

# EXPLORING DYNAMO DB AND API GATEWAY

## 1. INTRODUCTION

**Relational databases** (which use SQL) are traditional databases where data is structured in tables and each table has a schema that relates columns and data types. Different tables are related to each other using keys. Each table has a unique identifier.

They are very robust but have some limitations, especially concerning data growth, data changes and scalability .

On the other hand, **non-relational databases** (also known as non-SQL), are designed to overcome the problems with traditional databases (they can fit more servers, more distributions, handle large volumes of data, and scale horizontally). They are suitable for big data applications and are very flexible. We can obtain the schemas without needing all attributes. Nevertheless, the disadvantages of non-SQL are the lack of robust structure, that the data is difficult to organize and there is less tools and experiences (since is relatively new).

There are 4 main types of NoSQL databases:

- a) Value key stores: Data stored as key-value pairs like in Amazon DynamoDB or Redis. This is the most simple storage.
- b) Document stores: Data stored as documents (for example JSON objects) like in MongoDB.
- c) Column-family stores: Data stored in columns rather than rows.
- d) Graph databases: Data stored as nodes and edges, like in Amazon Neptune.

## 2. OBJECTIVE

The objective of this part is to make a simple exercise where we are going to take a .csv file that contains genes and use it to create a nonSQL database inside the Amazon service DynamoDB. Afterwards, we will use an API gateway to create a service that allows us to ask questions to our database, and the questions will be asked over the internet).

The genomic file we will use in the exercise looks like this:

```
ID,Chromosome,Position,Reference_Allele,Alternate_Allele,Genotype,Quality,Depth
1,1,100001,A,T,AA,30,50
2,1,150002,C,G,CC,28,45
3,2,500003,G,A,GG,35,60
4,2,750004,T,C,TT,32,55
5,3,300005,A,G,AG,25,40
6,3,450006,C,T,CC,28,48
7,4,900007,G,T,GT,31,52
8,4,1200008,A,C,AC,27,44
9,5,600009,T,G,TG,33,58
10,5,900010,C,A,CA,29,46
```

### 3. METHODOLOGY

- First of all, we download the data in .csv and upload it into an s3 bucket we create for the occasion.
- Then, we go to the **dynamo DB** service, where we can either create the table from scratch or from a file that we have in s3. In this case, we will select the second option and import the .csv genomic file by clicking “s3 importation” and following the steps:

Propietario del bucket de S3

☒ Esta cuenta de AWS (975050214287)

☐ Una cuenta de AWS diferente

Compresión de archivos de importación

Elige el tipo de compresión que coincida con los datos de S3 de origen.

☒ Sin compresión

☐ GZIP

☐ ZSTD

Importar formato de archivo [Información](#)

Elige el formato de archivo que coincida con su fuente de almacenamiento de Amazon S3.

☐ JSON de DynamoDB

☐ Amazon Ion

☒ CSV

Encabezado CSV

Elige cómo se definen los encabezados al importar un archivo CSV.

☒ Usar la primera línea del archivo de origen

Genere automáticamente valores de encabezado a partir de la primera línea del archivo de origen.

☐ Definir los encabezados

Defina los valores de encabezado específicos que desea importar separados por una coma.

Carácter delimitador CSV

El carácter delimitador debe coincidir con los datos del archivo de origen.

Coma (",")

- Then we click the default table configuration. If we need more resources we can choose the custom option. Finally, we end the importation.
- After creating the table we can:
  - a) Scan elements, which retrieves all the items in the table.
  - b) Consult, which first queries a key and then filters based on some fields.

▼ Scan or query items

☒ Scan ☐ Query

Select a table or index: Table - genomics

Select attribute projection: All attributes

▼ Filters

Attribute name	Type	Condition	Value	
Depth	String	Greater t...	50	Remove

Add filter

Run Reset

- When the database is finished we expose it to the internet. We will use an API, which is a service that we can have over the internet to ask questions to a database or other services. In very few words, it is a protocol to communicate over the internet and trigger actions.
- We will create a REST API (the most common and simple API type) with amazon **API gateway**. After creating the API , we define our path (/genomics) and inside this path, we create the necessary methods (ANY, DELETE, PUT...)

