

---

# **Proceso ETL en un Data Mart**

---



**ASIGNATURA: ALMACENES Y MINERÍA DE DATOS**

**PRÁCTICA 2**

**CURSO: 2025/26**

---

Pascual Albericio, Irene (NIP 871627)

Vázquez Martín, Lucía (NIP 871886)

# Índice

<b>Índice.....</b>	<b>2</b>
<b>1. Introducción.....</b>	<b>3</b>
<b>2. Selección de herramienta ETL.....</b>	<b>4</b>
<b>3. Proceso de extracción de los datos.....</b>	<b>4</b>
<b>4. Adaptación del diseño esquema en estrella de vuelos.....</b>	<b>5</b>
<b>5. Proceso de transformación de los datos.....</b>	<b>6</b>
<b>6. Implementación de vuelos comerciales sin pasajeros.....</b>	<b>8</b>
6.1. Script de creación de tablas.....	8
6.2. Script de inserción de datos.....	10
6.3. Script de consultas.....	18
<b>7. Análisis de cargas incrementales.....</b>	<b>27</b>
<b>8. Análisis del rendimiento.....</b>	<b>27</b>
<b>9. Distribución del trabajo.....</b>	<b>28</b>
<b>10. Problemas encontrados.....</b>	<b>29</b>
<b>11. Conclusiones.....</b>	<b>30</b>

# 1. Introducción

El objetivo de esta práctica es modificar el diseño y la implementación del data mart desarrollado en la práctica anterior, centrado en el análisis del rendimiento de vuelos comerciales en Estados Unidos, para incorporar información sobre las ciudades en las que se encuentran los aeropuertos.

Asimismo, se repetirán los procesos de creación, inserción y análisis de datos, esta vez utilizando datos reales obtenidos de las fuentes públicas **GeoNames** y **RITA**, correspondientes al mes de marzo de 2025.

Debido al uso de los nuevos datos, ha sido necesario ajustar el esquema en estrella del data mart original, de forma que refleje adecuadamente la información seleccionada de las webs.

El proceso ETL implementado en esta práctica se alinea con los 34 subsistemas propuestos por Ralph Kimball en su metodología de data warehouse, aunque adaptados al contexto específico del data mart de vuelos y a las herramientas disponibles:

## **1. SUBSISTEMAS DE EXTRACCIÓN DE DATOS**

---

Este subsistema se implementó mediante la consolidación de múltiples fuentes en el archivo **datos\_vuelos.xlsx**, actuando como un **área de staging** unificada. Aunque no se implementó un Sistema de Detección de Cambios porque se trabajó con un snapshot estático de marzo de 2025, se establecieron las bases para futuras implementaciones incrementales mediante la identificación de claves naturales (matrículas de aviones, códigos IATA, códigos DOT, códigos de vuelo...) que permitirán identificar las modificaciones en cargas posteriores.

## **2. SUBSISTEMAS DE LIMPIEZA Y CONFORMACIÓN**

---

Se realizó mediante múltiples nodos en **KNIME**, lo cual está explicado más adelante en el apartado de “Proceso de Extracción de los Datos”.

## **3. SUBSISTEMAS DE ENTREGA DE DATOS**

---

El **Subsistema de Carga de Dimensiones** se implementó procesando cada dimensión por separado en **KNIME**, eliminando duplicados con *GroupBy* y generando IDs autoincrementales. El **Subsistema de Carga de Hechos** utilizó *JOINs* secuenciales para unir todas las dimensiones con la tabla de vuelos, calculando las métricas pertinentes.

## **4. SUBSISTEMAS DE GESTIÓN DEL DATA WAREHOUSE**

---

El **Subsistema de Gestión del Flujo de Trabajo** se llevó a cabo en la estructura modular del workflow de **KNIME**, permitiendo ejecutar y monitorear cada paso del ETL. El **Subsistema de Gestión de Metadatos** se abordó mediante la documentación de transformaciones y nomenclatura clara en cada nodo.

## **5. ADAPTACIONES ESPECÍFICAS DEL PROYECTO**

---

Como esta práctica se trata de un proyecto académico, algunos subsistemas se simplificaron. Sin embargo, la estructura implementada proporciona una base sólida que podría permitir incorporar más subsistemas de Kimball en futuras versiones.

