Arquitectura de Datos

PO3: APACHE CASSANDRA

Curso 2025/2026

Grupo de prácticas: 10

Nombre	NIA	Correo Electrónico	Grupo
Javier Rosales Lozano	100495802	100495802@alumnos.uc3m.es	81
Alonso Rios Guerra	100495821	100495821@alumnos.uc3m.es	81
Nahuel Sebastián Vargas	100495715	100495715@alumnos.uc3m.es	81
Alejandro Rodríguez García	100495745	100495745@alumnos.uc3m.es	81

Fecha de entrega: 5/12/2025

Índice

1. Introducción

La siguiente memoria documenta el proceso de desarrollo de la tercera y última práctica correspondiente a la asignatura Arquitectura de Datos del curso 2025/26. La práctica se centra en el uso de un dataset del Titanic compuesto por dos ficheros CSV, en el que se recogen tanto información personal de los pasajeros como datos relacionados con su billete y trayecto.

El objetivo de la práctica es diseñar e implementar un modelo de datos en Apache Cassandra que permita resolver de forma óptima un conjunto de consultas analíticas sobre este dataset.

La realización de la práctica se organiza según los siguientes procedimientos, atendiendo al enunciado de la práctica:

- **Limpieza y normalización de datos**: aunque los datos son relativamente sencillos, presentan variabilidad, por lo que se debe realizar una exploración previa antes del diseño del modelo. Es importante realizar una limpieza de datos, y en la memoria, se documentará el trabajo realizado en esta sección.
- **Diseño del modelo de datos**: realización del modelo de datos orientado a la resolución de consultas en los siguientes apartados.
- **Implementación del diseño y carga de datos**: programación e implementación de las tablas en CQL y proceso de carga de datos que garantice que los datos iniciales puedan usarse para la resolución de consultas.
- **Consultas**: resolución de consultas; adjuntación de capturas de pantalla con las salidas y reflexiones de problemas encontrados o alternativas de diseño consideradas. En esta sección, se evaluarán evidencias básicas de rendimiento.
- **Cuestiones**: respuesta de cuestiones planteadas en el enunciado de la práctica.

2. Diseño del modelo de datos

Dado que Apache Cassandra es un **sistema orientado a columnas** y optimizado para lecturas rápidas mediante claves bien elegidas, para el diseño se han definido las tablas necesarias para cada consulta, eligiendo cuidadosamente las claves de partición y de clustering para evitar escaneos completos y garantizar un buen rendimiento.

El dataset original consta de dos ficheros CSV: uno con información personal de los pasajeros y otro con los datos del billete. Como Cassandra no soporta operaciones de tipo JOIN, se ha realizado una **integración previa de ambos ficheros mediante MongoDB**, generando un fichero CSV (`titanic_join.csv`) completo que agrega cada pasajero con la información de su billete.

Este fichero CSV conjunto ha pasado por un preprocesado de atributos que requerían las siguientes operaciones:

1. Primeramente, se han sustituido globalmente todos los **valores `undefined` de MongoDB por `NULL`**, para evitar inconsistencias entre lenguajes.
2. Después, se han imputado estos valores; en concreto, se debían imputar un atributo numérico y uno categórico.
    - En el caso del atributo categórico `Embarked`, añadimos una **nueva categoría denominada `'N'`**, indicando que no está identificado. De esta manera, este atributo tiene cuatro posibles valores (`'S'`, `'C'`, `'Q'` y `'N'`).
    - Por otro lado, para el atributo numérico `Age`, creamos una variable que almacenaba la **media de todas las edades de los pasajeros**, y sustituimos dicho valor por `NULL` en aquellas instancias que lo requerían.
3. Finalmente, se **normalizan** los valores de cada atributo para que todos compartan el mismo tipo de dato (`Age` y `Fare` se convierten en `Double`, y `Ticket` en `String`).

La adición del **atributo derivado `rango_edad`** en la preparación del CSV para las consultas específicas que lo requerían permite un filtrado más fácil, facilitando el diseño de claves de partición sin filtros numéricos en tiempo de ejecución. Los rangos específicos son:

- Infante: menores a 13 años
- Adolescentes: mayores o igual a 13 años y menores a 20
- Adulto Joven: edades entre 20 y 40 años
- Adulto: entre 40 y 60 años
- Senior: edades mayores a 60 años
- Desconocido: valores nulos o no clasificados (en este caso, no existirán).