Gateway de API > API > Recursos - genomics-api (mzkbne8oq9) > Crear recurso

### Crear recurso

**Detalles del recurso**

☒ **Recurso de proxy** Información  
 Los recursos de proxy gestionan las solicitudes a todos los subrecursos. Para crear un recurso de proxy, utilice un parámetro de ruta que termine con un signo más, por ejemplo {proxy+}.

Ruta de recurso: /

Nombre del recurso: genomics

☐ **CORS (uso compartido de recursos entre orígenes)** Información  
 Cree un método OPTIONS que permita todos los orígenes, todos los métodos y varios encabezados comunes.

Cancelar Crear recurso

When retrieving information we are getting (GET). We can use **lambda** for example to get the information (although there are other options like HTTP or other AWS services).

- We go to lambda and create a function named “genomics-retrieval”, for example. We implement this logic in lambda using python:

```
import json
import boto3
from decimal import Decimal

def lambda_handler(event, context):
    try:
        dynamodb = boto3.resource('dynamodb')
        table = dynamodb.Table('genomics')
        response = table.scan()
        items = response['Items']
        for item in items:
            for key, value in item.items():
                if isinstance(value, Decimal):
                    item[key] = float(value)
        return {
            'statusCode': 200,
            'body': json.dumps(items)
        }
    except Exception as e:
        return {
            'statusCode': 500,
            'body': f'Error retrieving data: {str(e)}'
        }
```

We are using the boto3 library to interact with other Amazon services. In this case, the lambda handler is asking the dynamo DB resource to obtain the genomics data from that table. In the example, the response is just to scan (no need to filter). If the retrieval works, a status code 200 (standard code for all ok) is returned, and we will be able to see the json version of the items in the browser. Otherwise, it will return a 500 code and print the error.

- Then, we deploy the lambda and go back to the API gateway configuration. When creating the GET method, we select the newly created lambda (the rest of the configuration is kept as default). Finally, we deploy the API. We copy the URL that appears in the GET method and paste it into the browser. We will see the status code and the data from the database:

```

1 {
2   "statusCode": 200,
3   "body": "[{"Depth": "45",
  "Alternate_Allele": "G", "Quality": "28",
  "Reference_Allele": "C", "Position": "1",
  "ID": "2", "Chromosome": "1",
  "Genotype": "CC"}, {"Depth": "44",
  "Alternate_Allele": "C", "Quality": "27",
  "Reference_Allele": "A", "Position": "120008",
  "ID": "8", "Chromosome": "4",
  "Genotype": "AC"}, {"Depth": "58",
  "Alternate_Allele": "G", "Quality": "33",
  "Reference_Allele": "T", "Position": "60009",
  "ID": "9", "Chromosome": "5",
  "Genotype": "TG"}, {"Depth": "50",
  "Alternate_Allele": "T", "Quality": "30",
  "Reference_Allele": "A", "Position": "10001",
  "ID": "1", "Chromosome": "1",
  "Genotype": "AA"}, {"Depth": "48",
  "Alternate_Allele": "T", "Quality": "28",
  "Reference_Allele": "C", "Position": "45006",
  "ID": "6", "Chromosome": "3",
  "Genotype": "CC"}, {"Depth": "40",
  "Alternate_Allele": "G", "Quality": "25",
  "Reference_Allele": "A", "Position": "30005",
  "ID": "5", "Chromosome": "3",
  "Genotype": "AG"}, {"Depth": "55",
  "Alternate_Allele": "C", "Quality": "32",
  "Reference_Allele": "T", "Position": "75004",
  "ID": "4", "Chromosome": "2",
  "Genotype": "TT"}, {"Depth": "52",
  "Alternate_Allele": "T", "Quality": "31",
  "Reference_Allele": "G", "Position": "90007",
  "ID": "7", "Chromosome": "4",
  "Genotype": "GT"}, {"Depth": "60",
  "Alternate_Allele": "A", "Quality": "35",
  "Reference_Allele": "G", "Position": "50003",
  "ID": "3", "Chromosome": "2",
  "Genotype": "GG"}, {"Depth": "46",
  "Alternate_Allele": "A", "Quality": "29",
  "Reference_Allele": "C", "Position": "90010",
  "ID": "10", "Chromosome": "5",
  "Genotype": "CA"}]"
4 }

```

We have made the database accessible through the internet in very few and simple steps.

