Experiences and observations on the industrial implementation of a system to search over outsourced encrypted data.

Patrick Grofig, Martin Haerterich, Isabelle Hang, Florian Kerschbaum, Mathias Kohler, Andreas Schaad, Axel Schroepfer, Walter Tighzert

SAP AG

andreas.schaad@sap.com

Abstract: This paper presents an industrial report on the implementation of a system that supports execution of queries over encrypted data. While this idea is not new, e.g. [HILM02, AKRX04, PRZB11], the implementation in a real world large scale in-memory database is still challenging.

We will provide an overview of our architecture and detail two use cases to give the reader an insight into how we technically realized the implementation. We then provide three main contributions, reporting that:

a) We significantly improve functionality by intelligently splitting query execution, i.e. which parts of a query can be performed in the cloud and which on the client.

b) We share some initial performance measurements with the community on basis of the TPCH benchmark.

c) We present a domain-specific analysis of three data sets that shows the effects of executing queries over encrypted data and what adjustments are required with respect to the encryption of individual columns.

The three made observations on query execution, execution time measurements and domain-specific query analysis will lead us to the conclusion that although searching over outsourced encrypted data is always a trade off between functionality, performance and security, it is realistic to assume that working solutions can be provided in the not too distant future to the market.

1 Introduction

1.1 The Problem

On-demand databases outsourced in the "Cloud" are vulnerable to additional attacks compared to on-premise databases. While the cloud provider organization is usually trusted, its employees like database operators may misuse their elevated privileges to access cloud data. One recent prominent example is that where a database administrator of a Swiss Bank sold the records of clients to German and US Tax authorities. Clouds also represent a valuable target for attackers and a single vulnerability in the provider's system landscape can put all customers at risk.

1.2 The naive approach

The obvious approach could be to encrypt all data with a secure encryption algorithm such as AES and store it in the cloud. However, while secure, all data can no longer be processed in the cloud but has to be downloaded and decrypted on the client to execute any query on it. This makes any serious Database as a Service offering questionable and is the way many traditional DBMS like Sybase, Oracle, DB2 or solutions like Dropbox appear to work when they claim to encrypt data and provide cloud storage.

1.3 The Solution: Searching over Encrypted Data

However, we can provide proof that it is possible to execute arbitrary SQL queries on encrypted data without any decryption on the server-side by following three key ideas [PRZB11]:

- first, a SQL-aware encryption strategy that maps specified SQL operations to "fitting" encryption schemes (e.g. deterministic, order-preserving, homomorphic or searchable) supporting the requested functionality;
- second, an "adjustable, layered, query-based encryption" (onion encryption) that can adjust the encryption level of each item to the required level for the desired functionality;
- third, a query optimization algorithm that can improve the trade-off between security and performance as well as a set of algorithms improving columnar re-encryption required for join operations.

We developed a framework (called “SEEED” – Search over Encrypted Data) that allows to provide scalable and fine-grained encryption at a columnar-level at the same time providing the ability to directly execute queries over encrypted data. One application could be in a JDBC context, another secure JPA persistency. The framework has been prototypically implemented in the SAP HANA database management system [FMLG+12].

1.4 The Delta: True Security for Outsourced Data

Our approach provides a significant delta to the current approach of securing outsourced data. As mentioned before, just encrypting tables is not sufficient as any application would require decryption before processing. If this happens in the cloud, we are vulnerable to any attacks from the cloud provider and if the decryption and query execution happens on the on-premise client any Database as a Service offering is pointless.

More specifically, we do not only provide table encryption but encryption of individual columns paired with the ability to execute SQL directly over the encrypted columns. Our approach scales as it also allows to leave non-critical columns in cleartext without any change to a query required.

Note, that primary key material never leaves the client system. The final result set of any query processed in the cloud is sent in an encrypted fashion back to the client where it is decrypted.

The abstract use case is that a database query (eg. select, insert, update) triggered by an application is intercepted by our SEEED Database driver (eg JDBC). The SEEED driver will encrypt the query elements and send the encrypted query to the database. The encrypted query is executed over the encrypted data and the encrypted result is sent back to the application. Only the application can decrypt the result as primary encryption keys never leave the client.

