Private and Scalable Execution of SQL Aggregates on a Secure Decentralized Architecture

QUOC-CUONG TO, Inria and University of Versailles St-Quentin, PRisM lab
BENJAMIN NGUYEN, INSA Centre Val de Loire, LIFO lab
PHILIPPE PUCHERAL, Inria and University of Versailles St-Quentin, PRisM lab

Current applications, from complex sensor systems (e.g. quantified self) to online e-markets acquire vast quantities of personal information which usually end-up on central servers where they are exposed to prying eyes. Conversely, decentralized architectures helping individuals keep full control of their data, complexify global treatments and queries, impeding the development of innovative services. This paper precisely aims at reconciling individual's privacy on one side and global benefits for the community and business perspectives on the other side. It promotes the idea of pushing the security to secure hardware devices controlling the data at the place of their acquisition. Thanks to these tangible physical elements of trust, secure distributed querying protocols can reestablish the capacity to perform global computations, such as SQL aggregates, without revealing any sensitive information to central servers. This paper studies how to secure the execution of such queries in the presence of honest-but-curious and malicious attackers. It also discusses how the resulting querying protocols can be integrated in a concrete decentralized architecture. Cost models and experiments on SQL/AA, our distributed prototype running on real tamper-resistant hardware, demonstrate that this approach can scale to nationwide applications.

Categories and Subject Descriptors: **[Security and Privacy]**: Privacy-Preserving Protocols

General Terms: SQL execution, privacy-preserving database protocols

Additional Key Words and Phrases: Trusted hardware.

ACM Reference Format:

Quoc-Cuong To, Benjamin Nguyen and Philippe Pucheral, 2014. Private and Scalable Execution of SQL on a Secure Decentralized Architecture. *ACM Transactions on Database Systems* X, Y, Article ZZ (March 2015), 46 pages. DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

With the convergence of mobile communications, sensors and online social networks technologies, an exponential amount of personal data - either freely disclosed by users or transparently acquired by sensors - end up in servers. This massive amount of data, the *new oil,* represents an unprecedented potential for applications and business (e.g., car insurance billing, traffic decongestion, smart grids optimization, healthcare surveillance, participatory sensing). However, centralizing and processing all one's data in a single server incurs a major problem with regards to privacy. Indeed, individuals' data is carefully scrutinized by governmental agencies and companies in charge of processing it [de Montjoye *et al.* 2012]. Privacy violations also arise from negligence and attacks and no current server-based approach, including cryptography based and server-side secure hardware [Agrawal *et al.* 2002], seems capable of closing the gap. Conversely, decentralized architectures (e.g., personal data vault), providing better control to the user over the management of her personal data, impede global computations by construction.

This paper aims to demonstrate that privacy protection and global computation are not antagonist and can be reconciled to the best benefit of the individuals, the community and the companies. To reach this goal, this paper capitalizes on a novel architectural approach called *Trusted Cells* [Anciaux *et al.* 2013]. Trusted Cells push the security to the edges of the network, through personal data servers [Allard *et al.* 2010] running on secure smart phones, set-top boxes, plug computers[1] or secure portable tokens[2] forming a global decentralized data platform. Indeed, thanks to the

[1] http://freedomboxfoundation.org/
[2] http://www.gd-sfs.com/portable-security-token

emergence of low-cost secure hardware and firmware technologies like ARM TrustZone[3], a full Trusted Execution Environment (TEE) will soon be present in any client device. In this paper, and up to the experiments section, we consider that personal data is acquired and/or hosted by secure devices but make no additional assumption regarding the technical solution they rely on.

Global queries definitely make sense in this context. Typically, it would be helpful to compute aggregates over smart meters without disclosing individual's raw data (e.g., *compute the mean energy consumption per time period and district*). Identifying queries also make sense assuming the identified subjects consent to participate (e.g., *send an alert to people living in Paris-La Defense district if their total energy consumption reaches a given threshold*). Computing SQL-like queries on such distributed infrastructure leads to two major and different problems: computing joins between data hosted at different locations and computing aggregates over this same data. We tackled the first issue in [To *et al.* 2015a] thanks to a trusted MapReduce-based system that can support joins and cover parallelizable tasks executed over a Trusted Cells infrastructure. This paper concentrates on the second issue: how to compute global queries over decentralized personal data stores while respecting users' privacy? Indeed, we believe that the computation of aggregates is central to the many novel privacy preserving applications such as smart metering, e-administration, etc.

Our objective is to make as few restrictions on the computation model as possible. We model the information system as a global database formed by the union of a myriad of distributed local data stores (e.g., nation-wide context) and we consider regular SQL queries and a traditional access control model. Hence the context we are targeting is different and more general than, (1) querying encrypted outsourced data where restrictions are put on the predicates which can be evaluated [Agrawal *et al.* 2004, Amanatidis *et al.* 2007, Popa *et al.* 2011, Hacigümüs *et al.* 2004], (2) performing privacy-preserving queries usually restricted to statistical queries matching differential privacy constraints [Fung *et al.* 2010, Fayyoumi and Oommen 2010] and (3) performing Secure-Multi-Party (SMC) query computations which cannot meet both query generality and scalability objectives [Kissner and Song 2005].

The contributions of this paper are[4]: (1) to propose different secure query execution techniques to evaluate regular SQL "group by" queries over a set of distributed trusted personal data stores, (2) to quantify and compare the respective information exposure of these techniques, (3) to study the range of applicability of these techniques and show that our approach is compatible with nation-wide contexts by thorough analysis of a cost model and performance measurements of a prototype running on real secure hardware devices.

The rest of this paper is organized as follows. Section 2 states our problem. Section 3 discusses related works. Section 4 introduces a framework to execute simple queries and Section 5 concentrates on complex queries involving *Group By* and *Having* clauses. Section 6 discusses practical aspects of the proposed solution.

[3] http://www.arm.com/products/processors/technologies/trustzone.php

[4] This paper is an extended and restructured version of [To *et al.* 2014a]. The new material covers a set of important problems that need to be solved to make the approach practical: cryptographic key management, accuracy and latency of the collection phase, access control management, ~~load balancing and fault tolerance~~. The security analysis was also improved to address stronger attackers with more knowledge. A solution was also proposed to prevent malicious attackers from deleting the data, ensuring the completeness of the result. In addition, it validates our cost model thanks to performance measurement performed on real secure hardware. This version also integrates more detailed results and a performance comparison with state of the art methods.

Section 7 presents a privacy analysis of each querying protocol. Section 8 analyzes the performance of these solutions through cost models while section 9 validates these cost models through performance measurements. Finally section 10 concludes. Appendix A is added at the end to clarify how we prevent malicious attacks.

2. CONTEXT OF THE STUDY

2.1 Scenarios and queries of interest

As discussed in [Anciaux *et al.* 2013], trusted hardware is more and more versatile and has become a key enabler for all applications where trust is required at the edges of the network. Figure 1 depicts different scenarios where a Trusted Data Server (TDS) is called to play a central role, by reestablishing the capacity to perform global computations without revealing any sensitive information to central servers. TDS can be integrated in energy smart meters to gather energy consumption raw data, to locally perform aggregate queries for billing or smart grid optimization purpose and externalize only certified results, thereby reconciling individuals' privacy and energy providers' benefits. Green button[5] is another application example where individuals accept sharing their data with their neighborhood through distributed queries for their personal benefit. Similarly, TDS can be integrated in GPS trackers to protect individuals' privacy while securely computing insurance fees or carbon tax and participating in general interest distributed services such as traffic jam reduction. Moreover, TDSs can be hosted in personal devices to implement secure personal folders like e.g., PCEHR (Personally Controlled Electronic Health Record) fed by the individuals themselves thanks to the Blue Button initiative[6] and/or quantified-self devices. Distributed queries are useful in this context to help epidemiologists performing global surveys or allow patients suffering from the same illness to share their data in a controlled manner.

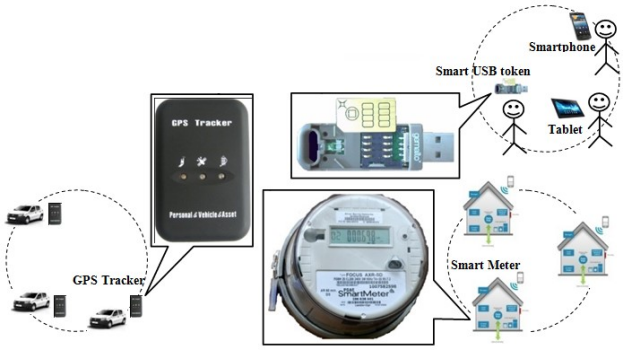


Fig. 1. Different scenarios of TDSs

For the sake of generality, we make no assumption about how the data is actually gathered by TDSs, this point being application dependent [Allard *et al.* 2010, Montjoye *et al.* 2012]. We simply consider that *local databases* conform to a common schema (Fig. 3) which can be queried in SQL. For example, power meter data (resp., GPS traces, healthcare records, etc) can be stored in one (or several) table(s) whose schema is defined by the national distribution company (resp., insurance company consortium, Ministry of Health[7], specific administration, etc). Since raw data can be highly sensitive, it must also be protected by an access control policy defined either

[5] http://www.greenbuttondata.org/
[6] http://healthit.gov/patients-families/your-health-data
[7] This is the case in France for instance.

by the producer organism, by the legislator or by a consumer association. Depending on the scenario, each individual may also opt-in/out of a particular query. For sake of generality again, we consider that each TDS participating in a distributed query protocol enforces at the same time the access control policy protecting the local data it hosts, with no additional consideration for the access control model itself, the choice of this model being orthogonal to this study. Hence, the objective is to let queriers (users) query this decentralized database exactly as if it were centralized, without restricting the expressive power of the language to statistical queries as in many PPDP works [Fayyoumi and Oommen 2010, Popa *et al.* 2011].

Consequently, we assume that the querier can issue the following form of SQL queries[8], borrowing the *SIZE* clause from the definition of windows in the *StreamSQL* standard [StreamSQL 2015]. This clause is used to indicate a maximum number of tuples to be collected, and/or a collection duration.

SELECT <attribute(s) and/or aggregate function(s)>
FROM <Table(s)>
[WHERE <condition(s)>]
[GROUP BY <grouping attribute(s)>]
[HAVING <grouping condition(s)>]
[SIZE <size condition(s)>]

For example, an energy distribution company would like to issue the following query on its customers' smart meters: "*SELECT AVG(Cons) FROM Power P, Consumer C WHERE C.accomodation='detached house' and C.cid = P.cid GROUP BY C.district HAVING Count(distinct C.cid) > 100 SIZE 50000*". This query computes the mean energy consumption of consumers living in a detached home grouped by district, for districts where over 100 consumers answered the poll and the poll stops after having globally received at least 50.000 answers. The semantics of the query are the same as those of a stream relational query [Abadi *et al.* 2003]. Only the smart meter of customers who opt-in for this service will participate in the computation. Needless to say that the querier, that is the distribution company, must be prevented to see the raw data of its customers for privacy concerns[9].

In other scenarios where TDSs are seldom connected (e.g., querying mobile PCEHR), the time to collect the data is probably going to be quite large. Therefore the challenge is not on the overall response time, but rather to show that the query computation on the collected data is tractable in reasonable time, given local resources.

Also note that unless specified otherwise, our semantics make the Open World Assumption: since we assume that data is *not* replicated over TDS, many true tuples will not be collected during the specified period and/or due to the limit, both indicated in the *SIZE* clause. Hence, the SIZE clause is mandatory, since having a complete answer is contradictory with the open world assumption. Under the closed world assumption (in which all TDS are always connected to the SSI), one can replace the keyword SIZE by ALL to collect all available data.

[8] As stated in the introduction, we do not consider *joins* between data stored in different TDSs in this article, the solution to this specific problem being addressed in [To *et al.* 2015a]. However, there is no restriction on the queries executed locally by each TDS.

[9] At the 1HZ granularity provided by the French Linky power meters, most electrical appliances have a distinctive energy signature. It is thus possible to infer from the power meter data inhabitants activities [Lam *et al.* 2007].

2.2 Asymmetric Computing Architecture

The architecture we consider is decentralized by nature. It is formed by a large set of low power TDSs embedded in secure devices. Despite the diversity of existing hardware platforms, a secure device can be abstracted by (1) a Trusted Execution Environment and (2) a (potentially untrusted but cryptographically protected) mass storage area (see Fig. 2)[10]. E.g., the former can be provided by a tamper-resistant microcontroller while the latter can be provided by Flash memory. The important assumption is that the TDS code is executed by the secure device hosting it and thus cannot be tampered, even by the TDS holder herself. Each TDS exhibits the following properties:

High Security. This is due to a combination of factors: (1) the microcontroller tamper-resistance, making hardware and side-channel attacks highly difficult, (2) the certification of the embedded code making software attacks also highly difficult, (3) the ability to be auto-administered, in contrast with traditional multi-user servers, precluding DBA attacks, and (4) the fact that the device holder cannot directly access the data stored locally (she must authenticate and can only access data according to her own privileges). This last point is of utmost importance because it allows the definition of distributed protocols where data is securely exchanged among TDSs with no confidentiality risk.

Low Availability. The Secure Device is physically controlled by its owner who may connect or disconnect it at will, providing no availability guarantee.

Modest Computing Resource. Most Secure Devices provide modest computing resources (see section 8) due to the hardware constraints linked to their tamper-resistance. On the other hand, a dedicated cryptographic co-processor usually handles cryptographic operations very efficiently (e.g., AES and SHA).

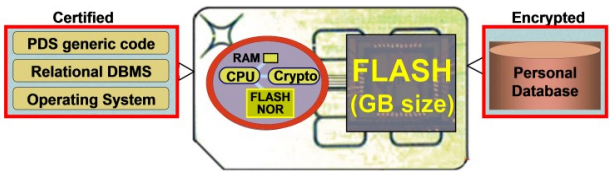


Fig. 2. Trusted Data Servers

Hence, even if there exist differences among Secure Devices (e.g., smart tokens are more robust against tampering but less powerful than TrustZone devices), all provide *much stronger security guarantees* combined with a *much weaker availability and computing power* than any traditional server.

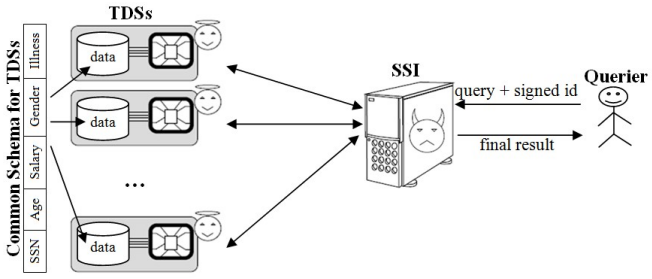


Fig. 3. The Asymmetric Architecture

[10] For illustration purpose, the secure device considered in our experiments is made of a tamper-resistant microcontroller connected to a Flash memory chip.

Since TDSs have limited storage and computing resources and they are not necessarily always connected, an external infrastructure, called hereafter *Supporting Server Infrastructure* (SSI), is required to manage the communications between TDSs, run the distributed query protocol and store the intermediate results produced by this protocol. Because SSI is implemented on regular server(s), e.g., in the Cloud, it exhibits the same low level of trustworthiness, high computing resources, and availability.

The computing architecture, illustrated in Fig. 3 is said *asymmetric* in the sense that it is composed of a very large number of low power, weakly connected but highly secure TDSs and of a powerful, highly available but untrusted SSI.

2.3 Threat Model

TDSs are the unique elements of trust in the architecture and are considered *honest*. As mentioned earlier, no trust assumption needs to be made on the TDS holder herself because a TDS is tamper-resistant and enforces the access control rules associated to its holder (just like a car driver cannot tamper the GPS tracker installed in her car by its insurance company or a customer cannot gain access to any secret data stored in her banking smartcard).

We primarily consider *honest-but-curious* (also called *semi-honest*) SSI (i.e., which tries to infer any information it can but strictly follows the protocol), concentrating on the prevention of confidentiality attacks. We additionally discuss (see Appendix A) how to extend our protocols with safety properties to detect attacks conducted by *malicious* SSI (i.e., which may tamper the protocol with no limit, including denial-of-service), although the probability of such attacks is supposed to be much lower because of the risk of an irreversible political/financial damage and even the risk of a class action against the SSI.

The objective is thus to implement a querying protocol so that (1) the querier can gain access only to the final result of authorized queries (not to the raw data participating in the computation), as in a traditional database system and (2) intermediate results stored in SSI are obfuscated. Preventing inferential attacks by combining the result of a sequence of authorized queries as in statistical databases and PPDP work (see section 3) is orthogonal to this study.

3. RELATED WORKS

This work has connections with related studies in different domains, namely protection of outsourced (personal) databases, statistical databases and PPDP, SMC and finally secure aggregation in sensor networks. We review these works below.

Security in outsourced databases. Outsourced database services or DaaS [Hacigumus *et al.* 2002] allow users to store sensitive data on a remote, untrusted server and retrieve desired parts of it on request. Many works have addressed the security of DaaS by encrypting the data at rest and pushing part of the processing to the server side. Searchable encryption has been studied in the symmetric-key [Amanatidis *et al.* 2007] and public-key [Bellare *et al.* 2007] settings but these works focus mainly on simple exact-match queries and introduce a high computing cost. Agrawal *et al.* [2004] proposed an order preserving encryption (OPE) scheme, which ensures that the order among plaintext data is preserved in the ciphertext domain, supporting range and aggregate queries, but OPE relies on the strong assumption that all plaintexts in the database are known in advance and order-preserving is usually synonym of weaker security. The assumption on the *a priori* knowledge of all

plaintext is not always practical (e.g., in our highly distributed database context, users do not know all plaintexts *a priori*), so a stateless scheme whose encryption algorithm can process single plaintexts on the fly is more practical. Bucketization-based techniques [Hacigumus *et al.* 2002, Hore *et al.* 2012] use distributional properties of the dataset to partition data and design indexing techniques that allow approximate queries over encrypted data. Unlike cryptographic schemes that aim for exact predicate evaluation, bucketization admits false positives while ensuring all matching data is retrieved. A post-processing step is required at the client-side to weed out the false positives. These techniques often support limited types of queries and lack of a precise analysis of the performance/security tradeoff introduced by the indexes. To overcome this limitation, the work in [Damiani *et al.* 2003] quantitatively measures the resulting inference exposure. Other works introduce solutions to compute basic arithmetic over encrypted data, but homomorphic encryption [Paillier 1999] supports only range queries, fully homomorphic encryption [Gentry 2009] is unrealistic in terms of time, and privacy homomorphism [Hacigumus *et al.* 2004] is insecure under ciphertext-only attacks [Mykletun, and Tsudik 2006]. Hence, optimal performance/security tradeoff for outsourced databases is still regarded as the Holy Grail. Recently, the Monomi system [Tu *et al.* 2013] has been proposed for securely executing analytical workloads over sensitive data on an untrusted database server. Although this system can execute complex queries, there can be only one trusted client decrypting data, and therefore it cannot enjoy the benefit of parallel computing. Another limitation of this system is that to perform the GROUP BY queries, it encrypts the grouping attributes with deterministic encryption, allowing frequency-based attacks.

