SQL/AA: Executing SQL on an Asymmetric Architecture

Quoc-Cuong To
INRIA, France
PRISM, UVSQ, France

Benjamin Nguyen
INRIA, France
PRISM, UVSQ, France

Philippe Pucheral
INRIA, France
PRISM, UVSQ, France

<Fname.Lname>@inria.fr, <Fname.Lname>@prism.uvsq.fr

1. INTRODUCTION

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. This information represents an unprecedented potential for user customized applications and business (e.g., car insurance billing, carbon tax, traffic decongestion, resource optimization in smart grids, healthcare surveillance, participatory sensing). However, the PRISM affair has shown that public opinion is starting to wonder whether these new services are not bringing us closer to science fiction dystopias. It has become clear that centralizing and processing all one's data on a single server is a major problem with regards to privacy concerns. Conversely, decentralized architectures, devised to help individuals keep full control of their data, complexify global treatments and queries, often impeding the development of innovative services and applications.

In [3], we proposed a novel architectural approach to this problem called *Trusted Cells*. This approach capitalizes on hardware advances representing a sea change in the acquisition and protection of personal data. Trusted Cells push the security to the edges of the network, through personal data servers [2] running on secure smart phones, set-top boxes, plug computers or secure portable tokens[1] forming a global secure decentralized data platform. In [10], we proposed a privacy-preserving querying protocol on this *asymmetric architecture* (see Section 2) aiming to reconcile individual's privacy on one side and global benefits for the community and business perspectives on the other.

This querying protocol makes as few restrictions on the computation model as possible. We model the information system as a horizontally partitioned global database formed by the union of a multitude of distributed local data stores. We consider regular SQL queries and any traditional access control model. Hence the context is different and more general than, (1) querying encrypted outsourced data where restrictions are put on the predicates which can be evaluated [1, 9], (2) performing privacy-preserving queries usually restricted to statistical queries matching differential privacy constraints [4] and (3) performing Secure-Multi-Party (SMC) query computations which cannot meet both query generality and scalability objectives [6].

In a nutshell, the contributions presented in [10] are: (1) to propose different secure query execution techniques to evaluate regular SQL queries (including `GROUP BY` clause) over a set of distributed trusted personal data stores and (2) to study the range of applicability of these techniques. The demonstration proposed here illustrates that such querying techniques can be put in practice, that is to say on real hardware platforms, at low cost and with a performance level ensuring a nation-wide scalability (millions of participants i.e. personal data stores).

2. ARCHITECTURE AND SCENARIOS

Asymmetric Architecture and threat model. As pictured in Figure 1, the architecture we consider is formed by a large set of low power personal Trusted Data Servers (TDSs) embedded in secure devices (e.g., GPS tracking units in cars, smart meters installed at home, PCEHR - Personally Controlled Electronic Health Records - embedded in portable smart tokens). Whatever its form factor, a secure device can be abstracted by (1) a Trusted Execution Environment hosting and running the code, and (2) a - potentially untrusted but cryptographically protected - mass storage area (see Fig. 1). TDSs cannot be tampered, even by the TDS holder herself, and are thus considered *honest*.

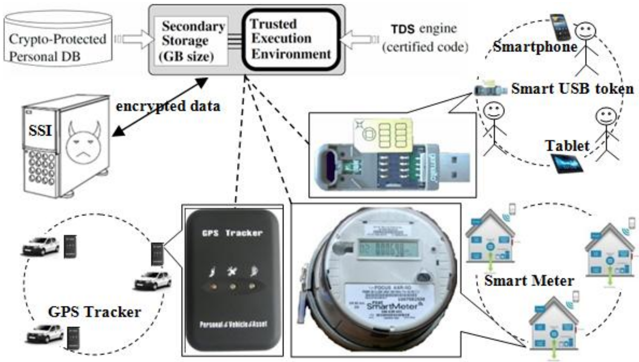

Crypto-Protected Personal DB
Secondary Storage (GB size)
Trusted Execution Environment
TDS engine (certified code)
Smartphone
Smart USB token
Tablet
SSI
encrypted data
GPS Tracker
Smart Meter

Figure 1. Trusted Data Servers

Since TDSs have limited resources and are not necessarily always connected, an external infrastructure, called *Supporting Server Infrastructure* (SSI), is required in the architecture to manage the communications between TDSs, run the distributed query protocol and store the intermediate results produced by this protocol, hence the name *asymmetric architecture*. Because the SSI is implemented on regular servers, e.g., the Cloud, it is considered as *honest-but-curious* (i.e., it may try to infer any information it can but strictly follows the protocol). Considering *malicious* SSI (i.e., which may tamper the protocol with no limit, including denial-of-service) is out the scope of this study.

