Data Engineering Design Patterns

Bartosz Konieczny

Published by O'Reilly Media, Inc.

Chapter 7. Data Security Design Patterns

Undoubtedly, the easily accessible and valuable datasets we've created so far with the data value and data flow design patterns are important business assets. They are also objects of envy for other market actors, including malicious ones.

Consequently, data engineering can't stop at writing data processing jobs. Nowadays, data engineers also need to think about the security aspects. One aspect that has received enormous traction over the last few years is compliance. Data privacy laws, such as the General Data Protection Regulation (GDPR) in Europe or the California Consumer Privacy Act (CCPA) in the US, define precisely the boundaries between you (the data provider) and a customer (the data consumer).

Another important aspect is access control. Imagine that your dataset is open within your organization and a different team accidentally overwrites it. The consequences for you and downstream consumers can be huge.

Data protection is another important security aspect. Even if you accidentally give access to a dataset location, such as a table or a path in the file system, it doesn't mean the user will be able to read the data. If you used an extra protection layer, such as encryption, the consumer would also need the decryption key to access the dataset.

Finally, data security also concerns connections. You, as either a human or an application user, will need to connect to a data store to read and write your work's outcome. Clearly, storing credentials in your Git repository increases the data leak risk and is therefore not the best idea. Instead, you should store them in an external and safer place.

It all may sound scary, but no worries. The design patterns presented in this chapter will address the introduced issues and give you some real-world application use cases. This chapter starts with the data removal patterns that show how to comply with users' right to be forgotten in your datasets. Next, you're going to discover fine-grained access patterns for tables and cloud resources. After that comes the data protection patterns that secure datasets with encryption and anonymization techniques. Finally, you're going to see two approaches that address the connectivity issue. One will use references instead of real credentials, while another will rely on identity-based access.

Hopefully, you now feel ready to start exploring the first category, data removal!

Data Removal

Privacy regulations, such as the CCPA and the GDPR, define several important compliance requirements. One of these important requirements focuses on personal data removal requests, in which you must delete a user's data upon receiving a removal request from the user. In this first section, you're going to see two possible implementation approaches that address this issue.

Pattern: Vertical Partitioner

Smart data organization or workflow isolation can often solve even the most challenging problems. This rule applies to the first data removal pattern.

Problem

You recently presented the first design doc for a new personal data removal pipeline. Your peers were very enthusiastic, but some of them pointed out an important storage overhead. Several of the columns of your dataset are immutable. Put differently, they never change, but at the same time, they're present in each record. Examples of these attributes are the birthday or personal ID number.

In their feedback, your peers asked you to store each immutable property only once. As a result, you'll have less data to delete in case of a data removal request.

Solution

The announced problem implies dividing the dataset into two parts, a mutable one and an immutable one. This is where the Vertical Partitioner pattern helps.

In data engineering, you can partition a dataset horizontally or vertically. Horizontal partitioning puts all the records with related attributes together. Some of the most frequent examples of this are date-based partitions, which are often used in batch jobs for incremental data processing. You're going to discover this storage approach in Chapter 8.

Vertical Works with Horizontal

You can combine vertical partitioning with horizontal partitioning and group related but split parts of a record in the same storage zone.

An alternative to horizontal partitioning is vertical partitioning. This approach splits each row and writes the parts into different places. The attribute-based division can help you implement an efficient personal data removal pipeline.

The first step consists of identifying the columns to split and an attribute that you will use to merge the split rows again. After this preparation step, you need to adapt your data ingestion job by adding the attribute-based split logic. This logic will divide a row into two parts and write each of them into a separate storage space. Figure 7-1 shows this logic, in which event attributes with values that are different with each new record are delivered to a separate storage, while the unchanging ones and those related to the personal data scope are written to a different place.

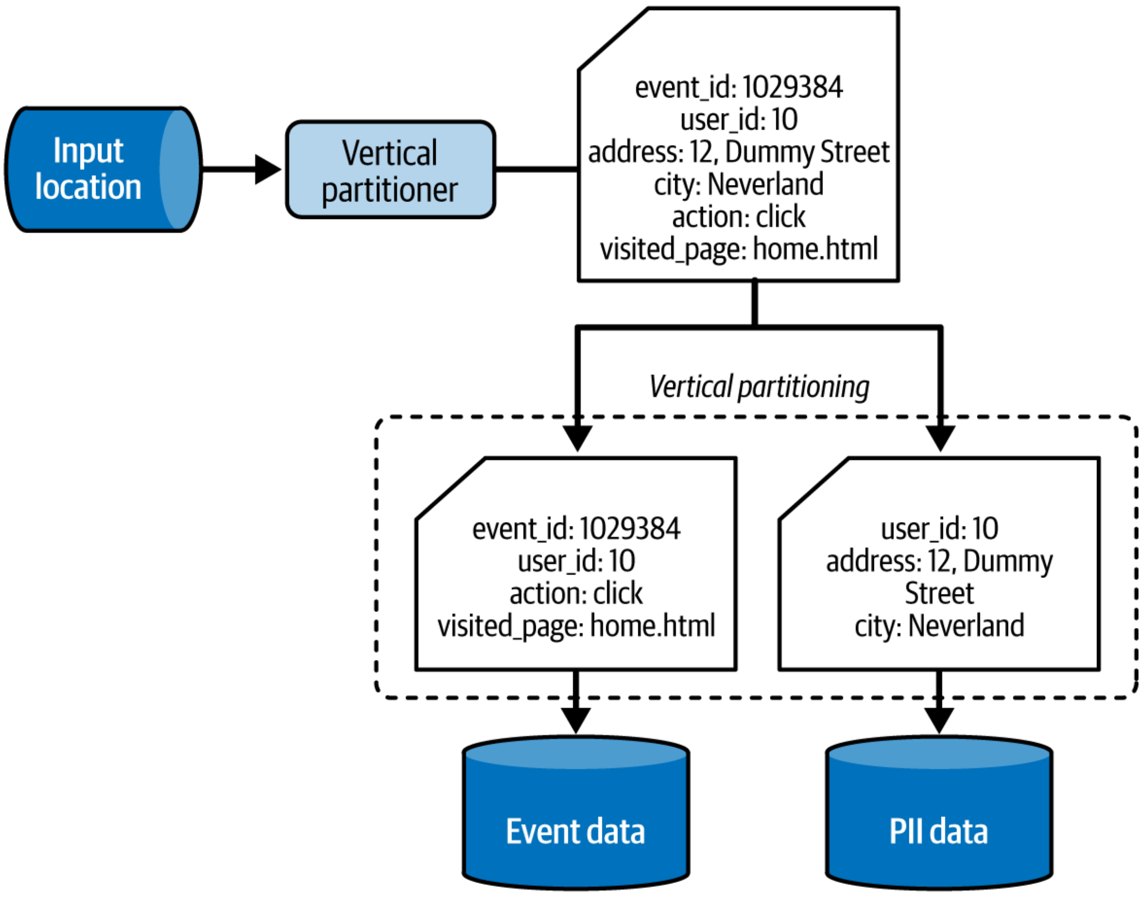

Figure 7-1. Vertical Partitioner extracting and writing PII and immutable attributes to a dedicated storage (note that user_id is the field a consumer can later use to merge the split rows)

The easiest way to implement the split is to use the SELECT statement or an equivalent attribute projection method in a data processing framework. That way, you issue multiple queries, each of which targets a different set of columns and writes the result to a dedicated data store. Eventually, each query can have some business rules, such as deduplication.

When it comes to the personal data removal request, the solution considerably reduces the amount of data to remove. The private attributes being repeated in each record are now written only once. Consequently, the removal process has less data to process. It should be less expensive from both the computational perspective and the financial perspective than a removal from nonpartitioned storage. Finally, thanks to this fully separated storage layer, you can apply different data access or data retention rules.

Consequences

Unfortunately, despite being a good performance optimization technique for the data removal use case, the Vertical Partitioner pattern has some drawbacks for consumers.

Query performance

Vertical partitioning introduces a kind of data normalization in which immutable properties live apart from mutable ones. This optimizes write performance by reducing volume, but it also reduces read performance as each reader will need to join split rows. Any reading operation will then involve network traffic, while normally, for the nonpartitioned version, at the row level, the reading would remain local.

Querying complexity

Data separation also brings some extra complexity to queries. Consumers will be aware that some properties may be located in a different place. Thankfully, it's pretty easy to mitigate this issue by exposing the dataset from a single entry point (such as a view) by providing data documentation (for example, with data catalogs), or by clearly exposing the data lineage (cf. "Pattern: Dataset Tracker").

Complexity in a polyglot world

Another challenge affects the polyglot persistence world, where one dataset lives in different kinds of storage (such as NoSQL databases and relational databases) at the same time. Polyglot persistence is great for readers, as it exposes a given record in technology that's adapted to the consumers. For example, a search engine feature would like to leverage a search-optimized database, while a low-latency microservice might need to get the same record but very quickly from a key-value store. As a consequence, you may need to apply this vertical partitioning across different storage systems, including multiple data removal pipelines.

A solution for that is depicted in Figure 7-2, where a job splits each row, and later, dedicated consumers process the split result according to their expected storage layer.