Statistical Database and PPDP. Statistical databases (SDB) [Fayyoumi and Oommen 2010] are motivated by the desire to compute statistics without compromising sensitive information about individuals. This requires trusting the server to perform query restriction or data perturbation, to produce the approximate results, and to deliver them to untrusted queriers. Thus, the SDB model is orthogonal to our context since (1) it assumes a trusted third party (i.e., the SDB server) and (2) it usually produces approximate results to prevent queriers from conducting inferential attack [Fayyoumi and Oommen 2010]. For its part, Privacy-Preserving Data Publishing (PPDP) [Fung *et al.* 2010] provides a non trusted user with some sanitized data produced by an anonymization process such as *k*-anonymity, *l*-diversity or differential privacy to cite the most common ones [Fung *et al.* 2010]. Similarly, PPDP is orthogonal to our context since it again assumes a trusted third party (i.e., the publisher) and produces sanitized data of lower quality to match the information exposure dictated by a specific privacy model. The work in [Allard *et al.* 2014] tackles the first limitation by pushing the trust to secure clients but keeps the objective of producing sanitized releases. Contrary to these works, our paper targets the execution of general SQL queries, considers a traditional access control model and does not rely on a secure server.

Secure Multi-party Computation. Secure multi-party computation (SMC) allows N parties to share a computation in which each party learns only what can be inferred from their own inputs (which can then be kept private) and the output of the computation. This problem is represented as a combinatorial circuit which depends on the size of the input. The resulting cost of a SMC protocol depends on the number of inter-participant interactions, which in turn depends exponentially on the size of the input data, on the complexity of the initial function, and on the number of participants. Despite their unquestionable theoretical interest, generic SMC

approaches are impractical where inputs are large and the function to be computed complex. Ad-hoc SMC protocols have been proposed [Kissner and Song 2005] to solve specific problems/functions but they lack of generality and usually make strong assumptions on participants' availability. Hence, SMC is badly adapted to our context.

Secure Data Aggregation. Wireless sensor networks (WSN) [Alzaid *et al.* 2008] consist of sensor nodes with limited power, computation, storage, sensing and communication capabilities. In WSN, an aggregator node can compute the sum, average, minimum or maximum of the data from its children sensors, and send the aggregation results to a higher-level aggregator. WSN have some connection with our context regarding the computation of distributed aggregations. However, contrary to the TDS context, WSN nodes are highly available, can communicate with each other in order to form a network topology to optimize calculations (In fact, TDSs can collaborate to form the topology through SSI, but because of the weak connectivity of TDSs, forming the topology is inefficient in term of time). Other work [Castelluccia *et al.* 2005] uses additively homomorphic encryption for computing aggregation function on encrypted data in WSN but fails to consider queries with GROUP BY clauses. Liu *et al.* [2010] protects data against frequency-based attacks but considers only point and range queries.

As a conclusion, and to the best of our knowledge, our work is the first proposal achieving a fully distributed and secure solution to compute aggregate SQL queries over a large set of participants.

4. BASIC QUERYING PROTOCOL

This section presents the protocol to compute Select-From-Where queries. This protocol is simple yet very useful in practice, since many queries are of this form. We also use it to help the reader get used to our approach. We tackle the more difficult Group By clause in section 5.

4.1 Core infrastructure

Our querying protocols share common basic mechanisms to make TDSs aware of the queries to be computed and to organize the dataflow between TDSs and queriers such that SSI cannot infer anything from the queries and their results.

Query and result delivery: queries are executed in pull mode. A querier posts its query to SSI and TDSs download it at connection time. To this end, SSI can maintain personal *query boxes* (in reference to mailboxes) where each TDS receives queries directed to it (e.g., *get the monthly energy consumption of consumer C*) and a global *query box* for queries directed to the crowd (e.g., *get the mean of energy consumption per month for people living in district D*). Result tuples are gathered by SSI in a temporary storage area. A query remains active until the SIZE clause is evaluated to *true* by SSI, which then informs the querier that the result is ready.

Dataflow obfuscation: all data (queries and tuples) exchanged between the querier and the TDSs, and between TDSs themselves, can be spied by SSI and must therefore be encrypted. However, an *honest-but-curious* SSI can try to conduct *frequency-based attacks* [Liu *et al.* 2010], i.e., exploiting prior knowledge about the data distribution to infer the plaintext values of ciphertexts. Depending on the protocols (see later), two kinds of encryption schemes will be used to prevent frequency-based attacks. With non-deterministic (aka probabilistic) encryption, denoted by *nDet_Enc*, several encryptions of the same message yield different ciphertexts while deterministic encryption (*Det_Enc* for short) always produces the

same ciphertext for a given plaintext and key [Bellare *et al.* 2007]. Whatever the encryption scheme, symmetric keys must be shared among TDSs: we note k_Q the symmetric key used by the querier and the TDSs to communicate together and k_T the key shared by TDSs to exchange temporary results among them. Note that these keys may change over time and the way they are delivered to TDSs is discussed more deeply in section 6.

4.2 Select-From-Where statements

Let us first consider simple SQL queries of the form:
SELECT <attribute(s)> FROM <Table(s)> [WHERE <condition(s)>] [SIZE <size condition(s)>]

These queries do not have a *GROUP BY* or *HAVING* clause nor involve aggregate functions in the *SELECT* clause. Hence, the selected attributes may (or may not) contain identifying information about the individuals. Though basic, these queries answer a number of practical use-cases, e.g., a doctor querying the embedded healthcare folders of her patients, or an energy provider willing to offer special prices to people matching a specific consumption profile. To compute such queries, the protocol is divided in two phases (see Fig. 4):

Collection phase: (step 1) the querier posts on SSI a query Q encrypted with k_Q, its credential $\mathcal{C}$ signed by an authority and S the SIZE clause of the query in cleartext so that SSI can evaluate it; (step 2) targeted TDSs[11] download Q when they connect; (step 3) each of these TDSs decrypts Q, checks $\mathcal{C}$, evaluates the AC policy associated to the querier and computes the result of the WHERE clause on the local data; then each TDS either sends its result tuples (step 4), or a small random number of dummy tuples[12] when the result is empty or the querier has not enough privilege to access these local data (step 4'), non-deterministically encrypted with k_T. The collection phase stops when the SIZE condition has been reached (i.e., the total number of collected encrypted tuples is S)[13]. The result of the collection phase is actually the result of the query, possibly complemented with dummy tuples. We call it Covering Result.

Filtering phase: (step 5) SSI partitions the Covering Result with the objective to let several TDSs manage next these partitions in parallel. The Covering Result being fully encrypted, SSI sees partitions as uninterpreted chunks of bytes; (step 6) connected TDSs download these partitions. These TDSs may be different from the ones involved in the collection phase; (step 7) each of these TDS decrypts the partition and filters out dummy tuples; (step 8) each TDS sends back the true tuples encrypted with key k_Q to SSI, which finally concatenates all results and informs the querier that she can download the result (step 9).

[11] Connected TDSs actually download all queries and decrypt them to check whether they can contribute to them or not. The SSI cannot perform this task since queries are encrypted.
[12] The objective is to hide which TDSs satisfy the Where clause of the query in the case SSI and Querier collude.
[13] The production of dummy tuples may slightly impact the evaluation of the SIZE clause. The Querier must thus oversize this parameter according to his perception of the selectivity of the query and the percentage of TDSs opting-out for the query. If this over sizing turns out to be insufficient, the query could need to be rerun. Note anyway that the SIZE limit is a coarse parameter in the open world assumption.

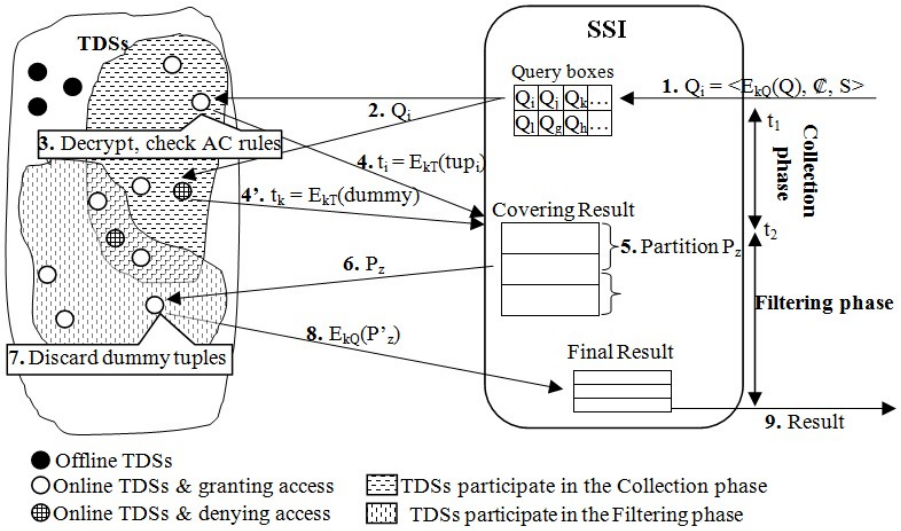


Fig. 4. Select-From-Where querying protocol

Informally speaking, the accuracy, security and efficiency properties of the protocol are as follows:

Accuracy. Since SSI is honest-but-curious, it will deliver to the querier all tuples returned by the TDSs. Dummy tuples are marked so that they can be recognized and removed after decryption by each TDS. Therefore the final result contains only true tuples. If a TDS goes offline in the middle of processing a partition, SSI resends that partition to another available TDS after a given timeout so that the result is complete. As discussed in Appendix A, accuracy is more difficult to achieve in the case of malicious SSI.

Security. Since SSI does not know key k_Q, it can decrypt neither the query nor the result tuples. TDSs use *nDet_Enc* for encrypting the result tuples so that SSI can neither launch any frequency-based attacks nor detect dummy tuples. There can be two additional risks. The first risk is that SSI acquires a TDS with the objective to get the cryptographic material. As stated in section 2, TDS code cannot be tampered, even by its holder. Whatever the information decrypted internally, the only output that a TDS can deliver is a set of encrypted tuples, which does not represent any benefit for SSI. The second risk is if SSI colludes with the querier. For the same reason, SSI will only get the same information as the querier (i.e., the final result in clear text and no more).

Efficiency. The efficiency of the protocol is linked to the frequency of TDSs connection and to the SIZE clause. Both the collection and filtering phases are run in parallel by all connected TDSs and no time-consuming task is performed by any of them. As the experiment section will clarify, each TDS manages incoming partitions in streaming because the internal time to decrypt the data and perform the filtering is significantly less than the time needed to download the data.

While important in practice, executing Select-From-Where queries in the Trusted Cells context shows no intractable difficulties and the main objective of this section was to present the query framework in this simple context. Executing Group By queries is far more challenging. The next section will present different alternatives to tackle this problem. Rather than trying to get an optimal solution, which is context dependent, the objective is to explore the design space and show that different querying protocols may be devised to tackle a broad range of situations.

5. GROUP BY QUERIES

The Group By clause introduces an extra phase: the computation of aggregates of data produced by different TDSs, which is the weak point for frequency-based attacks. In this section, we propose several protocols, discussing their strong and weak points from both efficiency and security points of view.

5.1 Generic query evaluation protocol

Let us now consider general SQL queries of the form[14]:
SELECT <attribute(s) and/or aggregate function(s)> FROM <Table(s)> [WHERE <condition(s)>] [GROUP BY <grouping attribute(s)>] [HAVING <grouping condition(s)>][SIZE <size condition(s)>]

These queries are more challenging to compute because they require performing set-oriented computations over intermediate results sent by TDSs to SSI. The point is that TDSs usually have limited RAM, limited computing resources and limited connectivity. It is therefore unrealistic to devise a protocol where a single TDS downloads the intermediate results of all participants, decrypts them and computes the aggregation alone. On the other hand, SSI cannot help much in the processing since (1) it is not allowed to decrypt any intermediate results and (2) it cannot gather encrypted data into groups based on the encrypted value of the grouping attributes, denoted by $A_G=\{G_i\}$, without gaining some knowledge about the data distribution. This would indeed violate our security assumption since the knowledge of A_G distribution opens the door to frequency-based attacks by SSI: e.g. in the extreme case where A_G contains both quasi-identifiers and sensitive values, attribute linkage would become obvious. Finally, the querier cannot help in the processing either since she is only granted access to the final result, and not to the raw data.

To solve this problem, we suggest a generic aggregation protocol divided into three phases (see Fig. 5):

Collection phase: similar to the basic protocol.

Aggregation phase: (step 5) SSI partitions the result of the collection phase; (step 6) connected TDSs (may be different from the ones involved in the collection phase) download these partitions; (step 7) each of these TDS decrypts the partition, eliminates the dummy tuples and computes partial aggregations (i.e., aggregates data belonging to the same group inside each partition); (step 8) each TDS sends its partial aggregations encrypted with k_T back to SSI; depending on the protocol (see next sections), the aggregation phase is iterative, and continues until all tuples belonging to the same group have been aggregated (steps 6', 7', 8'); The last iteration produces a Covering Result containing a single (encrypted) aggregated tuple for each group.

[14] For the sake of clarity, we concentrate on the management of distributive, algebraic and holistic aggregate functions identified in [Locher 2009] as the most prominent and useful ones.

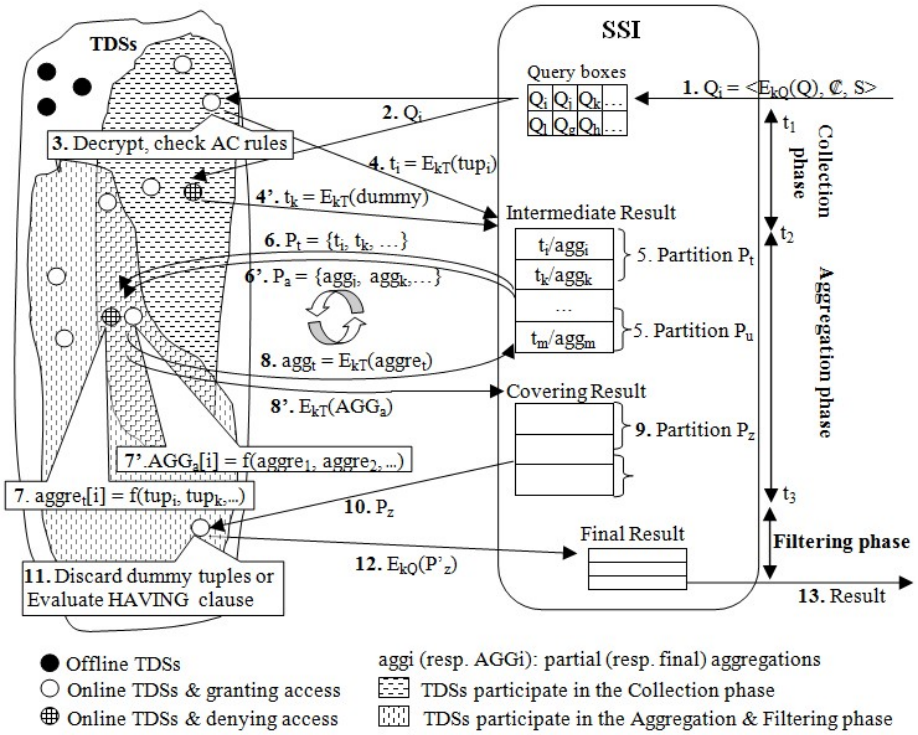


Fig. 5. Group By querying protocol

Filtering phase: this phase is similar to the basic protocol except that the role of step 11 is to eliminate the groups which do not satisfy the HAVING clause instead of eliminating dummy tuples.

The rest of this section presents different variations of this generic protocol, depending on which encryption scheme is used in the collection and aggregation phases, how SSI constructs the partitions, and what information is revealed to SSI. Each solution has its own strengths and weaknesses and therefore is suitable for a specific situation. Three kinds of solutions are proposed: secure aggregation, noise-based, and histogram-based. They are subsequently compared in terms of privacy protection (Section 7) and performance (Section 8).

5.2 Secure Aggregation protocol

This protocol, denoted by *S_Agg* and detailed in Algorithm 1, instantiates the generic protocol as follows. In the **collection phase**, each participating TDS encrypts its result tuples using *nDet_Enc* (i.e., $nE_{kT}(tup)$) to prevent any frequency-based attack by SSI. The consequence is that SSI cannot get any knowledge about the group each tuple belongs to. Thus, during step 5, tuples from the same group are randomly distributed among the partitions. This imposes the **aggregation phase** to be iterative, as illustrated in Fig. 6. At each iteration, TDSs download encrypted partitions (i.e., Ω^e) containing a sequence of (A_G, Aggregate) value pairs ((City, Energy_consumption) in the example), decrypt them to plaintext partitions (i.e., $\Omega \leftarrow nE_{kT}^{-1}(\Omega^e)$), aggregate values belonging to the same grouping attributes (i.e., $\Omega_{new} = \Omega_{old} \stackrel{?}{+} \Omega$), and sends back to SSI an (usually smaller) encrypted sequence of (A_G, Aggregate) value pairs where values of the same group have been aggregated. SSI gathers these partial

aggregations to form new partitions, and so on and so forth until a single partition (i.e., Ω_{final}) is produced, which contains the final aggregation.

