# Optimizing Integrity Checks for Join Queries in the Cloud

Sabrina De Capitani di Vimercati[1], Sara Foresti[1], Sushil Jajodia[2], Stefano Paraboschi[3], and Pierangela Samarati[1]

[1] Università degli Studi di Milano – 26013 Crema, Italy
{firstname.lastname}@unimi.it
[2] George Mason University – Fairfax, VA 22030-4444
jajodia@gmu.edu
[3] Università di Bergamo – 24044 Dalmine, Italy
parabosc@unibg.it

**Abstract.** The large adoption of the cloud paradigm is introducing more and more scenarios where users can access data and services with an unprecedented convenience, just relying on the storage and computational power offered by external providers. Also, users can enjoy a diversity and variety of offers, with the possibility of choosing services by different providers as they best suit their needs. With the growth of the market, economic factors have become one of the crucial aspects in the choice of services. However, security remains a major concern and users will be free to actually benefit from the diversity and variety of such offers only if they can also have proper security guarantees on the services. In this paper, we build upon a recent proposal for assessing integrity of computations performed by potentially untrusted providers introducing some optimizations, thus limiting the overhead to be paid for integrity guarantees, and making it suitable to more scenarios.

## 1 Introduction

The competitive pressures are driving the IT sector away from the classical model that assumed the processing and storage of an organization data within the internal information system, toward the use of storage and processing capabilities offered by providers, which can benefit from economies of scale deriving from the large size of the infrastructure and service catalogue, together with possible access to less expensive resources. Along this line, we can expect a continuous increase in the differentiation of the market for cloud services. For instance, in the area of cloud architectures, interest has emerged on hybrid clouds and on a distinction between cloud storage and cloud computational services. Storage and computational services respond in fact to separate requirements, with distinct profiles. The first should offer reliability for data storage, typically corresponding to providers with high reputation on the market. The second should offer availability of – possibly cheap – computational power, which can be offered by unknown providers. Reputation of the provider is, in this case, less critical,

V. Atluri and G. Pernul (Eds.): DBSec 2014, LNCS 8566, pp. 33–48, 2014.

as it is relatively easy to move computation from one provider to another, and the most important parameter becomes the price of the service. An obstacle to a stronger differentiation in the market between storage and computational resources is however represented by the security concerns of users, who can see the involvement of multiple parties in the processing of their information as increasing the risk of confidentiality and integrity violations.

In this paper, we present an approach for users to protect confidentiality of processed data and to assess the integrity of computations performed by potentially untrusted computational providers, operating over data stored at trusted storage providers. Our approach builds upon a recent proposal [5], aimed at controlling the behavior of a computational provider that joins data stored at independent trusted storage servers. We address the problem of optimizing integrity controls so to decrease their performance and economic overheads making them suitable to more scenarios and enabling their application with stronger integrity guarantees. In particular, we introduce two optimization techniques. The first technique (Sect. 3) exploits the execution of the join with a semi-join strategy, hence possibly decreasing data communication and consequent performance/economic costs, while leaving unaltered the offered guarantees. The second technique (Sect. 4) limits the application of the integrity checks to a small portion of the data, producing a considerable saving in terms of performance and economic cost, though at the price of a reduced integrity guarantee. The two optimizations are independent and orthogonal and can be used individually or in combination (Sects. 5 and 6).

## 2 Scenario and Basic Concepts

We present the basic idea of the approach on which we build our optimization techniques. The scenario is characterized by a client that wishes to evaluate a query involving a join over two relations, $B_l$ and $B_r$, stored at storage servers $\mathcal{S}_l$ and $\mathcal{S}_r$, respectively, by using a computational server $\mathcal{C}_s$. The storage servers are assumed to be trustworthy while the computational server is not. The query is 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 $A$ is a subset of 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. Typically, execution of such a query involves in pushing down, to each of the storage servers, the evaluation of the condition ($C_l$ and $C_r$) on its own relation. We assume that, regardless of the degree of the original schema, relations $L$ and $R$ resulting from the evaluation of $C_l$ and $C_r$, respectively, have schema ($I$, $Attr$), where $I$ and $Attr$ represent the set of join attributes and all the other attributes, respectively, as a unit. Without security concerns, relations $L$ and $R$ are then sent to the computational server, which performs the join, evaluates condition $C_{lr}$ and returns the result to the client. Since the computational server is not trusted, the proposal in [5]: *i)* provides data confidentiality by encrypting on the fly the relations sent to the computational server, with a key communicated by the client to the storage servers, *ii)* provides integrity guarantees by using a combination of controls as follows:

- *markers*: each of the storage servers inserts fake control tuples (markers), not recognizable by the computational server, in the relation to be sent to the computational server. Markers are inserted so to join (i.e., belong to the result) and to not collide with real join attribute values (to not create spurious joined tuples).
- *twins*: each of the storage servers duplicates (twins) some of the tuples in its relation before sending it to the computational server. The creation of twins is easily controlled by the client by specifying a percentage of tuples to be twinned and a condition for twinning.
- *salts/buckets*: used in alternative or in combination to destroy recognizable frequencies of combinations in one-to-many joins. Salts consist in salting the encryption at side "many" of the join so that occurrences of a same value become distinct; at the same time salted replicas are created at side "one" of the join so to create the corresponding matching. Bucketization consists in allowing multiple occurrences of the same (encrypted) value at the side many of the join, but in such a way that all the values have the same number of occurrences. Bucketization can help in reducing the number of salts to be inserted, while possibly requiring insertion of dummy tuples (to fill otherwise not complete buckets).

