Analyzing Encrypted Analytics

Luke Zhu

Abstract

Integrating column-level encryption into data analytics could make it possible for analysts to conduct queries on encrypted data by default. We found that adding encryption slows down queries by at least 2.5x and requires a somewhat sophisticated key management solution.

1 Intro

Around a decade ago, big data tools like Hadoop and Spark made analyzing TBs to PBs of data possible. Nowadays, these tools are widely used in enterprise [2]. While big data distributions come with features for protecting data-at-rest and data-in-transit [3, 4], data presented to analysts is left in plaintext. This means that data analysts typically have access to TBs of data, some of which is personally identifiable information (PII) like name, address, and social security number.

While basic authorization features exist in data lakes to solve this problem, the amount of work needed to ensure privacy standards are met cause corners to be cut. Access control and ETL are two components which require lots of time to configure:

Access Control: While data lakes and data warehouses come with access control features, there's a reason to believe that they aren't be used properly. In 2017, 207 HDFS clusters were found exposing 5.12 PBs of data were found through the Shodan search engine [1]. This means that hundreds of clusters were not configured to use basic security features at all. Also, data lakes and data warehouses contain hundreds of tables. Many of these tables have dozens of columns. It's impractical for admins to properly configure access for each of these tables.

ETL: PII needs to manually removed or masked before being put into the data lake or warehouse. This means that every new table created requires a custom ETL pipeline to be made. This is difficult when hundreds of tables needed to be updated from dozens of upstream databases.

To summarize, existing approaches require too much manual work given the scale of data. As a result, we decided to explore the potential of a more automated solution: encrypting all data by default and auditing all decryption requests.

The main contributions of this paper are as follows:

- Exploring the feasibility and performance impact of integrating encryption into Apache Spark. (Section 5)
- Describing a basic implementation of secure encrypted Spark involving delegating identity and access management to AWS. (Sections 3 and 4)

2 Background

2.1 Model

We protect against employees in the company are honest but curious. These employees may often work with the data and incidentally see large amounts of PII. The main change in our solution is that these users now see encrypted data by default. If they need to decrypt some data in order to properly do their job, their decryption request is audited. This dissuades employees from excessively decrypting zip codes or decrypting fields like SSN at all.

We also aim to identify malicious employees that want to extract inside knowledge.

Example Attack: A marketing analyst is ready to leave for another company and wants to get the contact information and subscription renew dates for existing customers. To do this, they would run a SQL query to fetch the query through the SQL dashboard they use daily.

In our solution, the marketing employee must decrypt the contact info and subscription renewal dates in order to see them. As a result, these queries will be audited. While this doesn't prevent the employee from running a query, the legal ramifications of doing so effectively do.

We also consider malicious employees who want to delete or modify. We aim to solve this issue by maintaining compatibility with time travel (also known as point-in-time recovery) tools and access control features.

It should be noted that the level of privacy described is weaker than differential privacy, which prevents analysts from running queries after exceeding a privacy loss budget.

2.2 Related Work

Big data distributions also come with access control features. Enterprise distributions of data lakes and data warehouses generally come with file and table access control [5,6]. Some now support column-level and row-level access control [7]. Free distributions do not. As noted earlier, these leave time consuming access management and ETL tasks to the user.

BigQuery supports column-level encryption functions [8]. We explore the feasibility of automatic encryption and decryption as a solution but do not implement it.

Researchers in 2019 have worked on incorporating fully homomorphic encryption into Spark [9, 10] using the Microsoft SEAL library [11]. We focus instead on the key management aspect in our implementation and avoid forking Spark. Customers often use managed services/distributions like AWS EMR or Apache Ambari, which use their own Spark JARs.

Open-source contributors have worked on integrating encryption into Spark by extending the Parquet file format [12, 13]. We survey into both format-specific approaches and format-agnostic ones.

3 Design

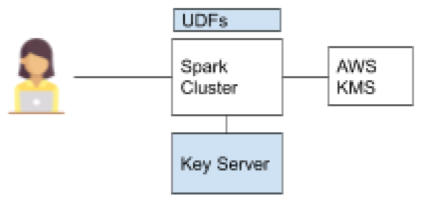

Figure 1: Users send queries using the encryption and decryption UDFs to the Spark cluster. Encrypted data keys are stored in the key server and decrypted through KMS.

3.1 Query Path

The user sends queries to the Spark SQL application through traditional options like `spark-shell` or `spark-submit`. The Spark driver schedules tasks to be run on the executors. A query looks like the following:

```
df.select(decrypt(col("SSN")))
  .first()
// or
spark.sql("SELECT decrypt(SSN) FROM
df").show()
```

If the user uses an encryption method like above, SparkSQL will fetch the corresponding AES secret key for the table-column pair or create it. The key is used for encrypting and decrypting all values in the column in memory.

3.1 Key Server

When a key for a (table, column) pair does not exist, code in the Spark executors generates a data key using AWS Key Management Service (KMS). AWS KMS generates data keys using a Customer Managed Key which cannot be viewed by all users.

We store an encrypted version of the (table, column) key in a Golang server connected to etcd. When the key is needed, the Spark application will fetch the key from the server and send a decryption request to AWS KMS. This ensures that only users with authorization to decrypt the key can use it.