ALGORITHM 1. Secure Aggregation: *S_Agg* (k_Q, k_T, Q, α)

Input: (TDS's side): The cryptographic keys (k_Q, k_T), query Q from Querier.
(SSI's side): reduction factor α ($\alpha \geq 2$).
Output: The final aggregation Ω_{final}.
Notations: encrypted partial aggregation Ω^e; Total number of partitions n^e_Ω
Non-deterministic encryption/decryption with key k_T $nE_{kT}()/nE_{kT}^{-1}()$;
The aggregation operator $\dot{+}$ to aggregate tuples having the same A_G;
begin *Collection phase*
Each connected TDS sends a tuple of the form $tup^e = nE_{kT}(tup)$ *to SSI*
end
begin *Aggregation phase*

SSI side

repeat
Randomly choose tup^e *or* Ω^e *to form partitions*
repeat
Send these partitions to connected TDSs
until *all partitions in SSI have been sent*
Receive Ω^e *from TDSs*
until $n^e_\Omega = 1$

TDSs side

while *(true)*
Reset $\Omega = 0$
Receive partition from SSI
Decrypt partition: $tup \leftarrow nE_{kT}^{-1}(tup^e)$; *(or* Ω^e*):* $\Omega \leftarrow nE_{kT}^{-1}(\Omega^e)$
Add to its partial aggregation: $\Omega = \Omega \dot{+} tup$; *or* $\Omega_{new} = \Omega_{old} \dot{+} \Omega$
Encrypt its partial aggregation: $\Omega^e \leftarrow nE_{kT}(\Omega)$
Send Ω^e *to SSI*
endwhile
end
Filtering phase //evaluate HAVING clause
return $nE_{kQ}(\Omega_{final})$ *by SSI to Querier*;

Accuracy. The requirement for *S_Agg* to terminate is that TDSs have enough resources to perform partial aggregations. Each TDS needs to maintain in memory a data structure called *partial aggregate* which stores the current value of the aggregate function being computed for each group. Each tuple read from the input partition contributes to the current value of the aggregate function for the group this tuple belongs to. Hence the *partial aggregate* structure must fit in RAM (or be swapped in stable storage at much higher cost). If the number of groups is high (e.g., grouping on a key attribute) and TDSs have a tiny RAM, this may become a limiting factor. In this case, FLASH memory can be used to store large intermediate results, at the cost of swapping.

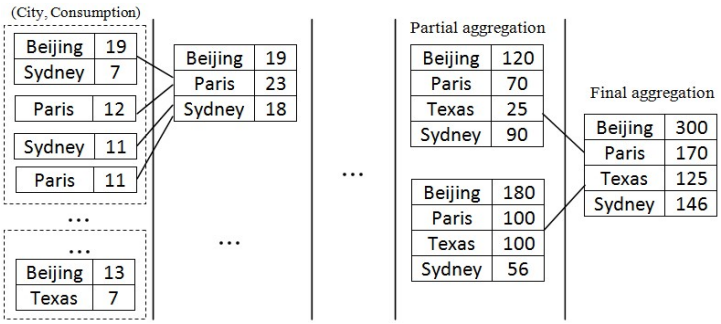


Fig. 6. (iterative partial) aggregation

Security. In all phases, the information revealed to SSI is a sequence of tuples or value pairs (i.e., tup^e and Ω^e) encrypted non-deterministically (*nDet_Enc*) so that SSI cannot conduct any frequency-based attack.

Efficiency. The aggregation process is such that the parallelism between TDSs decreases at each iteration (i.e., $\alpha = \frac{N_1^{TDS}}{N_2^{TDS}} = \cdots = \frac{N_{n-1}^{TDS}}{N_n^{TDS}}$, with N_i^{TDS} being the number of TDSs that participate in the i^{th} partial aggregation phase), up to having a single TDS producing the final aggregation (i.e., $N_n^{TDS} = 1$). The cost model is proposed in section 8 to find the optimal value for the reduction factor α. Note again that incoming partitions are managed in streaming because the cost to download the data significantly dominates the rest.

Suitable queries. Because of the limited RAM size, this algorithm is applicable for the queries with small G such as Q1: *SELECT AVG(Cons) FROM Power P, Customer C WHERE C.City='Paris' and C.cid=P.cid GROUP BY C.district* (Paris has only 20 districts) or Q2: *SELECT COUNT(*) FROM Customer C WHERE 25 < C.Age and C.Age < 45 GROUP BY C.Gender.*

5.3 Noise-based protocols

In these protocols, called *Noise_based* and detailed in Algorithm 2, *Det_Enc* is used during the **collection phase** on the grouping attributes A_G. This is a significant change, since it allows SSI to help in data processing by assembling tuples belonging to the same groups in the same partitions. However, the downside is that using *Det_Enc* reveals the distribution of A_G to SSI. To prevent this disclosure, the fundamental idea is that TDSs add some noise (i.e., fake tuples) to the data in order to hide the real distribution. The added fake tuples must have identified characteristics, as dummy tuples, such that TDSs can filter them out in a later step. The **aggregation** phase is roughly similar to *S_Agg*, except that the content of partitions is no longer random, thereby accelerating convergence and allowing parallelism up to the final iteration. Two solutions are introduced to generate noise: random (white) noise, and noise controlled by complementary domains.

Random (white) noise solutions. In this solution, denoted R_{nf}*_Noise*, n_f fake tuples are generated randomly then added. TDSs apply *Det_Enc* on A_G, and *nDet_Enc* on $\bar{A}_G$ (the attributes not appearing in the *GROUP BY* clause). However, because the fake tuples are randomly generated, the distribution of mixed values may not be different enough from that of true values especially if the disparity in frequency among A_G is big. To overcome this difficulty, a large quantity of fake tuples ($n_f >> 1$) must be injected to make the fake distribution dominate the true one.

ALGORITHM 2. Random noise: R_{nf}_Noise (k_Q, k_T, Q, n_f)

Input: (TDS's side): the cryptographic keys (k_Q, k_T), query Q from Querier.
Output: The final aggregation Ω_{final}.
Notation: $E_{kT}()/E_{kT}^{-1}()$ deterministic encryption/decryption with key k_T
begin *Collection phase*
Each connected TDS sends (n_f+1) tuples of the form $tup^e = (E_{kT}(A_G), nE_{kT}(\tilde{A}_G)$ *to SSI*
end
begin *Aggregation phase*
SSI side
repeat
Group data with the same $E_{kT}(A_G)$ *to form partitions*
repeat
Send these partitions to connected TDSs
until *all partitions in SSI have been sent*
Receive $[E_{kT}(A_G), nE_{kT}(AGG)]$ *from TDSs*
until $n^e_\Omega = 1$
TDSs side
while *(true)*
Reset $[A_G, AGG] = 0$
Receive partition from SSI
Decrypt tup_e: $A_G \leftarrow E_{kT}^{-1}(A^e_G)$; $\tilde{A}_G \leftarrow nE_{kT}^{-1}(\tilde{A}^e_G)$
Filter false tuples
Compute aggregate for A_G: $[A_G, AGG]$
Encrypt aggregate: $[E_{kT}(A_G), nE_{kT}(AGG)]$
Send $[E_{kT}(A_G), nE_{kT}(AGG)]$ *to SSI*
endwhile
end
Filtering phase //evaluate HAVING clause
return $nE_{kQ}(\Omega_{final})$ *to Querier by SSI;*

Noise controlled by complementary domains. This solution, called *C_Noise*, overcomes the limitation of R_{nf}_Noise by generating fake tuples based on the prior knowledge of the A_G domain cardinality. Let us assume that A_G domain cardinality is n_d(e.g., for attribute Age, $n_d \approx 130$), a TDS will generate n_d - 1 fake tuples, one for each value different from the true one. The resulting distribution is totally flat by construction. However, if the domain cardinality is not readily available, a cardinality discovering algorithm must be launched beforehand (see section 5.4).

Accuracy. True tuples are grouped in partitions according to the value of their A_G attributes so that the aggregate function can be computed correctly. Fake tuples are eliminated during the aggregation phase by TDSs thanks to their identified characteristics and do not contribute to the computation.

Security. Although TDSs apply *Det-Enc* on A_G, A_G distribution remains hidden to SSI by injecting enough white noise such that the fake distribution dominates the true one or by adding controlled noise producing a flat distribution.

Efficiency. TDSs do not need to materialize a large partial aggregate structure as in *S_Agg* because each partition contains tuples belonging to a small set of (ideally one) groups. Additionally, this property guarantees the convergence of the aggregation process and increases the parallelism in all phases of the protocol. However, the price to pay is the production and the elimination afterwards of a potentially very high number of fake tuples (the value is algorithm and data dependent).

Suitable queries. R_{nf}_Noise with small $n_f$is suitable for the queries in which there is no wide disparity in frequency between A_G such as Q3: *SELECT COUNT(Child)*

FROM Customer C GROUP BY C.Name HAVING COUNT(Child) < 3. In contrast, the white noise solution with big n_f is suitable for queries with big disparity such as Q4: *SELECT COUNT(*) FROM Customer C GROUP BY C.Salary* because the number of very rich people (i.e, salary > 1 M€/year) is much less than that of people having average salary. For the *C_Noise*, in term of the feasibility, because the process of calculating aggregation is divided among connected TDSs in a distributed and parallel way, better balancing the loads between TDSs, this protocol is applicable not only for the queries where G is small (e.g., Q1, Q2) but also for those with big G, such as Q5: *SELECT AVG(Cons) FROM Customer C WHERE C.Age>20 GROUP BY C.Age* (because the Age's domain is 130 at maximum). However, considering the efficiency, because the number of fake tuples is proportional to *G*, this solution is inappropriate for the queries with very big G (e.g., Q4) when it has to generate and process a large amount number of fake tuples.

5.4 Equi-depth histogram-based protocol

Getting a prior knowledge of the domain extension of A_G allows significant optimizations as illustrated by *C_Noise*. Let us go one step further and exploit the prior knowledge of the real distribution of A_G attributes. The idea is no longer to generate noisy data but rather to produce a uniform distribution of true data sent to SSI by grouping them into equi-depth histograms, in a way similar to [Hacigumus *et al.* 2002]. The protocol, named *ED_Hist*, works as follows. Before entering the protocol, the distribution of A_G attributes must be discovered and distributed to all TDSs. This process needs to be done only once and refreshed from time to time instead of being run for each query. The discovery process is similar to computing a *Count* function on *Group By* A_G and can therefore be performed using one of the protocol introduced above. During the **collection phase**, each TDS uses this knowledge to calculate *nearly equi-depth histograms* that is a decomposition of the A_G domain into buckets holding *nearly* the same number of true tuples. Each bucket is identified by a hash value giving no information about the position of the bucket elements in the domain. Then the TDS allocates its tuple(s) to the corresponding bucket(s) and sends to SSI couples of the form (h(bucketId), *nDet_Enc*(tuple)). During the partitioning step of the **aggregation phase**, SSI assembles tuples belonging to the same buckets in the same partitions. Each partition may contain several groups since a same bucket holds several distinct values. The first aggregation step computes partial aggregations of these partitions and returns to SSI results of the form (*Det_Enc*(group), *nDet_Enc*(partial aggregate)). A second aggregation step is required to combine these partial aggregations and deliver the final aggregation.

Accuracy. Only true tuples are delivered by TDSs and they are grouped in partitions according to the bucket they belong to. Buckets are disjoint and partitions contain a small set of grouping values so that partial aggregations can be easily computed by TDSs.

Security. SSI only sees a nearly uniform distribution of h(bucketId) values and cannot infer any information about the true distribution of A_G attributes. Note that h(bucketId) plays here the same role as *Det_Enc*(bucketId) values but is cheaper to compute for TDSs.

Efficiency. TDSs do not need to materialize a large partial aggregate structure as in *S_Agg* because each partition contains tuples belonging to a small set of groups during the first phase and to a single group during the second phase. As for *C_Noise*, this property guarantees convergence of the aggregation process and maximizes the

parallelism in all phases of the protocol. But contrary to *C_Noise*, this benefit does not come at the price of managing fake tuples.

ALGORITHM 3. Histogram-based: *ED_Hist* (k_Q, k_T, Q)

Input: (TDS's side): the cryptographic keys (k_Q, k_T), query Q from Querier.
Output: The final aggregation Ω_{final}.
Call distribution discovering algorithm to discover the distribution
begin *Collection phase*
Each connected TDS sends a tuple of the form $tup^e = (h(A_G), nE_{kT}(\bar{A}_G))$ *to SSI.*
// $h(A_G)$ is the mapping function applied on the A_G.
end
begin *First aggregation phase*

SSI side

Group tup^e *with the same* $h(A_G)$ *to form partitions*
repeat
Send these partitions to connected TDSs
until *all partitions in SSI have been sent*

TDSs side

while *(true)*
Reset $[A_{Gi}, AGG_i] = 0$
Receive partition from SSI
Decrypt tup_e: $A_G \leftarrow h^{-1}(A^e_G)$; $\bar{A}_G \leftarrow nE_{kT}^{-1}(\bar{A}^e_G)$
Compute aggregate for $h(A_G)$: $[A_{Gi}, AGG_i]$
Encrypt aggregate: $[E_{kT}(A_{Gi}), nE_{kT}(AGG_i)]$
Send $[E_{kT}(A_{Gi}), nE_{kT}(AGG_i)]$ *to SSI*
endwhile
end
begin *Second aggregation phase*

SSI side

repeat
Group data with the same $E_{kT}(A_G)$ *to form partitions*
repeat
Send these partitions to connected TDSs
until *all partitions in SSI have been sent*
Receive $[E_{kT}(A_G), nE_{kT}(AGG)]$ *from TDSs*
until $n^e_\Omega = 1$

TDSs side

while *(true)*
Reset $[A_G, AGG] = 0$;
Receive partition from SSI
Decrypt tup_e: $A_G \leftarrow E_{kT}^{-1}(A^e_G)$; $AGG \leftarrow nE_{kT}^{-1}(AGG^e)$
Compute aggregate for only group A_G: $[A_G, AGG]$
Encrypt aggregate: $[E_{kT}(A_G), nE_{kT}(AGG)]$
Send $[E_{kT}(A_G), nE_{kT}(AGG)]$ *to SSI*
endwhile
end
Filtering phase //evaluate HAVING clause
return $nE_{kQ}(\Omega_{final})$ *to Querier by SSI;*

Suitable queries. This solution is suitable for both kinds of queries (i.e., with small *G* like Q1, Q2 and big *G* like Q4, Q5) both in terms of efficiency (because it does not handle fake data) and feasibility (because it divides the big group into smaller ones and assigns the tasks for TDSs).

This section shows that the design space for executing complex queries with Group By is large. It presented three different alternatives for computing these queries and provided a short initial discussion about their respective accuracy, security and efficiency. Sections 8 and 9 compare in a deeper way these alternatives in terms of performance while Section 7 summarizes a comparison of these same alternatives in terms of security. The objective is to assess whether one solution dominates the others in all situations or which parameters are the most influential in the selection of the solution best adapted to each context.

6. PRACTICAL ISSUES

This section discusses complementary issues that are not core of the protocols but play an important role in ensuring that the protocols work in practice. These issues include: (i) how to manage the shared keys among TDSs and Querier; (ii) how to enforce the access control in the TDSs context; and (iii) how to calibrate the dataset subset.

6.1 Key Management

Our protocols rely heavily on the use of symmetric key cryptography. This section explains how these keys (k_Q for Querier-TDS communication and k_T for inter-TDS communication) can be managed and shared in a secure way.

State-of-the-Art. Group key exchange (GKE) protocols can be roughly classified into three classes: centralized, decentralized, and distributed [Rafaeli and Hutchison 2003]. In centralized group key protocols, a single entity is employed to control the whole group and is responsible for distributing group keys to group members. In the decentralized approaches, a set of group managers is responsible for managing the group as opposed to a single entity. In the distributed method, group members themselves contribute to the formation of group keys and are equally responsible for the re-keying and distribution of group keys. Their analysis [Rafaeli and Hutchison 2003] made clear that there is no unique solution that can satisfy all requirements. While centralized key management schemes are easy to implement, they tend to impose an overhead on a single entity. Decentralized protocols are relatively harder to implement and raise other issues, such as interfering with the data path or imposing security hazards on the group. Finally, distributed key management, by design, is simply not scalable. Hence it is important to understand fully the requirements of the application to select the most suitable GKE protocol.

Overview of Key Management. There are numerous ways to share the keys between TDSs and Querier depending on which context we consider.

In the closed context, we assume that all TDSs are produced by the same provider, so the shared key k_T can be installed into TDSs at manufacturing time. If Querier also owns a TDS, key k_Q can be installed at manufacturing time as well. Otherwise, Querier must create a private/public key and can use any GKE to exchange key k_Q. An illustrative scenario for the closed context can be: patients and physicians in a hospital get each a TDS from the hospital, all TDSs being produced by the same manufacturer, so that the required cryptographic material is preinstalled in all TDSs before queries are executed.

In an open context, a Public Key Infrastructure (PKI) can be used so that queriers and TDSs all have a public-private key pair. When a TDS or querier registers for an application, it gets the required symmetric keys encrypted with its own public key.

Since the total number of TDS manufacturers is assumed to be very small (in comparison with the total number of TDSs) and all the TDSs produced by the same producer have the same private/public key pair, the total number of private/public key pairs in the whole system is not big. Therefore, deploying a PKI in our architecture is suitable since it does not require an enormous investment in managing a very large number of private/public key pairs (i.e., proportional to the number of TDSs). PKI can be used to exchange both keys k_Q and k_T for both Querier cases i.e. owning a token or not. In the case we want to exchange k_T, we can apply the above protocol for k_Q with Querier being replaced by one of the TDSs. This TDS can be chosen randomly or based on its connection time (e.g., the TDS that has the longest connection time to SSI will be chosen).

An illustrative scenario for the open context can be: TDSs are integrated in smart phones produced by different smart phone producers. Each producer has many models and we assume that it installs the same private/public key on each model. In total, there are about one hundred models in the current market, so the number of different private/public keys is manageable. The phone's owner can then securely take part in surveys such as: what is the volume of 4G data people living in Paris consume in one month, group by network operators (Orange, SFR…).