The implicit trust assumption is that the client / on-premise environment is trusted, the network is untrusted and that the server / cloud environment is honest but curious. Prior to outsourcing, tables and individual columns have been encrypted following our onion approach. Data is always encrypted in a randomized fashion at the outermost encryption layer (for example, AES in CBC mode). This means that when leaving the on-premise environment, the dataset is encrypted following current security best practices.

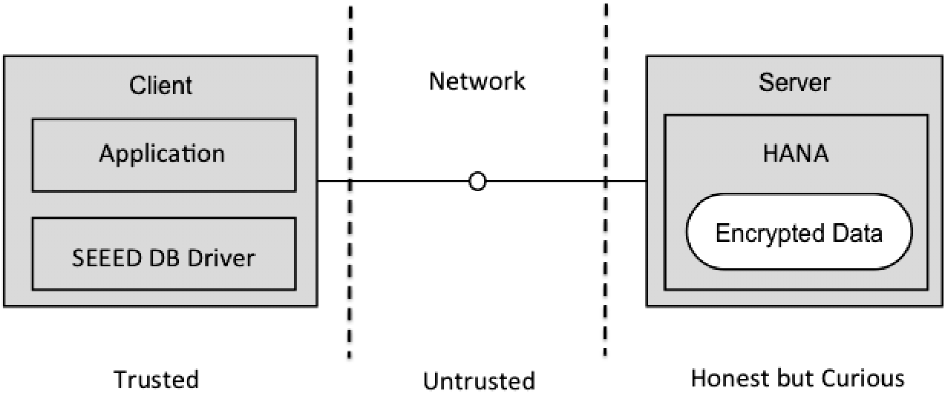


Figure 1: Simplified Data Flow Diagram

2 Architecture

2.1 Main Components

Extending the basic description provided in Section 1.4, Figure 2 now provides a more detailed description of the architectural components and their interaction introduced by SEEED. We again emphasize that the web application server hosting a business application and providing JDBC services is running in a trusted on-premise domain.

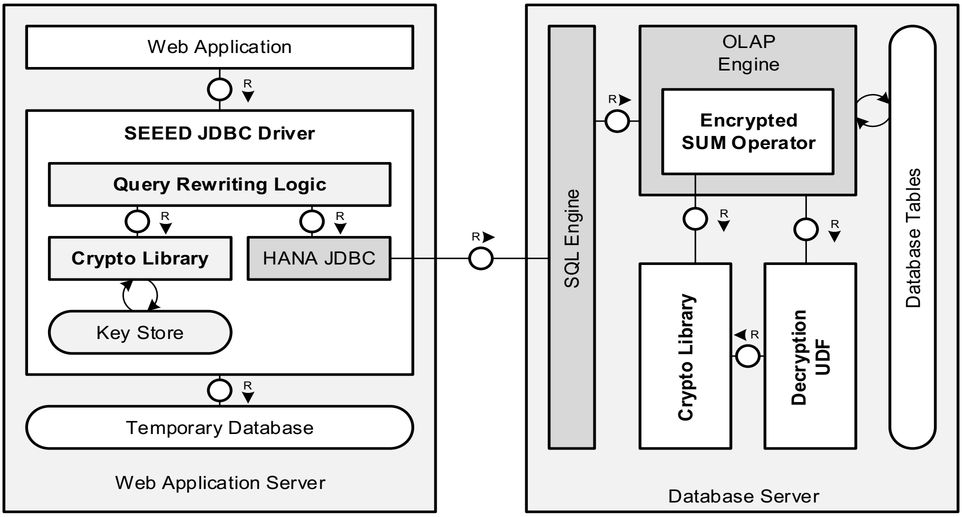


Figure 2: Architecture Overview

On the side of the Web Application Server, the SEEED JDBC Driver is the central component for connecting and querying the encrypted database. It contains a Query Rewriting Logic which transforms a plain SQL query given by the Web Application into an operator tree based on which all further processing is done. This includes encryption of single elements within the tree, selection of specific encryption schemes and their respective onions and layers, optimization of the execution sequence given the several operators in the tree, as well as the decisions which operators are executed on the server and which have to be processed on the client by means of post-processing.