Queries of interest. We consider that *local databases* hosted by TDSs conform to a common schema which can be queried in SQL. For example, power meter data (resp., GPS traces,

Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08

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

healthcare records, etc) can be stored in one or several table(s) whose schema is defined by the national distribution company (resp., an insurance company consortium, the Ministry of Health, etc) horizontally partitioned on the local stores. Queries are regular SQL queries[2], borrowing the SIZE clause from the definition of windows in the StreamSQL standard[3]. For example, an energy distribution company could issue the following query on customers' smart meters:

```
SELECT C.district, 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 current_date() >= 2014-04-1
```

This query computes the mean energy consumption of people living in a detached house, grouped by district, for districts where over 100 consumers answered the poll. The poll is open until the 1st of April 2014.

Global objective. In the example presented above, 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 from seeing the raw data of its customers for privacy concerns[4]. In terms of **privacy protection**, the querying protocol must guarantee that (1) the querier gains access only to the final result of authorized queries, as in a traditional database systems and (2) intermediate results stored in SSI are fully obfuscated. Preventing inferential attacks by combining the result of a sequence of authorized queries as in statistical databases and privacy-preserving data publishing (PPDP) work is orthogonal to this study. In terms of **performance**, the querying protocol must guarantee nation-wide scalability. The time to collect data is highly scenario dependent, i.e., very fast if TDSs are always connected (e.g., smart metering context), probably quite long otherwise (e.g., querying mobile PCEHR embedded in TDSs). Therefore the performance challenge is not on the overall response time, but rather query computation time on data collected, given local TDSs resources.

3. QUERYING PROTOCOLS

In [10], three querying protocols have been proposed and compared in terms of privacy protection and performance. Our querying protocols share common basic mechanisms and mainly differ in the way data is obfuscated from the SSI during the computation. The computation is split in three phases summarized in Figure 2. We first present these three phases and then discuss how they are instantiated by each protocol.

Basic mechanisms. Queries are executed in pull mode. A querier posts its query to SSI and TDSs download it at connection time. Result tuples are gathered by SSI until the SIZE clause is evaluated to *true*. 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. Moreover, an *honest-but-curious* SSI can try to conduct *frequency-based attacks* [8], i.e. exploiting prior knowledge about the data distribution to infer the plaintext values of ciphertexts. Depending on the protocols (see later), various encryption schemes are used to prevent these attacks. While `SELECT-FROM-WHERE` queries remain easy to compute in this context, `GROUP BY` clauses require performing set-oriented computations over intermediate results sent by TDSs to the 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, the SSI cannot help much in the processing since (1) it cannot 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. To solve this problem, we suggest the following generic protocol.

Collection phase: (step 1) the querier posts on the SSI a query Q encrypted with a key shared between the querier and the TDSs[5], its credential C signed by an authority and S the SIZE clause of the query in cleartext so that the SSI can evaluate it; (step 2) targeted TDSs download Q when they connect; (step 3) each of these TDSs decrypts Q, checks C, evaluates the access control policy AC 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 dummy tuple if the result is empty or TDS's owner refuses to send his data to answer the query (step 4'). The result tuples are encrypted with a key shared by the TDSs only. The collection phase stops when the `SIZE` condition has been reached. The result of collection phase (Covering Result) is actually the result of the query complemented with dummy tuples.