In PKI, only one entity creates the whole secret key, and securely transfers it to the others. In the distributed key agreement protocols, however, there is no centralized key server available. This arrangement is justified in many situations—e.g., in peer-to-peer or ad hoc networks where centralized resources are not readily available or are not fully trusted to generate the shared key entirely. Moreover, an advantage of distributed protocols over the centralized protocols is the increase in system reliability, because the group key is generated in a shared and contributory fashion and there is no single-point-of-failure [Lee *et al.* 2006]. Group AKE protocols are essential for secure collaborative (peer-to-peer) applications [Lee *et al.* 2006]. In these circumstances, every participant wishes to contribute part of its secrecy to generate the shared key such that no party can predetermine the resulting value. In other words, no party is allowed to choose the group key on behalf of the whole group. These reasons lead to another way to exchange the shared key between TDSs and Querier in the open context. In this way, GKE protocols [Wu *et al.* 2011, Amir *et al.* 2004, Wu *et al.* 2008] can be used so that Querier can securely exchange the secret contributive key to all TDSs. Some GKE protocols [Amir *et al.* 2004] require a broadcast operation in which a participant sends part of the key to the rest. These protocols are not suitable for our architecture since TDSs communicate together indirectly through SSI. This incurs a lot of operations for SSI to broadcast the messages (i.e., $O(n^2)$, with n is the number of participants). Other protocols [Wu *et al.* 2008] overcome this weakness by requiring that participants form a tree structure to reduce the communication cost. Unfortunately, SSI has no knowledge in advance about TDSs thus this tree cannot be built. The work in [Wu *et al.* 2011] proposes a protocol with two rounds of communications and only one broadcast operation. However, this protocol still has the inherent weakness of the GKE: all participants must connect during the key exchange phase. This characteristic does not fit in our architecture since TDSs are weakly connected. Finally, the Broadcast Encryption Scheme (BES) [Castelluccia *et al.* 2005] requires that all participants have a shared secret in advance, preventing us from using it in a context where TDSs are produced by different manufacturers.

In consequence, we propose an adaptive GKE scheme, fitting our architecture as follows.

The Adaptive Key Exchange Protocol. Based on the decisional and computational Diffie-Hellman (CDH) assumptions, we propose a GKE protocol in [To *et al.* 2015b] in which each participant contributes a secret to the shared key. In the first step, each TDS generates a secret and sends it to Querier. Then, Querier selects a random number and computes a secret. The shared key is generated using these contributive secrets and broadcast to TDSs. Finally, upon receiving the message, only the TDSs who can compute the secret can decrypt the message and compute the shared key.

The security of the proposed GKE comes from the difficulty of the CDH problem and the one-way hash functions. Formally, there are five notions of security [Bresson and Manulis 2007] used as the standards to evaluate the security of a GKE protocol: Authenticated key exchange (AKE), Forward/backward secrecy, Contributiveness, Universal Composability (UC), and Mutual Authentication (MA). As detailed in [To *et al.* 2015b], our proposed GKE satisfies AKE security, forward/backward secrecy, and contributiveness and can also achieve MA-security at an additional cost of only two communication rounds.

The resulting protocol has three advantages over other GKEs in literature. First, it does not require that all TDSs connect at the same time to form the group, the connection of a single TDS per manufacturer being enough. The encrypted k_T could be stored temporarily on SSI so that the offline TDS can get it as soon as it comes online and still take part in the protocol (i.e., using its private key to compute the secret, then the shared key). Second, even if a TDS opts out of a SQL query in the collection phase, it can still contribute to the parallel computation in the aggregation phase. With a traditional distributed key exchange, any TDS disconnected during setup will require a new key exchange to take place. With our protocol, each TDS contributes to part of the shared secret key, the only requirement is that at least one TDS per manufacturer participates in step 1 to contribute to the secret value representing this manufacturer. Third, in terms of performance, this protocol requires only 2-round of communications. Interested readers can refer to technical report [To *et al.* 2015b] for detail analysis of the computation comparison of this GKE with other related works.

In conclusion, our way of managing encryption keys can accommodate any situation, open or closed context, central PKI or fully distributed GKE, making our protocols very versatile.

6.2 Enforcing Access Control

Contrary to statistical databases or PPDP works where the protection resides on the fact that aggregate queries or anonymized releases do not reveal any information linkable to individuals, we consider here traditional SQL queries and a traditional access control model where *subjects* (either users, roles or applications) are granted access to *objects* (either tables or views). In the fully decentralized context we are targeting, this impacts both the definition of the access control (AC for short) and its enforcement.

AC policies can be defined and signed by trusted authorities (e.g., Ministry of Health, bank consortium, consumer association). As for the cryptographic material, such predefined policy can be either installed at burn time or be downloaded dynamically by each TDS using the key exchange protocols discussed in section 6.1. In more flexible scenarios, users may be allowed to modify the predefined AC policy to personalize it or to define it from scratch. The latter case results in a decentralized Hippocratic database [Agrawal *et al.* 2002] in the sense that tuples belonging to a

same table vertically partitioned among individuals may be ruled by different AC policies. Lastly, each individual may have the opportunity to opt-in/out of a given query. Our query execution protocol accommodates this diversity by construction, each TDS checking the querier's credentials and evaluating the AC policy locally before delivering any result (either true or dummy tuples depending on the AC outcome).

But how can AC be safely enforced at TDS side? The querier's credentials are themselves certified by a trusted party (e.g., a public organization or a company consortium delivering certificates to professionals to testify their identity and roles). TDSs check the querier's credentials and evaluates the AC policies thanks to an AC engine embedded on the secure chip, thereby protecting the control against any form of tampering. Details about the implementation of such a tamper-resistant AC module can be found in [Anciaux *et al.* 2009].

6.3 Collecting Data

In scenarios where TDSs are seldom connected (e.g., TDSs hosting a PCEHR), the collection phase of the querying protocol may be critical since its duration depends on the connection rate of TDSs. However, many of these scenarios can accommodate a result computed on a representative subset of the queried dataset (e.g., if Querier wants to find out the average salary of people in France with the total population of 65 millions, it is reasonable to survey only a fraction of the population). The question thus becomes *how to calibrate the dataset subset*? Larger subsets slow down the collection phase while smaller subsets diminish the accuracy and/or utility of the results. To determine if a sample population accurately portrays the actual population, we can estimate the sample size required to determine the actual mean within a given error threshold [Cochran 1977].

We propose to use the Cochran's sample size formula [Cochran 1977] to calculate the required sample size as follow:

$$spop^{est} = \frac{\varphi^2 * \zeta^2 * \left(\frac{pop_m}{pop_m - 1}\right)}{\lambda^2 + \left(\varphi^2 * \frac{\zeta^2}{pop_m - 1}\right)}$$

with pop_m the size of the actual population, λ the user selected error rate, φ the user selected confidence level, and ζ the standard deviation of the actual population. The meaning of each parameter in this formula is explained below.

The error rate λ (sometimes called the level of precision) is the range in which the true value of the population is estimated to be (e.g., if a report states that 60% of people in the sample living in Paris have salary greater than 1300 EUR/month with an error rate of ±5%, then we can conclude that between 55% and 65% of Parisian earn more than 1300 EUR/month).

The confidence level φ is originated from the ideas of the Central Limit Theorem which states that when a population is repeatedly sampled, the average value of the attribute obtained by those samples approaches to the true population value. Moreover, the values obtained by these samples are distributed normally around the real value (i.e., some samples having a higher value and some obtaining a lower score than the true population value). In a normal distribution, approximately 95% of the sample values are within two standard deviations of the true population value (e.g., mean).

The degree of variability ζ of the dataset refers to the distribution of attributes in the population. A low standard deviation indicates that the data points tend to be

very close to the expected value; a high standard deviation indicates that the data points are spread out over a large range of values. The more heterogeneous a population, the larger the sample size required to obtain a given level of precision and vice versa.

To take into account the fact that some TDS's holders may opt out of the query, let us call opt_{out} the percentage of TDSs that opt out of the survey. Then, the required sample size we need to collect in the collection phase is:

$$S = \frac{1}{1 - opt_{out}} * spop^{est}$$

Among the three parameters, λ and φ are user selected but ζ is data-dependent. Cochran [1977] listed four ways of estimating population variances for sample size determinations: (1) take the sample in two steps, and use the results of the first step to determine how many additional responses are needed to attain an appropriate sample size based on the variance observed in the first step data; (2) use pilot study results; (3) use data from previous studies of the same or a similar population; or (4) estimate or guess the structure of the population assisted by some logical mathematical results. Usually, φ = 1.96 (i.e., within two standard deviations of the mean of the actual population) is often chosen in statistics to reflect 95% confidence level. In the experiment, because ζ is data-dependent, we will vary this parameter to see its impact to S. We also vary the error rate reflecting Querier's preference.

7. INFORMATION EXPOSURE ANALYSIS

7.1 Security of Basic Encryption Schemes

In cryptography, indistinguishability under chosen plaintext attack (IND-CPA) [Katz and Lindell. 2007] is considered as a basic requirement for most provably secure cryptosystems. While *nDet_Enc* is considered to be IND-CPA [Arasu *et al.* 2014], the maximum level of security for *Det_Enc* is PRIV [Bellare *et al.* 2007], which is a weaker notion of security than IND-CPA, due to the lack of randomness in cipertext.Then, it is important to understand how much (quantitatively) less (more) secure the *Noise_based* and *ED_Hist* are, in compare with *nDet_Enc* (*Det_Enc*). To address this question, we use two ways to measure the security level of *Noise_based* and *ED_Hist*, given *nDet_Enc* as the highest bound of security level. Each way corresponds to each increasing level of adversary's knowledge. In the first level, we assume that adversary does not know the distribution within a bucket of equi-depth histogram but only the global distribution (section 7.2). In the higher level, we address stronger attackers with more knowledge in which he knows the probability distribution of the values within each bucket (section 7.3).

7.2 Information Exposure with Coefficient

To quantify the confidentiality of each algorithm, we measure the information exposure of the encrypted data they reveal to SSI by using the approach proposed in [Damiani *et al.* 2003] which introduces the concept of coefficient to assess the exposure. To illustrate, let us consider the example in Fig. 7 where Fig. 7a is taken from [Damiani *et al.* 2003] and Fig. 7b is the extension of [Damiani *et al.* 2003] applied in our context. The plaintext table Accounts is encrypted in different ways corresponding to our proposed protocols. To measure the exposure, we consider the probability that an attacker can reconstruct the plaintext table (or part of the table)

by using the encrypted table and his prior knowledge about global distributions of plaintext attributes.

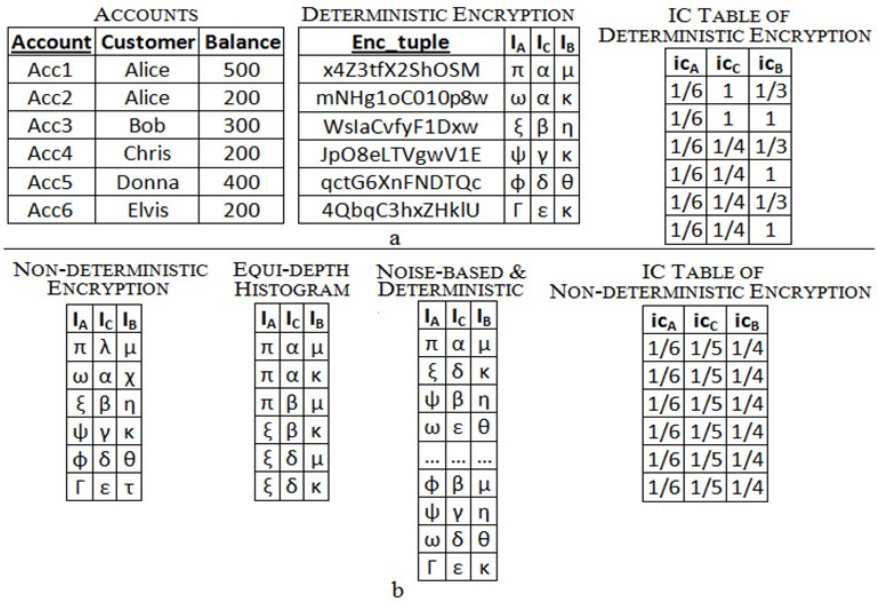


Fig. 7. Encryptions and IC tables

Although the attacker does not know which encrypted column corresponds to which plaintext attribute, he can determine the actual correspondence by comparing their cardinalities. Namely, she can determine that I_A, I_C, and I_B correspond to attributes Account, Customer, and Balance respectively. Then, the IC table (the table of the inverse of the cardinalities of the equivalence classes) is formed by calculating the probability that an encrypted value can be correctly matched to a plaintext value. For example, with *Det_Enc*, $P(a = Alice) = 1$ and $P(\kappa = 200) = 1$ since the attacker knows that the plaintexts *Alice* and *200* have the most frequent occurrences in the Accounts table (or in the global distribution) and observes that the ciphertexts a and κ have highest frequencies in the encrypted table respectively. The attacker can infer with certainty that not only a and κ represent values *Alice* and *200* (*encryption inference*) but also that the plaintext table contains a tuple associating values *Alice* and *200* (*association inference*). The probability of disclosing a specific association (e.g., *<Alice,200>*) is the product of the inverses of the cardinalities (e.g., $P(<a,\kappa> = <Alice,200>) = P(a = Alice) \times P(\kappa = 200) = 1$). The *exposure coefficient* $\mathcal{E}$ of the whole table is estimated as the average exposure of each tuple in it:

$$\varepsilon = \frac{1}{n}\sum_{i=1}^{n}\prod_{j=1}^{k} IC_{i,j}$$

Here, n is the number of tuples, k is the number of attributes, and $IC_{i,j}$ is the value in row i and column j in the IC table. Let's N_j be the number of distinct plaintext values in the global distribution of attribute in column j (i.e., $N_j \leq n$).

Using *nDet_Enc*, because the distribution of ciphertexts is obfuscated uniformly, the probability of guessing the true plaintext of a is $P(a = Alice) = 1/5$. So, $IC_{i,j} = 1/N_j$ for all i, j, and thus the exposure coefficient of *S_Agg* is:

$$\varepsilon_{S\_Agg} = \frac{1}{n}\sum_{i=1}^{n}\prod_{j=1}^{k}\frac{1}{N_j} = 1/\prod_{j=1}^{k}N_j$$

For the nearly equi-depth histogram, each hash value can correspond to multiple plaintext values. Therefore, each hash value in the equivalence class of multiplicity m can represent any m values extracted from the plaintext set, that is, there are $\binom{N_j}{m}$ different possibilities. The identification of the correspondence between hash and plaintext values requires finding all possible partitions of the plaintext values such that the sum of their occurrences is the cardinality of the hash value, equating to solving the NP-Hard *multiple subset sum problem* [Ceselli *et al.* 2005]. We consider two critical values of collision factor h (defined as the ratio G/M between the number of groups G and the number M of distinct hash values) that correspond to two extreme cases (i.e., the least and most exposure) of ε_{ED_Hist}: (1) $h = G$: all plaintext values collide on the same hash value and (2) $h = 1$: distinct plaintext values are mapped to distinct hash values (i.e., in this case, the nearly equi-depth histogram becomes *Det_Enc* since the same plaintext values will be mapped to the same hash value).

In the first case, the optimal coefficient exposure of histogram is:

$$\min(\varepsilon_{ED\_Hist}) = 1/\prod_{j=1}^{k}N_j$$

because $IC_{i,j} = 1/N_j$ for all i, j. For the second case, the experiment in [Ceselli *et al.* 2005] (where they generated a number of random databases whose number of occurrences of each plaintext value followed a Zipf distribution) varies the value of h to see its impact to ε_{ED_Hist}. This experiment shows that the smaller the value of h, the bigger the ε_{ED_Hist} and ε_{ED_Hist} reaches maximum value (i.e., $\max(\varepsilon_{ED\_Hist}) \approx 0.4$) when $h = 1$.

For Noise_based algorithms, when $n_f = 0$ (i.e., no fake tuples), R_{nf_Noise} becomes *Det_Enc* and therefore it has maximum exposure in this case. If n_f is not big enough, since each TDS generates very few fake tuples, the transformed distribution cannot hide some ciphertexts with remarkable (highest or lowest) frequencies, increasing the exposure. The bigger the n_f, the lower the probability that these ciphertexts are revealed. Exceptionally, when the noise is not random (but controlled by domain cardinality of A_G), *C_Noise* has better exposure since all ciphertexts have the same frequency ($IC_{i,j} = 1/N_j$ for all i, j):

$$\varepsilon_{C\_Noise} = \frac{1}{(n_f+1)*n}\sum_{i=1}^{(n_f+1)*n}\prod_{j=1}^{k}IC_{i,j}$$
$$= \frac{1}{n_d*n}\sum_{i=1}^{n_d*n}\prod\nolimits_{j=1}^{k}\frac{1}{N_j} = 1/\prod\nolimits_{j=1}^{k}N_j$$

The exposure coefficient gets the highest value when no encryption is used at all and therefore all plaintexts are displayed to attacker. In this case, $IC_{i,j} = 1\ \forall$ i, j, and thus the exposure coefficient of plaintext table is (trivially):

$$\varepsilon_{P\_Text} = \frac{1}{n}\sum_{i=1}^{n}\prod_{j=1}^{k}1 = 1$$

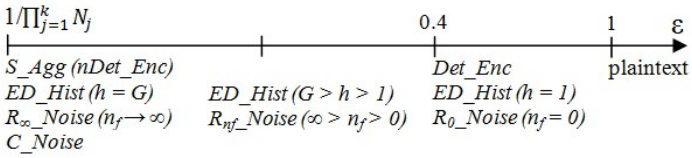


Fig. 8. Information exposure among protocols

The information exposures among our proposed solutions are summarized in Fig. 8. In conclusion, *S_Agg* is the most secure protocol. To reach the same highest security level as *S_Agg*, other protocols must pay a high price. Specifically, R_{nf}*_Noise* has to generate a very large amount of noise regardless of the value of *G*; *C_Noise* also incurs large noise if *G* is big; and *ED_Hist* must have a significant collision factor. Hence, as usual, there exists a trade-off between security and performance and the expected balance can be reached in each protocol by tuning a specific parameter (i.e., amount of noise in R_{nf}*_Noise* and *C_Noise* or number of histograms in *ED_Hist*).

