\overline{t}_r[I] \oplus \overline{t}_r[salt]))$.

```
Function Check_Integrity($JI^*, \langle N, N_{min}, N_{max} \rangle, C_{\rm twin}$)
$M$ := Generate_Markers($\langle N, N_{min}, N_{max} \rangle$)
$T := \emptyset$ /* twins found solo */
for each $t \in JI^*$ do
    if $t$ is a marker then
        $M := M \setminus \{t\}$; $JI^* := JI^* \setminus \{t\}$ /* remove marker from $JI^*$*/
    elseif $t$ satisfies $C_{\rm twin}$ then
        if $\exists \overline{t} \in T$ s.t. $t'[I] = t[I]$ then
            $T := T \setminus \{\overline{t}\}$; $JI^* := JI^* \setminus \{t\}$ /* remove twin from $JI^*$ */
        else $T := T \cup \{t\}$
if $M \neq \emptyset \vee T \neq \emptyset$ then return NULL else return $JI^*$
```

Figure 5. Function checking integrity

The combined use of twins and markers provides strong protection guarantees and allows a client to easily verify the integrity of a join result. Figure 5 illustrates function Check_Integrity used by the client to verify integrity and remove twins and markers from the result JI^* returned by the computational cloud. The function first generates the set M of expected markers and sets variable T to the empty set. It then considers each tuple t in JI^*. If t is a marker, it is removed it from both M and JI^*. If t is a twin, then if the corresponding twin $\overline{t}$ is in T, it removes it, also removing t from JI^*; else it adds t to T. At the end, if M or T is not empty, an integrity violation is detected.

IV. Join sequences

The approach and integrity checks presented for joins involving two relations/servers can be extended to queries involving an arbitrary number of relations/servers, that is, an arbitrary number of join operations. The extension requires some investigation and adjustment to ensure that joins remain one-to-one, so to maintain the join profile completely hidden from the computational cloud, and that the amount of integrity checks, in terms of number of markers and twins expected in the result, does not get lost due to the applications of the join sequence (to which only a limited number of tuples survive).

Before illustrating the approach for join sequences, we note that the case when, for every relation, the attributes participating in the joins are always the same does not require special treatment as it is a trivial extension of the case with two relations illustrated in Figure 2. *1)* Every storage server evaluates the local condition, projects the resulting relation over the join attributes, and - after adding markers, twins, and encrypting - sends the resulting relation (composed only of encrypted chunks) to the computational cloud. *2)* The computational cloud performs the join among all the n relations received and sends the join result to the client. *3)* The client checks the integrity of the join result, asks the storage servers to send the remaining needed attributes of the tuples belonging to the join, and then merges them to produce the query result.

When relations participate in different joins with different attributes, the situation is more complex. In fact, storage servers cannot simply send the projection of all the join attributes to the computational cloud as, in case of one-to-many or many-to-many joins, this would expose the join profile to the eyes of the computational cloud. Joins need therefore to be executed in sequence, and the information sent by a storage server for the execution of the i-th join should consider only the tuples in its relation that have already been filtered by the previous $i-1$ joins. Also, the requests to the storage servers to send the remaining attributes for the tuples belonging to the join result should be performed in a sequence (reversed in this case) so that only tuples actually belonging to the join result are communicated to the client (which then needs to do a simple merge).