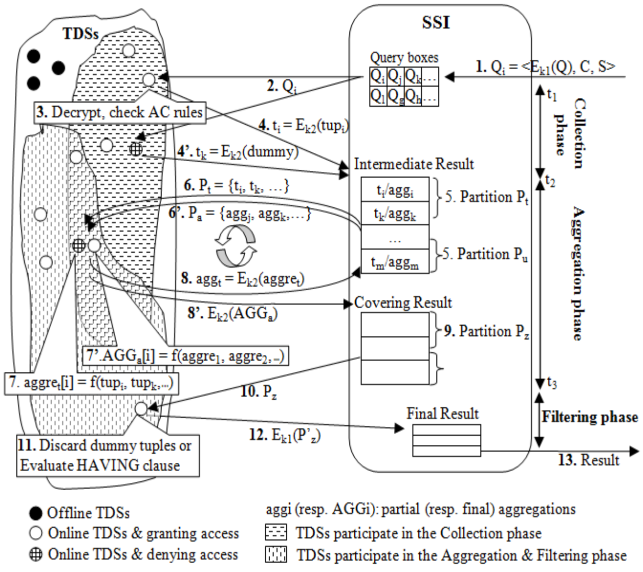

TDSs
SSI
Query boxes
Q_i Q_j Q_k …
Q_l Q_g Q_h …
1. Q_i = <E_{k1}(Q), C, S>
2. Q_i
3. Decrypt, check AC rules
4. $t_i = E_{k2}(tup_i)$
4'. $t_k = E_{k2}$(dummy)
Intermediate Result
t_i/agg_i
t_k/agg_k
…
t_m/agg_m
5. Partition P_t
5. Partition P_u
6. P_t = {t_i, t_k, …}
6'. P_a = {agg_j, agg_k,…}
7. $aggre_t[i] = f(tup_i, tup_k,…)$
7'.$AGG_a[i] = f(aggre_1, aggre_2,…)$
8. $agg_t = E_{k2}(aggre_t)$
8'. $E_{k2}(AGG_a)$
Covering Result
9. Partition P_z
10. P_z
11. Discard dummy tuples or Evaluate HAVING clause
12. $E_{k1}(P'_z)$
Final Result
13. Result
t_1
t_2
t_3
Collection phase
Aggregation phase
Filtering phase
● Offline TDSs
○ Online TDSs & granting access
⊕ Online TDSs & denying access
aggi (resp. AGGi): partial (resp. final) aggregations
▤ TDSs participate in the Collection phase
▥ TDSs participate in the Aggregation & Filtering phase

Figure 2. Querying protocols

[2] We simply do not consider *joins* between data stored in different TDSs in this study. However, internal joins which can be executed locally by each TDS are supported.

[3] http://www.streambase.com/developers/docs/latest/streamsql/

[4] 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 the inhabitants' activities using power meter data [7].

[5] key management is more deeply explained in [10].

Aggregation phase *(this phase is empty for simple `SELECT-FROM-WHERE` queries)*: (step 5) SSI partitions the Covering Result with the objective to let several TDSs manage these partitions in parallel. The Covering Result being fully encrypted, the SSI sees partitions as uninterpreted chunks of bytes; (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 back to the SSI; depending on the protocol (see later) the aggregation phase can be 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 per group.

Filtering phase: (step 9) the SSI partitions the Covering Result with the objective to let TDSs either eliminate dummy tuples (*`SELECT-FROM-WHERE`* queries) or manage the *`HAVING`* clause (queries with group by and aggregations); (step 10) connected TDSs download these partitions; (step 11) depending on the query, each of these TDS decrypts the partition and filters out dummy tuples or partitions which do not satisfy the *`HAVING`* clause; (step 12) each TDS returns the final tuples to the SSI, encrypted with a key known by the querier; the SSI finally concatenates all results and informs the querier that she can download the result (step 13).

This generic protocol is instantiated differently depending on which encryption scheme is used in the collection and aggregation phases, how the SSI constructs the partitions, and what information is revealed to the SSI. Each solution has its own strengths and weaknesses and therefore is suitable for a specific situation. We have implemented three solutions described below. The idea underlying each of this solution is the following.

Secure aggregation (S_Agg): During the collection phase, result tuples are *non-deterministically* encrypted (i.e., equality is not preserved) to prevent any frequency-based attack by SSI. The consequence is that SSI cannot get any knowledge about the group each tuple belongs to, forcing the aggregation phase to be iterative. At each iteration, TDSs download partitions containing a sequence of (A_G, Aggregate value) pairs and returns to the SSI a smaller sequence of (A_G, Aggregate value) pairs where values of the same group have been aggregated. The SSI gathers these partial aggregations to form new partitions, and so on and so forth until a single partition is produced. The security of this protocol is maximum at the price of an iterative aggregation phase, the parallelism of which decreases at each iteration. Termination is discussed in [10].

Noise-based (Noise): *Deterministic* encryption is used on the grouping attributes A_G during the collection phase, so that the SSI can assemble tuples belonging to the same groups in the same partitions. To prevent frequency-based attacks from the SSI, TDSs add some noise (i.e., fake tuples) to the data in order to hide the real distribution. Fake tuples are filtered out in a later step of the protocol. The performance benefit comes from the fact that the contents of partitions are no longer random, thereby accelerating convergence and allowing parallelism up to the final iteration. However, the information revealed to the SSI is directly linked to the amount of noise added in the intermediate results.