7.3 Privacy Measure using Variance

In this section, we propose a stronger assumption that the adversary A possesses more knowledge of encrypted dataset than the previous section: A *knows the entire bucketization scheme and the exact probability distribution of the values within each bucket*. For example, given that bucket B has 10 elements, we assume A knows that: 3 of them have value 85, 3 have value 87 and 4 have value 95, say. However, since the elements within each bucket are indistinguishable, this does not allow A to map values to elements with absolute certainty. Then, the A's goal is to determine the precise values of sensitive attributes of some (all) individuals (records) with high degree of confidence. Eg: What is the value of salary field for a specific tuple? [Hore *et al.* 2004] proposes the **Variance** of the distribution of values within a bucket B as its measure of privacy guarantee. They first define the term *Average Squared Error of Estimation (ASEE)* as follows.

Definition ASEE: Assume a random variable X_B follows the same distribution as the elements of bucket B and let P_B denote its probability distribution. For the case of a discrete (continuous) random variable, we can derive the corresponding probability mass (density) function denoted by p_B. Then, the goal of the adversary A is to estimate the true value of a random element chosen from this bucket. We assume that A employs a statistical estimator for this purpose which is, itself a random variable, X'_B with probability distribution P'_B.

In other words, A guesses that the value of X'_B is x_i, with probability $p'_B(x_i)$. If there are N values in the domain of B, then we define *Average Squared Error of Estimation (ASEE)* as:

$$ASEE(X_B, X'_B) = \sum_{j=1}^{N} \sum_{i=1}^{N} p'_B(x_i) * p_B(x_j) * (x_i - x_j)^2$$

Theorem [Hore *et al.* 2004]: ***ASEE(X, X') = Var(X) + Var(X') + (E(X) - E(X'))²*** *where X and X' are random variables with probability mass (density) functions p and* p_0*, respectively. Also **Var(X)** and **E(X)** denote variance and expectation of X respectively.*

Proof: refer to [Hore *et al.* 2004].

Note that unlike coefficient exposure, the smaller value of *ASEE* implies the bigger security breach because the distance between guessed values and actual values is smaller, and vice versa. So the adversary tries to minimize *ASEE* as much as he can. From the theorem above, it is easy to see that A can minimize $ASEE(X_B, X'_B)$ in two ways: 1) by reducing $Var(X'_B)$ or 2) by reducing the absolute value of the difference $E(X_B) - E(X'_B)$. Therefore, the best estimator of the value of an element from bucket B that A can get, is the constant estimator equal to the mean of the distribution of the elements in B (i.e., $E(X_B)$). For the constant estimator X'_B, $Var(X'_B) = 0$. Also, as follows from basic sampling theory, the "mean value of the sample-means is a good estimator of the population (true) mean". Thus, A can minimize the last term in the above expression by drawing increasing number of samples or, equivalently, obtaining a large sample of plaintext values from B. However, note that the one factor that A cannot control (irrespective of the estimator he uses) is the true variance of the bucket values, $Var(X_B)$. Therefore, even in the worst case scenario (i.e., $E(X'_B) = E(X_B)$ and $Var(X'_B) = 0$), A still cannot reduce the *ASEE* below $Var(X_B)$, which, therefore, forms the lowest bound of the accuracy achievable by A. Hence, the data owners try to bucketize data in order to maximize the variance of the distribution of values within each bucket. These two cases corresponds to the two extreme cases of nearly equi-depth histogram (when h = 1 and h = G) as analyzed below.

When h = 1 (*Det_Enc*), since each bucket contains only the same plaintext values, and with the assumption above about additional knowledge of adversary, he can easily infer that the expected value of X'_B equals to that of X_B: $E(X'_B) = E(X_B)$. For the variance, with h = 1, the variance of X'_B gets the minimum value $Var(X'_B) = 0$ (because variance measures how far a set of numbers is spread out, a variance of zero indicates that all the values are identical). In this case, the value of *ASEE* equals to the lowest bound $Var(X_B)$.

When h = *G*, since all plaintext values collide on the same hash value, the difference between $E(X^2_B) - (E(X'_B))^2$ is big, leading to the big value of $Var(X'_B)$. So, the value of *ASEE* approaches highest bound.

As you can see, although the *coefficient exposure* and *average squared error of estimation* are different ways to measure privacy of equi-depth histogram depending on the adversary's knowledge, they give the same result.

8. ANALYTICAL COST MODEL

This section proposes an analytical cost model for the evaluation of our protocols. We calibrate this model with basic performance measurements performed on a real hardware platform (see section 8.3). We also show in section 9 that this model is accurate when compared to real measures on a real system composed of a set of TDSs. Thus the objective of this section is to provide an analytical model to assess the efficiency of the deployment of a TDSs based infrastructure for a given application without having to set up such a costly experiment.

8.1 Metrics of interest

The metrics of interest in this evaluation are the following:

- $MaxP_{TDS}$: The maximum number of TDSs *concurrently* needed in the computation. In different phases of the protocol, the optimal number of TDSs needed for the parallel computation varies and can exceed the number of

connected TDSs available at that time (i.e., demanding resource is greater than available resource), reducing the parallelism degree. Nonetheless, this value should be considered to measure the parallelism level of the protocol.

- ***Load$_Q$***: Global resource consumption for evaluating a query Q, expressed as the total size of data that all TDSs and SSI have to process. This metric reflects the scalability of the solution in terms of capacity of the system to manage a large set of queries in parallel and/or a large set of TDSs to be queried. It also provides a global view of the resource consumption (i.e., the bigger $Load_Q$, the more resource spent to process that data).
- ***Load$_{AVG}$***: Average load of all participating TDSs in the computation. While $Load_Q$ reflects the global resource consumption, this metric reflects the local resource consumption (i.e., how much load that each TDS has to incur locally in average).
- ***Load$_{MAX}$***: Maximum load of participating TDSs in the computation. Each TDS that participates in the computation incurs different load because the same TDS can participate in different steps of the protocol if connection time of that TDS is long enough. Load$_{MAX}$ reflects the possible worst case of load that a TDS can incur. This is important to measure the feasibility of the protocol. If Load$_{Max}$ is too large, maybe no TDS will ever connect for long enough.
- ***Load$_{BL}$***: Load balance among participating TDSs in the parallel computation. It is measured as the ratio of $Load_{MAX}/Load_{AVG}$. It reflects the protocol's ability to evenly divide and deliver the parallel tasks to connected TDSs.
- ***T_Q***: query response time, reflecting the responsiveness of the protocol. Since the time in the collection phase is application-dependent and is similar for all protocols, and since the time in the filtering phase is also similar for all protocols, T_Q focuses on the time spent on the aggregation phase, which is actually the most complex phase.
- ***T_{local}***: Average time that each participating TDS spends to compute the query. This metric reflects the feasibility of the solution because the longer this time, (1) the lower the probability that TDS stays connected during this time and (2) the higher the burden for an individual to accept participating in distributed queries.
- ***s_{RAM}***: Size of RAM required in each participating TDS for the computation.

The above metrics can be classified into: (i) Local resource consumption, reflecting the resource consumed locally in each TDS; (ii) Global resource consumption, reflecting the global resource needed for the whole system to answer a query. The weight associated to each of these metrics is context-dependent, as discussed in Section 8.6. These metrics are computed based on the following main parameters which reflect the characteristics and resources of the architecture:

- N_t total number of encrypted tuples sent to SSI by TDSs (without loss of generality, we consider in the model that each TDS produces a single tuple in the collection phase, hence N_t reflects also the number of TDSs participating in the collection phase);
- P_{TDS}: total number of TDSs that participate in the computation (depending on the protocol, not all connected TDSs may be involved in a computation);
- G number of groups;
- s_t size of an encrypted tuple (this size depends on the schema of the database, number of attributes needed in the query, and size of each attribute);
- T_t time spent by each TDS to process one tuple (including transfer, cryptographic and aggregation time);

- N_i^{TDS} number of TDSs that participate in the i^{th} partial aggregation phase (protocol dependent);
- α, n_{NB}, n_{ED}, reduction factors in the aggregation phase in *S_Agg*, *Noise_based* and *ED_Hist* respectively;
- n_f number of fake tuples per true tuple in *Noise_based* protocols;
- h average number of groups corresponding to each hash value in *ED_Hist*.

8.1.1 Secure Aggregation protocol

Because the aggregation phase is iterative, the time spent in this phase is the total time for all iterative steps. In the first step of this phase, the time required to download data from SSI and return temporary result is:
$t_1 = \frac{N_t}{N_1^{TDS}} * T_t$ (since there are total N_t number of tuples which are evenly divided to N_1^{TDS} connected TDSs); $t_1' = G * T_t$ (since each TDS returns at most G tuples).
Similarly, in step i of the aggregation phase, we have:
$t_i = \frac{N_{i-1}^{TDS}}{N_i^{TDS}} * G * T_t$; $t_i' = G * T_t$ ($i = 2 \text{–} n$), with n is the total number of iterative steps in this phase.
For simplicity, we assume that the reduction factor α in every step is similar:
$\alpha = \frac{N_t/G}{N_1^{TDS}} = \frac{N_1^{TDS}}{N_2^{TDS}} = \cdots = \frac{N_{n-1}^{TDS}}{N_n^{TDS}}$.
Since $N_n^{TDS} = 1$ (there are only one TDS who computes the final aggregation), the number of iterative steps is $n = \left\lceil \log_\alpha \frac{N_t}{G} \right\rceil$
The computation time of *S_Agg* is the total time of all iterative steps:

$$T_Q^{S\_Agg} = \sum_{i=1}^{n} (t_i + t_i') = \left[(\alpha + 1) \log_\alpha \frac{N_t}{G} \right] * G * T_t$$

To find the optimal time for aggregation phase, let $f(\alpha) = (\alpha + 1)\log_\alpha(N_t/G)$ (if N_t and G are fixed, the computation time of *S_Agg* is a function of α and therefore its optimal value depends on α)
We have: $\frac{df}{d\alpha} = \frac{\alpha * ln\alpha - (\alpha+1)}{\alpha * (ln\alpha)^2} * ln\left(\frac{N_t}{G}\right)$
Solving the equation $\frac{df}{d\alpha} = 0$ gives $\alpha \approx 3.6$.
We call $\alpha_{op} = 3.6$ the optimal reduction factor (i.e., $T_Q^{S\_Agg}$ gets the minimum value when $\alpha_{op} = 3.6$).
These other metrics are calculated as follows:
In each step, the participating TDSs reduces α times: $P_{TDS}^{S\_Agg} = \sum_{i=1}^{n} N_i^{TDS} = \frac{N_t}{G} * \sum_{i=1}^{n} \alpha^{-i}$
The first step requires the most number of TDSs: $Max P_{TDS}^{S\_Agg} = \frac{N_t}{\alpha G}$
The total size of data in all iterative steps:

$$Load_Q^{S\_Agg} = (N_t + \alpha G \sum_{i=2}^{n} N_i^{TDS} + G \sum_{i=1}^{n} N_i^{TDS}) * s_t$$
$$= (1 + 2\sum_{i=1}^{n} \alpha^{-i}) * N_t * s_t$$

The maximum load of the TDS that participates in all iterative steps:
$Load_{MAX}^{S\_Agg} = (n+1)\alpha G \times s_t$
The average load of each TDS can be calculated by dividing the total load of all TDSs by the total number of participating TDSs:

$$Load_{AVG}^{S\_Agg} = \begin{cases} \frac{\left(N_t + \alpha G \sum_{i=2}^{n} N_i^{TDS}\right)}{P_{TDS}^{S\_Agg}} \times s_t, if\ P_{TDS}^{S\_Agg} < N_t \\ \frac{\left(N_t + \alpha G \sum_{i=2}^{n} N_i^{TDS}\right)}{N_t} \times s_t, if\ P_{TDS}^{S\_Agg} \geq N_t \end{cases}$$

Finally, the average time of each TDS is the division of total time by total number of TDSs: $T_{local}^{S\_Agg} = \frac{\left(N_t + \alpha G \sum_{i=2}^{n} N_i^{TDS}\right) * T_t}{P_{TDS}^{S\_Agg}}$.

8.1.2 Noise_based protocols

Because all tuples belonging to one group may spread over multiple partitions, the aggregation phase includes two steps.

In the first step, each group contains $(n_f + 1) * N_t / G$ tuples in average, and we assume that there are n_{NB} TDSs handling tuples belonging to one group. The time required to download data from SSI and return temporary result in this step is:

$t_1 = \frac{(n_f+1)*N_t}{n_{NB}*G} * T_t$; $t_1' = T_t$;

In the second step, each TDS receives n_{NB} tuples belonging to one group to compute the final aggregation, so the time required is:

$t_2 = n_{NB} * T_t$; $t_2' = T_t$;

The computation time of R_{nf}_Noise is:

$$T_Q^{R_{nf}\_Noise} = \left(n_{NB} + \frac{(n_f + 1) * N_t}{n_{NB} * G} + 2\right) * T_t$$

Apply the Cauchy's inequality, we have:

$$n_{NB} + \frac{(n_f + 1) * N_t}{n_{NB} * G} \geq 2 * \sqrt{\frac{(n_f + 1) * N_t}{G}}$$

The computation time of R_{nf}_Noise gets optimal value when the optimal reduction factor is: $n_{NB} = \sqrt{\frac{(n_f+1)*N_t}{G}}$.

$P_{TDS}^{R_{nf}\_Noise} = (n_{NB} + 1) * G$

$MaxP_{TDS}^{R_{nf}\_Noise} = n_{NB} * G$

$Load_Q^{R_{nf}\_Noise} = \left[(n_f + 1) * N_t + 2n_{NB} * G + G\right] * s_t$

$Load_{MAX}^{R_{nf}\_Noise} = \left(\frac{(n_f + 1) * T_{tuple}}{n_{NB} * G} + n_{NB}\right) * s_t$

$$Load_{AVG}^{R_{nf}\_Noise} = \begin{cases} n_{NB} \times s_t \text{ , if } P_{TDS}^{R_{nf}\_Noise} < N_t \\ \frac{\left((n_f + 1) \times N_t + n_{NB} \times G\right) s_t}{N_t} \text{ , if } P_{TDS}^{R_{nf}\_Noise} \geq N_t \end{cases}$$

$T_{local}^{R_{nf}\_Noise} = Load_{AVG}^{R_{nf}\_Noise} \times \frac{T_t}{s_t}$

8.1.3 Histogram-based protocol

Let's h be the average number of groups corresponding to each hash value. By applying the Cauchy's inequality and the same mechanism as in R_{nf}_Noise, the optimal computation time is:

$T_{Q(op)}^{ED\_Hist} = \left(3 * \sqrt[3]{\frac{h*N_t}{G}} + h + 2\right) * T_t$ when the reduction factors in each step are:

$n_{ED} = \sqrt[3]{\left(\frac{h*N_t}{G}\right)^2}$; $m_{ED} = \sqrt[3]{\frac{h*N_t}{G}}$

Then, the other metrics are based on these factors as follows:

$P_{TDS}^{ED\_Hist} = \left(\frac{n_{ED}}{h} + m_{ED} + 1\right) * G$

$$MaxP_{TDS}^{ED\_Hist} = max\left\{\frac{n_{ED}}{h} \times G, m_{ED} \times G\right\}$$

$$Load_{Q}^{ED\_Hist} = (N_t + 2n_{ED} * G + 2m_{ED} * G + G) * s_t$$

$$Load_{MAX}^{ED\_Hist} = \left(\frac{h \times N_t}{n_{ED} \times G} + \frac{n_{ED}}{m_{ED}} + m_{ED}\right) \times s_t$$

$$Load_{AVG}^{ED\_Hist} = \begin{cases} \dfrac{(N_t + n_{ED} * G + m_{ED} * G)s_t}{\left(\frac{n_{ED}}{h} + m_{ED} + 1\right)G}, if\ P_{TDS}^{ED\_Hist} < N_t \\ \dfrac{(N_t + n_{ED} * G + m_{ED} * G)s_t}{N_t}, if\ P_{TDS}^{ED\_Hist} \geq N_t \end{cases}$$

$$T_{local}^{ED\_Hist} = \frac{(N_t + n_{ED} * G + m_{ED} * G) * T_t}{(n_{ED}/h + m_{ED} + 1) * G}$$

Note that this is just a subset of the complete cost model which can be found in the technical report [To *et al.* 2013].

8.2 Sample size in the collection phase

Let us consider a nation-wide study, taking the population of France, pop_m = 65*10^6, as a representative example. We fix φ = 1.96 for 95% confidence level (the most popular value used in statistics). We assume there are 10% people who do not want to answer the query (i.e., opt_{out} = 10%). We vary two parameters to see their impact to the sample size required in the collection phase: λ = 0.05, 0.01, 0.005, 0.002, 0.001; the variance ζ^2 = 5, 10, 20, 50, 100.

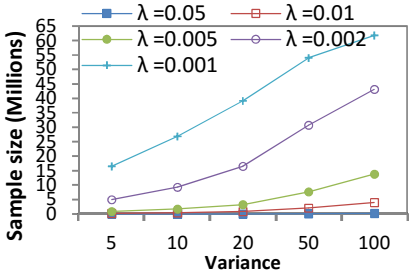


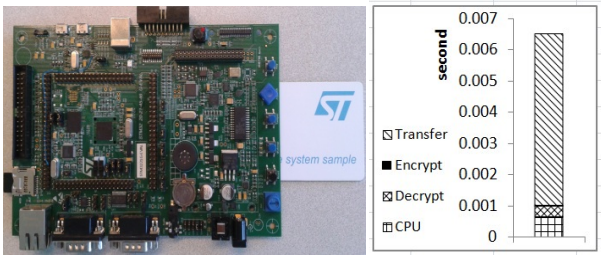
Fig. 11. Estimated sample size

As shown in Fig. 11, the higher the precision of the query result and the bigger the variability of the dataset, the higher the number of data we need to collect and therefore the longer the collection time that querier has to wait for and vice versa. For instance, if the querier wants 0.2% precision and the dataset has high variability, he has to wait until almost all the dataset is collected. However, if the querier queries low variable dataset, or if he does not want high precision, collecting only 10% or even 1% of the population is enough. Note that our system also supports opinion poll quota methods.