## 2. Selección de herramienta ETL

Para el diseño e implementación del proceso ETL, se proponían diversas alternativas que permiten realizar dicho proceso. Antes de comenzar con las tareas se analizaron varias de las herramientas sugeridas, entre ellas **Pentaho Data Integration** (PDI), **Talend Open Studio**, **Apache Hop** y **KNIME**, las cuales ofrecen potentes entornos visuales para el diseño de flujos de datos y la gestión de transformaciones complejas.

Tras una valoración inicial, se decidió optar por **KNIME** para llevar a cabo esta tarea. La principal razón de esta elección se debe a la familiaridad previa de los integrantes con este entorno, adquirida en otras asignaturas del grado. Esta decisión permitió centrar los esfuerzos en la lógica de negocio del proceso ETL, en lugar de invertir el tiempo en aprender a utilizar una nueva herramienta. Asimismo, **KNIME** demostró ser una solución robusta, ofreciendo la modularidad necesaria a través de sus nodos.

## 3. Proceso de extracción de los datos

Dado que en este caso no se partía desde cero, no fue necesario repetir por completo el proceso de la metodología de Kimball. En su lugar, se realizó una búsqueda exhaustiva de datos reales que permitieran minimizar los cambios necesarios en el esquema en estrella.

El proceso de obtención de datos presentó varios desafíos significativos. Tal y como se indicaba en el enunciado, se utilizaron tanto la fuente **RITA** como **GeoNames**, aunque no ambas en igual medida. Tras analizar exhaustivamente las fuentes disponibles, se detectó que los datos de **RITA** proporcionaban prácticamente toda la información completa necesaria. Por tanto, **GeoNames** se utilizó exclusivamente para obtener los datos de población de cada una de las ciudades.

Esta combinación de fuentes permitió consolidar toda la información necesaria en un único archivo llamado *datos\_vuelos.xlsx*. Este archivo contiene cinco tablas principales que capturan toda la información requerida para el análisis:

- **Tabla VUELOS:** registros de vuelos comerciales del mes de marzo de 2025. Los atributos clave son:
  - Identificadores temporales: *FL\_DATE*, *YEAR*, *MONTH*, *DAY\_OF\_MONTH*, *DAY\_OF\_WEEK*
  - Códigos de vuelo: *OP\_UNIQUE\_CARRIER*, *OP\_CARRIER\_FL\_NUM*
  - Aeropuertos: *ORIGIN*, *DEST*, *ORIGIN\_AIRPORT\_ID*, *DEST\_AIRPORT\_ID*
  - Tiempos y Distancias: *CRS\_DEP\_TIME*, *CRS\_ARR\_TIME*, *CRS\_ELAPSED\_TIME*, *ACTUAL\_ELAPSED\_TIME*, *DISTANCE*
  - Operadora/Aerolínea: *OP\_CARRIER\_AIRLINE\_ID*, *TAIL\_NUM*
  - Aviones: *TAIL\_NUM*
- **Tabla AVIONES:** inventario de aeronaves registradas. Los atributos principales son:
  - Identificación: *TAIL\_NUMBER*, *AIRLINE\_ID*
  - Características técnicas: *MANUFACTURER*, *MODEL*, *AIRCRAFT\_TYPE*, *NUMBER\_OF\_SEATS*, *CAPACITY\_IN\_POUNDS*
  - Información temporal: *MANUFACTURE\_YEAR*
  - Estado operativo: *OPERATING\_STATUS*, *AIRCRAFT\_STATUS*

- **Tabla AEROPUERTOS:** información detallada de aeropuertos estadounidenses. Los atributos principales son:
  - Identificación: *AIRPORT\_ID*
  - Características: *DISPLAY\_AIRPORT\_NAME, AIRPORT\_COUNTRY\_NAME, AIRPORT\_STATE\_CODE, DISPLAY\_CITY\_MARKET\_NAME, LATITUDE, LONGITUDE*
- **Tabla AEROLINEA\_OPERADORA:** catálogo de aerolíneas y operadoras. Los atributos esenciales son:
  - Identificadores: *AIRLINE\_ID, UNIQUE\_CARRIER*
  - Denominación: *UNIQUE\_CARRIER\_NAME*
- **Tabla LUGARES:** información complementaria de ubicaciones que se utilizó únicamente para obtener la población de cada una de las ciudades.
  - Identificadores: *NAME*
  - Características: *FEATURE\_CLASS, POBLACION*

Las relaciones identificadas entre tablas son las siguientes:

- **VUELOS ↔ AVIONES:** mediante *TAIL\_NUM = TAIL\_NUMBER*.
- **VUELOS ↔ AEROLINEA\_OPERADORA:** mediante *OP\_CARRIER\_AIRLINE\_ID = AIRLINE\_ID*.
- **VUELOS ↔ AEROPUERTOS:** mediante *ORIGIN\_AIRPORT\_ID = AIRPORT\_ID* y *DEST\_AIRPORT\_ID = AIRPORT\_ID*.
- **AVIONES ↔ AEROLINEA\_OPERADORA:** mediante *AIRLINE\_ID*.
- **LUGARES ↔ AEROPUERTOS:** mediante *NAME = CIUDAD*.