Figure 6 illustrates the pseudocode describing the operations executed by the client, storage servers, and computational cloud for a generic sequence of joins. To simplify the notation, for readability and simplicity of the pseudocode, we assume joins to be in a chain, while noting that the approach applies to arbitrary joins (i.e., a relation can be involved in a join with more than two other relations). The work of client and storage servers is divided in two phases: phase 1 retrieves the join attributes in the join result (via the computational cloud to which the join computations - the most expensive operations - are delegated); phase 2 completes the join result with the remaining attributes (which are transmitted by the storage servers to the client only for those tuples actually belonging to the result as computed in phase 1). In the first phase, the communication between the client and the storage servers is mediated by the computational cloud, which simply forwards packets with the requests encrypted with the key of the receiving storage server. The requests specify the query to be executed, the marker distribution strategy, the twin condition, the key to be used for the query and a query identifier *qid* (which is used by each storage server to operate at every step on partial results already retrieved instead of on the base relations). Basically, let $B_1, B_2, \ldots, B_n$ be the sequence of relations involved in a join, with I_{ij} the attribute participating in the join between B_i and B_j, $\mathcal{S}_i$ the storage server storing B_i, and C_i the condition on it in the query. First, the client requests $\mathcal{S}_1$ and $\mathcal{S}_2$ to evaluate the conditions (C_1 and C_2) on their respective relations obtaining B_1^{qid} and B_2^{qid}, and then to project over attributes I_{12} retrieving (via the join from the computational cloud over the encrypted projection completed with markers and twins) JI^*_{12}. At the i-th join between B_i and B_j, the client requests $\mathcal{S}_i$ to provide I_{ij} of all the tuples B_i^{qid} (i.e., the tuples of B_i that have satisfied all the conditions evaluated up to that point of the computation) for which $I_{(i-1)i} \in JI^*_{(i-1)i}$ (i.e., values for the join attribute of the previous join belonging to the partial result). It requests instead $\mathcal{S}_j$ to provide the projection of I_{ij} for all tuples in B_j that satisfy C_j. At the end of the sequence, the client will proceed in reverse order, asking each storage server $\mathcal{S}_i$ (from $\mathcal{S}_n$ to $\mathcal{S}_1$) to complete the tuples belonging to the results with all attributes $Attr_i$ of B_i appearing in the query. The reason for considering relations in reverse order is that the join is refined at every step and then $JI_{(n-1)n}$ is the most selective, certainly including only tuples that belong to the final result. Proceeding backwards guarantees therefore to request to each storage server only the information for the tuples actually belonging to the result. Figure 7 (from left to right) illustrates relations sent by storage servers and retrieved by the client in phase 1 for the execution of query "SELECT * FROM B_1 JOIN B_2 ON $B_1.I_{12} = B_2.I_{12}$ JOIN B_3 ON $B_2.I_{23} = B_3.I_{23}$ JOIN B_4 ON $B_3.I_{34} = B_4.I_{34}$". For instance, for the join between B_3 and B_4, $\mathcal{S}_3$ needs to provide attribute I_{34} only for those tuples that have attribute I_{23} equal to 10 or 20 (i.e., belonging to the join of the previous step). In phase 2, the storage servers are contacted in reverse order, every server $\mathcal{S}_i$

```
/* $q$ : user query SELECT $A$ FROM $B_1$ JOIN $B_2$ ON $B_1.I_{12}$=$B_2.I_{12}$
                              ...JOIN $B_n$ ON $B_{n-1}.I_{(n-1)n}$=$B_n.I_{(n-1)n}$
              WHERE $C_1$ AND ... AND $C_n$ AND $C_{ij}$
   $qid$ : identifier of the query
   $k_i$ : encryption key shared between the client and $\mathcal{S}_i$
   $\langle N, N_{min}, N_{max}\rangle$ : marker distribution strategy
   $C_{\text{twin}}$ : twinning condition
   $f$ : assignment function of the computational cloud
   $W$ : workers of the MapReduce computational cloud */
```