Data encryption is done by a local Crypto Library which supports a range of different crypto schemes (compare section 2.3) each providing its own specific search characteristics. For key management we use a primary key provided by the Web Application based on which the secondary encryption keys for de- and encryption are stored in a local Key Store (realized by using the standard Java KeyStore implementation).

The SEEED JDBC driver uses the original HANA JDBC driver to execute the encrypted queries on HANA. For possible post processing of query results (compare section 3.1), a local Temporary Database is used.

In Figure 2, the Database Server is a SAP HANA system where a new operator for aggregations on encrypted data was implemented with the OLAP Engine. Besides many other query expressions, this extension enables a client to explicitly execute SUM operations directly on HANA. The additional crypto library on the server provides the respective cryptographic algorithms. Moreover, it also supports the Decryption UDFs which are used to update the respective encrypted data if onion layers have to be removed or re-encrypted (compare section 2.2).

2.2 Onions & Re-encryption

The two fundamental concepts of our approach are based on so called onions of encryption, i.e. cryptographic schemes that a) support specific operations over the encrypted data and b) allow transformation from one scheme to another.

One plaintext column is encrypted and stored on the database by multiple onions. Each onion consists of different layers. The center layer is always the plaintext. Each layer then encrypts the result of the previous layer with a specific encryption scheme.

For example, one possible onion may be the following: AES Randomized (AES Deterministic (Order Preserving (Cleartext Column))) as shown in Figure 3. This basically means that the plaintext of a column is first encrypted under an Order Preserving scheme, the result is then encrypted under an AES scheme in deterministic mode and that result using AES in Randomized mode.

Another onion might have a plaintext layer and only one encryption layer featuring homomorphic encryption. A third onion might have a plaintext layer and one encryption layer with searchable (SCR) encryption schema. A last fourth one might only have plaintext and an AES Randomized encryption layer (a so called “retrieval onion” as it is only used for transferring data from the server to the client).

Each layer(!) of an onion is used for a specific purpose of an SQL query. So, for simplicity, let us assume, a column c1 is currently encrypted using AES Deterministic (DET) which supports an SQL operation such as join() or equals(). If another SQL query wants to perform a range query it may be more efficient to re-encrypt c1 into an Order Preserving Scheme (OPE).

This is the core of our approach and requires careful planning about how to initially encrypt data and when to do a re-encryption at runtime.

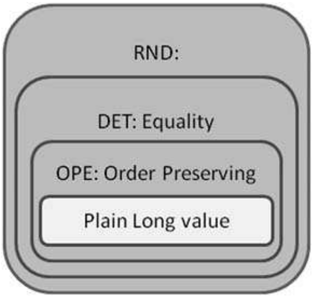


Figure 2: One possible Encryption Onion

2.3 Encryption Schemes

As part of current framework we currently use the following encryption schemes:

- Randomized: Cipher texts belonging to a sequence of plaintexts are indistinguishable from random values. Used if just the data is retrieved from the database and no operation on server side is required.

  Example: AES in CBC mode.

- Deterministic: Cipher texts belonging to same plaintexts are identical. Here security depends on the entropy of the data (for example, if there are only two distinct values, e.g. „female“ and „male“ we may face a frequency attack).

  Example: AES in ECB mode.

- Order-preserving: For ordered domains only (numerical values or lexicographically ordered texts).

  Example: Boldyreva et al. [BCLN09]

- Partially Homomorphic: Enables aggregation functions, in particular SUM()

  Example: Paillier [Pai99]

- Searchable Encryption: Used to perform exact searches without revealing anything else than the result set.

  Example: Song et al. [SWP00]

- Re-Encryption (DET-JOIN): Deterministic scheme where it is possible to change the key (re-encrypt) on without intermediary decryption. For an optimal re-encryption strategy see [KHGK+13].