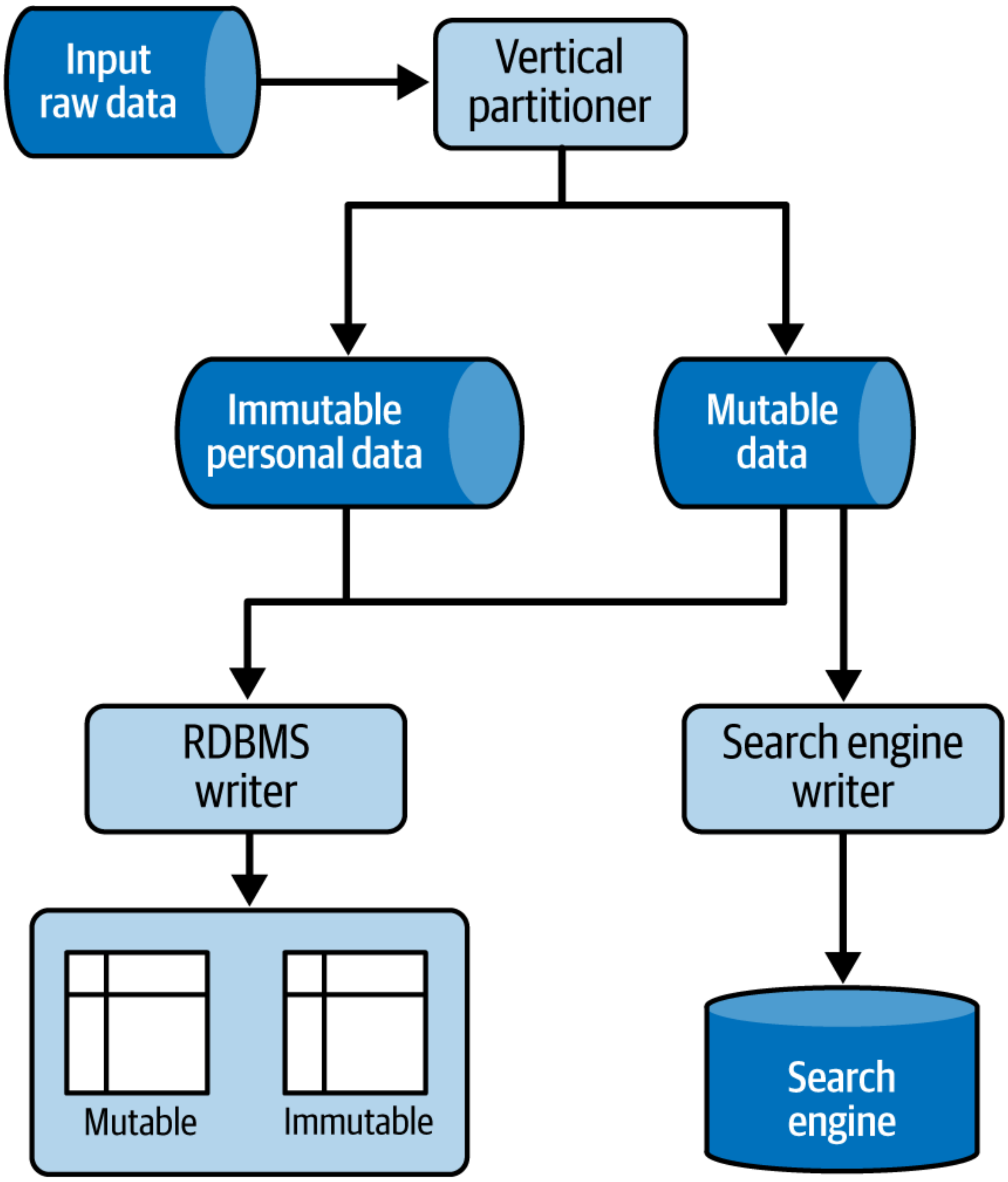

Figure 7-2. Polyglot persistence with vertical partitioning

Raw data

If you need to keep the raw (not divided) data for some time, you'll need a complementary solution to deal with data removal. The Vertical Partitioner pattern applies only from the first data transformation step.

An easy solution would be to use a short retention period for the unsplit data if it complies with the data removal request delay. However, that can reduce data availability for backfilling purposes.

Examples

Now, let’s see how to implement the Vertical Partitioner with Apache Kafka, Apache Spark, and Delta Lake. The first part consists of the job that splits the incoming records. As shown in Example 7-1, it leverages the foreachBatch output operation to create two datasets, each of which is written to a different Apache Kafka topic.

Example 7-1. Vertical partitioning with Apache Spark’s foreachBatch

```
def split_visit_attributes(visits_to_save: DataFrame, batch_number: int):
 visits_to_save.persist()


 visits_without_user_context = (visits_to_save
  .filter('user_id IS NOT NULL AND context.user.login IS NOT NULL')
  .withColumn('context', F.col('context').dropFields('user'))
  .select(F.col('visit_id').alias('key'), F.to_json(F.struct('*')).alias('value')))
 # save to visits_without_user_context


 user_context_to_save = (visits_to_save.selectExpr('context.user.*', 'user_id')
  .select(F.col('user_id').alias('key'), F.to_json(F.struct('*')).alias('value')))
 # save to user_context_to_save


 visits_to_save.unpersist()
```

The split method uses simple column-based transformations to remove user information from the visits dataset and to transform only the user attributes for the user context dataset. The partitioned user context topic can be later converted into a Delta Lake table, with another job using the MERGE operation to deduplicate the entries (see Example 7-2).

Example 7-2. Converting input rows into a Delta Lake table

```
def save_most_recent_user_context(context_to_save: DataFrame, batch_number: int):
 deduplicated_context = context_to_save.dropDuplicates(['user_id']).alias('new')


 current_table = DeltaTable.forPath(spark_session, get_delta_users_table_dir())
 (current_table.alias('current')
  .merge(deduplicated_context, 'current.user_id = new.user_id')
  .whenMatchedUpdateAll().whenNotMatchedInsertAll()
  .execute()
 )
```

After these preparation steps, it's time to talk about data removal. In the Delta Lake table, it relies on the delete action (see Example 7-3).

Example 7-3. Data removal from a Delta Lake table

```
user_id_to_delete = '140665101097856_0316986e-9e7c-448f-9aac-5727dde96537'

users_table = DeltaTable.forPath(spark_session, get_delta_users_table_dir())

users_table.delete(f'user_id = "{user_id_to_delete}"')
```

This action requires running an additional VACUUM operation to remove the files from the deleted user that exceeded the retention period. Otherwise, you will still be able to retrieve the user's data by reading an older version of the table.

When it comes to Apache Kafka, the cleaning part is a bit different as it relies on a so-called *tombstone message*, which is a special marker record for a deleted row. A tombstone message is composed of a key of the removed record and a null value. If you send it to a topic with a compaction cleanup policy (cleanup.policy), Apache Kafka will run a background compaction process that will delete all tombstone messages. One way to generate a tombstone message is by using the console producer (see Example 7-4).

Example 7-4. Deletion marker (aka tombstone) message

```
docker exec -ti ...  kafka-console-producer.sh  --bootstrap-server .... \

--topic ... --property parse.key=true --property key.separator=, \

--property null.marker=NULL


140665101097856_0316986e-9e7c-448f-9aac-5727dde96537,NULL
```

After you execute the compaction process, user 140665101097856_0316986e-9e7c-448f-9aac-5727dde96537 shouldn't be in the topic anymore. As you can see, this approach works for our vertically partitioned user_context topic, as there is always one occurrence per key. But it will not work for other types of records, such as user visits, where multiple events share the same visit ID key. Having a compaction process running on that topic would mean eventually exposing only the last visit entry to consumers.

Pattern: In-Place Overwriter

The Vertical Partitioner pattern is great if you start a new project or have enough time and compute resources allocated to migrate existing workloads. However, if you are not in one of these comfortable positions, you may have no other choice than to rely on the tried and true overwriting strategy.

Problem

You inherited a legacy system in which terabytes of data are stored in time-based horizontal partitions. There is no personal data management strategy defined. Despite this legacy character, the project is still widely used within your organization, and because of that, it needs to comply with new privacy regulations from your government. One of them requires personal data removal upon user request.

Solution

The current architecture and lack of resources for refactoring leave you with no choice but to apply the In-Place Overwriter pattern.

The implementation heavily depends on the data storage technology. If your data store natively supports in-place operations, the implementation consists of running a DELETE statement that targets entities to remove via a WHERE condition.

Eventually, you may need to complete this deletion query with a data cleaning operation. This applies especially to open-table file formats such as Apache Iceberg and Delta Lake, which often provide a time travel feature to restore the dataset to a prior version in case of an erroneous action. If you delete personal data but don't reclaim the data blocks storing the removed rows, the personal data will still be there.

Deletion Vectors

There are two approaches to managing deletes in table file formats. The first simply identifies removed rows and writes them to a smaller *side file* to reduce the writing footprint; this is commonly known as a *deletion vector*. With that approach, the consumer has to remove deleted rows at reading time. An opposite approach is more writer heavy as it writes all but removed entries to the files so that the consumer can directly use them.

If your storage doesn't provide this native deletion capacity—for example, when you use raw file formats (JSON and CSV)—you have to simulate it. The simulation job will process the whole dataset and filter out all records representing the removed users. It's a compute-intensive operation that replaces the existing data with the filtered dataset.

To avoid any side effects, you shouldn't replace the dataset directly, as the job can retry or even fail, causing data loss. But that doesn't mean you should overwrite each file individually, as this will be both time- and compute-intensive. As an alternative, you can rely on a job writing data to an internal storage area, also known as a *staging area*, where the results are kept private as long as the job doesn't complete. Only once the data removal succeeds can you run a data promotion job that will overwrite the existing public dataset. Figure 7-3 shows this workflow.

