Initial Encryption of large Searchable Data Sets using Hadoop

Feng Wang
SAP SE
Karlsruhe, Germany
feng.wang02@sap.com

Mathias Kohler
SAP SE
Karlsruhe, Germany
mathias.kohler@sap.com

Andreas Schaad
SAP SE
Karlsruhe, Germany
andreas.schaad@sap.com

ABSTRACT
With the introduction and the widely use of external hosted infrastructures, secure storage of sensitive data becomes more and more important. There are systems available to store and query encrypted data in a database, but not all applications may start with empty tables rather than having sets of legacy data. Hence, there is a need to transform existing plaintext databases to encrypted form. Usually existing enterprise databases may contain terabytes of data. A single machine would require many months for the initial encryption of a large data set. We propose encrypting data in parallel using a Hadoop cluster which is a simple five step process including the Hadoop set up, target preparation, source data import, encrypting the data, and finally exporting it to the target. We evaluated our solution on real world data and report on performance and data consumption. The results show that encrypting data in parallel can be done in a very scalable manner. Using a parallelized encryption cluster compared to a single server machine reduces the encryption time from months down to days or even hours.

Categories and Subject Descriptors
H.2.0 [**Database Management**]: General – *security;* H.3.4 [**Information Storage and Retrieval**]: Systems and Software – *Distributed systems, Performance evaluation*.

General Terms
Performance, Security

Keywords
Database; Searchable Encryption; Hadoop; Performance

1. INTRODUCTION
Storing data externally with a hosted provider becomes more and more attractive to companies which want to save costs on their own physical infrastructure. Since the cloud service providers have access to all data, clearly, storing sensitive data externally asks for solutions providing respective privacy precautions such that only the data owner is able to access the data.

Inspired by CryptDB [1], we developed a system to encrypt and store data on a database, all happening transparently to the application. Hence, it can be used in the same way traditional databases are accessed to execute a large set of queries over the encrypted data. Introducing such an encrypted database solution to an existing application landscape requires an initialization phase where all unencrypted data is transferred and encrypted into the new database.

Since most of the encryption operations are slow, encrypting a plaintext table is very time-consuming, especially when the table is large. Using tables enabled for adjustable encryption [13] having multiple encryption layers for one database entry adds significantly to the computation time for the initial encryption. We show how to make the initial table encryption an automated process and much more efficient using a Hadoop cluster for parallelized data processing.

The rest of the report is structured as follows. Section 2 describes the technical details. In Section 3, we discuss the performance test and its result, and conclude with Section 4.

SACMAT '15, June 1–3, 2015, Vienna, Austria

ACM 978-1-4503-3556-0/15/06…$15.00
http://dx.doi.org/10.1145/2752952.2752960

2. IMPLEMENTATION
In this section, we will discuss the technical details of how to encrypt the database in parallel.

2.1 Search over Encrypted Data
Inspired by CryptDB [1], we implemented a JDBC driver for connecting to database containing encrypted data. The goal of the driver is to realize accessing encrypted data transparently to the client application, which means a large set of regular SQL queries can be used to search over encrypted data.

SQL operations (equality search, range search, aggregations, etc.) require that data is stored with different encryption schemes. When encrypting a database, each plaintext data item is encrypted using multiple schemes, depending on the SQL operations to be executed on the data. In total there are five types of encryption schemes we use:

- Random (RND): RND has the highest security level. Two same plaintexts result in different ciphertexts. This scheme is used for secure data storage and retrieval only, no other SQL operation is supported on these ciphertexts. We use AES in CBC mode (with Padding).
- Deterministic (DET): For DET holds $DET(x) = DET(y)$, if $x = y$. Hence, equality operations on the ciphertext is supported. AES in ECB mode (with padding) is used.
- Order-preserving encryption (OPE): For OPE, if $x < y$, then $OPE(x) < OPE(y)$ holds. It supports SQL range queries on the ciphertext. We use the algorithm introduced by Boldyreva et al. [2].

- Additive homomorphic encryption (HOM): We use the Paillier algorithm [3] for aggregating encrypted values as it holds: $HOM(x) \cdot HOM(y) = HOM(x + y)$.
- Re-encryption (JOIN): For maximum security, all columns are encrypted with different keys. Joining two columns, however, requires them to be deterministically encrypted with the same key. With the encryption scheme introduced by Pohlig and Hellman [4] we support on demand re-encryption of a column to be on the same key as another column for join operations (see [13] for more details).

For the support of an initial maximum security and storage optimization, we use adjustable encryption [13]. This means a plaintext item *x* is encrypted in a layered approach with multiple encryption schemes such that, for instance, *RND(DET(OPE(x)))* holds. If during an SQL execution an in-between-layer is required, top layers are removed on demand to reveal the respective layer.

