Primera parte:

// Busca un dataset de al menos 100MB y 50K registros y describe sus datos.
https://www.kaggle.com/datasets/sriharshaeedala/financial-fraud-detection-dataset
Registra datos sobre transacciones financieras como el tipo de transacción, la cantidad, el origen, el destinatario y los saldos antes y después de la transacción por unidad de tiempo y una columna donde se califica como fraude o no.

// Explica por qué lo han elegido y explica la calidad y utilidad de los datos.
Por nada en particular. En cuanto a la calidad de los datos, teniendo en cuenta que no son reales sino que están generados artificialmente con fines de aprendizaje, pues están perfectos.

// Busca dashboards que representen algo similar a lo que se podría hacer con los datos que has elegido y habla de él desde el punto de vista de cómo de útiles serían los análisis y los cuadros que pueden representar la información elegida.
https://www.slideteam.net/blog/top-10-fraud-detection-dashboard-templates-with-samples-and-examples
En cuanto a la utilidad de los análisis, el dataset que ha tocado en suerte diría que está centrado en los fraudes a "pringados" de los que obtienen sus datos de ccc o tarjeta y les vacían la cuenta en cuestión de horas por lo que ahí lo importante es tener mecanismos automáticos de detección y bloqueo ya que es imposible analizar y responder con mediación humana dado el volumen de transacciones que se realizan en cada momento. Los análisis detallados con gráficas, cuadros de mando etc... serían útiles en el caso de grandes cuentas, que puedan implicar grandes pérdidas para la entidad bancaria y requieran de un análisis más largo y pormenorizado.

// Indica si hay alguna API que nos facilite información similar y qué cantidad de datos podemos extraer de ella.
https://www.transactionlink.io/blog/fraud-detection-apis-to-use
Al no ser información pública, no deberían existir APIs (al menos legales) que sirvan para obtener este tipo de datos, así que he leído el artículo sobre APIs para que las empresas se protejan de posibles fraudes y algunas dicen poder establecer perfiles de posbiles defraudadores sólo con datos como el número de teléfono o de la seguridad social... No se si es bueno o malo...

Segunda parte:

```
// Importaciones
import org.apache.spark.sql.SparkSession
import org.apache.spark.sql.functions._

// Crear la sesión
val spark = SparkSession.builder().appName("Examen").master("local[*]").getOrCreate()

// Carga del dataframe inicial
val dfInicial = spark.read.option("header", "true").option("inferSchema",
"true").csv("311_Service_Requests_from_2010_to_Present_20250520.csv")

// Elimina registros con valores nulos en las columnas esenciales: fecha de creación (Created Date),
tipo de incidente (Complaint Type), barrio (Borough)
val dfNulless = dfInicial.na.drop(cols = Seq("Created Date", "Complaint Type", "Borough"))

scala> dfInicial.count()
res28: Long = 1145936
scala> dfNulless.count()
res29: Long = 1145936

// Convierte la columna de fecha a tipo Date
spark.conf.set("spark.sql.legacy.timeParserPolicy","LEGACY")
val dfFechas = dfNulless.withColumn("Created Date", to_timestamp(col("Created Date"),
"mm/dd/yyyy hh:mm:ss aa")) .withColumn("Resolution Action Updated Date",
to_timestamp(col("Resolution Action Updated Date"), "mm/dd/yyyy hh:mm:ss aa"))
```

```
+----------+------------------+----------------------------+
|Unique Key|Created Date      |Resolution Action Updated Date|
+----------+------------------+----------------------------+
|64802502  |2025-04-30 17:36:05|2025-04-30 20:10:27         |
|64804984  |2025-04-30 17:36:05|NULL                        |
|64803860  |2025-04-30 17:36:01|2025-05-01 11:05:56         |
|64800957  |2025-04-30 17:35:58|2025-04-30 19:12:35         |
|64805078  |2025-04-30 17:35:55|NULL                        |
|64808941  |2025-04-30 17:35:52|2025-05-01 12:01:25         |
|64802636  |2025-04-30 17:35:50|2025-04-30 18:06:51         |
|64805518  |2025-04-30 17:35:27|2025-04-30 18:08:36         |
|64802327  |2025-04-30 17:35:27|NULL                        |
|64805596  |2025-04-30 17:35:25|2025-04-30 18:15:41         |
|64805044  |2025-04-30 17:35:13|2025-04-30 17:43:06         |
|64801988  |2025-04-30 17:35:00|2025-05-01 09:20:53         |
|64809609  |2025-04-30 17:35:00|2025-04-30 22:10:50         |
|64808036  |2025-04-30 17:35:00|2025-05-01 09:48:00         |
|64820934  |2025-04-30 17:35:00|2025-05-01 12:00:00         |
|64808254  |2025-04-30 17:34:59|2025-04-30 22:08:09         |
|64803720  |2025-04-30 17:34:43|2025-05-01 19:16:17         |
|64804982  |2025-04-30 17:34:39|NULL                        |
|64806250  |2025-04-30 17:34:35|NULL                        |
|64802537  |2025-04-30 17:34:26|2025-04-30 22:16:00         |
+----------+------------------+----------------------------+
```