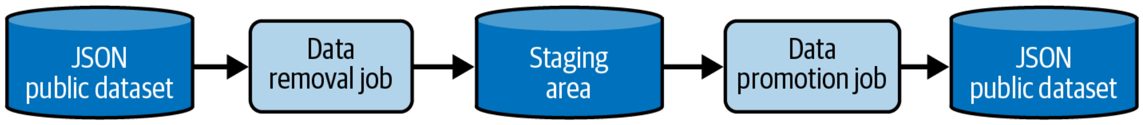

Figure 7-3. Data removal with an intermediary (aka staging) storage

Although this implementation adds some extra complexity with intermediary storage, it avoids the data quality issues that might be introduced by a retried data removal job. Of course, the data promotion job can still fail, but the failure will not lead to data loss since the dataset is already fully computed in the staging area.

Finally, overwriting can also work for compactable data stores, such as an Apache Kafka topic with the compaction strategy enabled and written key-based records. This implementation follows these steps:

1. Read the records stored in the topic.

2. If a record belongs to the removed entity, use the record key and send it with a null payload to the topic.

That way, the compaction process, since it always keeps the most recent entry for each key, will remove previous records with personal data and store empty delete markers, exactly like you saw in the Examples section of “Pattern: Vertical Partitioner”. However, the implementation requires key-based records and a compaction strategy defined in the topic.

Retention Period

If your data retention period is shorter than the delay for taking data removal action, you might consider it as a data removal compaction strategy. After all, the data will be automatically removed within the legal delay period. However, this is just an implementation suggestion, and before considering it as a final solution, please double-check with your Chief Data Officer (CDO) and legal department.

Consequences

At this point, you can probably already see the issue: the pattern performs a lot of read and write operations. Both have consequences for the system.

I/O overhead

Reading and overwriting files incurs serious I/O overhead. Over time, the storage space can nearly double in size and lead to increased throughput.

This overhead will be smaller if your data storage layer can avoid reading irrelevant files for the filter condition. That’s the case with Apache Parquet and the table file formats that rely on it, such as Delta Lake and Apache Iceberg. Apache Parquet, besides storing the records in data blocks, has a metadata layer where it saves statistics about the rows from each data block. If the query engine analyzes these statistics and sees there are no rows matching the removed user or users, it can skip the more costly data block reading operation.

Cost

As the pattern requires reading all the data, it’s a more costly solution than the Vertical Partitioner pattern. Let’s take a look at an example. If you have 2,000 records for one removed entity, with the Vertical Partitioner, you will need to read and drop only one entry, while with the In-Place Overwriter pattern, 2,000 records will be impacted.

To mitigate this impact, you might try to group data removal requests and instead of running one pipeline for each demand, execute one for all the requests.

Examples

To help us analyze the complexity of the In-Place Overwriter pattern, let’s see how to delete a row in the Delta Lake and JSON file formats with Apache Spark. In fact, the Delta Lake example is just a formality as it uses the same code as in Example 7-3, so instead of repeating it, I’ll show you why overwriting data in flat file formats like JSON is more challenging. Let’s take a look at the Apache Spark job that removes a user, shown in Example 7-5.

Example 7-5. Removing rows in a flat file format with PySpark

```
input_raw_data = spark_session.read.text(get_input_table_dir())
```

```
df_w_user_column = input_raw_data.withColumn(
  'user', F.from_json('value', 'user_id STRING')
)

user_id = '139621130423168_029fba78-15dc-4944-9f65-00636566f75b'
to_save = df_w_user_column.filter(f'user.user_id != "{user_id}"').select('value')
to_save.write.mode('overwrite').format('text').save(get_staging_table_dir())
```

There are two important things to keep in mind here. First, we don't want to alter the dataset, hence our use of the simplest writing API possible, which is the text one. Second, to save some space, we don't extract all attributes but only the one(s) used in the filter (user_id in our case).

There is one remaining detail: the writer generates a new filtered dataset in a staging location. This is a temporary storage area where you can keep your data private before exposing it to end users. As Apache Spark is a distributed and transactionless processing layer, we can't simply overwrite the files in the final location as this may leave a partially valid result in cases of failure. For that reason, the writer first generates a dataset in this staging location and only later promotes it to the final output with the rename-like[1] command that's adapted to your storage layer. Example 7-6 demonstrates this for moving files on S3.

Example 7-6. Dataset promotion from staging area to final location with AWS CLI

```
aws s3 rm ${BUCKET}/output --recursive
aws s3 mv ${BUCKET}/staging ${BUCKET}/output --recursive
```

Of course, the copy can also fail, but in that case, you still have the new dataset ready to be copied in the staging location.

Impossible Rollback

The staging-based approach works for our scenario, but it's not perfect either. Let's imagine that your data removal job has some bugs and you need to replay it. Unfortunately, you won't be able to use the original dataset because it was overwritten. To mitigate this issue, you can rely on the Proxy pattern or enable data versioning at your infrastructure level.[2]

Access Control

However, even the most efficient data removal pattern will not be enough to provide the basic security that should come with access control. Indeed, besides being compliant, you also want to let only authorized users access the most critical sections of your data. Keeping personal data private is crucial, but the data itself can be the biggest competitive asset in your possession.

Pattern: Fine-Grained Accessor for Tables

The first pattern fits perfectly into the classical analytical world, where you create users or groups and assign them permissions to access particular tables. As it turns out, it's possible to have finer access control than that.

Problem

After migrating your previous HDFS/Hive workloads to a cloud data warehouse, you need to implement a secure access policy. The first part of the requirement is relatively easy to fulfill as the new data warehouse supports classical users and the creation of groups to manage access to the tables. However, there is an extra demand from your stakeholders. Users, despite their authorization to access a given table, may not have the permissions to read all column and rows. You need to implement an authorization mechanism for these low-level resources as well.

Solution

The Fine-Grained Accessor for Tables pattern solves the low-level data access issue.

When it comes to column-based access, the easiest implementation relies on the GRANT operator, which lets you define the columns within the authorized action scope. It's supported in Amazon Redshift and PostgreSQL. Example 7-7 shows an authorization policy to read only two columns (col_A and col_B).

Example 7-7. Granting access on two columns with the GRANT **operator**

```
GRANT SELECT(col_A, col_B) ON my_table TO some_user;
```

The second implementation for column-level access relies on a data catalog and its tags set to the columns. That's how the pattern works for GCP BigQuery. The implementation starts by creating policy tags in Data Catalog and assigning them to the protected columns in a BigQuery table. The next step is to authorize users to access protected tables by assigning them a Fine-Grained Reader role for each policy tag.

Finally, there is a third implementation based on the data masking feature. Here, users can see protected columns, but their content will be hidden if users don't have access to it. That's how Databricks with Unity Catalog and Snowflake implement column-level access policies. Example 7-8 shows how to achieve this with a column-masking function that shows the column content only if the user is part of a group of engineers.

Example 7-8. Example of a column-masking function in Databricks where only members of a group of engineers can see the ip column

```
CREATE FUNCTION ip_mask(ip STRING)
  RETURN CASE WHEN is_member('engineers') THEN ip ELSE '.' END;


CREATE TABLE visits (
  visit_id STRING,
  ip STRING MASK ip_mask);
```

Column-level security does not provide complete fine-level access protection, but you can make it complete with row-level security. The row-level approach often relies on somewhat dynamic functions that add a WHERE condition to the executed query on the fly. They may have a different name, though. Databricks calls them ROW FILTER, Amazon Redshift calls them Row-Level Security, while GCP BigQuery and Snowflake classify them as row access policies. This implementation consists of defining a separate database object that will add some dynamic condition to all select requests made on a protected table.

On the other hand, if the database doesn't provide native support for row-level access authorization, you can simulate it by exposing a table from a view with an access guard condition (see Example 7-9).

Example 7-9. Row-level access with a view

```
CREATE VIEW users_blogs AS

SELECT ... FROM blogs WHERE table.blog_author = current_user
```

The created view returns the blogs owned by the user issuing the query. The condition will then be different for each user, and consequently, each user will have a dedicated version of the view.

Consequences

This feature is natively supported in databases, so its number of drawbacks is relatively small compared to those of the other patterns you've seen so far.

Row-level security limits

Most of the row-level security implementations have a limited scope of applications that consist of the attributes that you can get directly from the connection session, such as user name, user group, and IP.

Data type

If your column is of a complex type, such as a nested structure, you may not be able to apply the simple column-based access strategy. Column-based permissions, as the name indicates, refer to column, so to apply them to initially nested attributes, you'll need to unnest them first and expose them from another table or use the Dataset Materializer pattern if the latter supports fine-grained permissions.

Query overhead

As you saw in the Solution section, row-level and column-level security protections may be expressed as SQL functions dynamically added to the queries executed against the secured table. If this overhead causes unexpected latency, you can try to mitigate it by creating a dedicated table or view with the Dataset Materializer pattern, only with the data the user or group is allowed to read. Of course, this mitigation also has some limitations, such as data duplication or governance for many access groups.

Examples

PostgreSQL is a great open source implementation supporting both column- and row-level access policies. Let's begin with column-based access, which relies on the GRANT statement that's used to define the scope of access of a user or group. Example 7-10 demonstrates this.

Example 7-10. GRANT **access to a subset of columns in PostgreSQL**

```
GRANT SELECT(id, login, registered_datetime) ON dedp.users TO user_a;
```

After that, whenever user_a issues a query that includes the unlisted columns, such as SELECT *, the database will return an error like "ERROR: permission denied for table users."