Histogram-based (EDHist): EDHist exploits a prior knowledge of the real distribution of A_G attributes. The idea is to produce a uniform distribution of true data sent to the SSI by grouping them into *equi-depth histograms*, in a way similar to [5]. Hence, the SSI can group data in partitions based on histogram identifiers; each tuple is marked with the identifier of the histogram its grouping attributes A_G belongs to; each histogram represents a small set of A_G domain values and histograms are formed in such a way that they all have nearly the same cardinality. The SSI only sees a nearly uniform distribution of the tuples and the performance penalty incurred either by an iterative aggregation phase or by managing fake tuples is avoided.

We refer the reader interested by a deeper presentation of the generic querying protocol and by a complete discussion about the performance and information leakage linked to each of its instantiation to [10].

4. DEMONSTRATION

In this section, we present our prototype platform and describe how we will demonstrate the proposed protocols and their scalability and parallelism, through a scenario illustrating a distributed architecture where a SSI connects to various TDSs. To make the demonstration user-friendly and easy to follow, we use a graphical interface (Figure 4) that helps understand the overview of the system and how data flows through the system.

4.1 Demonstration platform

The Hardware Platform. The demonstration platform is an instance of the architecture presented in Figure 1. A PC plays the role of the SSI, listens to connections from TDSs, manages the communication between TDSs, runs the distributed protocols, stores intermediate results, and shows encrypted data and results it receives from TDSs. A number of development boards (Figure 3) represent the TDSs and host the client application. This application can open a connection to the SSI using an Ethernet connection via a switch. These boards exhibit hardware characteristics representative of secure tokens-like TDSs, including those provided by Gemalto (the smartcard world leader), one of our industrial partners. 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. Other devices used to represent the TDSs are tokens built by the ZED company (Figure 3) that can connect to a host (e.g. a laptop connected to SSI via ethernet) by USB port. The ZED tokens have the same characteristics as the boards: they are equipped with a crypto coprocessor, run the same client application to receive encrypted data from the SSI, decrypt data, compute the aggregation, encrypt the result, and return the result to the SSI. Because both boards and ZED token are by design unobservable, they are connected to the PC through a COM port used by our demonstration to trace their behavior.

The Graphical User Interface (GUI). A GUI is used to control the system and show what information each actor can see in our system. The GUI is divided into three parts: Set of TDSs, SSI, and Querier. The first part shows the (fictional) geographic location of the TDSs. The original cleartext distribution is displayed next to it. The real distribution will be compared with the distribution of the ciphered data seen by the SSI during each protocol. The second part displays the encrypted query that the SSI receives from Querier, the encrypted data from the collection phase of each protocol and its visualization to compare the difference between protocols. The final part consists of a textbox that allows users to input any SQL query and a table to display the final cleartext result of the query.

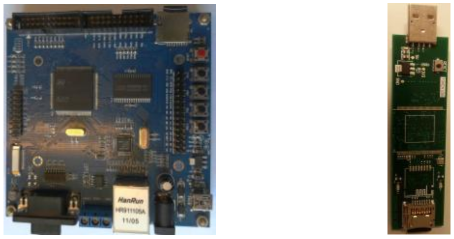
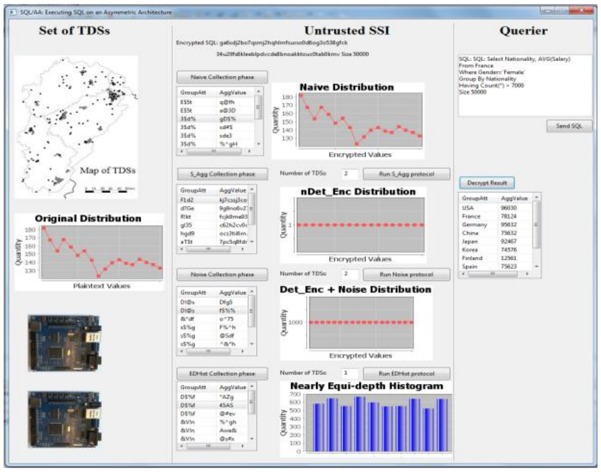


Figure 3. TDS development board & ZED token



Encrypted SQL: ga6sdj2bo7qsmj2hqhlmfsurso0d6og3o538gfck
34u28fh8kleeblpdvcde8bnoakktoucOtab0kmv Size 50000

SQL: SQL: Select Nationality, AVG(Salary)
From France
Where Gender='Female'
Group By Nationality
Having Count(*) > 7000
Size 50000

GroupAtt	AggValue
E$5t	q@fh
E$5t	a@3D
3$d%	gD$%
3$d%	sd#$
3$d%	sde3
3$d%	%^gH

GroupAtt	AggValue
F1d2	kj7cssj3co
dTGe	9g9no6v2
R!kt	fcjk9me93
gl35	c62h2cv0
hgd9	ocs3ti8m.
aT3t	7pc5q8fdr