La diferencia entre operadora y aerolínea se realizó gracias a la tabla de aviones, ya que en esta tabla se mostraba la compañía que operaba realmente el vuelo (porque ofrece el avión, empleados...), mientras que la compañía que aparecía en *tabla\_vuelos* era la que organizaba el vuelo.

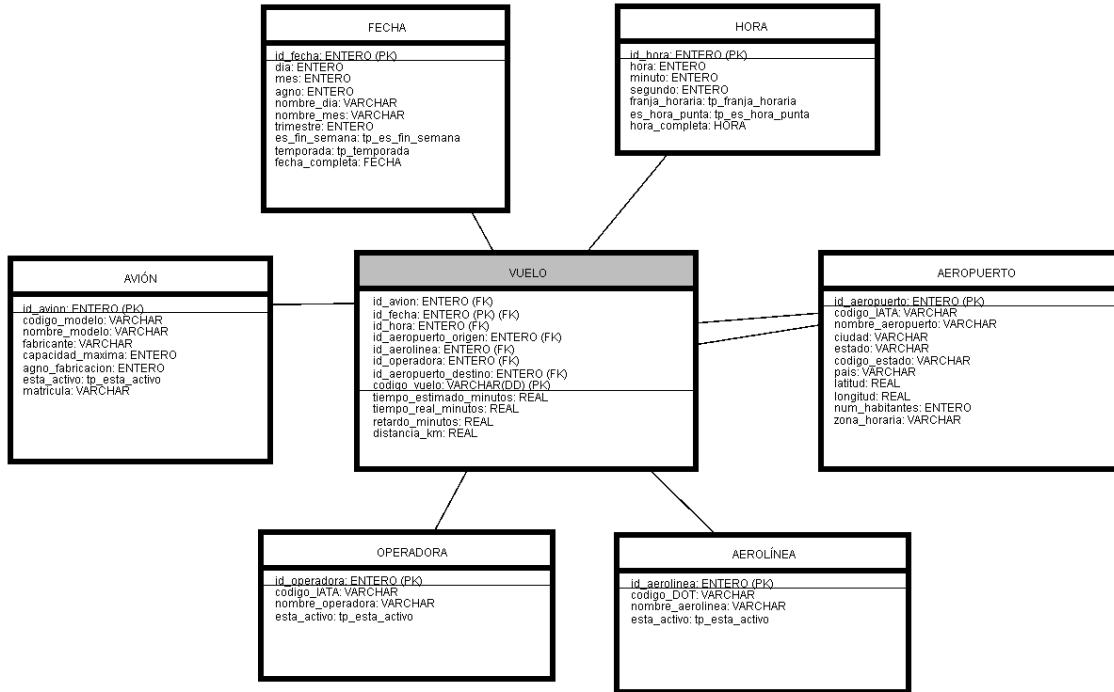
## 4. Adaptación del diseño esquema en estrella de vuelos

Una vez completado el análisis de los datos reales disponibles e identificadas las relaciones entre las tablas, fue necesario adaptar el esquema en estrella original para reflejar adecuadamente la información contenida en el archivo *datos\_vuelos.xlsx*.

El esquema en estrella final apenas difiere del planteado originalmente en la práctica anterior, ya que únicamente fue necesario incorporar un atributo adicional (*num\_habitantes*) a las tablas existentes, debido a que la estructura inicial resultaba suficiente para representar la información disponible.

Las modificaciones realizadas se centraron principalmente en la eliminación de ciertos atributos que se consideraron no relevantes o para los cuales no existían datos en las fuentes consultadas. Por ejemplo, en la tabla **AVIÓN** se eliminaron los atributos *tipo\_de\_motor, num\_motores, velocidad\_maxima* y *agnos\_comienzo\_uso*.