8.3 Unit test

To calibrate our model, we performed unit tests on the development board presented in Fig. 12a. This board exhibits hardware characteristics representative of secure tokens-like TDSs, including those provided by Gemalto (the smartcard world leader), our industrial partner. This board has the following characteristics: the microcontroller is equipped with a 32 bit RISC CPU clocked at 120 MHz, a crypto-coprocessor implementing AES and SHA in hardware (encrypting or decrypting a block of 128bits costs 167 cycles), 64 KB of static RAM, 1 MB of NOR-Flash and is

connected to a 1 GB external NAND-Flash and to a smartcard chip hosting the cryptographic material. The device can communicate with the external world through USB full speed. The speed in theory is 12 Mbps but the real speed measured with the device is around 7.9 Mbps.



a) b)

Fig. 12. Hardware device & its internal time consumption

We measured on this device the performance of the main operations influencing the global cost, that is: encryption, decryption, hashing, communication and CPU time, and put these numbers as constants in the formulas. Fig. 12b depicts the internal time consumption of this platform to manage partitions of 4KB. The transfer cost dominates the other costs due to the network latencies. The CPU cost is higher than cryptographic cost because (1) the cryptographic operations are done in hardware by the crypto-coprocessor and (2) TDS spends CPU time to convert the array of raw bytes (resulting from the decryption) to the number format for calculation later. Encryption time is much smaller than decryption time because only the result of the aggregation of each partition needs to be encrypted.

Other TDSs (e.g., smart meters) may be more powerful than smart tokens, although client-based hardware security is always synonym of low power. Anyway, as this section will make clear, the internal time consumption turns out not to be the limiting factor. Hence our choice of considering low-power TDSs in this experiment is expected to broaden our conclusions.

8.4 Performance comparisons

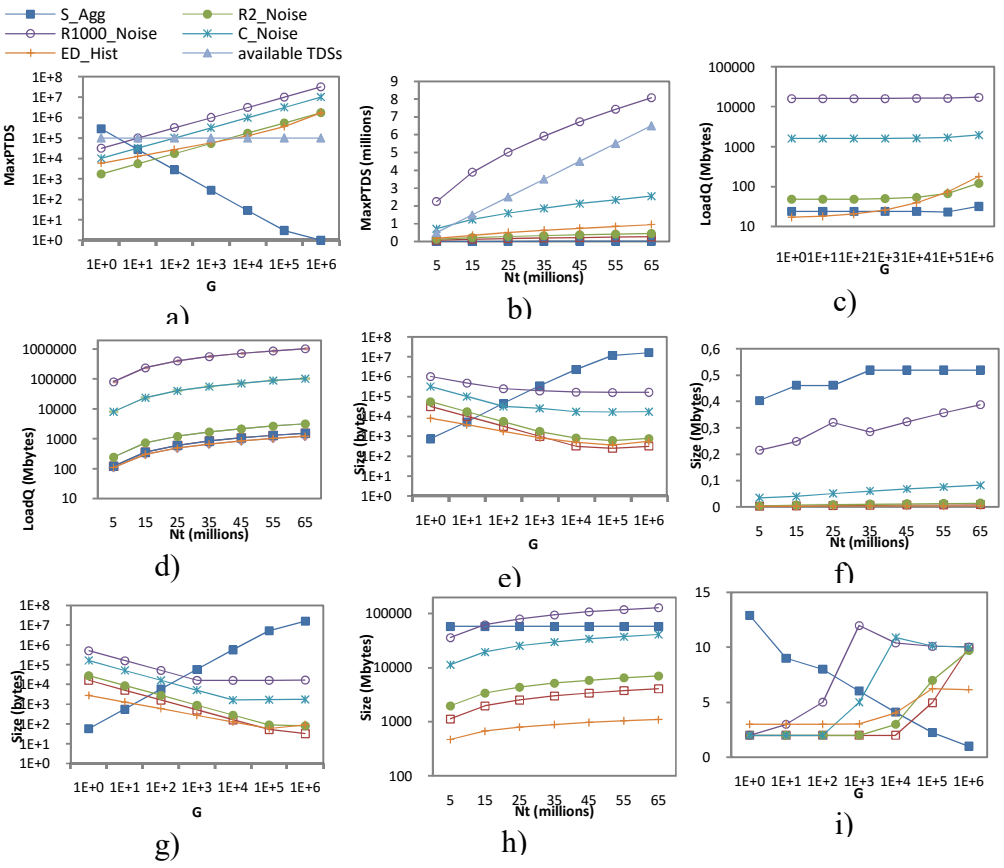
In this study, we concentrate on the performance of Group By queries since they are the most challenging to compute. We vary the dataset size (N_t varies from 5 to 65 million), the number of groups (G varies from 1 to 10^6) as well as the number of TDSs participating in the computation as a percentage of all TDSs connected at a given time (varying from 1% to 100%). For each study, we fix two parameters and vary the others. When the parameters are fixed, N_t=10^6, G=10^3, s_t=16b, T_t=16µs, h=5 and the percentage of TDS connected is 10% of N_t. We also compute and use the optimal value for all reduction factors as well as for N_i^{TDS}. In the figures, we plot two curves for R_{nf}_Noise protocols, R_2_Noise (n_f = 2) and R_{1000}_Noise (n_f = 1000) to capture the impact of the ratio of fake tuples. We summarize below the main conclusions of the performance evaluation. A more detailed study is provided in a technical report [To *et al.* 2013].

In what follows, we study each of the aspects of the protocol that seem important. We draw conclusions on the use cases for each protocol in section 8.6.

Parallelism requirement ($MaxP_{TDS}$). Fig. 13a presents $MaxP_{TDS}$ with varied G. Since S_Agg does not need too many TDSs for parallel computing, the demand of

connected TDSs for computation is almost satisfied. Unlike *S_Agg*, the other solutions need a lot of TDSs for the parallel computation, and when G increases to a specific point, the available resource does not meet these demands, reducing the parallel deployment of these solutions. In Fig. 13b, when G is not too big (i.e., G=1000), most of the protocols can fully deploy de parallel computation (except R_{1000}*_Noise*).

Resource consumption (Load$_Q$). Fig. 13c and 13d show $Load_Q$ respectively in terms of G and N_t. Not surprisingly, the total load of *Noise_based* protocols is highest because of the extra processing incurred by fake tuples. However, n_f depends only on N_t, so when G increases, the total load of *Noise_based* protocols remains constant. Other protocols generate much lower and roughly comparable loads. In general, in Fig. 13d, $Load_Q$ increases steadily due to the increase of N_t.



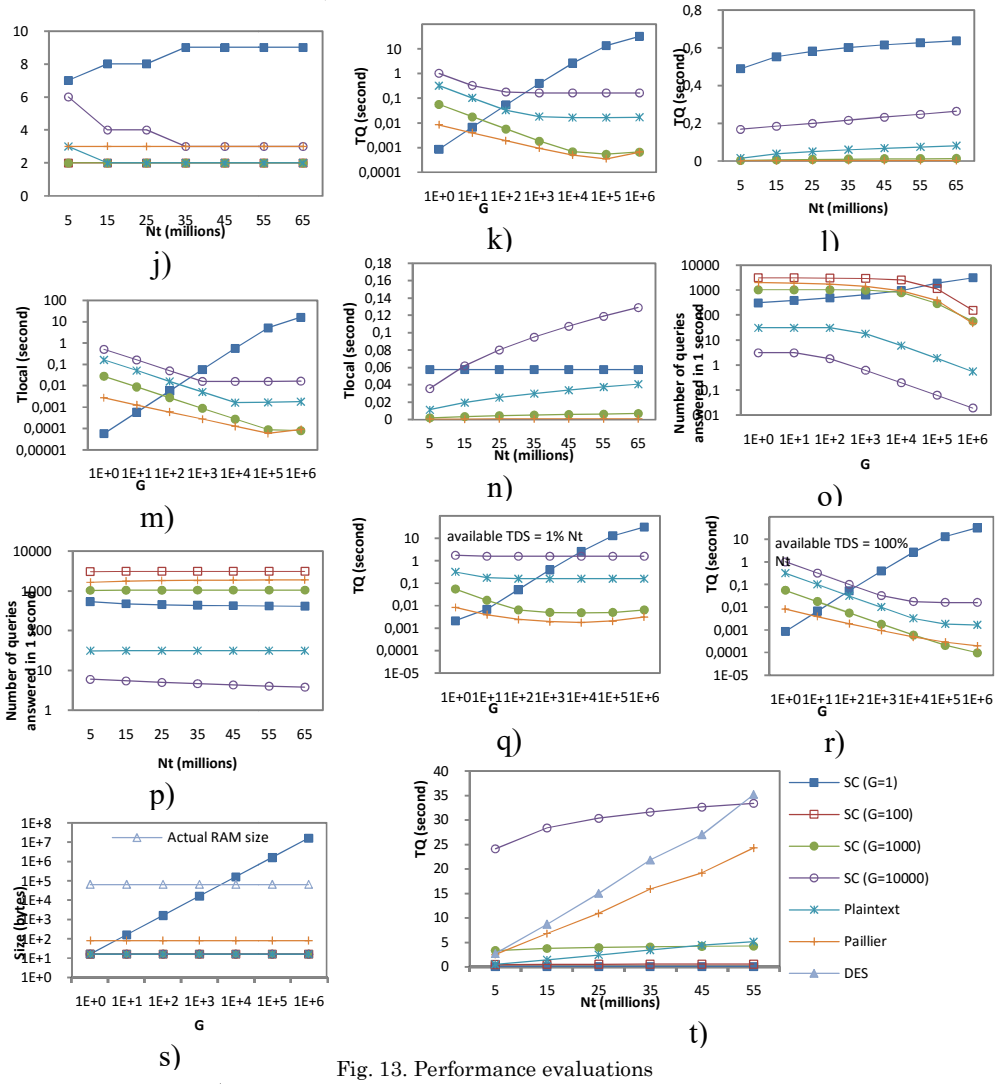


Fig. 13. Performance evaluations

Maximum load (Load$_{MAX}$). The maximum load of a particular TDS is illustrated in Fig. 13e. In *S_Agg*, when G increases, due to the increasing size of partial aggregation, each TDS has to process bigger aggregation, resulting in the increase of *Load$_{MAX}$*. Also, when G increases, the number of participating TDSs decreases, so each participating TDS has to incur higher load. For others, when G increases, since N_t remains unchanged, the number of tuples in each group decreases and the number of participating TDSs increases. Consequently, each TDS processes less tuples, and thus *Load$_{MAX}$* decreases. In other words, the parallel level in this case is high, reducing the maximum load that a particular TDS incurs. In Fig. 13f, when N_t increases, the number of participating TDSs also increases proportionally. So, in general, the *Load$_{MAX}$* remains stable except a slight increase in R_{1000}*_Noise* and *C_Noise*.

Average load (Load$_{AVG}$). Fig. 13g is the average load of every participating TDS. In *S_Agg*, since the total load stays almost constant and the number of participating TDSs declines steeply when G increases, the average load goes up. In the R_{1000}*_Noise*and *C_Noise*, the high total load is constant and all available connected TDSs participate in the computation when G varies from 10^3-10^6, thus every TDSs

incur the same amount of load. For the rest, $Load_{AVG}$ decreases when G increases, because there is more number of participating TDSs but the total load is almost unchanged. In Fig. 13h, although *C_Noise* has higher $Load_Q$ than *S_Agg*, the number of participating TDSs in *S_Agg* is much less than that in *C_Noise*, and therefore the $Load_{AVG}$ of *C_Noise* is less than that of *S_Agg*.

Load balance (Load$_{BL}$). Fig. 13i and 13j presents the load balance of solutions. Because of the low parallelism, *S_Agg* is the most unbalanced protocol. R_2*_Noise* divides the load evenly among participating TDSs. *ED_Hist* has worse load balance than R_2*_Noise* since each TDS has to process a partition including h groups while in R_2*_Noise* a partition composes of only one group.

Query response time (T_Q). Fig. 13k shows the impact of G over T_Q. In all protocols but *S_Agg*, T_Q depends on the total number of tuples in each group (resp. bucket for *ED_Hist*) because all groups (resp. buckets) are processed in parallel. Hence, when G increases while N_t remains constant, the number of tuples in each group (resp. bucket) decreases and so does T_Q. In *S_Agg*, when G increases, the size of each partial aggregation increases accordingly, and so does the time to process it and in consequence, so does T_Q. Fig. 13l shows that, for *ED_Hist*, when N_t increases, the number of TDSs which can be mobilized for processing increases accordingly, leading to a minimal impact on execution time. This statement is true also for R_{nf}*_Noise* protocols with the difference that the greater number of fake tuples generates extra work which is not entirely absorbed by the increase of parallelism. For *S_Agg*, the number of iterative steps increases with N_t and so does T_Q.

Local execution time (T_{local}). Fig. 13m and 13n plot the average execution time of every participating TDSs varying G and N_t respectively. It shows that all protocols benefit from an increase of G except *S_Agg*. This is due to the fact that, in *S_Agg*, less TDSs can participate in the parallel computation, and therefore each TDS has to process a higher load of bigger partial aggregations. Other protocols benefit from the fact that the computing load is shared evenly between TDSs. Fig. 13n shows that all protocols but *Noise_based* protocols are insensitive to an increase of N_t again thanks to independent parallelism. The bad behavior of *Noise_based* protocols is explained by the fact that the number of fake tuples increases linearly with N_t and this increased load cannot be entirely absorbed by parallelism because the number of TDSs available for the computation is bounded in this setting by 10% of the participating TDSs.

Throughput. In general, throughput is the amount of work that a computer can do in a given period of time. Applied in our case, throughput is measured as the number of queries that our distributed system can answer in a given time period, reflecting the efficiency of our protocols (cf., Fig. 13o and 13p). In Fig. 13o, when G increases, the number of participating TDSs for each query increases and the execution time for each query does not reduce considerably, resulting in the reduction of throughput for all solutions. The throughput of *S_Agg*, however, increases because P_{TDS} reduces much faster than the execution time for each query when G increases. In Fig. 13p, when N_t increases, the throughput remains constant for all solutions due to the proportional increase of participating TDSs. The *ED_Hist* solution has the highest throughput because it needs least participating TDSs and shortest execution time for each query. For *S_Agg*, although the response time for each query is long, the P_{TDS} is very low, resulting in high throughput. For the R_{1000}*_Noise*, since it not only demands very high number of P_{TDS} (to process fake tuples), but also responses slowly for each query, its throughput is worst.

Elasticity issues. A distributed and parallel system is said to be elastic if it can mobilize smoothly a variable part of its computing resources to meet run time requirements. Fig. 13q,r,k measures the elasticity of all protocols by varying the computing resource and assessing its impact on T_Q. The computing resource is materialized here by the number of TDSs which can be mobilized to contribute to a given computation. It is expressed by a percentage of the TDSs contributing to the collection phase. Fig. 13q (resp. Fig. 13r, Fig. 13k) considers scarce (resp. abundant, intermediate) computing resource in the sense that only 1% (resp. 100%, 10%) of the TDSs contributing to the collection phase contributes to the rest of the query computation. Comparing these figures shows that, when the resource is scarce, the parallel computation is not completely deployed, resulting in a longer time to answer the query and vice-versa. Since *S_Agg* does not depend on the number of available TDSs (but on G and on the memory size of TDS), its performance is not impacted by a fluctuation of the resource available. In other words, *S_Agg* has lowest elasticity.

Memory size. Fig. 13s details the memory's size required for the computation in each TDS when G is varied. Because the only factor that impacts the memory's size requirement is G but not N_t, we assess this metric by varying only G. The *Noise_based* solutions require least memory because each partition sent to TDS contains tuples belonged to only one group due to the *Det_Enc*, and thus TDSs store only one group in memory regardless of the value of G. The *ED_Hist* requires more memory because each TDS needs to process the partition having the same hash value and each hash value corresponds to multiple (i.e., h) groups in the first aggregation phase. The *S_Agg* needs highest memory because each TDS has to store the whole partial aggregation (which includes many groups) in the RAM. So, when G increases, the memory needed for storing the whole aggregation also increases linearly. When G is too big (i.e., G >1000), the s_{RAM} exceeds the actual RAM's size of TDS, and thus *S_Agg* is not feasible in this case[15].

8.5 Comparison with State of the Art

In order to provide a baseline comparison in terms of performance (and not security), Fig. 13t compares the performance of *S_Agg*, our most secure solution, with server-based solutions working on encrypted data. We consider the performance of two well-known encryption schemes, a symmetric one (i.e., DES) and a homomorphic one (i.e., Paillier [Paillier 1999]), as measured in [Ge and Zdonik 2007]. In DES method, each value is decrypted on the server and the computation is performed on the plaintext. Clearly this method is not a viable solution in our security model, because the database server must have access to the secret key or plaintext to answer the query, violating the security requirements. In Paillier's method, the secure modern homomorphic encryption scheme, which typically operates on a much larger (encryption) block size (say 2K bits) than single numeric data values, is used to densely pack data values in an encryption block. Then, the database server performs the computation directly on ciphertext blocks which are then passed back to a trusted agent (i.e., the Key Holder) to perform a final decryption and simple calculation of the final result. The strength of this method is due to the dense packing of values to reduce the number of modular multiplications and the minimization of the number of expensive decryption operations. We refer to the author's experimentations, which were run on now outdated hardware[16], since both methods were implemented in C-

[15]Swapping between FLASH memory and RAM is used in this case

[16] However, this hardware is still orders of magnitude superior to the secure tokens we use.

Store[17]which was run on a Linux workstation with an AMD Athlon-64 2Ghz processor and 512 MB memory [Ge and Zdonik 2007]. We also compare its performance with C-Store using no encryption at all. We ran an AVG query varying G and the database size. The result (Fig. 13t) shows that, with homomorphic encryption scheme (generalized Paillier), C-Store runs slightly faster than using DES for encryption due to the saving in the decryption cost during execution. It turns out that *S_Agg* outperforms DES and Paillier when the number of grouping attributes is small (i.e., $G \leq 1000$) since it can exploit the parallel calculation of TDSs to speed up the computation and becomes worse after this threshold.

Although these algorithms are a little dated, the objective is simply to provide a baseline comparison, to show the effectiveness of our approach and demonstrates the strength of large-scale parallel computation even when modest hardware is available on the participant's side. Fig. 13t matches this objective explicitly.