Esta discretización permite usar el atributo como clave de partición en tablas, accediendo directamente a particiones específicas sin ALLOW FILTERING ni escaneos por rangos continuos de `Age`.

Para cada consulta del apartado 5, se ha creado una tabla específica en Cassandra con una **clave primaria** diseñada para que la consulta se resuelva con un solo acceso por partición. Este diseño está centrado en cumplir con las consultas sin usar ALLOW FILTERING ni índices secundarios, aprovechando claves de partición y clustering bien definidas para un rendimiento óptimo. Por otro lado, las **claves de clustering** ordenan las filas dentro de la partición cuando se necesita un orden específico o una agrupación adicional.

A continuación, se definen claves claves de partición y clustering para cada tabla.

Consulta	Clave de partición	Clave(s) de clustering
1	`pclass`	`survived, passengerid`
2	`embarked`	`age, passengerid`
3	`pclass`	`sex, survived, passengerid`
4	`rango_edad`	`age, passengerid`
5	`embarked`	`survived, passengerid`
6	`rango_edad`	`pclass, survived, passengerid`

3. Implementación del diseño y carga de datos

3.1. Implementación

Para la implementación del modelo de datos haciendo uso de CQL (Cassandra), para ello se han integrado una serie de creación de tablas, creando una tabla específica para cada consulta del apartado 5. Esto evita ALLOW FILTERING e índices secundarios, optimizando lecturas mediante claves de partición y clústering que distribuyen datos uniformemente y soportan directamente los patrones de acceso.

Se han incluido campos redundantes (`edad`, `nombre`, `sexo`) para evitar JOINs y campos derivados (`rango_edad`) donde facilitan filtros eficientes. Todas las tablas usan passengers como discriminador final para unicidad, asumiendo aproximadamente 1300 pasajeros únicos en el dataset Titanic.

- **Distribución de particiones**: las claves de partición que uso (`pclass`, `embarked`, `survived`, `rango_edad`) tienen pocos valores posibles. Eso hace que los datos se repartan en varias particiones, pero ninguna se hace gigante. Así se evita que un solo nodo reciba casi todas las lecturas/escrituras (hot partition) y el cluster se aprovecha mejor.
- **Ordenamiento por clustering**: el formato de clustering ORDER BY (`age ASC`, `passengerid ASC`) ordena las filas dentro de cada partición en disco. Gracias a eso, las consultas que necesitan filtrar u ordenar por edad (por ejemplo, “pasajeros de un puerto ordenados por edad”) se resuelven leyendo un rango ordenado, sin escanear toda la tabla.
- **Eficiencia de las consultas:** en todas las tablas las consultas se hacen indicando siempre la partition key con el operador `WHERE` (por ejemplo `WHERE embarked='S'` o `WHERE pclass=1 AND survived=1`) y, cuando hace falta, añadiendo filtros sobre las columnas de clustering (como `age`). Esto permite que Cassandra vaya directamente a la partición correcta y lea solo el tramo necesario, en lugar de revisar todas las filas de la tabla.

A continuación, analizaremos la creación de las tablas según el código proporcionado a continuación.

```
DROP TABLE IF EXISTS titanic.mujeres_supervivientes_por_clase;

CREATE TABLE titanic.mujeres_supervivientes_por_clase (
    pclass int,
    sex text,
    survived int,
    passengerid int,
    name text,
    age float,
    PRIMARY KEY (pclass, sex, survived, passengerid)
);
```

Para agilizar el trabajo de importación y exportación de los datos durante el desarrollo del proyecto, se mantiene un fragmento de código que elimina la tabla por completo en el KeySpace si ésta existe, y luego, se vuelve a crear.

Por motivos de consistencia, se ha decidido nombrar la tabla de la misma manera que el respectivo CSV que contiene los datos que se deben importar. De este modo también podremos identificar cada importación de datos de manera distinta a las demás.