En cuanto a las tablas **AEROPUERTO**, **OPERADORA** y **AEROLÍNEA**, además de eliminar algunos campos secundarios, se realizaron ajustes en los códigos identificadores.



## 5. Proceso de transformación de los datos

El proceso de transformación de datos se implementó utilizando **KNIME Analytics Platform** mediante un workflow estructurado que procesó secuencialmente las cinco tablas del archivo *datos\_vuelos.xlsx*.

El flujo comenzó con la lectura de cada hoja Excel mediante **nodos Excel Reader**, seguido de la creación de las tablas dimensionales (**FECHA**, **HORA**, **AVIÓN**, **AEROPUERTO**, **OPERADORA** y **AEROLÍNEA**) a través de operaciones complejas de limpieza, transformación y agregación.

Para la dimensión **FECHA** se partió del campo *FL\_DATE* de la tabla **VUELOS**, implementando validaciones para regenerar fechas faltantes a partir de los componentes *YEAR*, *MONTH* y *DAY\_OF\_MONTH*, posteriormente se utilizaron **nodos String Manipulation** y **Math Formula** para generar atributos derivados como *nombre\_dia*, *nombre\_mes*, *trimestre*, *temporada* (basada en fechas de cambio de estación) y *es\_fin\_semana*.

La dimensión **HORA** se construyó extrayendo componentes del campo *CRS\_DEP\_TIME* mediante operaciones matemáticas (división y módulo para separar horas y minutos), y calculamos *franjas\_horaria* y *es\_hora\_punta* según rangos predefinidos.

Las dimensiones de entidades (**AVIÓN**, **AEROPUERTO**, **OPERADORA**, **AEROLÍNEA**) requirieron JOINs complejos entre múltiples tablas:

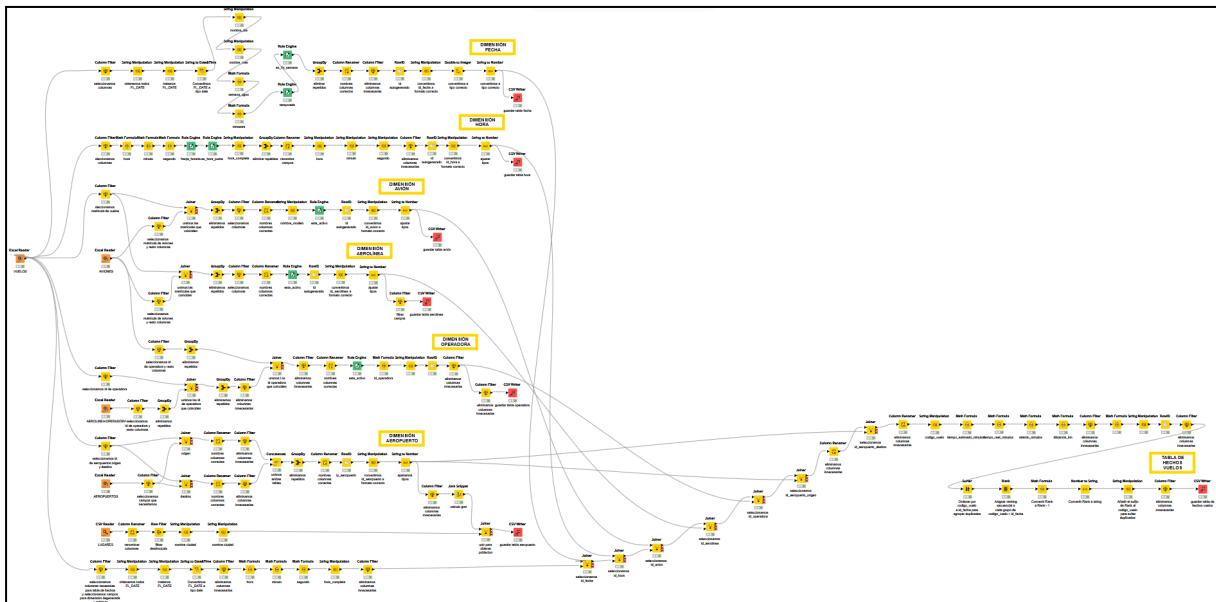
- **AVIÓN** se generó uniendo **VUELOS** y **AVIONES** por *TAIL\_NUM=TAIL\_NUMBER*, eliminando duplicados mediante el **nodo GroupBy** y derivando *codigo\_modelo* del

campo **AIRCRAFT\_TYPE** y **nombre\_modelo** concatenando **MANUFACTURER+MODEL**.