CLIENT

```
/* Phase 1: retrieve sub-joins */
$q_1$ := "SELECT $I_{12}$ FROM $B_1$ WHERE $C_1$"
for $i = 1, \ldots, n-1$ do
    $j := i+1$
    if $i \neq 1$ then
        $h := i-1$
        $q_i$ := "SELECT $I_{ij}$ FROM $B_i$ WHERE $I_{h,i} \in JI_{h,i}$"
    $q_j$ := "SELECT $I_{ij}$ FROM $B_j$ WHERE $C_j$"
    pick a key $k$
    toi := Encrypt($q_i$,$\langle N, N_{min}, N_{max}\rangle$,$C_{\text{twin}}$,$k$,qid) with $k_i$
    toj := Encrypt($q_j$,$\langle N, N_{min}, N_{max}\rangle$,$C_{\text{twin}}$,$k$,qid) with $k_j$
    send "SELECT $*$ FROM toi NATURAL JOIN toj" to computational cloud
    receive $JI^{*k}_{ij}$ from computational cloud
    $JI^*_{ij}$ := Decrypt($JI^{*k}_{ij}$) with $k$
    $JI_{ij}$ := Check_Integrity($JI^*_{ij}$, $\langle N, N_{min}, N_{max}\rangle$, $C_{\text{twin}}$)
    if $JI_{ij}$ = NULL then
        return "integrity error"
/* Phase 2: build final result */
Res := $\emptyset$
for $i = n, \ldots, 1$ do
    if $i = n$ then
        send ("SELECT $Att_n$ FROM $B_n$ WHERE $I_{(n-1)n} \in JI_{(n-1)n}$",qid) to $\mathcal{S}_n$
    elseif $i = 1$ then
            send ("SELECT $Att_1$ FROM $B_1$ WHERE $I_{12} \in P_2.I_{12}$",qid) to $\mathcal{S}_1$
    else
        $j := i+1; h := i-1$
        send ("SELECT $Att_i$ FROM $B_i$ WHERE $I_{ij} \in P_j.I_{ij}$",qid) to $\mathcal{S}_i$
    receive $P_i$ from $\mathcal{S}_i$
    Res := merge Res with $P_i$
Res := Evaluate "SELECT $A$ FROM Res WHERE $C_{ij}$"
return Res
```

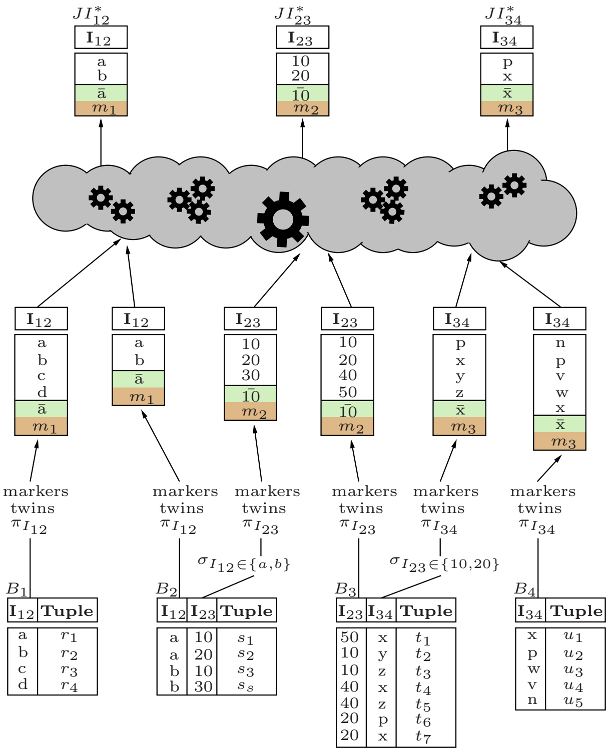
COMPUTATIONAL CLOUD

```
receive "SELECT $*$ FROM toi NATURAL JOIN toj" from the client
send toi to $\mathcal{S}_i$
send toj to $\mathcal{S}_j$
receive $BI^{*k}_i$ from $\mathcal{S}_i$
receive $BI^{*k}_j$ from $\mathcal{S}_j$
for each $w \in W$
    assign $L_w$ := $\{\tau \in BI^{*k}_i : f(\tau[I^k]) = w\}$ to worker $w$
    assign $R_w$ := $\{\tau \in BI^{*k}_j : f(\tau[I^k]) = w\}$ to worker $w$
    $w$ computes $Res_w$ := Evaluate "SELECT $*$ FROM $L_w$ NATURAL JOIN $R_w$"
$JI^{*k}$ := $\bigcup\{Res_w : w \in W\}$
send $JI^{*k}$ to the client
```