  Example: Pohlig and Hellmann [PH78]

We must strongly emphasize that it is not valuable to simply „compare“ the encryption schemes in a „A is more secure than B“ fashion. As part of section 3.3 we discuss the security implications of individual schemes in the context of domain-specific data.

3 Accepting the trade-offs

It is an accepted viewpoint that searching over encrypted data is a continuous trade-off between Functionality, Security and Performance. The following sections will now present some results in the context of our implementation efforts that relate to each one of the three variables.

3.1 Functionality

Since most relational operators work without modification on encrypted data, rewriting the SQL Query becomes simple. The entire rewriting process can be performed on the abstract syntax tree. This is efficient, but also limited. Some operators, such as like() or bitwise operators, as well as some sequences of operators, such as sorting or selecting after aggregation, cannot be executed with unmodified operators. As a database developer one can now either reject these queries asking the user to rewrite or – in these rare cases – retrieve the data and execute the query locally. Of course, it is still beneficial to execute as much as possible on the server. Hence, the query needs to be split into a part that can be executed on the server and a part that must be executed at the client. Such a split can no longer be performed on the declarative SQL syntax. Instead we parse the query and construct the query in relational algebra. A tree of relational operators can then be split into a remote part on the server and a local part on the client. The leaves of the tree are database tables (scan operations). These are always executed on the server. The tree above and including the bottom most operator that can no longer be executed on the server is executing at the client. We translate each part into a SQL query that is either executed on the server on encrypted data or on the temporary database of the client on decrypted data. Intermediate tables are transferred to the client. This local/remote split of the operator tree enables executing all SQL queries, since the local database can execute any query on the decrypted, retrieved data. As such our encrypted database supports the entire SQL functionality. For a detailed description of the algorithm see [KHKH+13].

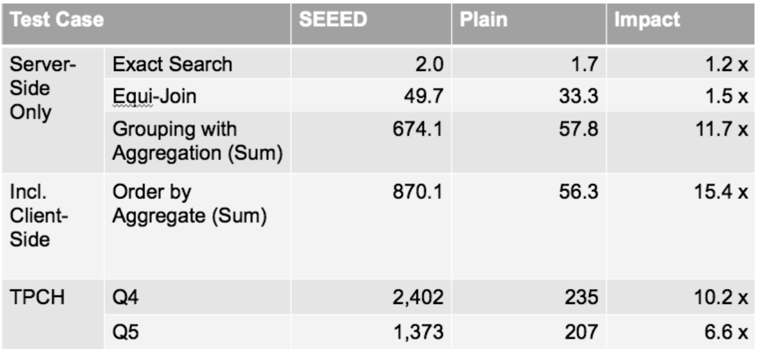
3.2 Performance

On basis of current prototypical implementation we made the following initial performance measurements (in ms). These are based on running queries for a table size of 1,000,000 rows. Both, server and client are running on HP Z820 workstations with 8 quad core CPUs and 128 GB RAM, operating SUSE Linux Enterprise Server 11 SP2.

Looking at only an exact search and only taking the server-side, i.e. the computation on the encrypted data, into consideration we can see that the impact (Factor 1.2) is marginal and the same is the case for an equi-join (Factor1.5). On the other hand, a grouping with aggregation, i.e. sum(), operation is costly and will incur a penalty of factor 11.7. It is an open research question to identify more efficient additively homomorphic encryption schemes. Lattice-based cryptography seems to be a promising candidate.We can observe the impact of the aggregation as well in the TPCH query 5, which besides some basic selection, join and a range condition includes a sum() operation.

```
SELECT
   N_NAME, SUM(L_P_DISC_PRICE)
FROM CUSTOMER,
   ORDERS, LINEITEM, SUPPLIER, NATION, REGION
WHERE C_CUSTKEY = O_CUSTKEY AND L_ORDERKEY = O_ORDERKEY
[...]
AND O_ORDERDATE >= '1994-01-01' AND O_ORDERDATE < '1995-01-01'
GROUP BY N_NAME
ORDER BY SUM(L_P_DISC_PRICE) DESC
```