- **AEROPUERTO** se extrajo directamente de los campos **ORIGIN/DEST** de **VUELOS**. Respecto a los campos de zona horaria y población, el primero de ellos se calculó mediante la latitud y longitud, y el otro se obtuvo gracias a la unión con la tabla **LUGARES**.
- **OPERADORA** se obtuvo agrupando por **OP\_CARRIER\_AIRLINE\_ID** y enlazando con **AEROLINEA\_OPERADORA** para obtener otras características.
- **AEROLÍNEA** se derivó de la tabla **AVIONES** agrupando por **AIRLINE\_ID** para identificar compañías propietarias.

Finalmente, se construyó la tabla de hechos **VUELO** mediante múltiples **JOINS** secuenciales que integraron todas las dimensiones, junto con las métricas como *tiempo\_estimado\_minutos* y *tiempo\_real\_minutos*, *retardo\_minutos* (con validación para evitar valores negativos en caso de que los vuelos llegasen con antelación), *distancia\_km* convirtiendo de centésimas de milla a kilómetros, y *codigo\_vuelo* concatenando **OP\_UNIQUE\_CARRIER** con **OP\_CARRIER\_FL\_NUM**.

Todo el proceso fue gestionado con **nodos especializados** como **Column Filter**, **String to Number**, **Rule Engine...**, resultando en un modelo dimensional consistente con aproximadamente 1.700 registros de vuelos válidos, cada dimensión con claves primarias autogeneradas secuencialmente, y manteniendo integridad referencial preparada para su carga en Oracle Database.



## 6. Implementación de vuelos comerciales sin pasajeros

Al igual que en la primera sesión, el modelo en estrella se implementó en Oracle Database.

Siguiendo la estructura de la práctica anterior, el proceso de implementación se ha hecho mediante 3 scripts SQL:

- *crear\_tablas\_vuelos.sql*
- *insertar\_datos\_vuelos.sql*
- *consultas\_vuelos.sql*

### 6.1. Script de creación de tablas

El script *crear\_tablas\_vuelos.sql* mantiene la misma estructura general, incluyendo los bloques PL/SQL encargados de eliminar las tablas existentes en orden inverso a sus dependencias mediante el uso de *EXECUTE IMMEDIATE*, capturando las excepciones correspondientes cuando una tabla no existe.

Se han realizado ajustes menores respecto al esquema original:

- **DIM\_FECHA:** Se han eliminado los atributos *semana\_agno* y *festivo*.

SQL

```
CREATE TABLE FECHA (
    id_fecha INTEGER PRIMARY KEY,
    fecha_completa DATE UNIQUE NOT NULL, -- Clave Natural
    dia INTEGER NOT NULL,
    mes INTEGER NOT NULL,
    agno INTEGER NOT NULL,
    nombre_dia VARCHAR2(20) NOT NULL,
    nombre_mes VARCHAR2(20) NOT NULL,
    trimestre INTEGER NOT NULL,
    es_fin_semana VARCHAR2(20) NOT NULL CHECK (es_fin_semana IN ('es_fin_semana',
    'no_es_fin_semana')),
    temporada VARCHAR2(20) NOT NULL CHECK (temporada IN ('primavera', 'verano', 'otono',
    'invierno'))
);
```

- **DIM\_HORA:** Se ha establecido el valor del atributo *segundo* siempre a 0.

SQL

```
CREATE TABLE HORA (
    id_hora INTEGER PRIMARY KEY,
    hora_completa DATE UNIQUE NOT NULL, -- Clave Natural
    hora INTEGER NOT NULL,
    minuto INTEGER NOT NULL,
    segundo INTEGER NOT NULL,
    franja_horaria VARCHAR2(20) NOT NULL CHECK (franja_horaria IN ('00:00 - 04:59', '05:00 -
    11:59', '12:00 - 19:59', '20:00 - 23:59')),
    es_hora_punta VARCHAR2(20) NOT NULL CHECK (es_hora_punta IN ('es_hora_punta',
    'no_es_hora_punta'))
);
```

- **DIM\_AEROPUERTO:** Se han eliminado los atributos *region*, *altitud* y *tipo\_aeropuerto* y se ha reemplazado el *codigo\_ICAO* por el *codigo\_IATA*.