STORAGE SERVER $\mathcal{S}_i$

```
/* Phase 1: contribute information to compute sub-joins */
receive toi from the computational cloud
($q_i$,$\langle N, N_{min}, N_{max}\rangle$,$C_{\text{twin}}$,$k$,qid) := Decrypt(toi) with $k_i$
    where $q_i$ = "SELECT $I$ FROM $B_i$ WHERE Condition"
if $B_i^{qid}$ = NULL then $B_i^{qid}$ := $B_i$
$B_i^{qid}$ := Evaluate "SELECT * FROM $B_i^{qid}$ WHERE Condition"
$BI$ := Evaluate "SELECT DISTINCT $I$ FROM $B_i^{qid}$"
$M$ := Generate_Markers($N$,$N_{min}$,$N_{max}$)
$T$ := Generate_Twins($BI$, $C_{\text{twin}}$)
$BI^*$ := $BI \cup M \cup T$
$BI^{*k}$ := Encrypt($BI^*$) with $k$
send $BI^{*k}$ to computational cloud
/* Phase 2: contribute information for final result */
receive ($q$,qid) from the client
$BJ$ := Evaluate $q$ over $B_i^{qid}$
send $BJ$ to the client
```

Figure 6. Pseudocode of the algorithms executed by the client, the computational cloud, and the storage servers for the evaluation of query q



$\mathbf{I}_{12}$	Tuple
a	r_1
b	r_2
c	r_3
d	r_4

$\mathbf{I}_{12}$	$\mathbf{I}_{23}$	Tuple
a	10	s_1
a	20	s_2
b	10	s_3
b	30	s_s

$\mathbf{I}_{23}$	$\mathbf{I}_{34}$	Tuple
50	x	t_1
10	y	t_2
10	z	t_3
40	x	t_4
40	z	t_5
20	p	t_6
20	x	t_7

$\mathbf{I}_{34}$	Tuple
x	u_1
p	u_2
w	u_3
v	u_4
n	u_5

Figure 7. An example of phase 1 of a join sequence execution

is asked to provide its tuples, and its result P_i is considered for producing the result. In particular, the client asks to $\mathcal{S}_4$ the tuples with $I_{34} \in JI_{34}$, obtaining P_4; to $\mathcal{S}_3$ the tuples with $I_{23} \in JI_{23}$ and $I_{34} \in P_4.I_{34}$ (i.e., $\{p$,$x\}$), obtaining P_3; to $\mathcal{S}_2$ the tuples with $I_{12} \in JI_{12}$ and $I_{23} \in P_3.I_{23}$ (i.e., $\{20\}$), obtaining P_2; to $\mathcal{S}_1$ the tuples with $I_{12} \in P_2.I_{12}$ (i.e., $\{a\}$).

V. Analysis

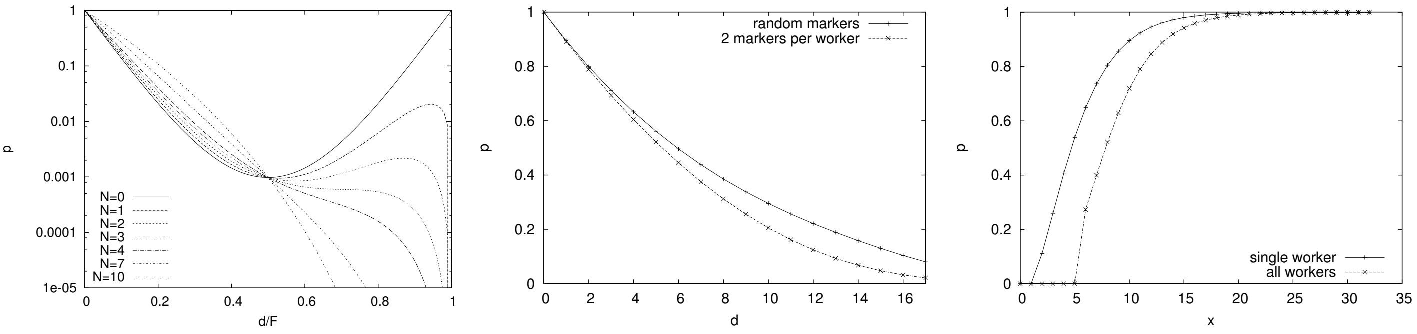
We discuss the integrity guarantees offered by our approach and the effectiveness of markers and twins for detecting violations.