8.6 Conclusion: Trade-off between criteria

Fig.14 summarizes and complements the experimental results described above through a qualitative comparison of our proposed protocols over all criteria of interest to perform a choice.

Each axis can be interpreted as follows. Local resource consumption axis refers to T_{local} metrics and compares the protocols in terms of feasibility, i.e., is the resource consumed by a single TDS compatible with the actual computing power of the targeted TDSs. This question is particularly relevant for low-end TDSs (e.g., smart tokens) and of lesser interest for high-end TDSs. *S_Agg* is at the worst extremity of this axis because the final aggregation must be done by a single TDS while *ED_Hist* occupies the other extremity thanks to its capacity to evenly share the load among all TDSs. That also explains why in Load Balance axis *ED_Hist* better balances the load among TDSs than *S_Agg*. *Noise_based* protocols are in between because they also share the load evenly but at the price of managing a large number of fake tuples. Note that the relative position of *S_Agg* and *ED_Hist* is reversed in the Global Resource Consumption and Satisfied Level of Parallel Deployment axis which refers to $Load_Q$ and $MaxP_{TDS}$ metrics and compares the scalability of the protocols in terms of number of parallel queries which can be computed and their ability of fully parallel computation, respectively. Indeed, the total number of TDSs mobilized by *S_Agg* for one single query computation is much smaller than that of *ED_Hist*. Regarding the Responsiveness axis, the relative ordering of *S_Agg* and *ED_Hist* actually differs depending on G. According to Fig. 13, *S_Agg* outperforms *ED_Hist* for small G (smaller than 10) and is dominated by *ED_Hist* for larger G. Finally, Elasticity axis is a direct translation of the conclusions drawn in Section 8.4 and Confidentiality axis recalls the conclusion of Section 7.

[17] http://db.csail.mit.edu/projects/cstore/

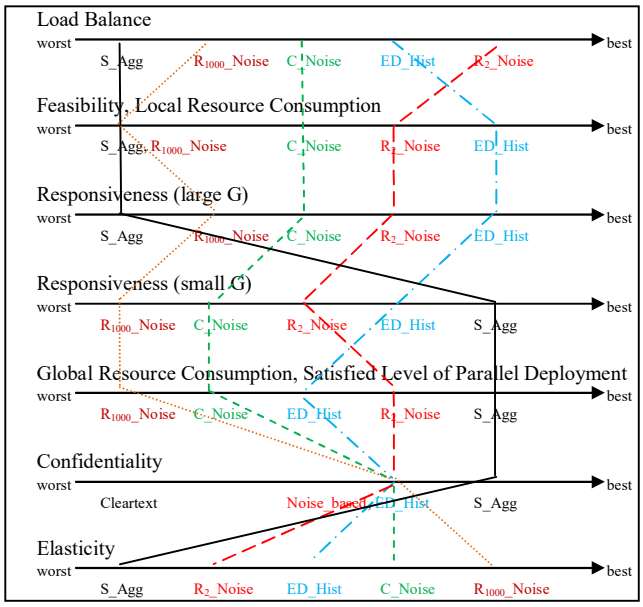


Fig. 14. Comparison among solutions

This figure makes clear that *Noise_based* protocols are always dominated either by *S_Agg* or *ED_Hist* and should be avoided. However, choosing between the other two depends on the application's characteristics, and Fig. 14 should be used to decide. Let us consider a first scenario where individuals manage their data (e.g., their medical folder) using a secure Personal Data Server embedded in a smart token-like TDS [Allard *et al.* 2010]. In such a scenario, individuals are likely to connect their TDS seldom, for short periods of time (e.g., when visiting a doctor) and would prefer to save resource for executing their own tasks rather than being slowed down by the computation of external queries. According to Fig. 14, *ED_Hist* best matches the above requirements. Conversely, let us consider a smart metering platform composed of power meter-like TDSs, connected all the time and mostly idle. In this case, TDSs' owners do not care how much resources are monopolized to compute queries and the primary concern is for the distribution company to maximize the capacity to perform global computation. *S_Agg* is more appropriate in this case. In short, *ED_Hist* and *S_Agg* are the two best solutions and the final choice depends on the weight associated to each axis for a given application.

9. PERFORMANCE MEASUREMENTS ON REAL HARDWARE

To test the accuracy of our proposed cost models given in previous section, we compare the values taken from experiments conducted on real hardware with that of the cost models.

9.1 Experiment Setting

This section experimentally verifies the proposed cost models using 20 ZED secure tokens[18] (Fig. 15) playing the role of a pool of TDSs used during the processing phase (ie. after the collection phase has been performed). The experiment is tested on a

[18] These secure tokens are used in different universities and FabLabs in France and will be soon distributed under an open-hardware licence. In terms of hardware resources, they share many commonalities with the development board described in Section 9.3.

Centrino Core 2 Duo PC with 2.4 Ghz CPU and 4Gbytes RAM, playing the role of SSI. The 20 ZED tokens communicate with the PC through USB port (Fig. 16). We verify our cost models on (i) Query response time (T_Q), (ii) Resource consumption ($Load_Q$), (iii) Local execution time (T_{local}) and (iv) Load balance ($Load_{BL}$) among tokens. The low number of tokens has an influence on a certain number of results, but overall we believe that our prototype demonstrates that the cost model is accurate.

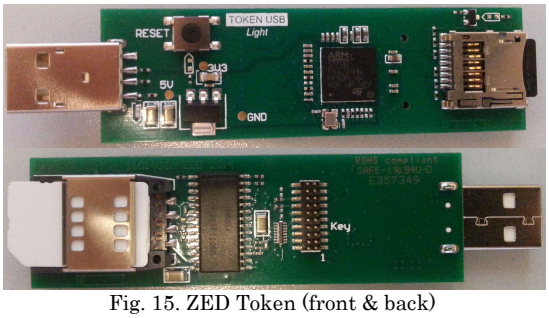


Fig. 15. ZED Token (front & back)

The prediction accuracy is measured as the error between actual and estimated values in answering a query. Specifically, let *act* be the actual values when running on real tokens and *est* be the estimated values when applying our proposed cost model, we adopt the following error rate definition [Tao *et al.* 2003]:

$$Err = \frac{|est - act|}{act}$$

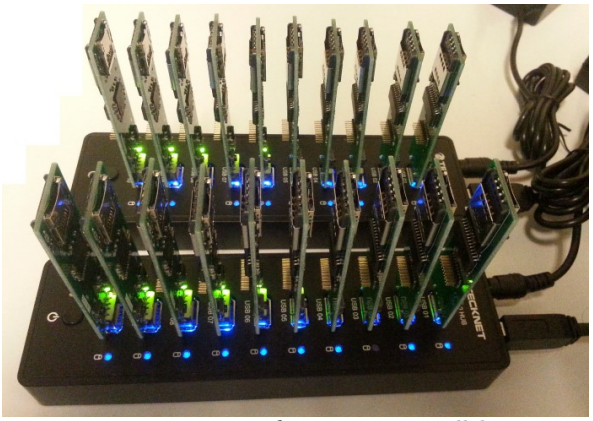


Fig. 16. Twenty tokens running parallel

Similar to the performance comparison done with the cost model in the previous section, we vary two parameters (i.e., G and N_t) to see its impact to the error rate. When N_t varies up to one million, G is fixed at 100, and when G varies from 50 to 400 groups, N_t is fixed to one million tuples.

9.2 Comparison

In the following figures, for each metric, the first graph represents the real absolute value measured using the 20 ZED tokens while the second graph represents the relative error between these real values and the values predicted by the cost model. This second graph captures the accuracy of our cost model.

The first set of experiments verifies the correctness of the query response time. Figure 17a plots T_Q varying G. The Noise protocol has the longest execution time due to fake tuples, and S_Agg runs longer than ED_Hist since each token has to process large partial aggregation. This observation is similar to that in figure 13k, giving a maximum estimation error under 7% in figure 17b. When N_t varies, T_Q increases linearly in figure 17c, similarly to figure 13l. However, the increase rate of figure 17c is bigger than that of figure 13l because in the case of 20 participating tokens, parallelism is not fully deployed due to the limited number of tokens. On the contrary, in figure 13l where we have many participating TDSs, the parallel computation is completely deployed, resulting in a lower increase rate when the data load increases. The maximum error is around 10% in figure 17d.

Figures 17 e-h show the resource consumption error rate. Similar to figure 13c, all protocols in figure 17e incur constant loads (except a very small increase in case of S_Agg) when G varies because the total number of tuples is fixed. This gives a very low error rate for EDHist and Noise protocols (around 2%) and a rather low error rate for S_Agg (less than 8%). Similarly, the variation of N_t yields the linear increase of $Load_Q$ in both figures 13d and 17g, giving an accurate result (around 2%-3% error) in figure 17h.

Figure 17 i-l depicts the error rate on local execution time. Except the small linear increase of S_Agg in figure 17i, Noise and ED_Hist remain constant. This contradicts the decreasing trend of Noise and ED_Hist in figure 13m when G varies. This can be explained again by the limited number of tokens. If the global data load keeps unchanged, and the number of tokens remains at twenty, each token processes the same amount of data in average even when G varies (except for S_Agg since the size of the aggregations depends on G). In contrast, when G increases in figure 13m, the number of participating tokens also increases, reducing the average connecting time for each token to process less load. Notice that when G increases over 1000 in figure 13m, the T_{local} of C_Noise and R_{1000}_Noise also remains constant since the number of connecting TDSs is less than the required TDSs to fully deploy parallel computation. We believe this explanation reinforces the credibility of our cost model since this trend repeats in figure 17i. When varying N_t, all protocols increase linearly in the experiment (figure 17k), while they remain unchanged in the cost model (figure 13n), except for Noise protocols. The reason of this difference is that when the total load increases while the number of tokens remain fixed (figure 17k), or when the number of tokens increases but does not meet the demand for an optimal parallel computing (Noise protocols in figure 13n), each token has to connect longer to process a bigger load. This is not the case for S_Agg and ED_Hist in the cost model since the increase rate of total load is less than that of connecting TDSs (in the cost model we assume that the percentage of connected TDSs is 10% of N_t).

Figure 17m displays the error rate of load balance among tokens. Since the total load is divided evenly among twenty tokens, the load balance remains at approximately 1 because all twenty tokens incur nearly the same load, yielding extremely accurate prediction (with maximum error less than 2% in figure 17n, except for S_Agg). Similarly, when N_t varies in figure 17o, Noise and ED_Hist have better load balance than S_Agg since some tokens in S_Agg have to process big aggregations to produce the final result. This observation conforms to the figure 13j where S_Agg has also the most unbalanced load among protocols.

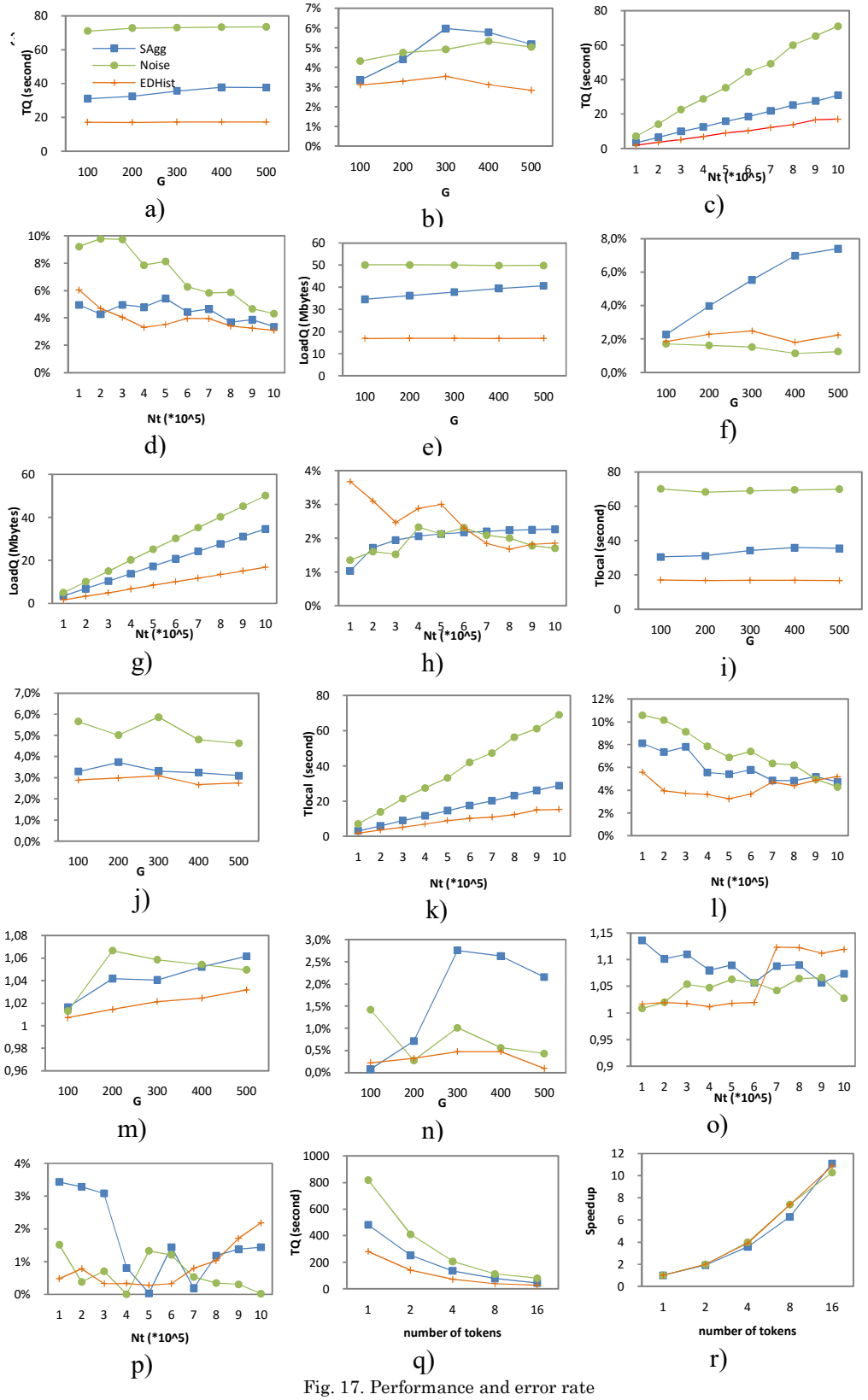


Fig. 17. Performance and error rate

As a summary of this section, although we can measure some differences between the cost model predictions and the real measurements, the error rate remains around few percents and the trends of all graphs in figure 17 are similar to the trends observed in figure 13. We believe the differences arise mostly from the inability to fully deploy the parallel computation due to limited connecting TDSs in the experiments. We plan on experimenting on larger sets of tokens in the future.

9.3 Scalability of the System

To test the ability of our system to scale up to millions of tokens in real life applications, we measure its speedup when increasing the number of tokens. Specifically, the speedup of our system is measured as follow:

$$S(n) = T(1)/T(n)$$

with $T(n)$ being the execution time using n tokens.

We vary the number of tokens to measure the execution time in figure 17q. From that, we calculate the speedup when doubling the number of tokens each time.

In figure 17r, the speedup approaches 12x when we use 16 tokens. When the number of tokens doubles, the average speedup ratios of *S_Agg, Noise* and *EDHist* are 1.82, 1.81 and 1.83 respectively. These speedup ratios let us expect that our system should be able to scale to millions of tokens (given an equivalent increase in power of the SSI) in real applications with reasonable execution time and speedup. This result is not surprising considering that all protocols exhibit mainly independent parallelism.

10. CONCLUSION

An ever increasing amount of personal data is collected and ends-up on servers. Decentralized architectures, devised to help individuals better protect their privacy, hinder global treatments and queries, impeding the development of services of great interest. This article presents a first attempt to fill this gap. It capitalizes on secure hardware advances promising soon the presence of a Trusted Execution Environment at low cost in any client device (trackers, smart meters, sensors, cell phones and other personal devices).

Based on this statement, we have proposed new query execution protocols to compute general SQL queries while maintaining strong privacy guarantees. The objective was not to find the most efficient solution for a specific problem but rather to perform a first exploration of the design space. We proposed three very different protocols and compared them according to different axes. The encouraging conclusion is that a good performance/security trade-off can be found in many situations and that the proposed protocols can scale up to nation-wide contexts.

We expect that this work will pave the way for the definition of future fully decentralized privacy-preserving querying protocols. The main research directions we foresee are: (1) extend the threat model to (a small number of) compromised TDSs and (2) perform performance study on large scale TDS platforms. The on-going deployment of very large TDS platforms (e.g., the Linky power meters installed by EDF in France or the growing interest for PCEHR hosted in secure tokens) would enable point (2) while providing a strong motivation to investigate issue (1).

ACKNOWLEDGMENTS

The authors wish to thank Anne Cantaut from INRIA-SECRET team and Matthieu Finiasz from CryptoExperts for their help in defining and proving the security of our GKE protocol, and Philippe Bonnet from University of Copenhagen for fruitful

discussions on this paper. This work is partly supported by ANR Grant KISS (*Keep your Information Safe and Secure*) n° ANR-11-INSE-0005, by the Paris-Saclay Institut de la Société Numérique funded by the IDEX Paris-Saclay, ANR-11-IDEX-0003-02 and by INRIA Project Lab CAPPRIS.

REFERENCES

Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan B. Zdonik. 2003. Aurora: a new model and architecture for data stream management. *VLDB Journal*. 12, 2, 120-139.

Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2002. Hippocratic databases. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*. Hong Kong, 143-154.

Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. 2004. Order-preserving encryption for numeric data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'04)*. Paris, 563-574.

Tristan Allard, Nicolas Anciaux, Luc Bouganim, Yanli Guo, Lionel Le Folgoc, Benjamin Nguyen, Philippe Pucheral, Indrajit Ray, Indrakshi Ray and Shaoyi Yin. 2010. Secure Personal Data Servers: a Vision Paper. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB'10)*. Singapore, 25-35.

Tristan Allard, Benjamin Nguyen, and Philippe Pucheral. 2014. MetaP: Revisiting Privacy-Preserving Data Publishing using Secure Devices. *Distributed and Parallel Databases*. 32, 2, 191-244.