SQL

```
CREATE TABLE AEROPUERTO (
    id_aeropuerto INTEGER PRIMARY KEY,
    codigo_IATA VARCHAR2(50) UNIQUE NOT NULL, -- Clave Natural
    nombre_aeropuerto VARCHAR2(100) NOT NULL,
    ciudad VARCHAR2(100) NOT NULL,
    estado VARCHAR2(100) NOT NULL,
    codigo_estado VARCHAR2(50) NOT NULL,
    pais VARCHAR2(100) NOT NULL,
    latitud REAL NOT NULL,
    longitud REAL NOT NULL,
    num_habitantes INTEGER,
    zona_horaria VARCHAR2(50) NOT NULL
);
```

- **DIM\_AVION:** Se han eliminado los atributos *tipo\_de\_motor*, *num\_motores*, *velocidad\_maxima* y *agno\_comienzo\_uso*.

SQL

```
CREATE TABLE AVION (
    id_avion INTEGER PRIMARY KEY,
    matricula VARCHAR2(50) UNIQUE NOT NULL, -- Clave Natural
    codigo_modelo VARCHAR2(50) NOT NULL,
    nombre_modelo VARCHAR2(100) NOT NULL,
    fabricante VARCHAR2(100) NOT NULL,
    capacidad_maxima INTEGER NOT NULL,
    agno_fabricacion INTEGER NOT NULL,
    esta_activo VARCHAR2(20) NOT NULL CHECK (esta_activo IN ('esta_activo', 'no_esta_activo'))
);
```

- **DIM\_OPERADORA:** Se han eliminado los atributos *pais\_origen* y *agno\_fundacion* y se ha reemplazado el *codigo\_ICAO* por el *codigo\_IATA*.

SQL

```
CREATE TABLE OPERADORA (
    id_operadora INTEGER PRIMARY KEY,
    codigo_IATA VARCHAR2(50) UNIQUE NOT NULL, -- Clave Natural
    nombre_operadora VARCHAR2(100) NOT NULL,
    esta_activo VARCHAR2(20) NOT NULL CHECK (esta_activo IN ('esta_activo', 'no_esta_activo'))
);
```

- **DIM\_AEROLINEA:** Se han eliminado los atributos *nombre\_comercial*, *pais\_origen*, *tipo\_aerolinea* y *agno\_fundacion* y se ha reemplazado el *codigo\_ICAO* por el *codigo\_DOT*.

SQL

```
CREATE TABLE AEROLINEA (
    id_aerolinea INTEGER PRIMARY KEY,
    codigo_DOT VARCHAR2(50) UNIQUE NOT NULL, -- Clave Natural
    nombre_aerolinea VARCHAR2(100) NOT NULL,
    esta_activo VARCHAR2(20) NOT NULL CHECK (esta_activo IN ('esta_activo', 'no_esta_activo'))
);
```

- **TABLA DE HECHOS (VUELO):** Se mantiene igual que en el original.

SQL

```
CREATE TABLE VUELO (
    codigo_vuelo VARCHAR2(50) NOT NULL,
    id_fecha INTEGER NOT NULL,
    id_hora INTEGER NOT NULL,
    id_avion INTEGER NOT NULL,
    id_aeropuerto_origen INTEGER NOT NULL,
    id_aeropuerto_destino INTEGER NOT NULL,
    id_aerolinea INTEGER NOT NULL,
    id_operadora INTEGER NOT NULL,
    tiempo_estimado_minutos REAL NOT NULL,
    tiempo_real_minutos REAL NOT NULL,
    retardo_minutos REAL NOT NULL,
    distancia_km REAL NOT NULL,
    PRIMARY KEY (codigo_vuelo, id_fecha),
    FOREIGN KEY (id_avion) REFERENCES AVION(id_avion) ON DELETE CASCADE,
    FOREIGN KEY (id_fecha) REFERENCES FECHA(id_fecha) ON DELETE CASCADE,
    FOREIGN KEY (id_hora) REFERENCES HORA(id_hora) ON DELETE CASCADE,
    FOREIGN KEY (id_aeropuerto_origen) REFERENCES AEROPUERTO(id_aeropuerto) ON DELETE CASCADE,
    FOREIGN KEY (id_aeropuerto_destino) REFERENCES AEROPUERTO(id_aeropuerto) ON DELETE CASCADE,
    FOREIGN KEY (id_aerolinea) REFERENCES AEROLINEA(id_aerolinea) ON DELETE CASCADE,
    FOREIGN KEY (id_operadora) REFERENCES OPERADORA(id_operadora) ON DELETE CASCADE,
    CONSTRAINT chk_tiempo_positivo CHECK (tiempo_estimado_minutos > 0 AND tiempo_real_minutos > 0),
    CONSTRAINT chk_distancia_positiva CHECK (distancia_km > 0),
    CONSTRAINT chk_retardo_no_negativo CHECK (retardo_minutos >= 0),
    CONSTRAINT chk_aeropuertos_diferentes CHECK (id_aeropuerto_origen != id_aeropuerto_destino)
);
```