Join computation, illustrated in Fig. 1(a), works now as follows. Each storage server receives in encrypted form its sub-query, together with the key to be used to encrypt the sub-query result, and the needed information to regulate the use of markers, twins, salts and buckets. It then executes the received sub-query (as before), and applies over the resulting relation $L$ ($R$, resp.) markers, twins, salts and buckets as appropriate, producing a relation $L^*$ ($R^*$, resp.). Relation $L^*$ ($R^*$, resp.) is then encrypted producing relation $L^*_k$ ($R^*_k$, resp) to be sent to the computational server. Encrypted relation $L^*_k$ ($R^*_k$, resp.) contains two encrypted chunks for each tuple: $L^*_k.I_k$ ($R^*_k.I_k$, resp.) for the join attribute, and $L^*_k.Tuple_k$ ($R^*_k.Tuple_k$, resp.) for all the other attributes (including the join attribute). The computational server receives the encrypted relations from the storage servers and performs the join returning the result $J^*_k$ to the client. The client receives the join result, decrypts it, checks whether the tuples have been correctly joined (i.e., $L^*.I$ obtained decrypting $L^*_k.Tuple_k$ is equal to $R^*.I$ obtained decrypting $R^*_k.Tuple_k$), and discards possible tuples with dummy content. Then, it checks integrity by analyzing markers and twins: an integrity violation is detected if an expected marker is missing or a twinned tuple appears solo. Note that the combined use of markers and twins offers strong protection guarantees. In fact, when omitting a large number of tuples in query results, the probability that the omission goes undetected increases with respect to twins, and decreases with respect to markers (e.g., one marker is sufficient to detect that an empty result is not correct), and vice versa. Figure 1(b) illustrates an example of relations $L$ and $R$ and of their extensions obtained by assuming: the presences of one marker (with value $x$ for the join attribute), twinning tuples with join attribute equal to $b$, and adopting 2 salts and buckets with 2 tuples each. Figure 1(c) reports

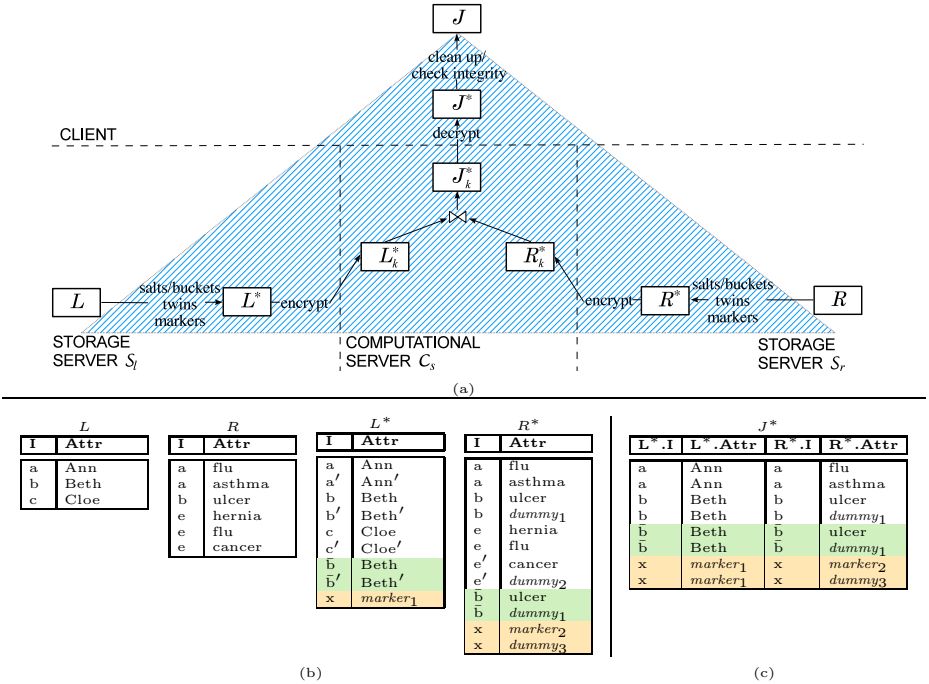



**Fig. 1.** Join computation as a regular join (a) and an example of relations $L$ and $R$ and their extensions with markers, twins, salts and buckets (b) along with the join computed over them (c)

the join result $J^*$ obtained by the client decrypting relation $J_k^*$ received from the computational server.

## 3 Semi-join

The first optimization we illustrate consists in performing the join according to a semi-join strategy. Without security concerns, a semi-join simply implements a join operation by first considering only the projection of the join attribute over the stored relations. Only after the join is computed, the join attribute is extended with the other attributes from the source relations to produce the final result. In our distributed setting, semi-joins – while requiring additional data flows – avoid communication of unnecessary tuples to the client and of non-join attributes to the computational server, producing a saving of the total communication costs for selective joins and/or relations with tuples of considerable size (see Sect. 6). Our approach for executing a semi-join in conjunction with the security techniques illustrated in the previous section works as follows. The execution of the join at the computational server basically works as before: it again receives from the storage servers encrypted relations on which markers, twins,

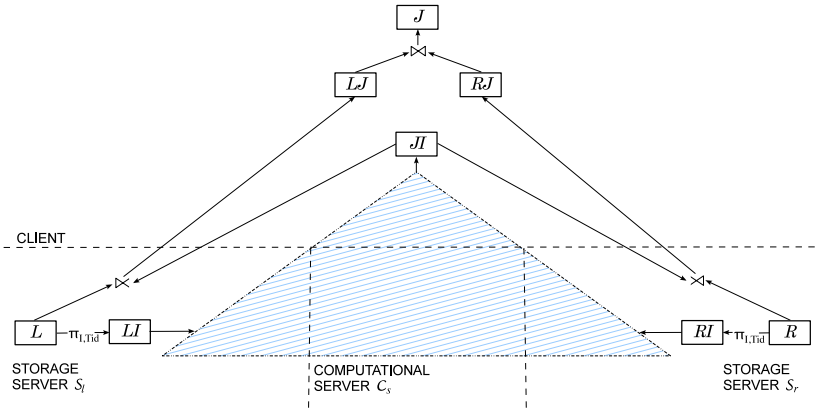



**Fig. 2.** Join execution as a semi-join

salts and buckets have been applied; it computes the join between them; and it sends the result to the client. However, in this case:

- the storage servers do not communicate to the computational server their entire tuples (relation $L$) but rather much slimmer tuples (relation $LI$) with only the join attribute and the tuple identifier (both in encrypted form). The tuple identifier ($Tid$) is needed to keep tuples with the same value for the join attribute distinct.
- after checking/cleaning the result of the join (relation $JI$), the client asks the storage servers to complete the tuples in the join with the attributes in $Attr$ in their relations (obtaining relations $LJ$ and $RJ$), and combines their results.