3.2. Carga de datos

Como se ha mencionado anteriormente, a partir del CSV conjunto de todos los datos, se han generado seis archivos CSV derivados (uno por cada consulta) destinados a la importación de los datos en las tablas creadas, tal y como se muestra en el siguiente fragmento de código.

```
// Consulta 1
COPY titanic.supervivientes_por_clase (passengerid, name, sex, age,
pclass, survived)
FROM '/home/lab/data_csv/supervivientes_por_clase.csv'
WITH HEADER=true AND NULL='null';

// Consulta 2
COPY titanic.pasajeros_por_puerto_edad (passengerid, name, sex, age,
pclass, embarked)
FROM '/home/lab/data_csv/pasajeros_por_puerto_edad.csv'
WITH HEADER=true AND NULL='null';

// ...
```

El parámetro **`WITH HEADER=true`** indica que el archivo CSV que se importa tiene una primera fila con los nombres de las columnas, por lo que Cassandra la utiliza para mapear correctamente los datos a las columnas de la tabla, sin considerarla como un dato más.

El parámetro **`NULL='null'`** especifica que el texto `'null'` en el archivo CSV debe interpretarse como un valor nulo (`NULL`) en Cassandra, permitiendo que los campos que contengan esa cadena se almacenen como valores nulos en la tabla.

4. Consultas

En este apartado se explicarán las consultas junto con imágenes de las salidas de estas. Además se comentarán cualquier inconveniente o problema encontrado.

4.1. Supervivientes por clase

Para comprobar el número de supervivientes por clase usaremos el siguiente formato de consulta:

```
SELECT pclass, COUNT(*) as supervivientes
FROM titanic.supervivientes_por_clase
WHERE pclass = N AND survived = 1;        //Donde N es la clase(1,2,3)
```

Por ejemplo, para ver los supervivientes de la clase 1 usaremos:

```
SELECT pclass, COUNT(*) as supervivientes
FROM titanic.supervivientes_por_clase
WHERE pclass = 1 AND survived = 1;
```

pclass	supervivientes
1	3016

4.2. Pasajeros por puerto ordenados por edad

Para comprobar los pasajeros en base a su puerto de embarque y ordenados de menor a mayor edad, usamos la siguiente consulta:

```
SELECT embarked, age, passengerid, name, sex, pclass
FROM titanic.pasajeros_por_puerto_edad
WHERE embarked IN ('S','C','Q');
```

Esta consulta nos devuelve el siguiente resultado ordenado por edad:

embarked	age	passengerid	name	sex	pclass
C	0.1	2535	Torres, Col. Christopher	male	3
C	0.1	8922	Marsh, Lady. Pamela	female	3
C	0.2	1398	Hernandez, Mrs. Melanie	female	3
C	0.2	7587	Taylor, Col. Christopher	male	2
C	0.2	8950	Lamb, Countess. Rachael	female	2

4.3. Mujeres supervivientes por clase

Esta consulta es bastante similar a la primera pero incluyendo una condición más, que la superviviente sea mujer. Para ello se usará el siguiente formato:

```
SELECT pclass, COUNT(*) AS num_mujeres_supervivientes
FROM titanic.mujeres_supervivientes_por_clase
```

```
WHERE pclass = N AND sex = 'female' AND survived = 1;
// Donde N es la clase (1,2,3)
```

Por ejemplo, para ver las mujeres supervivientes de la clase 3 usaremos:

```
SELECT pclass, COUNT(*) AS num_mujeres_supervivientes
FROM titanic.mujeres_supervivientes_por_clase
WHERE pclass = 3 AND sex = 'female' AND survived = 1;
```

Dando como resultado:

pclass	num_mujeres_supervivientes
3	425

4.4. Pasajeros por rango de edad

A la hora de realizar esta consulta, nos dimos cuenta que era necesario crear un nuevo atributo, rango_edad. Este atributo tomará los siguientes valores: {Infante, Adolescente, Adulto_Joven, Adulto, Senior}. En este caso el formato de la consulta es el siguiente:

```
SELECT *
FROM titanic.pasajeros_por_rango_edad
WHERE rango_edad = RANGO; // Donde RANGO tiene los valores anteriores
```

Por ejemplo, si queremos ver a los pasajeros cuya rango de edad es Senior, ordenados por edad:

```
SELECT *
FROM titanic.pasajeros_por_rango_edad
WHERE rango_edad = 'Senior';
```

Dando como resultado, el mostrado en la siguiente captura:

rango_edad	age	passengerid	name	pclass	sex	survived
Senior	60	367	Warren, Mrs. Frank Manley (Anna Sophia Atkinson)	1	female	1
Senior	60	588	Frolicher-Stehli, Mr. Maxmillian	1	male	1
Senior	60	685	Brown, Mr. Thomas William Solomon	2	male	0
Senior	60	695	Weir, Col. John	1	male	0
Senior	60	2109	Cooper, Rev. Michael	1	male	0