Integrity controls and accountability for violations. Integrity of the join result implies guaranteeing integrity of individual tuples and that the join result is complete. As for individual tuples, tampering of data or the matching of tuples that do not join can be trivially detected given the use of encryption, for the first, and by the fact that the client would see the non matching plaintext values, for the second. Twins and markers allow assessing (with probabilistic guarantees) the completeness of the join result. The following theorem proves that for a join using N markers and a twinning condition C_{twin}: *1)* if the join has been computed correctly, the client will see in the result N markers and all tuples satisfying C_{twin} in pairs; *2)* for any missing marker/twin, the client can recognize the worker that was responsible for producing it. (While identifying the worker, the failure for a missing marker/twin could also be due to a misbehavior of the manager, which is then always co-responsible. Also, a manager not correctly enforcing the assignment function produces failures distributed over all workers.)

Theorem 5.1: Let M be the set of markers used for a query, C_{twin} be the twinning condition, and JI^* be the relation computed by the computational cloud. *1)* If the computational cloud behaves correctly: *i)* $M \subseteq JI^*$, and *ii)* $\forall t \in JI^*$ satisfying C_{twin}, $\bar{t} \in JI^*$. *2)* If $\exists m \in M$ s.t. $m \notin JI^*$ or $\exists t \in JI^*$ s.t. t satisfies C_{twin} and $\bar{t} \notin JI^*$, the client can identify the worker(s) responsible for the missing tuple(s).

Effectiveness of markers and twins. We start by considering the basic level of protection offered by N markers and Z twins [1]. Given a number d of dropped tuples over a total number F (containing the original tuples, N markers and Z twins), the probability that the markers are maintained in the result will be[1] $p_m = (1 - d/F)^N$. The probability of keeping the twins consistent, either presenting both a tuple and its twin in the result or dropping both, is $p_t = ((1-d/F)^2+(d/F)^2)^Z$. These formulas show that twins and markers have a complementary structure: markers are half as effective as twins for low values of d/F, but twins lose their effectiveness when considering large values of d/F. Following these formulas, we aim at characterizing what is a good configuration (w.r.t. integrity guarantees offered vs. cost to be paid) in terms of number of markers and twins to adopt. We consider as cost the number of tuples that are added to the result, be them markers or twins. Figure 8(a) shows the probability $p = p_m \cdot p_t$ that omissions of the computational cloud go undetected for configurations with 10 tuples added, varying the number N of markers and $10 - N$ of twins. Clearly, the configuration that uses only twins ($N = 0$) is inadequate, as it is not able to detect complete omissions (i.e., with the computational cloud simply producing an empty result). The configurations with a low number of markers ($N = 1$ and $N = 2$) also show an inversion of the detection probability, with higher chance for the computational cloud of not being detected when increasing the dropping rate. Preferable configurations are those that have an always negative derivative ($N \geq 3$). We note that in Figure 8(a) an increase in the value of N leads to configurations that have lower values of p for higher dropping rates, when p is already quite low, and have higher values of p for configurations with $d/F < 0.5$. We consider preferable the configurations that, as far as the derivative for p is negative for the whole range of d/F, minimize p when $d/F < 0.5$. In other words, considering 10 added tuples, the configuration with 3 markers and 7 twins is the preferred one. We also note that while markers are guaranteed to belong to the join result, the presence of twins depends on the actual instance of the underlying relations (i.e., how many original tuples satisfy C_{twin}). This suggests that the best strategy is to identify a number of markers that is able to provide adequate guarantee for the detection of omissions for large values of d/F, then dedicating all the remaining resources to the introduction of twins. While, for simplicity, we presented our reasoning for small configurations (where the probability of a misbehavior not being detected may appear not negligible), we note that such a probability is really negligible for configurations expected in real life scenarios. For instance, applying our formula, we can observe that using 50 markers and twinning 5% of the tuples, a computational cloud that omits 50 tuples will be detected with probability greater than

[1] For small configurations a less readable but more precise formula is $p_m = F! \cdot (F+N-d)!/((F-d)! \cdot (F+N)!)$, because each tuple removed from the set has an impact on the probability.