The join process is illustrated in Fig. 2, where the striped triangle corresponds to the process in Fig. 1(a).

Note that, while entailing more flows (the previous process is a part of this), the semi-join execution limits the transfer of non-join attributes, thus reducing the amount of data transferred. Note also that while the storage servers and the client are involved in the execution of some computation to combine tuples, this computation does not entail an actual join execution but rather a simple scan/merging of ordered tuples that then requires limited cost.

Figure 3 illustrates an example of join computation over relations $L$ and $R$ in Fig. 1(b) according to the semi-join strategy. The semi-join execution strategy leaves unchanged the integrity guarantees offered by our protection techniques. In fact, the join computed by the computational server relies on the same protection techniques used for the computation of a regular join, and the storage servers are assumed to correctly evaluate the queries received from the client. The probability $\wp$ that the computational server omits $o$ tuples without being

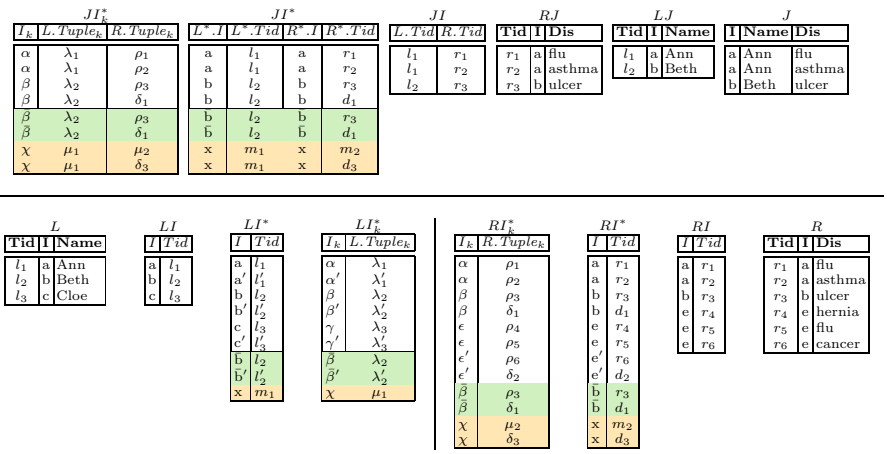



**Fig. 3.** An example of query evaluation process with twins on $b$, one marker, two salts, and buckets of size two

detected is then the same of regular joins, that is, $\wp = (1-\frac{o}{f})^m \cdot (1-2\frac{o}{f}+2(\frac{o}{f})^2)^t \approx e^{-2\frac{t}{f}o}$, where $f$ is the cardinality of relation $J^*$ with $t$ twin pairs and $m$ markers [5]. In fact, the probability that no marker is omitted is $(1-\frac{o}{f})^m$, while the probability that, for each twin pair, either both tuples are omitted or both are preserved is $(1-2\frac{o}{f}+2(\frac{o}{f})^2)^t$. This probabilistic analysis has also been confirmed by experimental analysis [5].

## 4 Limiting Salts and Buckets to Twins and Markers

The overhead caused by the adoption of our protection techniques is mainly due to salts and buckets [5], which are used to protect the frequency distribution of the values of the join attribute in case of one-to-many joins. In the following discussion, we refer to the execution of the join according to the process described in Sect. 2. Let $s$ be the number of salts and $b$ be the size of buckets defined by the client. Relation $L^*$ includes $s$ copies of each original tuple in $L$ and of each twin. Relation $R^*$ instead includes $b$ tuples for each marker (one marker and $(b-1)$ dummy tuples) and between 0 and $(b-1)$ dummy tuples for each value of the join attribute appearing in $R$ and in the twinned tuples. Hence, also the join result $J^*$ will have $b$ tuples for each marker and between 0 and $(b-1)$ dummy tuples for each original and twinned value of the join attribute. For instance, with respect to the example in Fig. 1(b), the adoption of our protection techniques causes the presence of six additional tuples in $L^*$ and $R^*$, and five additional tuples in $J^*$.

The second optimization we propose aims at limiting the overhead caused by the adoption of salts and buckets by applying them only to twins and markers rather than to the whole relations. Twins and markers (properly bucketized

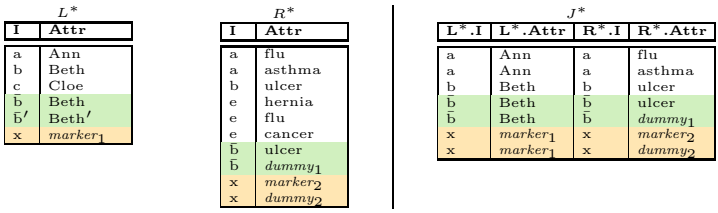
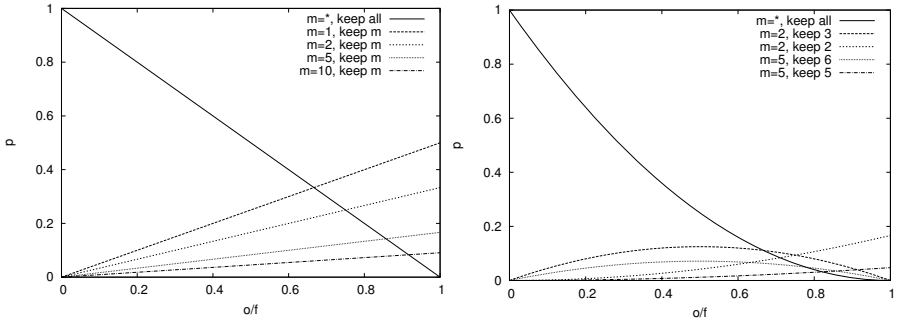