4.5. Análisis por puerto y supervivencia

Esta consulta se encarga de mostrarnos el número de supervivientes por puerto. Para ello se usa el siguiente formato de consulta:

```
SELECT survived, COUNT(*) AS num_pasajeros
FROM titanic.pasajeros_por_puerto_supervivencia
WHERE embarked = P
GROUP BY survived; // Donde P puede ser {S,C,Q}
```

Por ejemplo, vamos a analizar los supervivientes del puerto C:

```
SELECT survived, COUNT(*) AS num_pasajeros
FROM titanic.pasajeros_por_puerto_supervivencia
WHERE embarked = 'C'
GROUP BY survived;
```

```
survived | num_pasajeros
---------+---------------
       0 |           985
       1 |           903
```

4.6. Análisis por clase, edad y supervivencia

Al igual que en la consulta 4, en esta consulta es necesario implementar el atributo de rango_edad. Esta consulta nos permite ver la supervivencia de un rango de edad dividido por clase y estado de supervivencia. Las consultas seguirán el siguiente formato:

```
SELECT pclass, survived, COUNT(*) AS num_pasajeros
FROM titanic.pasajeros_por_clase_edad_supervivencia
WHERE rango_edad = RANGO // Donde RANGO es el rango_edad
GROUP BY pclass, survived;
```

Por ejemplo, vamos a analizar la supervivencia para los adolescentes:

```
SELECT pclass, survived, COUNT(*) AS
num_pasajeros
FROM
titanic.pasajeros_por_clase_edad_supervivencia
WHERE rango_edad = 'Adolescente'
GROUP BY pclass, survived;
```

Nos da como resultado:

```
pclass | survived | num_pasajeros
-------+----------+---------------
     1 |        0 |           100
     1 |        1 |           207
     2 |        0 |           162
     2 |        1 |           129
     3 |        0 |           262
     3 |        1 |           126
```

5. Cuestiones

5.1. ¿Cómo habrían cambiado las consultas si hubierais permitido ALLOW FILTERING? ¿serían eficientes?

ALLOW FILTERING permite ejecutar consultas con cláusulas `WHERE` que no usen necesariamente claves primarias, pero se fuerza un escaneo completo de la partición. La presencia de ALLOW FILTERING en las consultas habría simplificado el diseño de las tablas, pudiendo usar una tabla única con filtros en campos no clave, como edad. Sin embargo, aunque más sencillo en ocasiones, no serían eficientes, ya que provocan un full scan en particiones grandes, ocasionando inconvenientes asociados las full scan, tales como mayor latencia, carga del ordenador y riesgo de timeouts en datasets crecientes.

5.2. ¿Por qué los índices secundarios pueden parecer útiles, pero no son adecuados para entornos de producción en Cassandra?

Los índices secundarios resultan útiles porque permiten buscar por cualquier columna sin cambiar el diseño de las tablas, como filtrar pasajeros por edad o puerto directamente. Sin embargo, no son adecuados para producción porque distribuyen los datos por muchos nodos, lo que genera muchas lecturas lentas y sobrecarga un nodo principal que coordina todo. Es mejor crear tablas específicas para cada tipo de consulta en lugar de usarlas.

5.3. ¿Qué impacto tiene un mal diseño de claves sobre aspectos como el rendimiento, la compactación, la distribución del anillo o las lecturas excesivas?

Un mal diseño de claves crea particiones desiguales o muy grandes, donde algunos nodos reciben demasiados datos y se saturan mientras otros están vacíos. Esto ralentiza las consultas, complica la limpieza de datos antiguos y fuerza lecturas innecesarias de muchas filas. Como resultado, el sistema es más lento y puede fallar bajo carga.

6. Anexo

6.1. Declaración de uso de IA

Para el desarrollo de la práctica nos hemos apoyado en herramientas de IA generativa como Perplexity o ChatGPT. Estas fueron usadas para ayudarnos a entender conceptos, sintaxis de comandos y, ocasionalmente, ayudarnos a detectar errores en nuestro código. Para ello creamos un espacio de trabajo en modo estudio donde la IA basaba sus respuestas en las fuentes subidas como apuntes de clase y laboratorios.