(a) (b) (c)

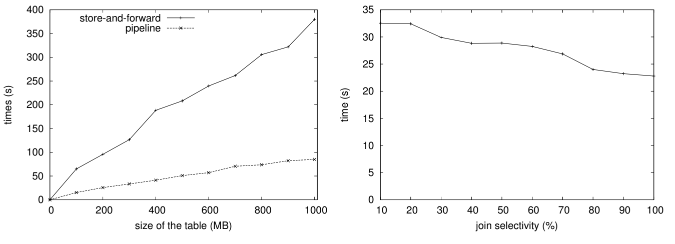
Figure 8. Probability p that the computational cloud omits a fraction d/F of tuples without being detected (a), probability p that the computational cloud omits d tuples from a single worker without being detected (b), and probability p that workers receive 2 markers each, increasing the number of generated markers (c)

99% for any cardinality of the input relations (e.g., 50 out of 10^4 tuples or 50 out of 10^9 tuples), with a low cost.

Marker distribution strategy. We want to assess the effectiveness of the different marker distribution strategies in terms of integrity guarantees. We then examine the impact of the distribution of markers on the ability of the client to detect violations (i.e., of markers being missing) when one worker fails. We assume to have $|W|$ workers and the join with F tuples plus N markers. If the distribution of tuples and markers derives from the application of a pseudorandom function, we can expect them to be distributed to the workers following a binomial distribution: the probability p_j of having j of the x (with x either F or N) tuples assigned to worker w will be $p_j = \binom{x}{j} \cdot (1/|W|)^j \cdot ((|W|-1)/|W|)^{x-j}$. For instance, if we have 51 tuples and 3 workers, the probability that worker w_i has 17 tuples will be $p_i^{17} = \binom{51}{17} \cdot (1/3)^{17} \cdot (2/3)^{34} = 0.118$. The probability to avoid detection for a worker that has received n_w markers and dropped d tuples would then be $p_n = (1 - d/F)^{n_w}$; with p_n assuming larger values for lower values of n_w (with greatest risk represented by workers that receive zero or one marker). Imposing a lower bound on the number of markers that each worker should receive (with an at-least-n or a uniform distribution strategy) avoids such risky configurations. Figure 8(b) compares, for a configuration with $|W| = 3$ workers, $F = 51$ tuples and $N = 6$ markers, the probability of a worker that has received n_w markers to go undetected varying the number d of dropped tuples, when using a random distribution strategy and a uniform distribution strategy. We note that imposing a lower bound on the number of markers to be assigned to every worker has little impact on the evaluation cost of function Generate_Markers in Figure 4 (which needs to generate and check markers until the strategy is satisfied), even for the (stricter) uniform distribution. Figure 8(c) shows the probability for a worker (upper curve) or all workers (lower curve) to receive at least two markers after x markers are generated in our example. The curve confirms that the generation of markers quickly identifies a uniform distribution (this behavior is maintained also for larger configurations).