Figure 5: TPCH Query 5



Test Case		SEEED	Plain	Impact
Server-Side Only	Exact Search	2.0	1.7	1.2 x
	Equi-Join	49.7	33.3	1.5 x
	Grouping with Aggregation (Sum)	674.1	57.8	11.7 x
Incl. Client-Side	Order by Aggregate (Sum)	870.1	56.3	15.4 x
TPCH	Q4	2,402	235	10.2 x
	Q5	1,373	207	6.6 x

Figure 6: Performance measurement

3.3 Security

It makes no practical sense to reason about security just on basis of the applied encryption schemes. For example, any order-preserving scheme will be rather more subject to a statistical attack than a fully randomized scheme. What is more important is to understand the impact of queries on individual columns in terms of possible re-encryption operations as well as the domain-specific semantics of the involved data.

Example I: The TPCH Industrial Benchmark

The TPCH benchmark is a standard ERP database benchmark, with in total 61 different columns in 8 tables. Running all 22 queries of the TPCH benchmark results in the following adjustment at the individual column level: 27,90 % of all columns stay randomly encrypted and 39,30 % of all columns stay deterministic or randomly encrypted.

	TPCH	TPCC	Customer X
Total Queries	22	20	406
Total Tables	8	9	2
Total Columns	61	71	248
RND (columns / %)	17 / 27,9%	49 / 69%	157 / 63,3%
DET (columns / %)	24 / 39,3%	17 / 24%	74 / 29,8%
OPE (columns / %)	20 / 32,8%	5 / 7%	17 / 7,9%

Figure 7: Distribution of Encryption Onions

So, the interesting result is that only 32,80 % of all columns have to be encrypted with lower security encryption schemes to achieve the full functionality of the TPC-H benchmark. Some of these columns appear to be rather descriptive, i.e. P_Name or P_Brand, where it may be arguable whether even cleartext would be of any value to an attacker.

Example II: The TPCC Industrial Benchmark

The TPCC is another benchmark focused on OLTP scenarios. For the TPCC benchmark, the results are even more promising as only 7.9% of all data appear to be subject to a re-encryption to the OPE scheme and in fact 69% of all data remain in a fully deterministic encryption. Looking in more detail at the semantics of the OPE affected columns, we again find rather descriptive names such as C_First (Customer Firstname), O_ID (Order ID) or S_QUANTITY (Order Quantity).

Example III: Major International Consumer Goods Producer

The ERP database we evaluated consumes about 43 Gigabyte pure disk space. As part of our evaluation we only considered two tables with in total 248 columns. Initially, we faced 936 queries touching these tables, but could reduce these to 406 by leaving out blanks and batches resulting in the following observations: Many sum() operations included in the queries will lead to many Paillier encryptions (about 118 columns). All in all we did not encounter any complex queries, some queries requiring DET / DETJOINs but mostly “retrieval” of Randomized and Paillier Encryptions. More significantly, from 248 columns only 17 need an OPE encryption for these 406 queries. Those 17 columns again appear to be rather non-critical, eg. time of record creation, company code, business area or year.

4 Discussion

Our current research has from the beginning onwards focused on providing functionality as our customers have a business to run. We strongly believe that any solution comparable to what we propose will have to implement mechanisms to split query execution between the cloud and an on-premise environment if we want to support complex business queries. We saw that very initial performance measurements on somewhat realistic datasets appear to be acceptable in some cases, e.g. for exact search and equi-join, while other operations such as grouping with aggregation may not be acceptable in an industrial context. However, what may be acceptable from a functional and performance perspective has to be evaluated in the context of different types of data storage scenarios, e.g. a customer requiring secure data archiving for quarterly audit purposes as opposed to a manufacturer with real-time production data analysis needs.