When it comes to row-level controls, PostgreSQL uses policies that add conditions whenever someone queries a protected table. Example 7-11 shows a policy that adds the login = current_user condition to the reading query.

Example 7-11. Row-level access policy in PostgreSQL

```
ALTER TABLE dedp.users ENABLE ROW LEVEL SECURITY;


CREATE POLICY user_row_access ON dedp.users USING (login = current_user);
```

Besides PostgreSQL, other data stores, including managed cloud services, also support row-level access. That's the case with AWS DynamoDB, which is a NoSQL database with key-based access. If you want to simulate the row access policy based on the user login, you can do this with an identity and access management (IAM) policy (see Example 7-12).

Example 7-12. Fine-grained access policy in AWS DynamoDB

```
{
 "Statement":[{
  "Sid": ...,
  "Effect":"Allow",
  "Action":[...],
  "Resource":["arn:aws:dynamodb:us-west-1:123456789012:table/users"],
  "Condition":{
   "ForAllValues:StringEquals":{
    "dynamodb:LeadingKeys":["${www.amazon.com:user_id}"]
   }
  }
 }]
}
```

The condition in DynamoDB relies on the dynamodb:LeadingKeys attribute, which allows each user to read only the rows starting with their user_id value (${www.amazon.com:user_id}).

Pattern: Fine-Grained Accessor for Resources

The access-based pattern is great when it comes to table-based datasets. However, databases are not the only data stores used by data engineers. A lot of data engineers, probably including you, also work with other data stores that are often fully managed by their cloud provider. The next pattern applies to these cloud-based resources.

Problem

A security audit detected overly broad permissions in your cloud account. One of the spotted dangers is the possibility that one data processing job could overwrite all datasets that are available in your object store. The auditor presented you with a security best practice called *at-least privilege*, which assigns the least required permissions for each component of your system, so that a data processing job can only manipulate the dataset it's really working on.

You're now looking for a technical solution to implement the at-least privilege on your cloud provider.

Solution

The good news for you is that all major cloud providers—including AWS, Azure, and GCP—come with an implementation for the at-least privilege principle, which is the backbone of the Fine-Grained Accessor for Resources pattern.

Cloud providers have two different strategies when it comes to limiting access to resources. The first approach is resource based because it defines the access scope at the resource level directly. For example, to control access to a GCS bucket on GCP, you'll need to assign the IAM policy from Example 7-13.

Example 7-13. IAM policy for GCS in Terraform

```
data "google_iam_policy" "admin_access" {

 binding {

  role = "roles/storage.admin"

  members = [

   "user:admingcs@waitingforcode.com",

  ]

 }

}


resource "google_storage_bucket_iam_policy" "policy" {

 bucket = google_storage_bucket.default.name

 policy_data = data.google_iam_policy.admin_access.policy_data

}
```

The second approach uses identities instead of resources. Here, the access permissions are defined directly at the identity level (for example, a human or an application user). AWS supports this mode with IAM roles that are assumed by the services. Example 7-14 shows how an Apache Spark EMR job on AWS gives read and write permissions to Kinesis Data Streams.

Example 7-14. Read and write access to AWS Kinesis Data Streams with Terraform

```
data "aws_iam_policy_document" "emr_assume_role" {
 statement {
  effect = "Allow"
  principals {
   type      = "Service"
   identifiers = ["elasticmapreduce.amazonaws.com"]
  }
  actions = ["sts:AssumeRole"]
 }
}
resource "aws_iam_role" "job_role" {
 name           = "visits-processor-role"
 assume_role_policy = data.aws_iam_policy_document.emr_assume_role.json
}
resource "aws_iam_policy" "visits_read_writer_policy" {
 name      = "visits_rw"
 policy = jsonencode({
   Version = "2012-10-17"
   Statement = [{
   Action = ["kinesis:Get*", "kinesis:Describe*", "kinesis:List*",
         "kinesis:Put*"]
   Effect  = "Allow"
   Resource = ["arn:aws:kinesis:us-east-1:1234567890:streams/visits"]}]
 })
}
resource "aws_iam_role_policy_attachment" "policy_attachment" {
 role      = aws_iam_role.job_role.name
 policy_arn = aws_iam_policy.visits_read_writer_policy.arn
}
```

Figure 7-4 summarizes both approaches. As you can see, for the resource-based approach, the access controls are directly attached to each resource, while for the identity-based approach, they're located at the user level.

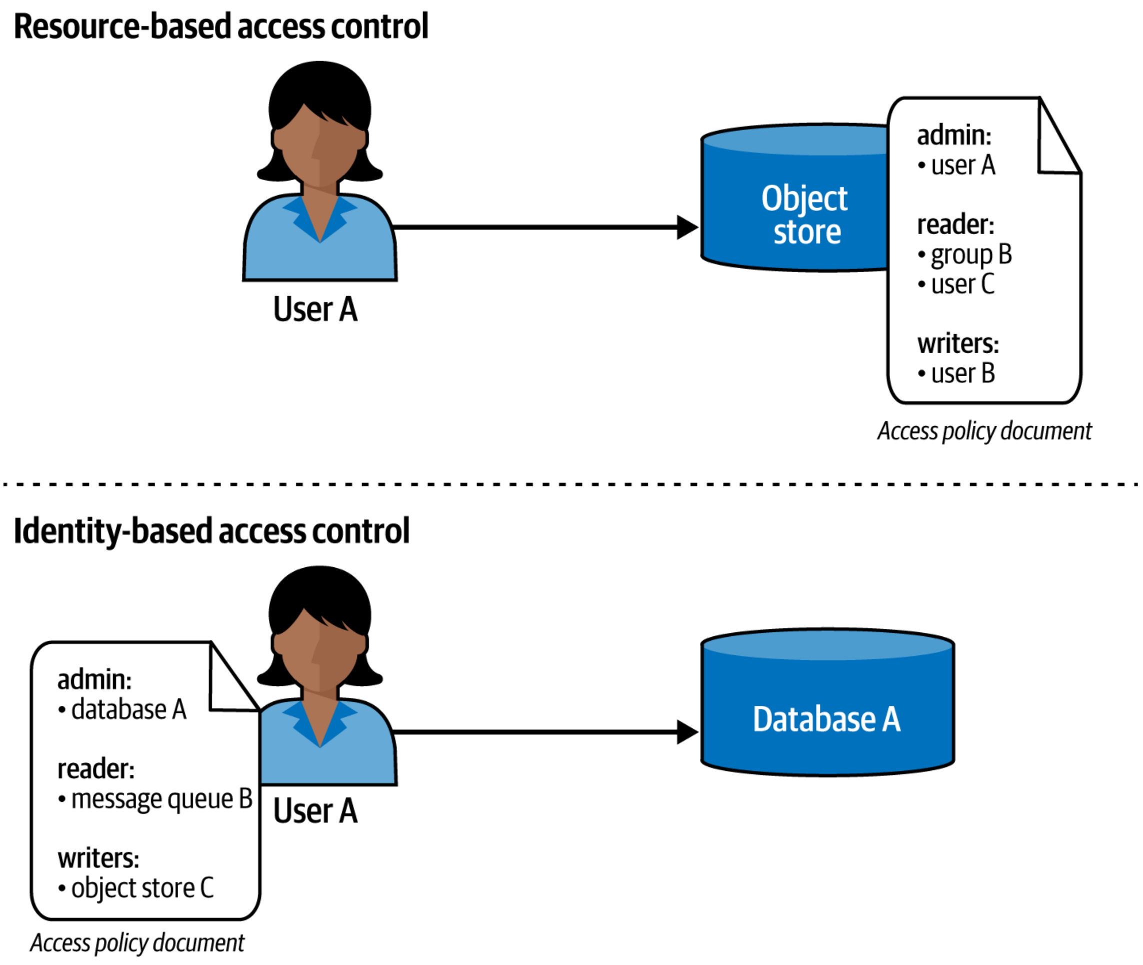

Figure 7-4. Simplified illustration of resource-based and identity-based access controls

Fine-grained permissions are often flexible. They can target a specific resource, a set of resources starting with the same prefix, or even a resource based on runtime conditions (as you saw with user ID in Example 7-12). The last implementation looks like the row-level access from the previous pattern because it controls access at the lowest levels of the infrastructure. Example 7-15 shows an example of a tag-based access control for AWS S3 where you rely on custom metadata attributes (tags) that are associated with each resource individually.

Example 7-15. Tag-based access policy for AWS S3

```
"Statement": [{
 "Effect": "Allow", "Action": "s3:PutObject",
 "Resource": "*", "Condition": {
```

```
 "ForAllValues:StringEquals": {"aws:TagKeys": ["${www.amazon.com:user_id}"]}}
}]
```

Consequences

Defining access policies shouldn't be difficult from a technical standpoint as you can rely on an infrastructure as code (IaC) tool or a custom script. That doesn't mean there are no consequences, though.

Security by the book trade-off

A security rule of thumb is the at-least privilege principle, which holds that a user or group should have access only to the resources it needs at the moment. Although this is a great principle, it can lead to the creation of many small access policy definitions that will be difficult to maintain in complex environments.

To mitigate this issue and keep resources down to a manageable size, you might use wildcard-based access, so instead of defining each of your cloud resources individually, you could use a prefix like *visits**. As a result, the access policy would apply to all resources starting with "visits." However, this may violate the at-least privilege principle because you can't guarantee that the user should have access to all visits-prefixed resources created in the future. For that reason, you should discuss this simplification strategy with your security department.