**Fig. 4.** An example of extensions of relations $L$ and $R$ in Fig. 1(b) and their join when salts and buckets are limited to twins and markers

and salted) would form a *Verification Object* (VO) that can be attached to the original (encrypted) relation. As an example, Fig. 4 illustrates the extended version of relations $L$ and $R$ in Fig. 1(b) where salts and buckets operate only on twins and markers. It is immediate to see that this optimization saves three tuples in $L^*$, two in $R^*$, and one in $J^*$. The strategy of limiting salts and buckets to twins and markers can be adopted in combination with both the regular and the semi-join strategies, reducing the computational overhead in both cases. While providing performance advantages, this strategy may reduce the integrity guarantee provided to the client. Let us consider a relation $J^*$, with $f$ original tuples, $t$ twin pairs, and $m$ markers. We examine the probability $\wp$ that the computational server omits $o$ original tuples without being detected. We build a probabilistic model considering the worst case scenario, assuming that the computational server: *i)* is able to recognize the tuples in VO (only the tuples in VO have a flat frequency distribution), *ii)* knows the number $m$ of markers in VO, *iii)* but cannot recognize which tuples in VO are twins and which are markers, or which of the original tuples have been twinned. We also consider all the tuples in a bucket as a single tuple. In fact, the computational server either preserves or omits buckets of tuples in their entirety as omissions of subsets of tuples in a bucket can always be detected. If the server omits $o$ tuples out of $f$, the probability for each twin to be omitted will be $\frac{o}{f}$. To go undetected, the server should provide a configuration of VO consistent with the $f - o$ returned tuples. There is only one such configuration, which contains a number of tuples between $m$ and $(m+t)$. The goal of the computational server is to maximize the probability of being undetected. We can model the behavior of the server considering two phases. In the first phase, the server determines the number of tuples that should belong to VO after the omission. Since there is a uniform and independent probability for the omission of twins, the number of expected tuples in VO follows a binomial distribution. This means that the probability that VO contains $(m+t)-k$ tuples is $\wp_{omit} = \binom{t}{k}(1-\frac{o}{f})^{t-k}(\frac{o}{f})^k$ (e.g., the probability that VO does not miss any tuple, $k = 0$, is $\wp_{omit} = (1 - \frac{o}{f})^t$). In the second phase, the server tries to guess the correct configuration including $(m + t) - k$ tuples. The number of such configurations depends on the number of missing tuples: if the server knows that $k$ of the $(m+t)$ tuples in VO are missing, the number of possible configurations is $\binom{m+t}{k}$, and the probability $\wp_{guess}$ of guessing it right




(a) (b)

**Fig. 5.** Probability $\wp$ that the server omits a fraction $\frac{o}{f}$ of the tuples without being detected, considering several strategies, assuming $t = 1$ (a) and $t = 2$ (b)

is the inverse of this quantity. For instance, if no tuple is omitted ($k = 0$), the server is certain of guessing the right configuration, whereas if all the twins have been omitted, the probability of randomly guessing the right configuration with exactly $m$ tuples is $1/\binom{m+t}{m}$, that is, $m! \cdot t!/(m+t)!$. The server can estimate the probability of a correct guess for each strategy by multiplying the first term with the second term, that is, $\wp = \wp_{omit} \cdot \wp_{guess}$. The server, to maximize the chance of not being detected, will have to choose the strategy that exhibits the greatest value. Since the second term exhibits an exponential growth with the increase in $k$, the server will typically prefer the strategy where no tuple in VO is omitted.

Figure 5 shows the result of the analysis for configurations with 1 or 2 twins and a variable number of markers. For the configuration with 1 twin in Fig. 5(a), we can see that the strategy that keeps all the elements in VO (independently from the number of markers) is preferable for a large range of omissions. When the number of markers increases, the cutoff between the selection of the strategy that tries to guess the tuple to omit from VO moves progressively to the right. A similar behavior characterizes the configuration with 2 twins described in Fig. 5(b), which shows that there is a range of values for $\frac{o}{f}$ where each configuration is preferable, but as the number of markers increases, the "keep all" strategy extends its benefit. The ability to avoid detection when omitting tuples becomes negligible when we consider configurations with the number of twins and markers that we expect to use in real systems.

As said above, it is convenient for the server to keep all the tuples in VO. In this case, the omission goes undetected if the server does not omit any original tuple that has been twinned. The probability $\wp$ for the server to go undetected is then equal to $\wp = (1 - \frac{o}{f})^t \approx e^{-\frac{t}{f} \cdot o}$. Figure 6 compares the results obtained when salts and buckets protect the whole relations ("full salts and buckets" in the figure) with the case where salts and buckets protect only VO ("VO-only salts and buckets" in the figure), assuming $\frac{t}{f}$=15%. We note that limiting salts and buckets to twins and markers offers roughly half the protection of applying salts and buckets to the whole relation. Intuitively, the client can obtain the

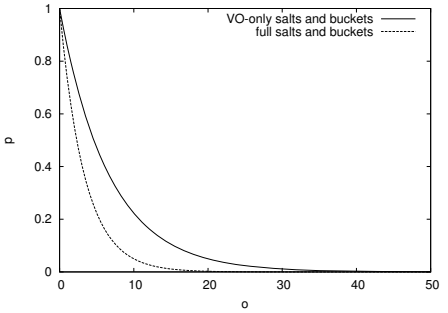



**Fig. 6.** Probability of the omission of $o$ tuples to go undetected when applying salts and buckets on the whole relations and on twins and markers only

same protection guarantee by doubling the ratio $\frac{t}{f}$ of twins. We note however that, even limiting salts and buckets to VO, the probability that the server is not detected when omitting more than 30 tuples is negligible and is independent from the number $f$ of tuples in the original relation. Since the computational server cannot selectively omit tuples from the join result (i.e., it cannot recognize tuples that have a twin), the advantage obtained from the omission of less than 30 tuples does not justify the risk of being detected in its omission.