It is possible to make the responses more fancy without having to edit lambda, only by editing the integration response. We could create a “Mapping template” like this one:

```

{
  "statusCode": $input.json('$.statusCode'),
  "headers": {
    "Content-Type": "application/json"
  },
  "body": $input.path('$.body')
}

```

Then we should deploy again the API and this would be how the response looks like this:

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": [
    {
      "Depth": "45",
      "Alternate_Allele": "G",
      "Quality": "28",
      "Reference_Allele": "C",
      "Position": "150002",
      "ID": "2",
      "Chromosome": "1",
      "Genotype": "CC"
    },
    {
      "Depth": "44",
      "Alternate_Allele": "C",
      "Quality": "27",
      "Reference_Allele": "A",
      "Position": "1200008",
      "ID": "8",
      "Chromosome": "4",
      "Genotype": "AC"
    },
    {
      "Depth": "58",
      "Alternate_Allele": "G",
      "Quality": "33",
      "Reference_Allele": "T",
      "Position": "600009",
      "ID": "9",
      "Chromosome": "5",
      "Genotype": "TG"
    }
  ],
}
```

# ETLs WITH AWS

## 1. INTRODUCTION

An ETL pipeline is a set of processes that extract data from a source, transform it, and load it into a destination. ETL stands for “Extract, Transform, Load”.

- ❖ Extract: In the extraction phase, data is gathered from various sources such as databases, files, APIs, or other systems. This raw data can be structured (like tables in a database), semi-structured (like XML or JSON files), or unstructured (like text documents or multimedia files). The goal is to collect the necessary data for analysis and processing.
- ❖ Transform: After extraction, the data often needs to undergo transformation to make it suitable for analysis and loading into the target system. This transformation step involves cleaning, filtering, aggregating, enriching, and restructuring the data. Common transformation tasks include data normalization, deduplication, joining datasets, and applying business rules or calculations.
- ❖ Load: Once the data has been extracted and transformed, it is loaded into the target destination, typically a data warehouse, data lake, or a database optimized for analytics. The loading phase involves efficiently inserting the transformed data into the target system while ensuring data integrity and maintaining performance. Loading can be done in batches or in real-time depending on the requirements of the analytics process.

On the other hand, **AWS Glue** is a fully managed extract, transform, and load (ETL) service. It simplifies the process of building, managing, and running ETL jobs for data preparation and analytics. You can create and run ETL jobs with a few clicks in the AWS Management Console.

Here's an overview of AWS Glue's key features and capabilities:

- Data Catalog: AWS Glue includes a centralized metadata repository called the Data Catalog. It stores metadata information about various data sources, tables, and schemas, making it easy to discover, search, and manage data assets across different AWS services and external data stores.

- **ETL Jobs:** AWS Glue allows users to define ETL jobs using a visual interface or by writing custom code in Python or Scala. These jobs can extract data from different sources such as Amazon S3, RDS, Redshift, DynamoDB, and more, apply transformations using built-in or custom scripts, and load the transformed data into a target destination.