Complexity

If you use both resource- and identity-based approaches in the same project, you may be inadvertently increasing the complexity of your system. Whenever possible, it's better to prefer one solution, preferably the one that covers more of your use cases.

Quotas

As for any other cloud resource, even access policies have limitations. For example, the AWS IAM service allows 1,500 custom policies by default, and GCP IAM has a limit on custom roles within a project (300). Some limits are flexible, though, so that you can ask your cloud provider to increase them.

Examples

To help you understand the Fine-Grained Accessor for Resources pattern, let's see how it behaves in the context of reading data from an S3 bucket. Assuming you don't have access to the tested bucket, you should get the permission error from Example 7-16.

Example 7-16. Exception from a reading operation without permissions

```
$ aws s3 ls s3://dedp-visits-301JQN/


An error occurred (AccessDenied) when calling the

ListObjectsV2 operation: Access Denied
```

As you can see, the error message explicitly says what the issue is. To start, let's fix it with the identity-based approach. To do so, we need to create a role with the S3 reading actions scoped to our bucket (see Example 7-17).

Example 7-17. Permissions required to read an S3 bucket

```
{"Version": "2012-10-17", "Statement": [{"Sid": "VisitsS3Reader",
 "Effect": "Allow", "Action": ["s3:Get*", "s3:List*"],
 "Resource": ["arn:aws:s3:::dedp-visits-301JQN/*",
   "arn:aws:s3:::dedp-visits-301JQN"]
}]}
```

After creating this role and assigning it to your user, you should see the listing operation succeed.

As an alternative, you can use the resource-based approach. Example 7-18 shows a command that authorizes a visits-s3-reader user to read data from our S3 bucket.

Example 7-18. A command to authorize an AWS role to read objects from an S3 bucket

```
$ aws s3api put-bucket-policy --bucket dedp-visits-301JQN --policy file://policy.json


# policy.json
{"Statement": [{"Effect": "Allow",
  "Principal": {"AWS": "arn:aws:iam::123456789012:user/visits-s3-reader"},
  "Action": ["s3:Get*", "s3:List*"],
  "Resource": "arn:aws:s3:::dedp-visits-301JQN/*"
}]}
```

Data Protection

You might be thinking that controlling data access at the logical level, meaning either the database or the cloud services scope, is enough to build a fully secure system. Unfortunately, it's only part of the solution. A missing part concerns securing data itself, hence protecting it against unexpected usage.

Pattern: Encryptor

Even though you run your infrastructure on the cloud, the data is still physically stored somewhere, and unauthorized people may try to read it. If you need to reduce the access risk, you can first implement the access policies from the previous section, and then make sure the data will be unusable if access controls get compromised.

Problem

After you implement fine-grained access policies for both your tables and your cloud resources, you're tasked with enforcing the security of your data at rest and in transit. Your stakeholders are worried that an unauthorized person could intercept the data transferred between your streaming brokers and jobs, or that the same person could physically steal your data from the servers.

Solution

One way to reduce this data intrusion risk is with the Encryptor pattern. As there are two protection levels required, the pattern has two implementations.

The first implementation is for the data at rest. It encrypts the stored data with a client-side or server-side approach, and the difference between the two approaches boils down to encryption key management. In the client-side implementation, the data producer is responsible for encrypting the data prior to sending it to storage. Naturally, it’s also responsible for managing the encryption key.

When it comes to server-side encryption, all the encryption and decryption work is done on the server side. A consumer or producer only issues a request, and the server does the rest, including encryption key management.

Server-side encryption, while it looks challenging, is widely supported by public cloud providers. Each of the major providers has its own encryption keys store that you can freely apply to all offered services supporting server-side encryption. On AWS and GCP, you’ll use the *Key Management Service* (KMS), while on Azure you’ll use the *Key Vault* service.

Figure 7-5 depicts the encryption workflow based on these services.

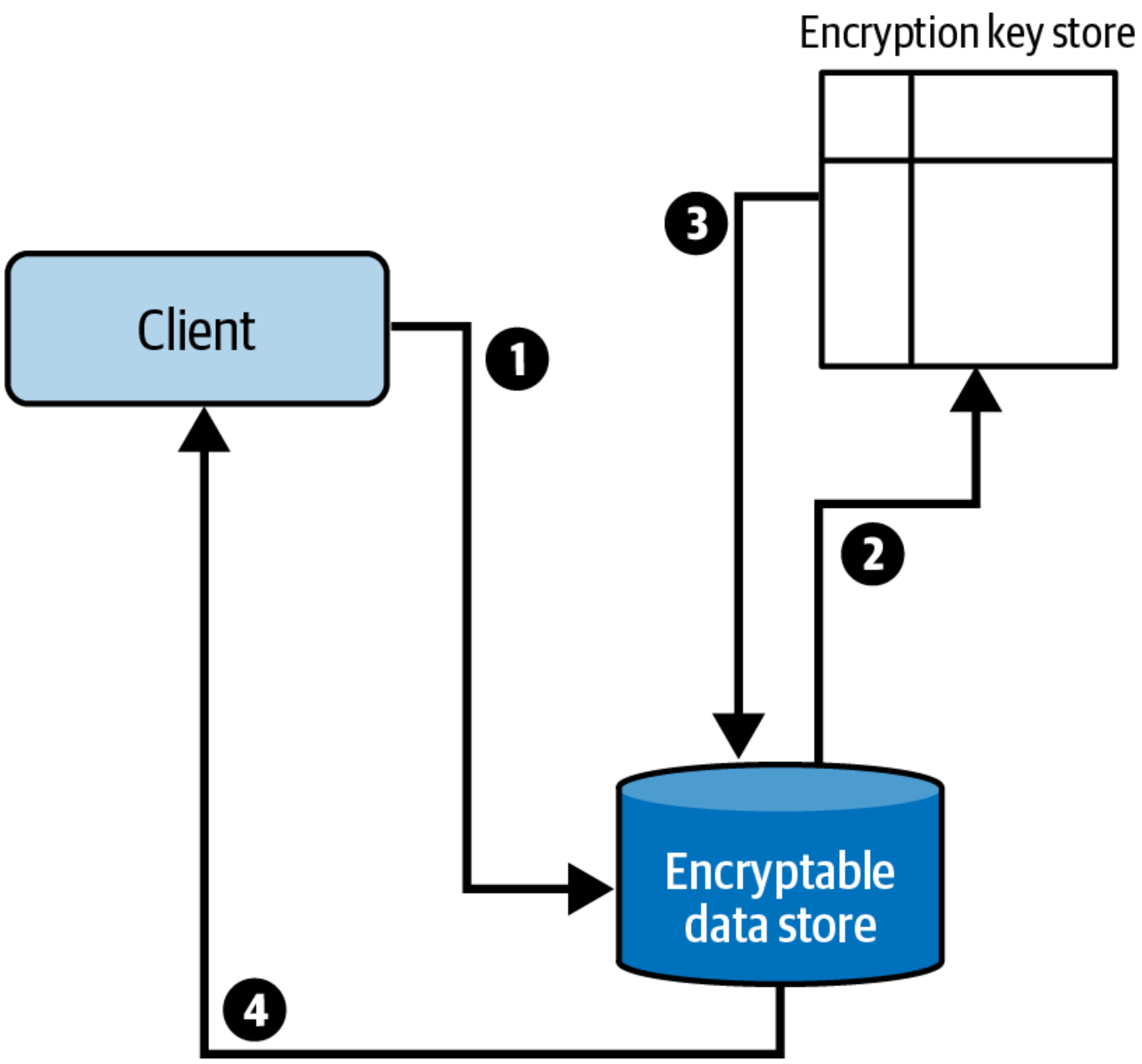

Figure 7-5. Encryption workflow for data-at-rest server-side encryption with an encryption store service

The interaction with an encrypted data store is based on the four main steps that are numbered in the diagram:

1. The request first reaches the encrypted data store. However, it doesn't return directly.
2. Instead of returning the data, the store identifies the encryption key and asks the encryption key store for the decryption key. If the client is not authorized to decrypt the data, the request will fail. Otherwise, it continues.
3. Next, the data store decrypts the data with the retrieved decryption key.
4. Finally, the decrypted records are sent back to the client.

If you are a cloud user, this complex exchange is fully abstracted to you by the cloud provider. Your only responsibility is to configure an appropriate encryption strategy for the data store and manage access to the data store and encryption key store for all reading and writing clients.

But data at rest is not the only encryption layer. Another layer is data in transit, meaning the place where clients and data stores exchange data across the network. The implementation on the cloud is relatively easy as it's limited to enabling secure communication at the SDK level on the client side and configuring the required protocol version on the service. That way, the exchanged data will remain encrypted.

Consequences

Encryption is a data access security strategy applied to the physical data storage level. Like other security protections, you can't get it for free.

Encryption/decryption overhead

CPU overhead is probably the first consequence that comes to mind. As you've seen, the data is not stored in plain text but in a somehow altered, unreadable version. Without an encryption and decryption action, it's unusable. For that reason, each writing and reading request puts some extra pressure on the CPU.

Data loss risk

Even though the Encryptor pattern tends to protect data at rest against unauthorized access, it can also, as a side effect, block access for authorized users. That said, this can happen only if you either lose the encryption key or simply lose access to it.

To mitigate the issue, cloud providers often implement *soft deletes* on the encryption stores. That means any delete request you send will not take immediate effect. Instead, it will benefit from a grace period during which you can restore the key in case of an accidental deletion.