## 5 Performance Analysis

We now evaluate the performance benefits obtained with the introduction of the semi-join strategy and the use of salts and buckets only on twins and markers. The performance costs depend on both the computational and communication costs. The computational costs are however dominated by the communication costs. In fact, the computational costs at the client and at the storage servers can be considered limited with respect to the computational costs at the computational server, which however can rely on a high amount of computational resources and on traditional techniques for optimizing join evaluation. In the following, we then focus our analysis first on the communication costs when evaluating a join as a regular join ($RegJ$) [5] or as a semi-join ($SemiJ$), according to the process described in Sect. 3. We then analyze the communication costs obtained when limiting salts and buckets to twins and markers.

**Semi-join vs Regular Join.** This analysis focuses on the amount of data exchanged among the involved parties (i.e., number of tuples transferred multiplied by their size). We first note that both the regular join and the semi-join require a common phase (Phase 1) where there is an exchange of data (tuples) between the storage servers and the computational server and between the computational server and the client. In this phase, regular join and semi-join differ in the size of the tuples transferred among the parties. The semi-join then requires an additional phase (Phase 2) where the client and the storage servers interact to compute the final join result. We analyze each of these two phases in details.

*Phase 1* [$\mathcal{S}_l$,$\mathcal{S}_r \rightarrow \mathcal{C}_s$; $\mathcal{C}_s \rightarrow$ Client]. As already discussed, the only difference between *SemiJ* and *RegJ* is the size of the tuples communicated, while the number of tuples in the join operands and in its result is the same for both strategies. *SemiJ* requires the transmission of the join attribute and of the tuple identifiers only, which are the two attributes forming the schema of relations $LI$ and $RI$. *RegJ* requires instead the transmission of all the attributes in $L$ and $R$. If the size of the tuples in $L$ and $R$ is higher than the size of the tuples in $LI$ and $RI$, *SemiJ* implies a lower communication cost than *RegJ*. Formally, the amount of data transmitted during this phase is:
*SemiJ*: $|L^*| \cdot size_L + |R^*| \cdot size_R + |J^*| \cdot (size_L + size_R)$
*RegJ*: $|LI^*| \cdot size_{IT} + |RI^*| \cdot size_{IT} + |JI^*| \cdot 2size_{IT}$
where $size_L$ is the size of the tuples in $L$, $size_R$ is the size of the tuples in $R$, $size_{IT}$ is the sum $size_I + size_{Tid}$, with $size_I$ the size of the join attribute $I$ and $size_{Tid}$ the size of the tuple identifier $Tid$. Since $LI^*$ ($RI^*$ and $JI^*$, resp.) has the same number of tuples as $L^*$ ($R^*$ and $J^*$, resp.), the difference in the communication cost is:
$|L^*| \cdot (size_L - size_{IT}) + |R^*| \cdot (size_R - size_{IT}) + |J^*| \cdot (size_L + size_R - 2size_{IT})$.
*Phase 2* [Client $\rightarrow \mathcal{S}_l$,$\mathcal{S}_r$; $\mathcal{S}_l$,$\mathcal{S}_r \rightarrow$ Client]. The number of tuples exchanged between the client and $\mathcal{S}_r$ is equal to the number of tuples resulting from the join computed by the computational server in the previous phase, after the removal of markers, twins, and dummies (i.e., $|JI|=|RJ|$). The number of tuples exchanged between the client and $\mathcal{S}_l$ depends on the type of join. In case of one-to-one joins, the number of tuples coincides with the number of tuples transmitted from the client to $\mathcal{S}_r$ (i.e., $|JI|=|LI|$). In case of one-to-many joins, the number of tuples is lower since the same tuple in $LI$ ($L$, resp.) may appear many times in the join result $JI$ ($J$, resp.). Assuming a uniform distribution of values, the number of different values for the join attribute in $RI$ is $\frac{2|RI|}{nmax}$. Given the selectivity $\sigma$ of the join operation, the number of different values for the join attribute in $JI$ is $\sigma \cdot \frac{2|RI|}{nmax}$, which corresponds to the number of tuples exchanged between the client and $\mathcal{S}_l$. The size of the tuples transmitted from the client to each storage server is $size_{Tid}$ since the client transmits only the values of the tuple identifier $Tid$. The size of the tuples transmitted from the storage servers to the client is equal to the size of the tuples in the original relations $L$ and $R$ (i.e., $size_L$ and $size_R$, resp.). Formally, the amount of data exchanged during this phase is:

*one-to-one join*: $2|JI| \cdot size_{Tid} + |JI| \cdot size_L + |JI| \cdot size_R$;
*one-to-many join*: $(|JI| + \sigma \cdot \frac{2|RI|}{nmax}) \cdot size_{Tid} + \sigma \cdot \frac{2|RI|}{nmax} \cdot size_L + |JI| \cdot size_R$.

By comparing the amount of data transmitted in Phase 2 with the additional amount of data transmitted in Phase 1 caused by the regular join, we note that the semi-join is convenient for relations with large tuples (i.e., $size_{IT} << size_L$ and $size_{IT} << size_R$), as also shown by our experimental analysis (Sect. 6). The advantage of the semi-join with respect to the regular join appears also more evident in case of one-to-many joins where a tuple in the left operand can appear many times in the join result (i.e., $|LJ| \approx \sigma \cdot \frac{2|RI|}{nmax} << |JI|$). In fact, with the semi-join strategy the client receives each tuple in $L$ that belongs to the final

result only once, while it receives many copies of the same tuple when adopting the regular join approach.