The data key is used to encrypt or decrypt every row in the dataset, so we cache the plaintext in memory for each Spark executor. This is a potential vulnerability only if the user is able to read the memory of other Spark applications running that have the desired data keys. In that case, they would also be able to access AWS credentials as they are also loaded into memory.

3.2 Encryption

Basic AES encryption and decryption provided by javax.crypto. Spark UDFs are used to encrypt and decrypt values.

3.3 Authorization

Users use AWS credentials in order to access resources.

```
spark.encryption.aws.access.key=AK...
spark.encryption.aws.secret.key=w/4...
```

Generally, users would also set the following variables in order to access files on S3:

```
spark.hadoop.fs.s3a.access.key
spark.hadoop.fs.s3a.secret.key
```

Using the same credentials allows admins to integrate S3 bucket-level and file-level access control with our application's table and column permissions.

4 Implementation

The current implementation consists of 250 lines of Go and 220 lines of Java and is available at https://github.com/luke-zhu/encrypted-spark.

It was difficult to find a way to modify Spark without significantly dropping performance. For example, not using the Parquet file format means losing the Parquet vectorized reader, which is around 9x faster than non-vectorized file readers [15].

5 Evaluation

5.1 Performance

We compared writing 100,000,000 small strings to disk with encrypting and then writing the strings. This was run on a local Spark cluster on a Dell XPS13 laptop.

The final size of the plaintext was 445 MB and the decrypted size was 2.5 GB. This is due to poorer compression of encrypted data and AES padding.

When using a single CPU in a single-core environment, it took 40x longer to save the encrypted DataFrame (875 seconds vs 19 seconds). When utilizing all 8 CPUs, it took 35 seconds to save the encrypted DataFrame vs 15 seconds to save the decrypted DataFrame.

In both cases, CPU was the bottleneck. This suggests that more efficient methods for encryption and decryption would improve performance. Using an optimized C++ implementation or writing a custom implementation that operates on Tungsten types could speed things up significantly.

The penalty for using encryption is significant but less than an order of magnitude large. This suggests that encryption could be useful in the future.

However, there are other problems with a UDF-based approach that makes some queries hard to write.

5.2 Feasibility of Automatic Encryption

An original goal of the project was to support automatic encryption. In this case, the user would write Spark SQL code without defining any UDFs. We decided to drop this from the implementation because (1) the Parquet DataSource contained significant optimizations related to vectorization and (2) the Parquet DataSource could not be extended to include encryption without modifying the Spark source code itself.

Without modifying the Spark source code, it's possible to add functionality to Spark through the following interfaces:

- DataSources
- Optimizer rules
- UDFs and native functions

Unless one heavily duplicates the code in the Parquet DataSource while creating a new DataSource, it seems difficult to integrate encryption into Spark.

Modifying the Spark source code could lead to a fairly performant implementation. For example, it's possible to add operators and rules to the optimizer to determine whether encryption or decryption is needed.

5.3 Other Performance Issues

Encryption leads to performance losses in various

- **File size:** Compression improves when there are patterns in the values. Encryption removes these patterns.
- **Partitioning:** Spark queries can run more quickly by only loading data from relevant

files. Order preserving encryption is needed in order to not lose locality.

5.4 Problems with Authorization

In order to make key performance feasible, we needed to generate data keys using KMS. However, this means that our application must manage the lifecycle of these keys. We were not able to get access management into the application.

A reason to use managed key services like AWS KMS, GCP KMS, and Azure Key Vault is that they come with features like key rotation. However, it would require a non-trivial amount of effort to integrate this into this system, as the data key storage is decoupled from the data storage.

6 Conclusion

Encryption in data lakes and data warehouses, if feasible and easy-to-integrate, could change how data analysis is done on PII. Our preliminary results show that the encryption in spark is capped by CPU and ends up slowing down file writes by least 2x.

In addition, encryption in Spark requires keys to be kept in the Spark executors to maximize throughput. Our

[1] https://blog.shodan.io/the-hdfs-juggernaut/

[2] https://www.forbes.com/sites/louiscolumbus/2018/12/23/big-data-analytics-adoption-soared-in-the-enterprise-in-2018/#1e5df70c332f

[3] https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-data-encryption-options.html

[4] https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.0/configuring-hdfs-encryption/content/hdfs_encryption_overview.html

[5] https://docs.databricks.com/administration-guide/access-control/table-acls/index.html

[6] https://cloud.google.com/bigquery/docs/access-control

[7] https://www.qubole.com/blog/data-governance-for-sparksql/

[8] https://cloud.google.com/bigquery/docs/reference/standard-sql/aead_encryption_functions

[9] https://github.com/SpiRITlab/SparkFHE-Examples/wiki

[10] https://www.slideshare.net/databricks/encrypted-computation-in-apache-spark

[11] https://github.com/Microsoft/SEAL

[12] https://databricks.com/session/efficient-spark-analytics-on-encrypted-data

[13] https://issues.apache.org/jira/browse/PARQUET-1178

[14] https://www.2ndwatch.com/blog/popular-aws-products-2018/

[15] https://spoddutur.github.io/spark-notes/second_generation_tungsten_engine.html