Protocol updates

Encryption in transit is much easier to set up than encryption at rest. However, it's still an extra component in your system that must be kept up-to-date. That's the case with the Transport Layer Security (TLS) protocol, which is a possible encryption channel in HTTPS. Over the years, there have been various security issues detected that led to the deprecation of the 1.0 and 1.1 versions of TLS.[3] For that reason, any services using these old releases must be upgraded.

This operation looks like a difficult extra maintenance task, but with cloud offerings, it is very often simplified and reduced to upgrading the new encryption protocol version at the service level.

Examples

Now, let's see encryption at rest in action for the AWS S3 object store with some Terraform code. The encryption configuration has two main implementation steps, and the first one is in Example 7-19. It defines the encryption key in the AWS KMS service and grants an AWS Lambda user decryption and encryption permissions.

Example 7-19. A Terraform definition for AWS KMS encryption

```
module "kms" {
 source    = "terraform-aws-modules/kms/aws"
 key_usage = "ENCRYPT_DECRYPT"
 deletion_window_in_days = 14
 aliases = ["visits-bucket-encryption-key"]
 grants = {
  lambda_doc_convert = {
   grantee_principal = aws_iam_role.iam_key_reader.arn
   operations       = ["Encrypt", "Decrypt", "GenerateDataKey"]
  }
 }
}
```

An AWS KMS key declaration is pretty self-explanatory. The only mysterious but essential parameter is the deletion_window_in_days. Do you remember when I mentioned the data loss drawback? Currently, this property is there to ensure an encryption key restore window in case of an erroneous removal. Also, the declaration assigns some grants to other services. In our example, the grant applies to an IAM role that you can later associate with a compute or a querying service.

After defining the encryption key, the next step is to combine it with an S3 bucket (see Example 7-20).

Example 7-20. An S3 bucket encryption at-rest configuration

```
resource "aws_s3_bucket_server_side_encryption_configuration" "visits" {
 bucket = aws_s3_bucket.visits.id
 rule {
  apply_server_side_encryption_by_default {
   kms_master_key_id = module.kms.key_arn
   sse_algorithm    = "aws:kms"
  }
 }
}
```

To sum up the two previous snippets, we could say that the encryption is just a matter of associating the key with the encrypted resource and all authorized identities.

Additionally, you can enforce the encryption in motion for some data stores and ensure that any client applications using older versions will be ignored. For the Azure Event Hubs streaming broker, you can define a minimum TLS version (see the last line of Example 7-21).

Example 7-21. Minimal TLS encryption version for Azure Event Hubs

```
resource "azurerm_eventhub_namespace" "visits" {
  name            = "visits-namespace"
  location         = azurerm_resource_group.dedp.location
  resource_group_name = azurerm_resource_group.dedp.name
  sku           = "Standard"
  capacity        = 2
  minimum_tls_version = "1.2"
}
```

Pattern: Anonymizer

As you saw in Chapter 6, you can improve the value of a dataset if you share it with other pipelines. However, it's not always that simple. If your dataset contains PII attributes and the user hasn't agreed to share those details with your partners, you will need to perform a special preparation step before sharing the dataset.

Problem

Your organization contracted an external data analytics company to analyze your customers' behavior and optimize your communication strategy. Since the dataset contains a lot of PII attributes and some of your users didn't agree to share them with third parties, your data engineering team was tasked to write a pipeline to make the shared dataset compliant with the privacy regulations.

Not Only PII

The examples refer to PII data as it's probably the most commonly discussed use case, but it's not the only data type requiring protection. Other examples are protected health information (PHI) and intellectual property (IP) data. In the following pages, we're going to stick to PII for the sake of simplicity.

Solution

The problem states that some parts of your dataset cannot be shared. Put differently, you need to remove or transform them. That's the perfect task for the Anonymizer pattern.

The goal of this pattern is to remove the sensitive data from the dataset, thereby transforming each row into anonymous information. Thanks to this process, the data consumer won't be able to identify the user.

The Anonymizer pattern supports various implementations:

1. Data removal
2. Data perturbation
3. Sythentic data replacement

All of them consist of removing or altering the initial sensitive attribute. How? Let's quickly look at each of the methods:

1. Data removal is the easiest one to implement as it takes the selected columns out of the input dataset.
2. Data perturbation is more complex. It adds some noise to the input value so that the value has a different meaning. For example, in an IP column, you could add extra attributes in random positions, so that “123.456.789.012” becomes “1823.456.7809.012”.
3. Synthetic data replacement substitutes the original values with the values coming from the synthetic data generator. What does that mean? It means that there is a smart model (probably a machine learning model) that is capable of interpreting the type of input column and generating a corresponding replacement. The replacement looks the same as the original attribute but has a different value. For example, in a country column, “Portugal” could be synthetically replaced with “Croatia.”

The easiest methods to implement are the first two. You can use either a mapping function or a column transformation to remove the value or to perturb it. The synthetic data method may involve some work with the data science team to build a model capable of analyzing and generating replacement values. In a less ideal version, you could eventually implement the solution on your own by creating a replacement function for each column that generates some random values.

Consequences

The Anonymizer pattern does indeed protect your sensitive data, but it greatly impacts the data’s usability.

Information loss

It’s clear that when you remove or replace information, your dataset becomes something new. This means your end users, including technical users like data analysts and data scientists, won’t be able to rely on these sensitive columns in their work. That can lead to many issues, including false data prediction models and incorrect data insights.

Examples

Let’s see how to anonymize a dataset by removing a birthday column and replacing an email address with a lower-quality version of the synthetic data. The pipeline leverages the Apache Spark API for the former task and the Faker Python library for the latter.

To start, you can remove a column two different ways. First, you can omit it from the list of columns to read in the SELECT statement. This approach is good if you don’t have a lot of columns to read. If that’s not the case, you can take the opposite approach and remove the columns with the drop function.

Next, to replace a value, you can use a column-based function such as withColumn or a row-based function like mapInPandas. Since our example only involves replacing one column, we’re going to use the former approach. Example 7-22 shows both the removal approach and the replacement approach. The replacement scenario uses the Faker library,[4] which can, among other things, generate random email addresses.

Example 7-22. Dropping a column and replacing its value with an anonymization example in PySpark

```
@pandas_udf(StringType())
def replace_email(emails: pandas.Series) -> pandas.Series:
  faker_generator = Faker()
  return emails.apply(lambda email: faker_generator.email())


users.drop('birthday').withColumn('email', replace_email(users.email))
```

As a result, the output dataset doesn't contain the birthday column and has the email value replaced by a randomly generated email address.

Pattern: Pseudo-Anonymizer

The "Pattern: Anonymizer" offers strong data protection. However, the pattern's impact on data science and data analytics pipelines can be very bad because of missing or altered values. The Pseudo-Anonymizer pattern reduces this impact.

Problem

An anonymized dataset you shared with the external data analysts' company doesn't contain all the columns. You chose to remove them because that was the simplest data anonymization strategy. However, because of that, the team can't answer most of the business queries. The team members asked you to provide them with a new dataset that still hides real PII values but replaces them with a more usable form.

Solution

Data sharing, PII data, and the requirement to keep some business meaning are perfect conditions in which to apply the Pseudo-Anonymizer pattern. Depending on your context, you can use one of its four implementations:

1. Data masking
2. Data tokenization
3. Hashing
4. Encryption

All of them consist of replacing the initial data with a different value that is more or less related to the original. How? Let's quickly look at each of the methods:

1. Data masking replaces sensitive data with meaningless characters or more realistic substitution values. For example, a Social Security number (SSN) like 999-55-1040 could be masked as XXX-XX-1040 or 9XX-5X-1XXX. As you can see, if you apply this masking strategy, several different users may share the same masked SSN.
2. Data tokenization substitutes initial values with fictive ones. However, it stores the mapping between the original values and their substitutes in a token vault store. The key here is to secure access to the vault to avoid compromising the protected

attributes. If the vault access security is compromised, an unauthorized person could access and thus reverse the tokenized values.

3. Hashing fully and irreversibly replaces the sensitive values. For example, an email like *contact@waitingforcode.com* could become a string like *gD0B+pUpXYVZ9nqhgLRuban0CilZRKVp4dcmvmocsYE=*, if it was hashed with an SHA-256 algorithm and encoded with a Base64 scheme.

4. Encryption relies on the encryption keys that are applied to columns or rows, a little bit like the keys for the datasets in the Encryptor pattern. Here, a user with access to the encryption key should be able to restore the original value.

Once you've identified the method that fits best with the data types of your dataset, you can proceed to implement the anonymization functions. Some of them, such as data masking, can be represented as easy column transformation functions. The others, such as tokenization that requires an extra mapping table, might need additional implementation effort.

Anonymization Versus Pseudo-Anonymization

The pseudo-anonymization techniques included in the currently described pattern are sometimes presented as part of the anonymization process. However, there is an important difference between them. Pseudo-anonymized PII data can become identifiable if it's combined with other datasets. That's not the case with anonymized datasets, where even in cases of combinations, the data will remain unidentifiable.

Consequences

The Pseudo-Anonymizer pattern, despite the fact that it protects personal data, doesn't provide as strong a protection guarantee as the Anonymizer pattern.

False sense of security

The Pseudo-Anonymizer, even though it blurs personal data, provides a weaker security guarantee than the pattern described in the Anonymizer pattern. One of the biggest issues comes from the combination of datasets, in which a pseudo-anonymized column can become a PII column identifying a person.

Let's take a look at an example of an imaginary but very popular data processing framework called Cheetach Processor, which was invented by a data engineer named John Doe who lived in San Marino, a small country in Europe. Your database has two pseudo-anonymized tables. The first, Table 7-1, stores user food preferences, and the second, Table 7-2, persists user registration information.