Obviously, even if stacked, we want the encryption schemes to keep their characteristics we listed above. Hence, schemes on a lower layer must have the same characteristics required for the layer stacked on top of it (e.g., OPE must be deterministic, if DET is stacked on top). We use four stack structures to support multiple SQL operations in one query (e.g., aggregation and a range condition). Moreover, we provide two stack structures for integers and one stack structure for strings. Table 1 gives an overview.

In Table 1, the row "Types" stands for the type of plaintext data the characteristics of the encryption schemes are actually applicable. For example, for integer or decimal values, the additive homomorphic encryption scheme is required, to actually enable aggregations; it is, however, not required for strings. For sorting items (with OPE) different paddings are used for integer and string values; moreover, in our use cases joins are only done with integer values such that we omit this layer for strings. Stack structure 1 is used for data retrieval to the client and group by selections only. Having AES-based encryption schemes in this stack is a very fast solution for decrypting result sets on the client.

Table 1 Encryption Structure

	Stack 1	**Stack 2**	**Stack 3**	**Stack 4**
Layer 1	DET	OPE	OPE	HOM
Layer 2	RND	RND	JOIN	
Layer 3			RND	
Types	Integer, String, Date, Decimal	String	Integer, Date, Decimal	Integer, Decimal
Usage	Retrieval, Group By	Range queries with strings	Joins, Equality, Range queries w/ numbers	Aggrega-tion

2.2 Encryption

We use the implemented JDBC driver and a Hadoop cluster to encrypt the plaintext table. There are five steps in total for the complete process. The basic workflow is shown in Figure 2.

2.2.1 Set up

Firstly the cluster to encrypt the data needs to be set up. We use Apache Hadoop [5], Apache Sqoop [6], and Apache Oozie [7]. These tools are installed in the cluster.

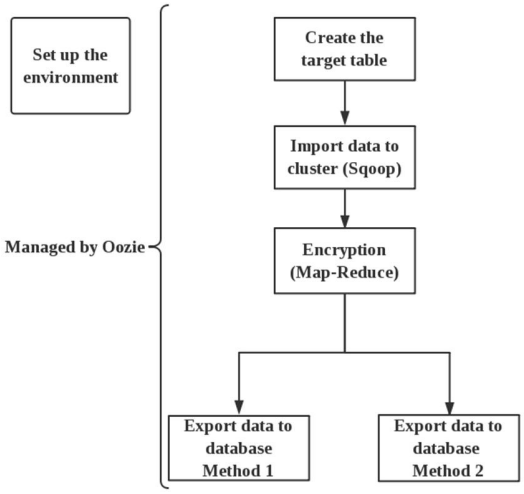

Figure 2. Workflow

Table 2 Plaintext Tables

Table Name	**Number of Columns**	**Number of Entries**	**Size on Disk (KB)**
BKPF	111	9,737,795	1,059,148
ORDERS	10	1,280,000	98,308
KNA1	175	14,554	3,076
KNB1	77	10,269	620
T001	79	269	448
T003	38	132	212
T005T	8	2,531	132
T016T	4	40	56
TBSLT	5	1,049	72

Table 3 Experiment Result

Table Name	**Import data to cluster**	**Encryption**	**Export data to database**	
			Method 1	**Method 2**
BKPF	1h42m	27h41m	About 24h	About 5d4h
ORDERS	1m40s	2h1m	23m	1h40m
KNA1	40s	19m	2m	10m
KNB1	1m	4m	50s	3m
T001	30s	2m	30s	40s
T003	30s	2m	30s	30s
T005T	30s	4m	30s	2m
T016T	30s	1m	30s	1m
TBSLT	30s	1m	30s	1m

Table 4 Encrypted Tables

Table Name	**Number of Columns**	**Number of Entries**	**Size on Disk (KB)**	**Size vs. Plain-text**
ENC_BKPF	227	9,737,795	~153,135k	144x
ENC_ORDERS	26	1,280,000	3,011,868	30x
ENC_KNA1	353	14,554	289,992	94x
ENC_KNB1	160	10,269	116,944	188x
ENC_T001	159	269	1,800	4x
ENC_T003	77	132	444	2x
ENC_T005T	17	2,531	4,324	33x
ENC_T016T	9	40	268	5x
ENC_TBSLT	11	1,049	516	7x

2.2.2 Create target table

We use the above mentioned stack structures to prepare and create the target table. Given the source table, for each of its plaintext columns, one or more encrypted columns will exist in in the encrypted table; each of these columns corresponds to exactly one stack. There will always be an encrypted column corresponding to Stack 1 for retrieving data. Which other stacks are created in addition is either selected automatically according to the source column's data type or is manually selected by the user. For instance, a plaintext integer column would have three encrypted columns in the target table, storing the encrypted data according to stack 1, 3, and 4. If it is an identifier column, the user might manually omit the column with HOM encryption (stack 4) as identifiers are usually not aggregated.