Hani Alzaid, Ernest Foo, and Juan G. Nieto. 2008. Secure Data Aggregation inWireless Sensor Networks: A Survey. In *Proceedings of the 6th Australasian Information Security Conference (AISC'08)*. 93-105.

Georgios Amanatidis, Alexandra Boldyreva, and Adam O'Neill. 2007. Provably-secure schemes for basic query support in outsourced databases. In *DBSec*. Lecture Notes in Computer Science, volume 4602, Springer. 14-30.

Yair Amir, Yongdae Kim, Cristina Nita-Rotaru, and Gene Tsudik. 2004. On the performance of group key agreement protocols. *ACM Transactions on Information and System Security (TISSEC)*. 7, 3, 457-488.

Nicolas Anciaux, Luc Bouganim, and Philippe Pucheral. 2009. Hardware Approach for Trusted Access and Usage Control. Handbook of research on Secure Multimedia Distribution (Chapter A). IGI Global.

Nicolas Anciaux, Philippe Bonnet, Luc Bouganim, Benjamin Nguyen, Philippe Pucheral and Iulian Sandu-Popa. 2013. Trusted Cells: A Sea Change for Personal Data Services. In *CIDR*. Asilomar, USA.

Arvind Arasu, Ken Eguro, Raghav Kaushik, and Ravi Ramamurthy. 2014. Querying Encrypted Data (Tutorial). In ACM SIGMOD Conference.

Mihir Bellare, Alexandra Boldyreva, and Adam O'Neill. 2007. Deterministic and efficiently searchable encryption. In *CRYPTO*. Lecture Notes in Computer Science, volume 4622. 535–552.

Emmanuel Bresson, Olivier Chevassut, Abdelilah Essiari, and David Pointcheval. 2004. Mutual authentication and group key agreement for low-power mobile devices. *Computer Communications*. 27, 17, 1730-1737.

Emmanuel Bresson and Mark Manulis. 2007. Malicious Participants in Group Key Exchange: Key. Control and Contributiveness in the Shadow of Trust. In *ATC*, 395–409.

Emmanuel Bresson, Mark Manulis and Joerg Schwenk. 2007. On Security Models and Compilers for Group Key Exchange Protocols. *IWSEC*. 4752, 292–307.

Claude Castelluccia, Einar Mykletun, and Gene Tsudik. 2005. Efficient Aggregation of Encrypted Data in Wireless Sensor Networks. In *Mobiquitous*. 109-117.

Ceselli, A., Damiani, E., De Capitani di Vimercati, S., Jajodia, S., Paraboschi, S., Samarati, P.: Modeling and assessing inference exposure in encrypted databases. ACM TISSEC, vol 8(1), pp. 119-152, (2005)

William Gemmell Cochran. 1977. Sampling Techniques. John Wiley, 3rd edition.

Ernesto Damiani , Sabrina De Capitani di Vimercati , Sushil Jajodia , Stefano Paraboschi, and Pierangela Samarati. 2003. Balancing confidentiality and efficiency in untrusted relational DBMSs. In *ACM CCS*. 93-102.

Ebaa Fayyoumi and B. John Oommen. 2010. A survey on statistical disclosure control and micro-aggregation techniques for secure statistical databases. *Software: Practice and Experience*. 40, 12, 1161-1188.

The World Economic Forum. Rethinking Personal Data: Strengthening Trust. May 2012.

Benjamin C. M. Fung, Ke Wang, Rui Chen, and Philip S. Yu. 2010. Privacy-Preserving Data Publishing: A survey of Recent Developments. *ACM Computing Surveys*. 42, 4, 1-53.

Tingjian Ge, and Stan Zdonik. 2007. Answering aggregation queries in a secure system model. In *VLDB*. Vienna, 519–530.

Craig Gentry. 2009. Fully homomorphic encryption using ideal lattices. In *STOC*. Maryland. 169-178.

Shafi Goldwasser and Silvio Micali. Probabilistic encryption. 1984. Journal of Computer and System Sciences, 28(2):270–299.

Hakan Hacigumus, Bala Iyer, Chen Li, and Sharad Mehrotra. 2002. Executing SQL over encrypted data in database service provider model. In *ACM SIGMOD*. Wisconsin, 216-227.

Hakan Hacigümüs, Balakrishna R. Iyer, and Sharad Mehrotra. 2004. Efficient execution of aggregation queries over encrypted relational databases. In *DASFAA*. Korea, 125-136.

Bijit Hore, Sharad Mehrotra, Tsudik, G.: A Privacy-Preserving Index for Range Queries. VLDB, pp. 223-235, (2004)

Bijit Hore, Sharad Mehrotra, Mustafa Canim, and Murat Kantarcioglu. 2012. Secure multidimensional range queries over outsourced data. *VLDB Journal*. 21, 3, 333-358.

Jonathan Katz and Yehuda Lindell. 2007. Introduction to Modern Cryptography: Principles and Protocols. Chapman and Hall/CRC

Jonathan Katz and J. S. Shin. 2005. Modeling Insider Attacks on Group Key-Exchange Protocols. In Proceedings of the 12th ACM Conference on Computer and Communications Security (*CCS*), pages 180–189. ACM Press, 2005.

Lea Kissner and Dawn Song. 2005. Privacy-Preserving Set Operations. In *CRYPTO*. 241–257.

H.Y. Lam, G.S.K. Fung, and W.K. Lee. 2007. A Novel Method to Construct Taxonomy Electrical Appliances Based on Load Signatures. *IEEE Transactions on Consumer Electronics*. 53, 2, 653-660.

Lee, P.P.C.; Lui, J.C.S.; Yau, D.K.Y.: Distributed collaborative key agreement and authentication protocols for dynamic peer Groups. *ACM Transactions on Networking*, vol.14, no.2, pp.263-276, 2006.

Hongbo Liu and Hui Wang and Yingying Chen. 2010. Ensuring Data Storage Security against Frequency-based Attacks in Wireless Networks. In *DCOSS*. California, 201-215.

Thomas Locher. 2009. Foundations of Aggregation and Synchronization in Distributed Systems. ETH Zurich, isbn 978-3-86628-254-4.

Yves-Alexandre de Montjoye, Samuel S Wang, Alex Pentland, Dinh Tien Tuan Anh, Anwitaman Datta. 2012. On the Trusted Use of Large-Scale Personal Data. *IEEE Data Eng. Bull*. 35, 4, 5-8.

Einar Mykletun, and Gene Tsudik. 2006. Aggregation queries in the database-as-a-service model. In *DBSec*. France, 89-103.

Pascal Paillier. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*. 223-238.

Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. 2011. CryptDB: protecting confidentiality with encrypted query processing. In *ACM SOSP*. New York, 85-100.

Sandro Rafaeli and David Hutchison. 2003. A Survey of Key Management for Secure Group Communication. *ACM Computing Surveys*. 35, 3, 309-329.

StreamSQL. 2015. Available at : http://www.streambase.com/developers/docs/latest/streamsql/

Yufei Tao, Jimeng Sun, and Dimitris Papadias. 2003. Analysis of predictive spatiotemporal queries. *ACM Transactions on Database Systems (TODS)*. 28, 4, 295–336.

Quoc-Cuong To, Benjamin Nguyen, and Philippe Pucheral. 2013. Secure Global Protocol in Personal Data Server. SMIS Technical report. INRIA, France. http://www.cse.hcmut.edu.vn/~qcuong/INRIA/TechReport.pdf

Quoc-Cuong To, Benjamin Nguyen, and Philippe Pucheral. 2014a. Privacy-Preserving Query Execution using a Decentralized Architecture and Tamper Resistant Hardware. In *EDBT*. Athens, 487-498.

Quoc-Cuong To, Benjamin Nguyen, and Philippe Pucheral. 2014b. SQL/AA : Executing SQL on an Asymmetric Architecture. *PVLDB*. 7, 13, 1625-1628.

Quoc-Cuong To, Benjamin Nguyen, and Philippe Pucheral. 2015a. TrustedMR: A Trusted MapReduce System based on Tamper Resistance Hardware. In *CoopIS*. Rhodes.

Quoc-Cuong To, Benjamin Nguyen, and Philippe Pucheral. 2015b. Key Exchange Protocol in the Trusted Data Servers Context. SMIS Technical report. INRIA, France. http://www.benjamin-nguyen.fr/papers/tr-gke2015.pdf

Stephen Tu, M. Frans Kaashoek, Samuel Madden, and Nickolai Zeldovich. 2013. Processing analytical queries over encrypted data. *PVLDB*. 6, 5, 289–300.

Bing Wu, Jie Wu, and Mihaela Cardei. 2008. A Survey of Key Management in Mobile Ad Hoc Networks. Handbook of Research on Wireless Security. 479-499.

Tsu-Yang Wu, Yuh-Min Tseng, and Ching-Wen Yu. 2011. Two-round contributory group key exchange protocol for wireless network environments. *EURASIP Journal on Wireless Communications and Networking*. 1, 1-8.

INRIA, LIRIS, UVSQ, GEMALTO, CryptoExperts, CG78. 2012. Use cases and functional architecture specification, KISS deliverable ANR-11-INSE-0005-D1, 21/12/2012.

APPENDIX A: SAFETY PROPERTIES AGAINST MALICIOUS ATTACKER

The protocols presented in this paper protect against confidentiality attacks conducted by honest-but-curious adversary. Additional counter-measures must be integrated in the protocols to defeat malicious attackers. Indeed, a malicious SSI could try to tamper with the intermediate results produced during the query execution, either to mislead the querier or to gain some benefit (e.g., save computing and storage resources) from skipping part of the computation. Hence, the integrity of the final result must be made controllable by the Querier. Checking the integrity of an outsourced query result sums up to checking data authenticity, data freshness and query completeness.

Authenticity means that the query result is generated from original database tuples. Authenticity is generally achieved by attaching a signature to a tuple or part of a tuple. As the signature is unforgeable by anyone who does not own the key, any TDS can check signatures produced by any other TDS.

Freshness means that the query result is computed over the latest version of the tuples. Since the query is executed directly in the client's side (TDS), the tuples returned in the collection phase are guaranteed to be up-to-date. However, SSI must be prevented from replacing fresh encrypted tuples sent by TDSs by old tuples resulted from previous queries by attaching the session id to the tuples.

Completeness means that all tuples participating to the collection phase are reflected once, and only once (hence completeness encompass accuracy in our definition), in the result. To ensure completeness, we must prevent duplicate and delete actions from malicious attackers. These actions are hard to detect since this requires having a global view of the dataset.

In order for the TDS and the querier to check authenticity, freshness and completeness, extra information, called herafter security information, has to be produced by the TDSs and bind both to the tuples produced during the collection phase and to the partitions produced during the aggregation phase. We introduce below the set of properties required to enforce authenticity, freshness and completeness and detail the associated security information and security test performed on it.

Let us call $\Gamma^e \leftarrow \{tup^e\}$ the set of encrypted tuples collected in the collection phase. Each tuple $tup^e \in \Gamma^e$ follows the format:

$tup^e \leftarrow (c, id)$ with $tup^e.c \leftarrow nE_k(tup)$, and $tup^e.id \leftarrow nE_k(identifier)$ with tup is the plaintext tuple and identifier is the identity of tup. We assume that identifiers are unique in the whole system, e.g., by concatenated the local tuple identifier with the TDS identifier.

Each encrypted partition/aggregation Ω^e_i contains a set of tuples' identifiers it is supposed to contain, called Ω-set, denoted as $\Omega^e_i.ID$, and the total number of tuples contributed to that aggregation, denoted as $\Omega^e_i.\psi$.

Definition 1 (Origin safety property). The Origin safety property guarantees that a tuple originates from a real TDS as target of a given query itself identified by Qid. Origin is considered as a proof of authenticity and freshness. To enforce this property, each tuple embeds a signature, denoted σ, which is the result of signing the encrypted tuple concatenated to its encrypted identifier and query identifier: $tup.\sigma \leftarrow sign(tup^e.c \,||\, tup^e.id \,||\, Qid)$.

As previously said, completeness is significantly harder to achieve since it requires building a global view of the collected dataset and be able to detect any delete and copy action on it from an adversary.

Delete actions means removing encrypted tuples from partitions or even removing complete partitions during the execution, making the final result incomplete. These actions reduce the size of the encrypted dataset to be processed. We use the Quantity Preservation safety property to prevent this kind of attack. Basically, this safety property preserves the number of tuples to be processed from the beginning of the protocol until the final step.

Definition 2 (Quantity Preservation safety property). Aggregations or partitions respect Quantity Preservation if $\sum_{i=1}^{p} |\Omega_i^e.\psi| = S$ with p being the total number of aggregations/partitions, and S the value in SIZE clause of the query.

Thank to this property, if SSI drops any tuple or partition during the execution, the total number of tuples for this query will be less than the required number, making this action detectable. Similarly, this property also prevents duplicate actions augmenting the number of tuples. However, it doesn't prevent a malicious SSI to delete arbitrary d encrypted tuples from a partition, and then copy another d existing encrypted tuples from this (or another) partition to replace them. This action satisfies both the Origin safety property (SSI does not forge any new tuples) and Quantity Preservation safety property (since the total number of tuples remains unchanged). However, it still makes the final result incorrect due to the difference between deleted tuples and duplicated ones. Replace actions (i.e., replacing a deleted tuple by a duplicated one) can be either *intra-partition* (tuples are replaced into their own partition) or *inter-partition* (the destination partition is different from the source partition). Intra-partition replace actions can be easily detected by checking the unicity of tuple identifiers within each partition. The Identifier Unicity safety property (Definition 3) serves this purpose.

Definition 3 (Identifier Unicity safety property) Let $tup^e \in \Omega^e_i$ be a tuple in the partition/aggregation Ω^e_i. Partition/Aggregation Ω^e_i respects the Identifier Unicity safety property if for every pair of tuples $tup^e_j, tup^e_k \in \Omega^e_i$, $tup^e_j.id = tup^e_k.id \Rightarrow j = k$.

Detecting inter-partition replace actions is more difficult and requires organizing the set of tuples such that each identifier is authorized to be part of a single partition (partitions intersections are empty). To this end, we define for each partition the set of identifiers it is supposed to contain, called Ω-set, denoted by Ω^e_i.ID. The Mutual Exclusion safety property (Definition 4) ensures that no Ω^e_i.ID overlaps, and the Membership safety property (Definition 5) ensures that each identifier must appear in the partition to which it is supposed to belong. As a result, Mutual Exclusion and Membership together guarantee that each identifier actually appears within a single partition (as stated in the Lemma).

Definition 4 (Mutual Exclusion safety property) Partitions respect Mutual Exclusion if for every pair of partitions Ω^e_i, Ω^e_j, $i \neq j \Rightarrow \Omega^e_i.\text{ID} \cap \Omega^e_j.\text{ID} = \emptyset$.

Definition 5 (Membership safety property) A partition respects Membership if for every tuple $tup^e_j \in \Omega^e_i$, then $tup^e_j.id \in \Omega^e_i.\text{ID}$.

Lemma. Enforcing together the Identifier Unicity, Mutual Exclusion, and Membership safety properties is necessary and sufficient to guarantee the absence of any (intra/inter-partition) replace action.

Proof We start by showing the sufficiency of these properties. First, Identifier Unicity is sufficient to preclude by itself intra-partition replace actions (recall that the authenticity of a tuple and its identifier is guaranteed by the Origin safety

property). Second, assume that a given tuple tup^e has been copied into two distinct partitions. Only the Ω^e-set of one of them contains tup^e identifier because otherwise Mutual Exclusion would be contradicted. Consequently there must be one partition's Ω^e-set that does not contain tup^e identifier. But this clearly contradicts the Membership safety property. As a result, Membership and Mutual Exclusion are together sufficient to preclude inter-partition replace actions.

We now show the necessity of these properties. First, since a distinct identifier is assigned to each tuple, the absence of intra-partition replace results immediately in the satisfaction of the Identifier Unicity property. Second, the absence of inter-partition replace implies that the partitioning is correct so that: (1) the Mutual Exclusion property is satisfied in that Ω^e-sets do not overlap (recall that a distinct identifier is assigned to each tuple) and (2) the Membership property too in that each tuple appears in the partition which Ω^e-set contains its identifier. □

Implementation Sketches.

The implementations of the Origin and Identifier Unicity safety properties are straightforward: when receiving a partition to compute, the given TDS simply checks the signatures of tuples and the absence of duplicate identifier[19].

The other properties are harder to check because they concern the complete dataset. We thus add an header in each partition to contain the summary information of that partition.

First, the header contains $\Omega^e_i.\psi$, the total number of tuples contributing to the aggregation Ω^e_i. Every times a TDS receives partitions from SSI and computes the new aggregation, it cumulates all $\Omega^e_i.\psi$ belonging to these partitions and stores the result in the header of the new aggregation. Note that only the TDS that combines the last partitions into the final result can check that $\sum_{i=1}^{p} |\Omega_i^e.\psi| = S$. But since a TDS does not know if he is calculating the final or intermediate result, he cannot check the Quantity Preservation safety property. Only when the final result is delivered to Querier, who knows for sure that he is obtaining the final result, this property ($\Omega^e_{final}.\psi = S$) can be checked.

Second, the header also contains Ω^e_i.ID, the set of identifiers of all tuples in the partition Ω^e_i. In every step of the protocol, when the partitions are accumulated, TDS also makes a union between these lists. When making this union, TDS can detect only the *partial* inter-partition replace actions in which these actions happen among the partitions that TDS is handling. Similar to $\Omega^e_i.\psi$, it is impossible for a TDS to detect the *full* inter-partition replace actions unless that TDS is handling the last partitions to create the Ω^e_{final}. Therefore, it is Querier, who obtains the Ω^e_{final}, which can check the *full* inter-partition replace actions.

While the extra-cost of these verifications has not been evaluated yet, it relies on rather simple tests compared to the usual complexity of checking the integrity of outsourced query results.

[19] In general, the identifier can be implemented simply by letting secure devices generate a random number. It has to be big enough with respect to the number of tuples to collect in order to make collisions improbable so that in the rare collision cases the recipient simply keeps one of the colliding tuples. For example, around 5 billion numbers have to be generated to reach 50% chance collision with a 64-bits random number.