User ID	Liked foods	Disliked foods
1000	carrot, broccoli, potato	chips, chocolate bar

Table 7-1. Food preferences table

User ID	Country	Role
1000	S*n M****o	C******h P*******r i******r

Table 7-2. User registration table

As you can see, Table 7-1 perfectly protects the identity of the user. However, if you combine it with Table 7-2, you'll get something like "C******h P*******r i******r living in S*n M****o likes carrot, broccoli, and potato." There are not a lot of countries whose names match the one in the masked country column. Once you get to San Marino here, you may also discover the "inventor" part and identify that the user ID number 1000 refers to our famous John Doe. I know, this example is pretty abstract, but its obviousness is there to clearly demonstrate the false sense of security pseudo-anonymization gives when combining datasets.

Identification wouldn't be possible with full anonymization because the Country and Role columns would both be removed or altered to, for example, "Europe" and "Software engineer." You can see now that even after combining both tables, it wouldn't be clear who this software engineer is.

Information loss

The best example illustrating this is the data masking strategy. As you saw in our previous example with the SSN, some of the numbers are preserved. Because of that, two different SSNs can now point to the same user. For example, numbers like 999-55-1040 and 999-13-1040 can both be masked in the same format, which is XXX-XX-1040.

Besides this loss, there is also a data type loss. A great example of this is the generalization method, in which a numeric value can be replaced by a numeric range represented by a text type.

Examples

Implementing the Pseudo-Anonymizer pattern relies on using the mapping function. In our example, let's see how to do this with PySpark's mapInPandas and column transformations that will work on the table in Example 7-23.

Example 7-23. Table to pseudo-anonymize

```
+-------+-------+--------------+------+
|user_id|country|           ssn|salary|
+-------+-------+--------------+------+
|      1| Poland|0940-0000-1000| 50000|
|      2| France|0469-0930-1000| 60000|
|      3|the USA|1230-0000-3940| 80000|
|      4|  Spain|8502-1095-9303| 52000|
+-------+-------+--------------+------+
```

All but the first column should be pseudo-anonymized, and we're going to use two different methods. The first one is the mapInPandas function shown in Example 7-24 that replaces the country value with a geographical area and masks the SSN.

Example 7-24. Pseudo-anonymization with generalization and data masking

```
def pseudo_anonymize_users(input_pandas: pandas.DataFrame) -> pandas.DataFrame:
  def pseudo_anonymize_country(country: str) -> str:
    countries_area_mapping = {
    'Poland': 'eu', 'France': 'eu', 'Spain': 'eu', 'the USA': 'na'
    }
    return countries_area_mapping[country]

 def pseudo_anonymize_ssn(ssn: str) -> str:
    return f'{ssn[0]}***-{ssn[5]}***-{ssn[10]}***'
 for rows in input_pandas:
   rows['country'] = rows['country'].apply(lambda c: pseudo_anonymize_country(c))
   rows['ssn'] = rows['ssn'].apply(lambda ssn: pseudo_anonymize_ssn(ssn))
   yield rows
```

The mapInPandas mapping doesn't include the range transformation for the salary column because of the type change. The salary is defined as an integer in the input dataset, while the range is of the string type. As you can see, it's a types-incompatible conversion, and to handle it, we need a simple column-based mapping (see Example 7-25).

Example 7-25. Column-based pseudo-anonymization with type conversion

```
pseud_anonymized_users = (users.mapInPandas(pseudo_anonymize_users, users.schema)
  .withColumn('salary', functions.expr('''
    CASE WHEN salary BETWEEN 0 AND 50000 THEN "0-50000"
        WHEN salary BETWEEN 50000 AND 60000 THEN "50000-60000"
        ELSE "60000+" END''')))
```

After we apply both types of transformations, the input dataset changes to the table in Example 7-26.

Example 7-26. User table after pseudo-anonymization

```
+-------+-------+--------------+-----------+
|user_id|country|           ssn|     salary|
+-------+-------+--------------+-----------+
|      1|     eu|0***-0***-1***|    0-50000|
|      2|     eu|0***-0***-1***|50000-60000|
|      3|     na|1***-0***-3***|     60000+|
|      4|     eu|8***-1***-9***|50000-60000|
+-------+-------+--------------+-----------+
```

Connectivity

So far, you have learned how to protect your data. Although this is the core part of data security, it might not be enough. Do you remember the data flow patterns from the previous chapter? Data is continuously flowing within the same system or across different systems, and you will need to access it. In this section, you'll learn about secure access strategies.

Pattern: Secrets Pointer

The login/password authentication method is still probably the most commonly used to access databases. It's simple but also dangerous if used without precautions, and the pattern presented next is one of the precautions you can apply.

Problem

The visits real-time processing pipeline from our use case leverages an external API to enrich each event with geolocation information. This API is provided to you by an external company, and the only authentication method is a login/password pair.

In the past, your team accidentally shared the login/password used for a different API. As the API was request billed, the leak led to increased billing. You want to reduce this risk right now and avoid storing the login/password for the code interacting with the new data enrichment API.

Solution

Credentials are sensitive parameters. One of the best ways to secure them is to...avoid storing them anywhere. Instead, you can use a reference (aka a *pointer*). That's what the Secrets Pointer pattern does.

The idea here is to leverage a secrets manager service (such as Google Cloud Secret Manager or AWS Secrets Manager) where you store all sensitive values, such as logins, passwords, and API keys. This approach has several advantages. First, it's a central place where you manage this sensitive data. Thanks to this centralization, access monitoring is easier. Second, the component also facilitates management. You can simply set a new set of credentials without having to update all the consumers.

Regarding the consumers, they won't reference the sensitive parameter values anymore. Instead, they'll use their names from the secrets manager service. Consequently, each consumer will retrieve the secret's value by issuing a query to the service at runtime. Additionally, to save some communication costs, consumers can store the credentials in their local cache for some time.

With the Secrets Pointer pattern, access is protected on two levels. At the first level, the consumer must have access to the secrets manager. Otherwise, it won't be able to retrieve the key. To secure this step, you can use one of the fine-grained access patterns. The second level of protection is natively guaranteed by the credentials themselves. If they're not valid, a consumer will simply not have access to the underlying API or database.

Consequences

A popular proverb in our engineering world came from Phil Karlton, who said, "There are only two hard things in computer science: cache invalidation and naming things."[5] The cache part applies to the Secrets Pointer too.

Cache invalidation and streaming jobs

If you cache the credentials, you'll never know whether you're using the most up-to-date ones, which may lead to connection issues. On the other hand, you may potentially avoid many credentials retrieval requests and hence optimize execution time if they're costly. That's a good thing, but if you implement the cache, you must also be able to refresh the credentials.

The simplest approach here, if you don't want to send a credentials refresh request, is to simply allow the job to fail. Normally, once restarted, the new execution version should reload the credentials from the secrets manager. That being said, this might not be optimal if the credentials change very often, as it would increase the number of failures. Also, remember to use one of the idempotency design patterns as they should keep your data correct even with retries.

You can also try to write an asynchronous refresh process, but here too, you may encounter some writing issues if the credentials change after you've started sending data to the output data store and before you refresh the connection parameters.

Logs

The Secrets Pointer pattern gives you a false sense of security that the credentials won't leak. Indeed, they're now stored in a secured place that only authorized entities can access. Even though this part doesn't get compromised, you can still leak the secrets if you inadvertently include them in the logs.

A secret remains secret

Even though consumers don't need to deal with credentials, it doesn't mean there are no credentials at all. In fact, to enable consumers to use references instead of secret values, there is a secrets producer that needs to securely generate secret values to the secrets storage.

In practice, this is either a human administrator who puts the secret values in storage or the IaC stack that defines random secret values while creating a database.

Examples

To see how to integrate the Secrets Pointer pattern into a pipeline, let's take a look at a simple Apache Spark job reading a PostgreSQL table and converting it to JSON files. Normally, the input dataset definition requires several parameters, including the login/password pair. However, as we're going to use the pattern, instead of the clear values, we're going to define the references. Example 7-27 shows this definition.

Example 7-27. A database connection from Apache Spark to PostgreSQL without plain text credentials

```
secretsmanager_client = boto3.client('secretsmanager')

db_user = secretsmanager_client.get_secret_value(SecretId='user')['SecretString']

db_password = secretsmanager_client.get_secret_value(SecretId='pwd')['SecretString']

spark_session.read.option('driver', 'org.postgresql.Driver').jdbc(

  url='jdbc:postgresql:dedp', table='dedp.devices',

  properties={'user': db_user, 'password': db_password})
```

Although the connection configuration still references the user and password, the code doesn't know about the values. Well, technically, it knows them, but only thanks to the secrets manager data store. Put differently, the credentials aren't tied to the codebase but are managed as a separate asset.

Besides keeping the secrets secret, the Secrets Pointer pattern simplifies the work on multiple environments. If you create the database with its connection attributes from a scripted project, like the IaC with Terraform, you can keep the same secret names for all environments and let the IaC automate their generation. That way, your code doesn't need to deal with any per-environment configuration files that would store different connection parameters for each environment.

Pattern: Secretless Connector

The Secrets Pointer pattern shows how to secure credentials, but what if I told you that it's even better to not have any credentials to manage? That is what the next pattern makes possible.

Problem

One of the teams in your organization has started integrating a new data processing service. All the code examples it found use API keys to interact with the cloud-managed resources available from the service.