// Estandariza los nombres de los barrios (por ejemplo, pasando todo a mayúsculas). (Ya están en mayúsculas así que los he "capitalizado")
val dfTuned = dfFechas.withColumn("Borough", initcap(col("Borough")))

```
 +-------------+------+
|      Borough| count|
+-------------+------+
|       Queens|260977|
|  Unspecified|   771|
|     Brooklyn|342392|
|Staten Island| 40199|
|    Manhattan|223431|
|        Bronx|278166|
+-------------+------+
```

// Calcula el número total de incidencias por barrio y tipo de incidente.
dfTuned.groupBy("Borough", "Complaint Type").count().orderBy(col("count").desc).show()

```
+---------+--------------------+-----+
|  Borough|      Complaint Type|count|
+---------+--------------------+-----+
|    Bronx| Noise - Residential|77771|
| Brooklyn|     Illegal Parking|76769|
|   Queens|     Illegal Parking|56000|
|    Bronx|      HEAT/HOT WATER|55742|
| Brooklyn|      HEAT/HOT WATER|37971|
|Manhattan|      HEAT/HOT WATER|35338|
| Brooklyn| Noise - Residential|29252|
|    Bronx|     Illegal Parking|26831|
|   Queens|    Blocked Driveway|22838|
|Manhattan| Noise - Residential|21604|
|Manhattan|     Illegal Parking|19694|
| Brooklyn|    Blocked Driveway|19692|
|   Queens|      HEAT/HOT WATER|19669|
|   Queens| Noise - Residential|19007|
|    Bronx|UNSANITARY CONDITION|11600|
|Manhattan|Noise - Street/Si...|10666|
| Brooklyn|UNSANITARY CONDITION|10333|
|   Queens|   Abandoned Vehicle| 9442|
|   Queens|    Street Condition| 8688|
|Manhattan|          Encampment| 8359|
+---------+--------------------+-----+
only showing top 20 rows
```

// Calcula el promedio de tiempo de resolución (Resolution Action Updated Date - Created Date) por barrio.
val dfTiempoResol = dfTuned.withColumn("time_diff_seconds", unix_timestamp(col("Resolution Action Updated Date")) - unix_timestamp(col("Created Date"))).withColumn("time_diff_hours", col("time_diff_seconds") / 3600).filter(col("time_diff_hours") > 0)
val dfAgrupado = dfTiempoResol.groupBy("Borough").agg(round(avg("time_diff_hours"), 2).alias("avg_resolution_hours"), round(avg("time_diff_seconds"), 2).alias("avg_resolution_seconds")).orderBy(desc("avg_resolution_hours"))

```
+-------------+--------------------+----------------------+
|      Borough|avg_resolution_hours|avg_resolution_seconds|
+-------------+--------------------+----------------------+
|  Unspecified|               78.89|              283998.8|
|    Manhattan|               39.62|             142633.38|
|Staten Island|               36.34|             130812.22|
|     Brooklyn|               36.03|             129705.18|
|        Bronx|                36.0|             129585.72|
|       Queens|               29.88|              107575.4|
+-------------+--------------------+----------------------+
```

// Guarda el DataFrame limpio y procesado particionado por año y barrio.
dfAgrupado.write.option("header", "true").mode("overwrite").csv("examen")

//Explica brevemente por qué el particionamiento es útil en Big Data.
El particionamiento es útil para poder manejar grandes volúmenes de información con más facilidad y sin tener que tener grandes requisitos de hardware.

// Realiza en Scala una gráfica con un porcentaje de los datos, puedes partir el CSV eliminando filas.