- **Dynamic Data Frames:** AWS Glue provides a feature called Dynamic Data Frames, which allows users to work with semi-structured and structured data using familiar programming constructs. It supports various data formats including JSON, CSV, Parquet, Avro, and ORC, enabling flexible data processing and transformation.
- **Serverless Architecture:** AWS Glue is serverless, meaning users don't need to provision or manage infrastructure. It automatically scales resources based on the workload, ensuring high availability and performance without the need for manual intervention.



**Crawlers** also play a crucial role in the data discovery and metadata management process within AWS Glue, enabling users to automate the cataloging of data sources and streamline the development of data workflows for analytics, reporting, and machine learning. They help ensure that data assets are properly documented, organized, and accessible for use in various data-driven applications and analyses.

We can also have triggers in AWS Glue to schedule jobs to run at specific times or in response to events.

The objective of this part will be to create a crawler in AWS Glue using the .csv genomic file.

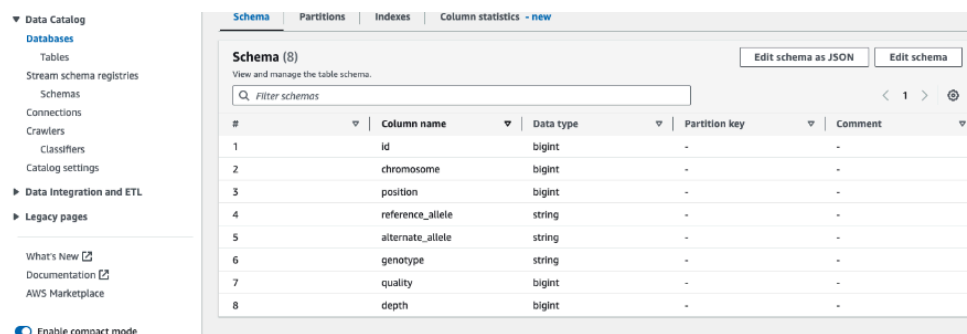


## 2. METHODOLOGY

- To create a crawler in AWS Glue, we need to specify the data source, the database, and the classifier. The crawler will automatically discover the schema of your data and create metadata tables in the AWS Glue Data Catalog.

When configuring, we select *crawl all -sub folders*, which means that every time we add some information we “crawl” everything. Moreover, we will also need to configure the output database and create a classifier to indicate that our data is in .csv (this will make it simpler for Amazon to read the crawler). Once the crawler is created, we run it.

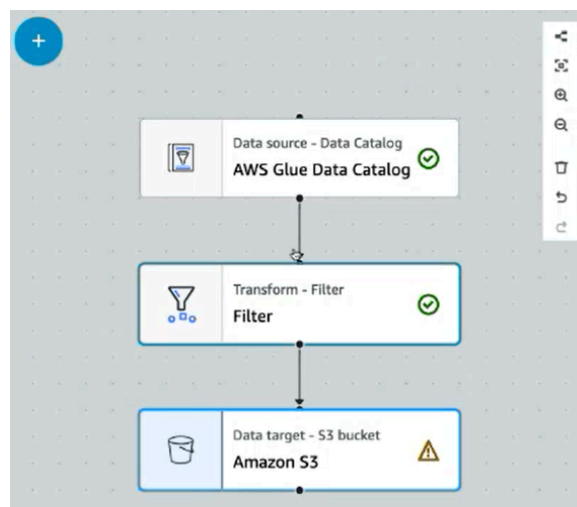
Afterward, we can go to the data catalog and visualize our database, check statistics, see the indexes, etc:



The screenshot shows the AWS Glue Data Catalog interface. On the left is a navigation menu with options like Data Catalog, Databases, Tables, Stream schema registries, Schemas, Connections, Crawlers, Classifiers, Catalog settings, Data Integration and ETL, and Legacy pages. The main panel displays the 'Schema (8)' for a table. It includes a search bar for 'Filter schemas' and buttons for 'Edit schema as JSON' and 'Edit schema'. Below is a table with 8 columns: #, Column name, Data type, Partition key, and Comment. The data rows are as follows:

#	Column name	Data type	Partition key	Comment
1	id	bigint	-	-
2	chromosome	bigint	-	-
3	position	bigint	-	-
4	reference_allele	string	-	-
5	alternate_allele	string	-	-
6	genotype	string	-	-
7	quality	bigint	-	-
8	depth	bigint	-	-

- In the visualETL subsection, we can make data extraction and transformation using a visual tool and doing drag and drop.



Here we indicate that it should filter the data and store the results in a specified s3 bucket.

- Once the job has finished running, we can view the filtered data in the S3 bucket. The data should be filtered based on the quality field, and the rows under a certain threshold should be discarded.

## ➤ REFERENCES

These two exercises were proposed and provided by Jordi Mateo, professor of the subject High Performance and Distributed Computing for Big Data (URV).

<https://github.com/JordiMateoUdL>