The team is small and would like to avoid managing these APIs. It called you for the second time to see if you have an alternative solution, ideally guaranteeing access to resources without any kind of credentials to reference in the code.

Solution

If you work with cloud services and don't want to manage the credentials, you can implement the Secretless Connector pattern. How? There are two main approaches.

The first implementation uses the IAM service that's available on your cloud provider. Here, a user or administrator assigns reading and writing actions to each user, group, or role, via document access policy. This IAM-based policy approach applies to application users, such as data processing jobs. They may work on the same cloud resources as you, a human user, but their interaction is automated. For example, their job may be scheduled to run at specific times of day. Therefore, they don't log in but must somehow be authorized to access cloud resources.

Generally, the workflow for IAM-based access for both physical and application users can be summarized as the schema shown in Figure 7-6.

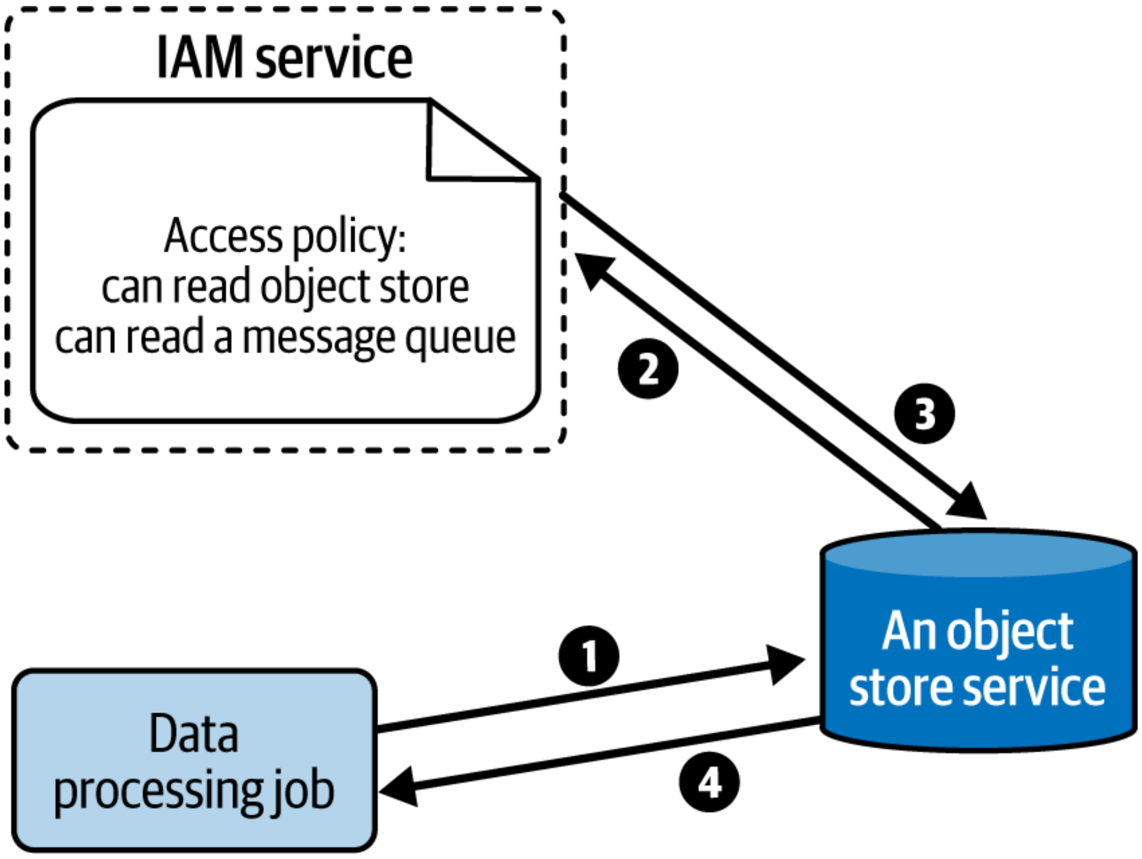

Figure 7-6. A generalized credentialless access workflow with a cloud service

The workflow is based on four main steps:

1. An application user issues a request to interact with a cloud service. In our example, it's an object store and the user wants to read one object.
2. The service doesn't return the object directly. Instead, it connects to the IAM service to validate that the user has all required permissions to fulfill the request.
3. The IAM responds to the service with the list of permissions' scope.
4. The service returns a response to the user. If the user has all required permissions, the response satisfies the request. Otherwise, the service returns an error.

The second implementation of the Secretless Connector uses certificates and is also known as *certificate-based authentication*. The workflow is similar to the one you saw in Figure 7-6, but instead of the IAM service, there is a certificate authority (CA) component. This authority validates the certificates used in the connection process before authorizing the workflow to move on.

Consequences

Despite the -less suffix that might indicate a lack of effort, the Secretless Connector pattern does require some work.

Workless impression

Although there are no credentials involved, there is still some work to do. You have to configure the entity to leverage the credentialless access. For example, on AWS, this requires setting an assume role permission that lets an entity use temporary credentials returned by the Security Token Service (STS) to interact with other services.

Rotation

This point is essentially valid certifcate-based authentication. Rotating access keys on a regular basis is often considered a security best practice to reduce the risk of leaks. However, with certificates, logins, and passwords, it adds extra management overhead.

To rotate them without impacting your consumers, you have to first generate and share new credentials with consumers. Meanwhile, you need to support both old and new credentials on your side, so that consumers who may not migrate at the same time can still use your data store. Only once the consumers confirm that they're using the new credentials can you drop the old ones.

Examples

Let's begin this section with a certificate-based connection to PostgreSQL from an Apache Spark job. As shown in Example 7-28, the connection attributes don't require any password. Instead, they rely on the certificate shared with the server.

Example 7-28. Certificate-based connection to PostgreSQL from Apache Spark

```
input_data = spark.read.option('driver', 'org.postgresql.Driver').jdbc(
  url='jdbc:postgresql:dedp', table='dedp.devices',
  properties={'ssl': 'true', 'sslmode': 'verify-full',
   'user': 'dedp_test', 'sslrootcert': 'dataset/certs/ssl-cert-snakeoil.pem',
})
```

The connection parameters use the verify-full SSL mode that ensures the server host name matches the name stored in the server certificate.

To help you understand the cloud component better, let's change the cloud provider and see what a GCP Dataflow job needs to process objects from a GCS object store. First, we need to create a *Service Account* resource, which is a GCP term for the application user. The setup requires a name and a GCP project where it should apply. Example 7-29 shows this.

Example 7-29. Service Account creation in GCP

```
resource "google_service_account" "visits_job_sa" {
  account_id = "dedp"
  display_name = "Dataflow SA for processing visits from GCS"
}
```

The next step consists of linking this Service Account to the GCS bucket the Dataflow job should process and the job itself. Example 7-30 shows how to create this link with read-only permissions in the role attribute and with the Dataflow job.

Example 7-30. Reading permissions in the visits bucket

```
resource "google_storage_bucket_iam_binding" "visits_access" {
  bucket = "visits"
  role   = "roles/storage.objectViewer"
  members = [
    "serviceAccount:${google_service_account.visits_job_sa.email}",
  ]
}


resource "google_dataflow_job" "visits_aggregator" {# ...
  service_account_email = google_service_account.visits_job_sa.email
}
```

This way, the visits_aggregator job gets an identity, and consequently, it doesn't need any credentials provided at runtime to read the visits bucket.

Summary

In this chapter, you learned about various aspects of identity security. In the first section, you learned how to comply with a data removal request, which is one of the important parts of data regulation policies such as the GDPR and the CPPA. There are two possible solutions. The first one is the Vertical Partitioner, which leverages data layout to perform cheaper delete operations on a reduced number of occurrences. An alternative is the In-Place Overwriter pattern, which changes the data in place for data stores without any prior data organization strategy. It's more expensive, but at the same time, it's more universal than the Vertical Partitioner.

Next, you learned about fine-grained access. In the Fine-Grained Accessor for Tables pattern, you saw how to implement column- and row-level access controls in table-oriented environments, such as data warehouses and lakehouses. However, nowadays, data engineers are not exclusively working on these storage layers. They also interact with cloud services, such as serverless NoSQL data stores. For them, you'll use the Fine-Grained Accessor for Resources pattern.

In the third part came data protection. First, you learned how to protect your data at rest and in motion with the Encryptor pattern. It's a great additional security protection against malicious users who access your encrypted data but can't use it as they don't have the encryption keys. Then, you learned about the Anonymizer and Pseudo-Anonymizer patterns that are great ways to secure the dataset in data sharing scenarios.

Finally, you learned about two connectivity patterns that can help you better secure your data applications. The Secrets Pointer pattern will help you use credentials such as passwords and authorization keys, without keeping them directly in your Git repository. But since the best strategy is to not have any credentials to manage, even outside the repository, there is also the Secretless Connector pattern that you can leverage for interactions without logins and passwords.

Even though, with the data security design patterns we're approaching the end of our journey, three important topics still remain. The first of them is data storage, which, besides helping with data removal requests, also optimizes data access. It'll be the topic of the next chapter.

1 Renames in the cloud object stores don't follow the same transactional semantic as in the local file system. They're often implemented as copy-and-remove operations, which in cases of failure can leave empty valid states.

2 All major object stores that are most often used for storing nontransactional file systems support versioning. These include S3, Azure Storage, and GCS.

3 You can learn more about the deprecation in RFC 8996.

4 You can learn more about Faker from the official repository.

5 Philip Lewis Karlton (1947–1997) was a software developer and one of the people who was responsible for the architecture of Netscape products.