Regarding the data provided when analyzing three industrial data sets with respect to resulting column encryption after query execution in section 3.3, we clearly demonstrated that „security“ is domain-specific. It is up to the customer to decide and accept the risk whether an outsourced data set containing post codes may be re-encrypted using an order-preserving scheme. This of course has an impact on the administrative capabilties of our solution regarding the entire database lifecycle. When we initially encrypt data using our onion approach before outsourcing it to the cloud, we have to be able to support selective and constrained columnar encryption. Selective means that a customer may from the beginning onwards decide to a leave column unencrypted, whereas constrained implies that a column may never be re encrypted using, for example, an order-preserving scheme. This of course immediatly suggests to create domain-specific templates, e.g. for the financial or healthcare industry, which will help the customer in optimizing his configurations. At runtime, i.e. when the data has been outsourced and query execution is running in the cloud, the made configurations will impact our query process as detailed in section 3.1. For example, if a legitimate query with a range condition encounters a column that should not be re encyrpted using an order-preserving scheme, then the query plan has to rewrite the query such that the data set and subset of the query including the range condition are executed on the client.

5 Summary & Conclusion

This paper is an industrial experience report on the prototypcial implementation of a system to execute SQL over encrypted data. Our writing style reflects the “hands-on” approach but we tried to compensate for any scientific imprecision by pointing the reader to the relevant existing research body of work. We have demonstrated that the idea of adjusting the encryption levels by Popa et al. [PRZB11] works in a real-world setting and significantly improves the security compared to order-preserving only approaches [AKRX04]. Furthermore we enhanced their algorithms, such that they work for arbitrary SQL queries. While this is mostly an algorithmic effort in database design, such functionality can play a crucial role in adoption, since the application’s SQL queries do not need to be rewritten when moving to encryption. Furthermore we present the resulting performance trade-off.

The most prominent recent result, however, is that of having analysed three data sets with respect to resulting column encryptions after query execution. We clearly demonstrated that security is domain-specific. Future work will now focus on automating data set and query analysis to constrain (onion)encryption before outsourcing data to the cloud as well as defining acceptable re-encryption operations or local / remote split query strategies at runtime.

References

[AKRX04] Rakesh Agrawal, Jerry Kiernan, Ramakrishnan Srikant, and Yirong Xu. Order preserving encryption for numeric data. In Proceedings of the 2004 ACM International Conference on Management of Data, SIGMOD, 2004.

[BCLN09] Alexandra Boldyreva, Nathan Chenette, Younho Lee, and Adam O'Neill. Order-preserving symmetric encryption. In Proceedings of the 28th International Conference on Advances in Cryptology, EUROCRYPT, 2009.

[FMLG+12] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees, "The SAP HANA database – an architecture overview," IEEE Data Engineering Bulletin 35(1), pp. 28–33, 2012.

[HILM02] Hakan Hacigümüs, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing sql over encrypted data in the database-service-provider model. In Proceedings of the ACM International Conference on Management of Data, SIGMOD, 2002.

[KHKH+13] Florian Kerschbaum, Martin Härterich, Mathias Kohler, Isabelle Hang, Andreas Schaad, Axel Schröpfer, Walter Tighzert. An Encrypted In-Memory Column-Store: The Onion Selection Problem. In Proceedings of the 9th International Conference on Information Systems Security, ICISS, 2013.

[KHGK+13] Florian Kerschbaum, Martin Härterich, Patrick Grofig, Mathias Kohler, Andreas Schaad, Axel Schröpfer, Walter Tighzert. Optimal Re-Encryption Strategy for Joins in Encrypted Databases. In Proceedings of the 27th IFIP WG 11.3 Conference on Data and Applications Security, DBSEC, 2013.

[Pai99] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In Proceedings of the 18th International Conference on Advances in Cryptology, EUROCRYPT, 1999.

[PH78] Stephen C. Pohlig and Martin E. Hellman. An improved algorithm for computing logarithms over gf(p) and its cryptographic significance. IEEE Transactions on Information Theory, 24(1):106–110, 1978.

[PRZB11] Raluca Ada Popa, Catherine M. S. Redfield, Nickolai Zeldovich, and Hari Balakrishnan. Cryptdb: protecting confidentiality with encrypted query processing. In Proceedings of the 23rd ACM Symposium on Operating Systems Principles, SOSP, 2011.

[SWP00] Dawn X. Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In Proceedings of the 21st IEEE Symposium on Security and Privacy, S&P, 2000.