**Limiting Salts and Buckets to Twins and Markers.** The saving, in terms of communication cost, provided applying salts and buckets to markers and twins rather than to the whole relation can be computed by analyzing the difference in the number of tuples in $L^*$, $R^*$, and $J^*$. We analyze each relation in detail.

- $L^*$. Since only twin tuples are salted, we save the salted copies of the tuples in $L$, that is, $(s-1) \cdot |L|$ tuples.
- $R^*$. Since buckets operate only on twins and markers, we save the dummy tuples of the buckets formed with the tuples in $R$. Since for each value of the join attribute, there is at most one bucket with dummy tuples with, on average, $\frac{b-1}{2}$ dummy tuples, and there are $\frac{2|R|}{nmax}$ distinct values for the join attribute (again assuming a uniform distribution of values), we save $\frac{b-1}{nmax} \cdot |R|$ tuples.
- $J^*$. The join result contains the subset of the tuples in $R^*$ that combine with the tuples in $L^*$. The number of tuples saved in $J^*$ is then a fraction of the number of tuples saved in $R^*$, that is, $\sigma \cdot \frac{b-1}{nmax} \cdot |R|$.

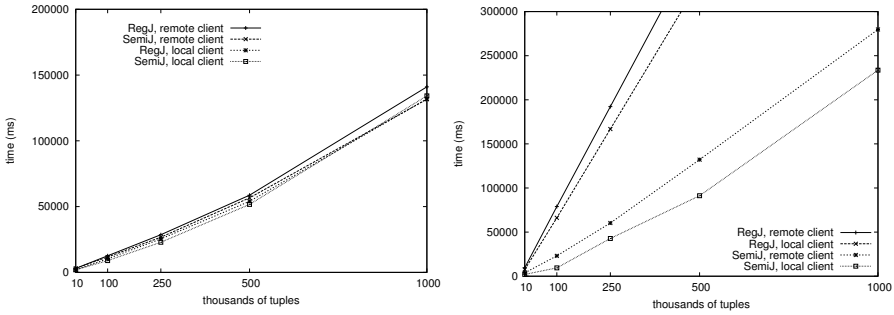
The overall advantage provided by limiting salts and buckets to twins and markers is: $(s-1) \cdot |L| \cdot size_L + \frac{b-1}{nmax} \cdot |R| \cdot size_R + \sigma \cdot \frac{b-1}{nmax} \cdot |R| \cdot (size_L + size_R)$.

## 6 Experimental Results

To assess the performance advantage of the semi-join strategy with respect to the regular join and of limiting salts and buckets to twins and markets, we implemented a prototype enforcing our protection techniques, and run a set of experiments. We used for the computational server a machine with 2 Intel Xeon Quad 2.0GHz, 12GB RAM. The client machine and the storage servers were standard PCs running an Intel Core 2 Duo CPU at 2.4 GHz, with 4GB RAM, connected to the computational server through a WAN connection with a 4 Mbps throughput. The values reported are the average over six runs.

**Regular Join vs Semi-join.** A first set of experiments was dedicated to the comparison between the regular join and the semi-join. The experiments also evaluated the impact of latency on the computation, comparing the response times for queries over local networks (local client configuration) with those obtained with a client residing on a PC at a distance of 1,000 Km connected through a shared channel that in tests demonstrated to offer a sustained throughput near to 80 Mbit/s (remote client configuration). The experiments used a synthetic database with two tables, each with between $10^4$ and $10^6$ tuples, with size equal to 30 and 2,000 bytes. We computed one-to-one joins between these tables, using 500 markers and 10% of twins. The results of these experiments are reported in Fig. 7.

The results confirm that the use of semi-join (*SemiJ* in the figure) gives an advantage with respect to regular join (*RegJ* in the figure) when the tuples




(a) tuples of 30 bytes (b) tuples of 2,000 bytes

**Fig. 7.** Response time for regular join and semi-join

have a large size, whereas the advantage becomes negligible when executing a join over compact tuples. This is consistent with the structure of the semi-join computation, which increases the number of exchanges between the different parties, but limits the number of transfers of non-join attributes. When the tuples are large, the benefit from the reduced transfer of the additional attributes compensates the increased number of operations, whereas for compact tuples this benefit is limited. The experiments also show that the impact of latency is modest, as the comparison between local client and remote client configurations of the response times for the same query shows a limited advantage for the local client scenario, consistent with the limited difference in available bandwidth. The results obtained also confirm the scalability of the technique, which can be applied over large tables (up to 2 GB in each table in our experiments) with millions of tuples without a significant overhead.

**Limiting Salts and Buckets to Twins and Markers.** A second set of experiments was dedicated to the analysis of the use of salts and buckets. The experiments considered a one-to-many join, evaluated as a regular join, over a synthetic database containing 1,000 tuples in both join operands. We tested configurations with at most 50 occurrences of each value, and used a number of salts $s$ varying between 1 and 100 and buckets of size $b{=}\lceil\frac{50}{s}\rceil$. The experiments evaluating the overhead of the protection techniques when salts and buckets are used only on markers and twins show that the overhead due to salts and buckets is proportional to the fraction of tuples that are twinned. For instance, if we add a 10% of twins, the overhead for salts and buckets will be one tenth of what we would have observed if applying the protection to all the tuples. Figure 8 compares the response time observed when executing the query without using our protection techniques ("base" in the figure), when using 50 markers and 15% of twins with salts and buckets on the whole table ("full salts and buckets" in the figure), and a configuration with 50 markers and 30% of twins with salts and buckets only on markers and twins ("VO-only salts and buckets" in the figure). The experiments confirm that the increase in response time represents a fraction $\frac{t}{f}$ (with $t$ the number of twins and $f$ the cardinality of the join result)