VI. Experimental results

To verify the working and applicability of our approach we developed a prototype in Java that implements the storage servers, the computational cloud, and the client, operating according to the pseudocode in Figure 6. We developed the approach for two MapReduce frameworks, Hadoop (the most popular – http://hadoop.apache.org) and Apache Storm (supporting real-time processing of streams in a parallel and distributed environment – http://storm.incubator.apache.org), and used the latter for additional experiments noting a better performance (the worse performance of Hadoop was due to the fact that it orders the $\langle$*key*,*value*$\rangle$ pairs it receives before sending them to workers). We tested the system using a machine with 2 Intel Xeon E5-2630 and 64GB RAM. Figure 9 illustrates some data extracted from the evaluation of a join between two relations: one of size 100 MB and the other one of size varying between 100 MB and 1 GB. Each tuple of the two relations sent by the storage servers was represented by a 16-byte block encrypted with AES-128. The tested configurations considered: $N = 100$ markers, a condition C_{twin} twinning 5% of the tuples in each relation, a 10% selectivity of the join (i.e., 10% of the tuples in the relations appear in the join result), and 20 workers. Figure 9(a) illustrates the execution time, varying the size of the second relation, in case of a *pipeline approach*, with workers processing inputs as they arrive (exploiting a local hash table where input not yet joined is maintained) and sends outputs to the manager as they are produced, and in case of a *store-and-forward approach*, with workers performing the join and sending results only when the input flow from the storage servers has terminated. As expected, the pipeline approach exhibits better performance, completing the join between a relation of 1 GB and one of 100 MB (the largest configuration) in less than 100 seconds. Figure 9(b) shows the effect of the selectivity of the join on the execution time. Keeping all the other parameters as above, considering the pipeline approach, we then varied the



(a) (b)

Figure 9. Execution time for a join with a pipeline and a store-and-forward approach (a), and execution time with a varying join selectivity (b)

join selectivity between 10% and 100%. Contrary to what one could have expected, a larger selectivity (meaning a larger number of tuples in the join result) leads to a decrease of the execution time. The reason for this behavior is that, for less selective joins, workers will often find matching tuples. Tuples for which a match is found are removed from the local hash table. Such tables then remain relatively slim hence providing more efficiency. The reduction in the execution time that we observe with the increase of the join selectivity is a clear indication of the critical role of the join operation.

VII. Related work

Previous related work falls in the area of security and privacy in emerging outsourcing and cloud scenarios (e.g., [2]–[6]). Research on data integrity focused on solutions for providing query results with guarantees of: *i)* correctness, traditionally provided through digital signature techniques (e.g., [7]); *ii)* completeness, which can be provided through authenticated data structures (e.g., [8]–[10]) or through probabilistic approaches (e.g., [1], [11], [12]); and *iii)* freshness, traditionally provided by making the integrity verification structure time-dependent (e.g., [9]). Our proposal has some affinity with the approaches in [1], [11]–[13]. In fact, the solution in [11] replicates the tuples that satisfy a certain condition, which then must appear twice in the query result. The approach in [12] instead includes fake tuples, according to a deterministic generation function, into the outsourced dataset and that should then selectively belong to the query result. These proposals, however, do not combine the two techniques and support operations involving only one relation with limited consideration of joins. The combined adoption of twins and markers to guarantee the integrity of join operations has first been proposed in [1], and enriched in [13] with the support of a semi-join evaluation strategy. These approaches, while effectively and efficiently managing one-to-one joins, introduce communication overheads in the evaluation of one-to-many joins and do not support many-to-many joins and join sequences. Other works for assessing integrity of query results have investigated hardware-based solutions, assuming the adoption of tamper-proof trusted co-processors (e.g., [14]). Other related work is represented by studies on the integrity of MapReduce computations (e.g., [15]–[17]). These proposals, however, are not directly applicable to our scenario because they either operate on specific computations (e.g., word-based computations such as count and grep [15]), or rely on the presence of trusted components for the verification of the computed results (e.g., [16], [17]). A different, though related, line of works considered the efficient evaluation of joins with MapReduce, which are specifically focused on the scalability issues arising when processing large data collections (e.g., [18], [19]). Our proposal is complementary to these solutions, as it provides integrity guarantees of the join result independently from the technique used for join evaluation.

VIII. Conclusions

We presented an approach for enabling clients to control and assess the integrity of join computations performed by an untrusted computational cloud, agains which confidentiality of the data is also protected. We believe the work to provide a step forward towards allowing users to fully take advantage of the wide and rich offer of cloud computing services without compromising on security guarantees, then also enabling a wider adoption of cloud solutions.