2.2.3 Import data to cluster

The third step is to import the plaintext data from the database to the cluster's file system called HDFS. To achieve this goal, we use Apache Sqoop. Sqoop is a tool for transferring data between Apache Hadoop and the relational database. The relational database can be from any database vendor. In our case, SAP HANA [9] is used. The imported data is stored in a CSV file type-like structure, which means each line in the file represents an entry of the table.

The performance of this step mainly depends on the database itself and the network between database and cluster. The data imported to the HDFS will be the input for next step.

2.2.4 Encryption

The forth step is to do the encryption on the cluster. It is a map-reduce process. As we have mentioned above, the imported data is in CSV file type, which means each line in the file represents an entry of the plaintext table. So in the map stage, the imported file is read line by line, and each line is encrypted using the encryption algorithms mentioned above. This step does not need the reduce stage. The output of this step is also CSV file(s) which is according to the encrypted table structure previously created.

The performance of this step mainly depends on the performance of the cluster.

2.2.5 Export data to database

The last step is to export the data from the cluster to the database. We implemented this step by two different methods.

The first method is to use Sqoop which is similar to Section 2.2.2. Sqoop export uses the INSERT SQL statement, which means it will use a statement like *INSERT INTO <TABLE> VALUES (<ROW>)* to export data from the Hadoop cluster to the target table. Since this method needs to iterate through all data line by line, it is quite slow. For some relational databases a batch mode is supported. Each time multiple lines can be inserted by using *INSERT INTO <TABLE> VALUES (<ROW1>), (<ROW2>), etc.*, and obviously this will make the exporting faster. As a result, the performance of the first method highly depends on if the database supports the batch mode INSERT. Moreover, it also depends on the condition of the network. And for some large tables, this exporting process requires the cluster connecting to the database for a long time, which may result in the connection being closed during the process.

The second method we implemented deals with exporting the data in a naïve way. This method requires the cluster having the Hadoop NameNode Web Interface installed. After the previous step (map-reduce encryption) is finished, the database server can directly download the output file(s) from the NameNode Web Interface. Then it obviously depends on the database which commands to be used to import the data into the database. In our case, using SAP HANA, the IMPORT FROM FILE [10] command is used.

The performance of the second method depends on the network and the database IMPORT performance.

2.2.6 Workflow management

The previous steps (Section 2.2.2 to Section 2.2.5) are all scheduled and managed by Apache Oozie. Oozie is a workflow scheduler tool to manage the Hadoop jobs. The reason to use Oozie is that it can make the implementation "cleaner", easier to modify, and more reliable.

3. EVALUATION

In this section, we will discuss our experiments with the above described setup and its performance.

3.1 Environment setup

SAP HANA is used as database. Typically for this database, all data is stored in memory in tables using column-based storage.

The Hadoop cluster resides on an internal virtual hosting environment. There is one master, twelve slaves, and one Ambari server [8]. The master machine has 8 CPUs, and 31.6 GB memory. The slave machines have each 8 CPUs, and 16 GB memory. All have between 100 and 300 GB available space.

3.2 Plaintext Tables

We have chosen 9 plaintext tables in total to test the performance. All the tables are standard SAP ERP tables and mainly come from the application of sFIN [11]. The details are shown in Table 3.

3.3 Time

The tables listed in Table 3 are encrypted one by one. There is only one job running in parallel, and before a new job is started, the cluster is cleaned. For all columns we create encrypted columns (stacks) according to the type of the plaintext column given with Table 1. We report on both methods to export the data to database (see Section 2.2.4). The results are showed in Table 4.

From Table 4, it is clear that the direct export using INSERT statements is much faster if it comes to large tables rather than using method 2 with CSV files. Method 2 takes about 5 times as much for processing then method 1. A reason we see is that in method 2 all encrypted data has to be extracted and copied from Hadoop's file system to the database server where it then has to be processed a second time for importing it with the IMPORT command.

3.4 Encrypted Tables

After the encryption, there are 9 encrypted tables corresponding to the plaintext tables. Their information is shown in Table 5.

3.5 Scalability test

We test the scalability of our solution with a midsized table ORDERS. We report on the scalability of the encryption with respect to the size of the table as well as with the size of the Hadoop cluster, i.e., number of working nodes.

3.5.1 Different table size

Firstly we use a fixed number of nodes (1 master server, 12 slave nodes) in cluster and change the table's size by randomly selecting a predefined number of entries from table ORDERS. And for exporting, we use method 1. The results are shown in Table 5.