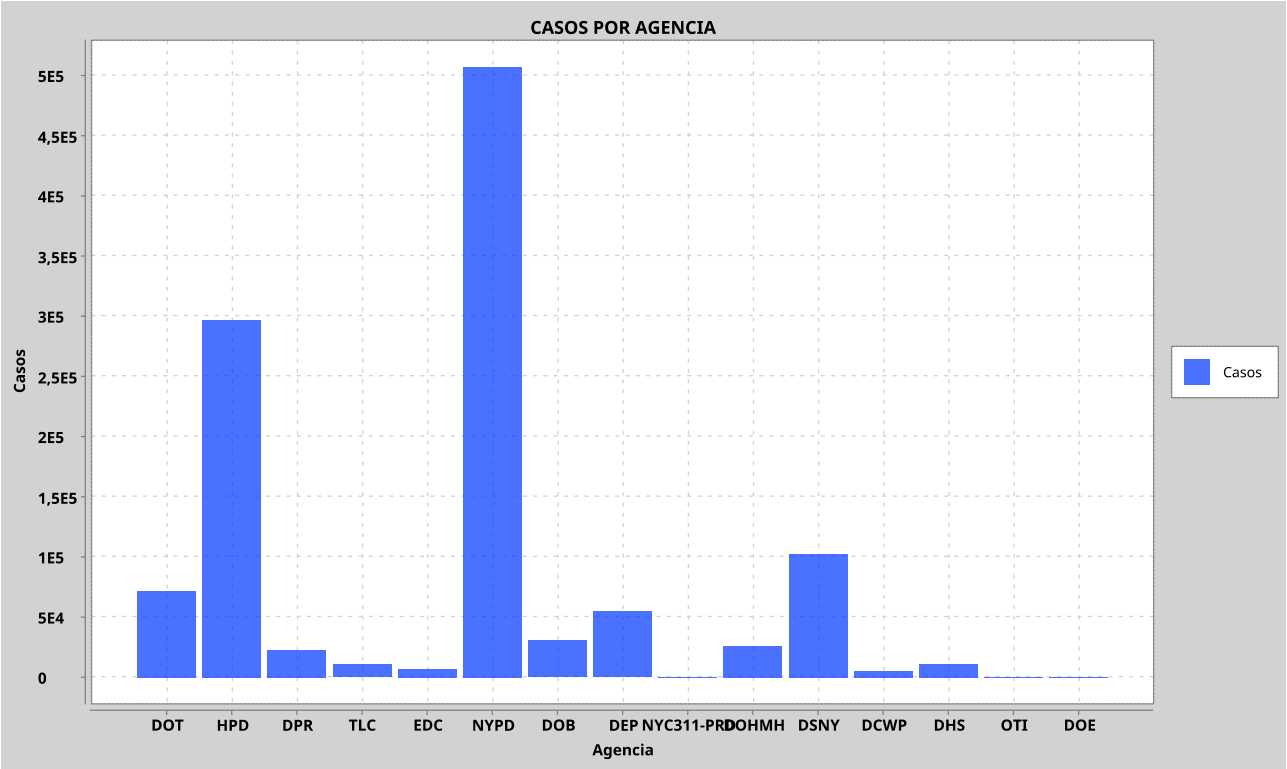
```
val dfGrafica = dfTuned.select("Agency").groupBy("Agency").count()
dfGrafica.write.option("header", "true").mode("overwrite").csv("graficaexamen")
```

```
import scala.io.Source
import scala.jdk.CollectionConverters._
import org.knowm.xchart.{CategoryChart, CategoryChartBuilder, SwingWrapper}
import org.knowm.xchart.VectorGraphicsEncoder
import org.knowm.xchart.VectorGraphicsEncoder.VectorGraphicsFormat

object Graficaexamen extends App {
  val filename = "examengrf.csv"
  val lines = Source.fromFile(filename).getLines().drop(1).toList

  val agencias = lines.map(_.split(",")(0))
  val casos = lines.map(_.split(",")(1).toInt)
  val chart: CategoryChart = new CategoryChartBuilder()
    .width(1500).height(1200)
    .title("CASOS POR AGENCIA")
    .xAxisTitle("Agencia")
    .yAxisTitle("Casos")
    .build()

  chart.addSeries("Casos", agencias.asJava, casos.map(_.asInstanceOf[Number]).asJava)
  new SwingWrapper(chart).displayChart()
  Thread.sleep(10000)
  VectorGraphicsEncoder.saveVectorGraphic(chart, "grafica_examen", VectorGraphicsFormat.SVG)
}
```



// Realiza al menos dos tipos de gráficas que se podrían crear (por ejemplo, mapa de incidencias por barrio, evolución temporal de un tipo de incidente...).