Acknowledgments

The authors would like to thank Enrico Bacis for support in the implementation of the system and in the experimental evaluation. This work was supported in part by: the EC within the 7FP under grant agreement 312797 (ABC4EU); the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH); and the NSF under grant IIP-1266147.

References

[1] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Integrity for join queries in the cloud," *IEEE TCC*, vol. 1, no. 2, pp. 187–200, July-Dec. 2013.

[2] P. Samarati and S. De Capitani di Vimercati, "Data protection in outsourcing scenarios: Issues and directions," in *Proc. of ASIACCS 2010*, Beijing, China, Apr. 2010.

[3] K. Ren, C. Wang, and Q. Wang, "Security challenges for the public cloud," *IEEE IC*, vol. 16, no. 1, pp. 69–73, Jan.-Feb. 2012.

[4] C. Wang, Q. Wang, K. Ren, N. Cao, and W. Lou, "Toward secure and dependable storage services in cloud computing," *IEEE TSC*, vol. 5, no. 2, pp. 220–232, Jan. 2012.

[5] C.-K. Chu, W.-T. Zhu, J. Han, J. Liu, J. Xu, and J. Zhou, "Security concerns in popular cloud storage services," *IEEE Pervasive Computing*, vol. 12, no. 4, pp. 50–57, Oct.-Dec. 2013.

[6] R. Jhawar, V. Piuri, and P. Samarati, "Supporting security requirements for resource management in cloud computing," in *Proc. of CSE 2012*, Paphos, Cyprus, Dec. 2012.

[7] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," *ACM TOS*, vol. 2, no. 2, pp. 107–138, May 2006.

[8] H. Pang and K. Tan, "Verifying completeness of relational query answers from online servers," *ACM TISSEC*, vol. 11, no. 2, pp. 5:1–5:50, May 2008.

[9] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in *Proc. of SIGMOD 2006*, Chicago, IL, June 2006.

[10] Y. Yang, D. Papadias, S. Papadopoulos, and P. Kalnis, "Authenticated join processing in outsourced databases," in *Proc. of SIGMOD 2009*, Providence, RI, June-July 2009.

[11] H. Wang, J. Yin, C. Perng, and P. Yu, "Dual encryption for query integrity assurance," in *Proc. of CIKM 2008*, Napa Valley, CA, Oct. 2008.

[12] M. Xie, H. Wang, J. Yin, and X. Meng, "Integrity auditing of outsourced data," in *Proc. of VLDB 2007*, Vienna, Austria, Sept. 2007.

[13] S. De Capitani di Vimercati, S. Foresti, S. Jajodia, S. Paraboschi, and P. Samarati, "Optimizing integrity checks for join queries in the cloud," in *Proc. of DBSec 2014*, Vienna, Austria, July 2014.

[14] S. Bajaj and R. Sion, "CorrectDB: SQL engine with practical query authentication," *PVLDB*, vol. 6, no. 7, pp. 529–540, May 2013.

[15] C. Huang, S. Zhu, and D. Wu, "Towards trusted services: Result verification schemes for MapReduce," in *Proc. of CCGrid 2012*, Ottawa, Canada, May 2012.

[16] W. Wei, J. Du, T. Yu, and X. Gu, "SecureMR: A service integrity assurance framework for MapReduce," in *Proc. of ACSAC 2009*, Honolulu, HI, Dec. 2009.

[17] Y. Wang and J. Wei, "VIAF: Verification-based integrity assurance framework for MapReduce," in *Proc. of CLOUD 2011*, Washington, DC, July 2011.

[18] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MapReduce," in *Proc. of SIGMOD 2010*, Indianapolis, IN, June 2010.

[19] A. Thusoo *et al.*, "Hive: A warehousing solution over a MapReduce framework," in *Proc. of VLDB 2009*, Lyon, France, Aug. 2009.