Table 5 Scalability test for different table size

Number of entries	Time (m)			
	Import	Encryption	Export	Total
1,280,000	1.6	121	23	145.6
1,120,000	1.3	98.3	32	131.6
960,000	1.5	87.3	22	110.8
800,000	2.1	73.25	10.6	85.95
640,000	1.5	59.3	10.5	71.3
480,000	1	43	7	51
320,000	0.8	33	4.5	38.3
160,000	0.5	16	2.5	19
10,000	0.5	5.3	0.6	6.4

A more intuitive result is showed in Figure 3. From it, it is clear that with the linear increasing table size, the computation time also increases linearly.

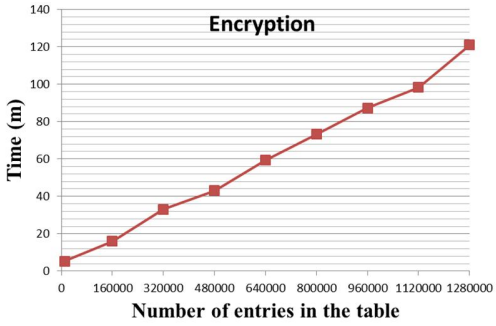

Figure 3: Computation time for different table sizes

3.5.2 Different number of nodes in cluster

We also test the influence of the number of nodes in cluster on the computation time. Because in the cluster only slave nodes store the data and do the computation, we increased the number of slave nodes from one node to 12 nodes in total. The complete table ORDERS is used for each encryption, remaining stable in its size. For exporting, again method 1 is used. The results are shown in Table 6.

Table 6 Scalability test for different number of nodes

Number of slave nodes	Time (m)			
	Import	Encryption	Export	Total
12	1.6	121	23	145.6
9	4	150	34	188
6	1.6	213.9	33	248.5
3	1.5	432.25	21.3	455.05
1	1.5	1498.5	25	1525

A more intuitive result is showed in Figure 4. From it, we can see the number of slave nodes affects the computation time and correlates directly with the number of nodes used. The time roughly drops to a third by using 3 nodes instead of only one. And it drops roughly to 1/12 when using 12 nodes.

4. CONCLUSION

In this report, we mainly talk about how to encrypt tables in parallel. After the environment setup, the idea is to divide the process in three main steps: import data to cluster, encryption, and export data to database. For step "import data to cluster", Apache Sqoop is used. For step "encryption", Map-Reduce is used. In step "export data to database", there are two potential methods. The first one uses Sqoop too, and the second one is a naïve method where the data is copied to the database with a database-specific import command.

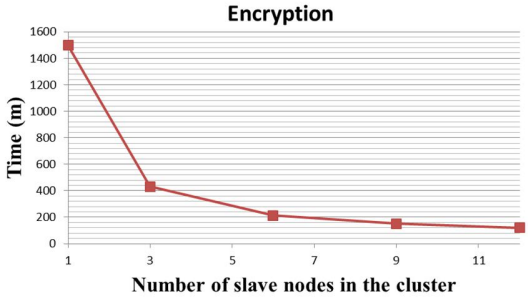

Figure 4: Computation time for different node numbers

We have run tests using real world tables. The results show that an automated encryption solution is feasible and using a Hadoop cluster reduces encryption time drastically in a very scalable manner.

5. REFERENCES

[1] Popa, R. A., Redfield, C., Zeldovich, N., & Balakrishnan, H. 2011. Cryptdb: protecting confidentiality with encrypted query processing. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*.

[2] Boldyreva, A., Chenette, N., Lee, Y., & O'neill, A. 2009. Order-preserving symmetric encryption. In *Advances in Cryptology – EUROCRYPT*.

[3] Paillier, P. 1999. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in cryptology – EUROCRYPT*.

[4] Pohlig, S. C., Hellman, M. E. 1978. An improved algorithm for computing logarithms over and its cryptographic significance (corresp.). In *Transactions on Information Theory*.

[5] *Apache Hadoop*. http://hadoop.apache.org/.

[6] *Apache Sqoop*. http://sqoop.apache.org/.

[7] *Apache Oozie*. http://oozie.apache.org/.

[8] *Apache Ambari*. http://ambari.apache.org/.

[9] *SAP HANA*. http://hana.sap.com/abouthana.html.

[10] *SAP HANA 'IMPORT FROM'*. http://help.sap.com/saphelp_hanaplatform/helpdata/en/20/f712e175191014907393741fadcb97/content.htm.

[11] *SAP Simple Finance aka sFIN*. http://scn.sap.com/docs/DOC-59882.

[12] Kerschbaum et al. 2013. Optimal Re-Emcryption Strategy for Joins in Encrypted Databases. In *Working Conference on Data and Applications Security and Privacy (DBSec)*.

[13] Florian Kerschbaum et al. 2013. Adjustably encrypted in-memory column-store. In *ACM Conference on Computer and Communications Security*.