GroupAtt	AggValue
D1@s	D#g5
D1@s	f$%%
&*df	o^T5
s$%g	F%^h
s$%g	@Sdf
s$%g	^&*h

GroupAtt	AggValue
D$%f	*AZg
D$%f	45AS
D$%f	@#ev
&V!n	%^gh
&V!n	Awe&
&V!n	@s#x

GroupAtt	AggValue
USA	96030
France	78124
Germany	95632
China	75632
Japan	92467
Korea	74576
Finland	12561
Spain	75623

Figure 4. Demonstration graphical interface

Dataset. We use a randomly-generated dataset for the demo with the objective to change the dataset distribution and analyze the impact on each querying protocol. We assume that the result of the collection phase is stored in an encrypted table and all boards and ZED tokens share the same key to encrypt/decrypt data. The cardinality of the encrypted table is one million tuples.

Algorithms. Our demonstration consists of three proposed protocols (i.e., S_Agg, Noise_based, EDhist) presented in Section 3, plus a Naïve protocol which simply uses deterministic encryption without any distribution obfuscation.

4.2 Demonstration results

Security. After explaining how the data is collected and stored in the encrypted table, we will demonstrate SSI frequency based inference attacks on the Naïve protocol. Then we run the three proposed protocols, show the difference between their distributions, and demonstrate how they prevent frequency-based attacks. During the execution of the protocols, we show what information (i.e., encrypted data) the SSI can see.

Performance. We also compare the execution times of these protocols to demonstrate their performances and show their feasibility (the protocols with a small number of TDSs participating in the computation can be executed in few seconds for a dataset of one million tuples). At the end of the execution, the plaintext result is printed on the TDSs' side so that audience can compare with the SQL result executed on the plaintext table. The audience will also be invited to propose aggregate SQL queries to be tested.

Scalability. To show the scalability and parallelism of our system, we vary the dataset's size and the number of TDSs used in our protocols. First, we run our protocol with one TDS, then we increase the number of TDSs to show that the execution time experimentally decreases by approximately the same value, demonstrating the scalability of the system.

Hence, this demonstration will show that decentralized architectures can be devised to help individuals better protect their privacy without impeding the development of global services of great interest. 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).

5. ACKNOWLEDGMENTS

This work was funded by project ANR-11-INSE-0005 "Keeping your Information Safe and Secure".

6. REFERENCES

[1] Agrawal, R., Kiernan, J., Stikant, R., Xu, Y. Order-preserving encryption for numeric data. *ACM SIGMOD*. Paris, 2004, 563-574.

[2] Allard, T., Anciaux, N., Bouganim, L., Guo, Y., Le Folgoc, L., Nguyen, B., Pucheral, P., Ray, Ij., Ray, Ik., Yin, S. Secure Personal Data Servers: a Vision Paper. *VLDB*. Singapore, 2010, 25-35.

[3] Anciaux, N., Bonnet, P., Bouganim, L., Nguyen, B., Sandu Popa, I., Pucheral, P. Trusted Cells: A Sea Change for Personal Data Services. *CIDR*. California, 2013.

[4] Fung, B. C. M., Wang, K., Chen, R., Yu, P. S. Privacy-Preserving Data Publishing: A survey of Recent Developments. *ACM Computing Surveys*, 42, 4 (2010).

[5] Hacigumus, H., Iyer, B., Li, C., Mehrotra, S. Executing SQL over encrypted data in database service provider model. *ACM SIGMOD*. Wisconsin, 2002, 216-227.

[6] Kissner, L., Song, D. X. Privacy-Preserving Set Operations. *CRYPTO*. California, 2005, 241–257.

[7] Lam, H. A Novel Method to Construct Taxonomy Electrical Appliances Based on Load Signatures. *TCE*, 2007.

[8] Liu, H., Wang, H., Chen, Y. Ensuring Data Storage Security against Frequency-based Attacks in Wireless Networks. *DCOSS*. California, 2010, 201-215.

[9] Popa, R. A., Redfield, C. M. S., Zeldovich, N., Balakrishnan, H. CryptDB: protecting confidentiality with encrypted query processing. *ACM SOSP*. Cascais, 2011, 85-100.

[10] To, Q.C., Nguyen, B., Pucheral, P. Privacy-Preserving Query Execution using a Decentralized Architecture and Tamper Resistant Hardware. *EDBT*. Athens, 2014, 487-498.