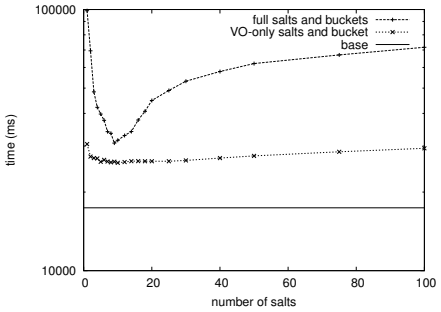



**Fig. 8.** Response time without adopting protection techniques, with salts and buckets on the whole relation, and with salts and buckets only on markers and twins

of the increase that would otherwise be observed using salts and buckets over all the tuples. The figure also shows that the overhead due to the adoption of our protection techniques is limited.

**Economic Analysis.** Besides the response time perceived by the user, the choice between regular and semi-join needs to take into consideration also the economic cost of each alternative. We focused on evaluating the economic advantage of the availability of the semi-join, besides regular join, strategy when executing queries [10]. In fact, in many situations the semi-join approach can be less expensive, since it entails smaller flows of information among the parties.

In our analysis, we assumed economic costs varying in line with available solutions (e.g., Amazon S3 and EC2, Windows Azure, GoGrid), number of tuples to reflect realistic query plans, and reasonable numbers for twins and markers. In particular, we considered the following parameters: *i)* cost of transferring data out of each storage server (from 0.00 to 0.30 USD per GB), of the computational server (from 0.00 to 0.10 USD per GB), and of the client (from 0.00 to 1.00 USD per GB); *ii)* cost of transferring data to the client (from 0.00 to 1.00 USD per GB);[1] *iii)* cost of CPU usage for each storage server (from 0.05 to 2.50 USD per hour), for the computational server (from 0.00 to 0.85 USD per hour), and for the client (from 1.00 to 4.00 USD per hour); *iv)* bandwidth of the channel reaching the client (from 4 to 80 Mbit/s); *v)* size $size_{IT} = size_I + size_{Tid}$ of the join attribute and the tuple identifier (from 1 to 100 bytes); *vi)* number of tuples in $L$ (from 10 to 1,000) and the size of the other attributes $size_L - size_{IT}$ (from 1 to 300 bytes); *vii)* number of tuples in $R$ (from 10 to 10,000) and the size of the other attributes $size_R - size_{IT}$ (from 1 to 200 bytes); *viii)* number $m$ of markers (from 0 to 50); *ix)* percentage $\frac{t}{f}$ of twins (from 0 to 0.30); *x)* number $s$ of salts (from 1 to 100); *xi)* maximum number $nmax$ of occurrences of a value in $R.I$ (from 1 to 100); *xii)* selectivity $\sigma$ of the join operation (from 0.30 to 1.00). Similarly to what is usually done in finance and economics to compare

[1] We did not consider the cost of input data for the storage and computational servers since all the price lists we accessed let in-bound traffic be free.

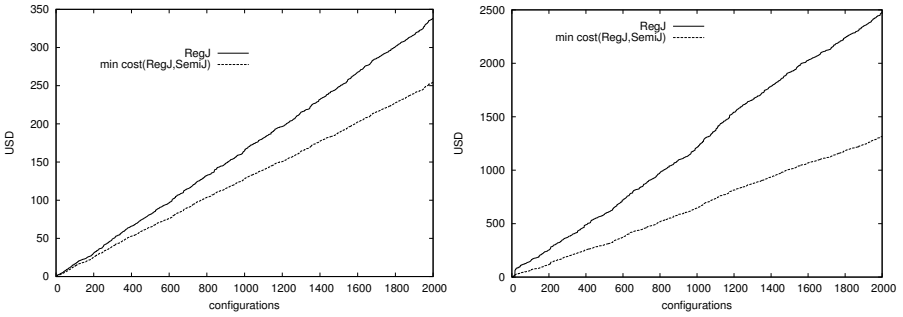



**Fig. 9.** Total economic cost of executing 2,000 one-to-one (a) and 2,000 one-to-many (b) join queries as a regular join or as the less expensive between regular and semi-join

alternative strategies in systems whose behavior is driven by a large number of parameters assuming values following a probability distribution, we used a Monte Carlo method to generate 2,000 simulations varying the parameters above and, for each simulation, we evaluated the cost of executing a join operation as a regular and as a semi-join.

We compared the cost of evaluating 2,000 one-to-one and 2,000 one-to-many join queries with (and without resp.) the availability of the semi-join technique for query evaluation. We assume that the query optimizer can assess which of the two strategies (i.e., *RegJ*, *SemiJ*) is less expensive for each query. Figures 9(a) and 9(b) illustrate the total costs as the number of query grows for the two scenarios, considering one-to-one and one-to-many queries, respectively. As visible in the figure, if all the queries are evaluated adopting the regular join approach, the total cost (continuous line) reaches higher values than with the availability of the semi-join approach (dotted line). This trend is more visible for one-to-many joins, where the total cost reached when all the queries are evaluated as regular joins is 2,475 USD while with the availability of the semi-join approach it remains at 1,321 USD, with a total saving of 1,160 USD (corresponding to 46,85%). In fact, out of the 2,000 one-to-many queries, 1784 were evaluated as semi-joins, while for the remaining 216 the regular join solution was cheaper. For one-to-one joins, as expected, the total saving is lower (24.77%) since half of the 2,000 queries considered are cheaper when evaluated as regular joins.