## 6.2. Script de inserción de datos

El script *insertar\_datos\_vuelos.sql* mantiene la misma estructura general que en la práctica anterior, comenzando con la eliminación ordenada de los registros existentes (de la tabla de hechos hacia las dimensiones) para permitir ejecuciones repetidas sin errores:

SQL

```
DELETE FROM VUELO;
DELETE FROM AEROLINEA;
DELETE FROM OPERADORA;
DELETE FROM AVION;
```

```
DELETE FROM AEROPUERTO;
DELETE FROM HORA;
DELETE FROM FECHA;
COMMIT;
```

A diferencia de la práctica anterior, en la que se utilizaron datos ficticios para el proceso de inserción y análisis de datos, que permitió comprobar la corrección del esquema en estrella desarrollado, en esta ocasión el data mart se puebla con datos reales obtenidos del fichero *datos\_vuelos.xlsx*, procesados y transformados mediante **KNIME** en el proceso ETL.

De este modo, los registros insertados reflejan vuelos reales del mes de marzo de 2025, junto con la información asociada a aeropuertos, aviones, aerolíneas y operadoras extraída de las fuentes **GeoNames** y **RITA**.

Las inserciones se realizan una vez generados los ficheros finales desde **KNIME**, asegurando la coherencia entre las claves primarias y foráneas del esquema en estrella y la consistencia con los atributos derivados.

Dado el elevado volumen de datos a insertar tanto en las dimensiones como en la tabla de hechos, se optó por automatizar el proceso mediante scripts en Python (.py). Estos scripts generan automáticamente las sentencias SQL de inserción correspondientes, que posteriormente se unificaron en el fichero *insertar\_datos\_vuelos.sql* para centralizar el proceso de carga. A continuación, se muestran los diferentes ficheros de generación de inserciones automáticas:

```
SQL
----- DIMENSIÓN FECHA -----
import csv

def generar_inserts(archivo_entrada='fecha.csv', archivo_salida='inserts_fecha.sql'):
    """
    Lee un archivo CSV de fechas y genera un archivo SQL con sentencias INSERT.

    Args:
        archivo_entrada: Ruta al archivo CSV de entrada
        archivo_salida: Ruta al archivo SQL de salida
    """
    try:
        with open(archivo_entrada, 'r', encoding='utf-8') as f_entrada:
            # Leer el CSV
            reader = csv.DictReader(f_entrada)

            # Abrir archivo de salida
            with open(archivo_salida, 'w', encoding='utf-8') as f_salida:
                for row in reader:
                    # Extraer los valores del CSV (con espacios eliminados)
                    fecha_completa = row['fecha_completa'].strip()
                    agno = row['agno'].strip()
                    mes = row['mes'].strip()
                    dia = row['dia'].strip()
                    nombre_dia = row['nombre_dia'].strip()
                    nombre_mes = row['nombre_mes'].strip()
                    trimestre = row['trimestre'].strip()

                    f_salida.write(f"INSERT INTO FECHA (fecha_completa, agno, mes, dia, nombre_dia, nombre_mes, trimestre) VALUES ('{fecha_completa}', '{agno}', '{mes}', '{dia}', '{nombre_dia}', '{nombre_mes}', '{trimestre}');\n")
```

```

temporada = row['temporada'].strip()
es_fin_semana = row['es_fin_semana'].strip()
id_fecha = row['id_fecha'].strip()

# Generar la sentencia INSERT siguiendo el orden de la tabla
insert = (
    f"INSERT INTO FECHA VALUES "
    f"({{id_fecha}} , {{fecha_completa}} , {{dia}} , "
    f"{{mes}} , {{agno}} , {{nombre_dia}} , "
    f"{{nombre_mes}} , {{trimestre}} , "
    f"{{es_fin_semana}} , {{temporada}} );\n"
)
f_salida.write(insert)

print(f"Archivo generado exitosamente: {{archivo_salida}}")

except FileNotFoundError:
    print(f"Error: No se encontró el archivo '{{archivo_entrada}}'")
except KeyError as e:
    print(f"Error: Columna no encontrada en el CSV: {{e}}")
except Exception as e:
    print(f"Error inesperado: {{e}}")

if __name__ == "__main__":
    # Ejecutar la función
    generar_inserts()

----- DIMENSIÓN HORA -----
import csv

def generar_inserts(archivo_entrada='hora.csv', archivo_salida='inserts_hora.sql'):
    """
    Lee un archivo CSV de horas y genera un archivo SQL con sentencias INSERT.

    Args:
        archivo_entrada: Ruta al archivo CSV de entrada
        archivo_salida: Ruta al archivo SQL de salida
    """
    try:
        with open(archivo_entrada, 'r', encoding='utf-8') as f_entrada:
            # Leer el CSV
            reader = csv.DictReader(f_entrada)

            # Abrir archivo de salida
            with open(archivo_salida, 'w', encoding='utf-8') as f_salida:
                for row in reader:
                    # Extraer los valores del CSV (con espacios eliminados)
                    hora_completa = row['hora_completa'].strip()
                    hora = row['hora'].strip()
                    minuto = row['minuto'].strip()
                    segundo = row['segundo'].strip()
                    franja_horaria = row['franja_horaria'].strip()
                    es_hora_punta = row['es_hora_punta'].strip()
                    id_hora = row['id_hora'].strip()

                    # Generar la sentencia INSERT
                    insert = (
                        f"INSERT INTO HORA VALUES "
                        f"({{id_hora}} , {{hora_completa}} , {{hora}} , "
                        f"{{minuto}} , {{segundo}} , "