## 7 Related Work

Our work falls in the area of security and privacy in emerging outsourcing and cloud scenarios [2,7]. In this context, researchers have proposed solutions addressing a variety of issues, including data protection, access control, fault tolerance, data and query integrity (e.g., [1,3,4,6,8,9,15]). In particular, current solutions addressing the problem of verifying the integrity (i.e., completeness, correctness, and freshness) of query results are based on the definition of a verification object

returned with the query result. Different approaches differ in the definition of the verification object and/or in the kind of guarantees offered, which can be deterministic or probabilistic. For instance, some proposals are based on the definition of an authenticated data structure (e.g., Merkle hash tree or a variation of it [12,19] or of signature-based schemas [13,14]) that allow the verification of the correctness and completeness of query results. These proposals provide deterministic guarantees, that is, they can detect integrity violations with certainty but only for queries involving the attribute(s) on which the authenticated data structure has been created. Some proposals have also addressed the problem of verifying the freshness of query results (e.g., [11,18]). The idea consists in periodically updating a timestamp included in the authenticated data structure or in periodically changing the data generated for integrity verification.

Probabilistic approaches can offer a more general control than deterministic approaches but they can detect an integrity violation only with a given probability (e.g., [5,16,17]). Typically, there is a trade-off between the amount of protection offered and the computational and communication overhead caused. The proposal in [16] consists in replicating a given percentage of tuples and in encrypting them with a key different from the key used for encrypting the original data. Since the replicated tuples are not recognizable as such by the server, the completeness of a query result is guaranteed by the presence of two instances of the tuples that satisfy the query and are among the tuples that have been replicated. The proposal in [17] consists in statically introducing a given number of fake tuples in the data stored at the external server. Fake tuples are generated so that some of them should be part of query results. Consequently, whenever the expected fake tuples are not retrieved in the query result, the completeness of the query result is compromised. In [5] we have introduced the idea of using markers, twins, salts and buckets for assessing the integrity of join queries. This paper extends such a solution proposing two optimizations that limit the overhead introduced by our protection techniques.

## 8 Conclusions

We presented two variations to markers, twins, salts and buckets proposed for assessing query integrity, offering significant performance benefits. In particular, we illustrated how markers, twins, salts and buckets can be easily adapted when a join query is executed as a semi-join, and how salts and buckets can be limited to twins and markers. The experimental evaluation clearly showed that these two variations limit the computational and communication overhead due to the integrity checks.

**Acknowledgements.** The authors would like to thank Michele Mangili 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) and by the Italian Ministry of Research within PRIN project "GenData 2020" (2010RTFWBH). The work of Sushil Jajodia was partially supported by NSF grant IIP-1266147.

## References


1. Basu, A., Vaidya, J., Kikuchi, H., Dimitrakos, T.: Privacy-preserving collaborative filtering on the cloud and practical implementation experiences. In: Proc. of IEEE Cloud, Santa Clara, CA (June-July 2013)
2. Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Balancing confidentiality and efficiency in untrusted relational DBMSs. In: Proc. of CCS, Washington, DC (October 2003)
3. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Livraga, G.: Enforcing subscription-based authorization policies in cloud scenarios. In: Cuppens-Boulahia, N., Cuppens, F., Garcia-Alfaro, J. (eds.) DBSec 2012. LNCS, vol. 7371, pp. 314–329. Springer, Heidelberg (2012)
4. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Encryption policies for regulating access to outsourced data. ACM TODS 35(2), 12:1–12:46 (2010)
5. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Paraboschi, S., Samarati, P.: Integrity for join queries in the cloud. IEEE TCC 1(2), 187–200 (2013)
6. De Capitani di Vimercati, S., Foresti, S., Samarati, P.: Managing and accessing data in the cloud: Privacy risks and approaches. In: Proc. of CRiSIS, Cork, Ireland (October 2012)
7. Hacigümüş, H., Iyer, B., Mehrotra, S., Li, C.: Executing SQL over encrypted data in the database-service-provider model. In: Proc. of SIGMOD, Madison, WI (June 2002)
8. Jhawar, R., Piuri, V.: Fault tolerance and resilience in cloud computing environments. In: Vacca, J. (ed.) Computer and Information Security Handbook, 2nd edn., pp. 125–142. Morgan Kaufmann (2013)
9. Jhawar, R., Piuri, V., Santambrogio, M.: Fault tolerance management in cloud computing: A system-level perspective. IEEE Systems Journal 7(2), 288–297 (2013)
10. Kossmann, D., Kraska, T., Loesing, S.: An evaluation of alternative architectures for transaction processing in the cloud. In: Proc. of SIGMOD, Indianapolis, IN (June 2010)
11. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: Proc. of SIGMOD, Chicago, IL (June 2006)
12. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Authenticated index structures for aggregation queries. ACM TISSEC 13(4), 32:1–32:35 (2010)
13. Mykletun, E., Narasimha, M., Tsudik, G.: Authentication and integrity in outsourced databases. ACM TOS 2(2), 107–138 (2006)
14. Pang, H., Jain, A., Ramamritham, K., Tan, K.: Verifying completeness of relational query results in data publishing. In: Proc. of SIGMOD, Baltimore, MA (June 2005)
15. Ren, K., Wang, C., Wang, Q.: Security challenges for the public cloud. IEEE Internet Computing 16(1), 69–73 (2012)
16. Wang, H., Yin, J., Perng, C., Yu, P.: Dual encryption for query integrity assurance. In: Proc. of CIKM, Napa Valley, CA (October 2008)
17. Xie, M., Wang, H., Yin, J., Meng, X.: Integrity auditing of outsourced data. In: Proc. of VLDB, Vienna, Austria (September 2007)
18. Xie, M., Wang, H., Yin, J., Meng, X.: Providing freshness guarantees for outsourced databases. In: Proc. of EDBT, Nantes, France (March 2008)
19. Yang, Z., Gao, S., Xu, J., Choi, B.: Authentication of range query results in MapReduce environments. In: Proc. of CloudDB, Glasgow, UK (October 2011)