```

```

        f"' {franja_horaria}', '{es_hora_punta}');\n"
    )

    f_salida.write(insert)

    print(f"Archivo generado exitosamente: {archivo_salida}")

except FileNotFoundError:
    print(f"Error: No se encontró el archivo '{archivo_entrada}'")
except KeyError as e:
    print(f"Error: Columna no encontrada en el CSV: {e}")
except Exception as e:
    print(f"Error inesperado: {e}")

if __name__ == "__main__":
    # Ejecutar la función
    generar_inserts()

----- DIMENSIÓN AEROPUERTO -----
import csv

def generar_inserts(archivo_entrada='aeropuerto.csv', archivo_salida='inserts_aeropuerto.sql'):
    """
    Lee un archivo CSV de aeropuertos y genera un archivo SQL con sentencias INSERT.

    Args:
        archivo_entrada: Ruta al archivo CSV de entrada
        archivo_salida: Ruta al archivo SQL de salida
    """
    try:
        with open(archivo_entrada, 'r', encoding='utf-8') as f_entrada:
            # Leer el CSV
            reader = csv.DictReader(f_entrada)

            # Diccionario de códigos para territorios
            codigos_territorios = {
                'Puerto Rico': 'PR',
                'Guam': 'GU',
                'American Samoa': 'AS',
                'Northern Mariana Islands': 'MP',
                'Virgin Islands': 'VI'
            }

            # Abrir archivo de salida
            with open(archivo_salida, 'w', encoding='utf-8') as f_salida:
                for row in reader:
                    # Extraer los valores del CSV (con espacios eliminados)
                    codigo_IATA = row['codigo_IATA'].strip()
                    nombre_aeropuerto = row['nombre_aeropuerto'].strip().replace(" ", " ")
                    pais = row['pais'].strip().replace(" ", " ")
                    estado = row['estado'].strip().replace(" ", " ")
                    codigo_estado = row['codigo_estado'].strip()
                    ciudad = row['ciudad'].strip().replace(" ", " ")
                    latitud = row['latitud'].strip()
                    longitud = row['longitud'].strip()
                    id_aeropuerto = row['id_aeropuerto'].strip()
                    zona_horaria = row['zona_horaria'].strip()
                    num_habitantes = row['num_habitantes'].strip() if
row['num_habitantes'].strip() else 'NULL'

```

```

# Si estado está vacío, usar el nombre del país como estado
if not estado or estado == '':
    estado = pais

# Si código de estado está vacío, buscar en el diccionario o usar el del país
if not codigo_estado or codigo_estado == '':
    codigo_estado = codigos_territorios.get(pais, pais[:2].upper())

# Generar la sentencia INSERT
insert = (
    f"INSERT INTO AEROPUERTO VALUES "
    f"({id_aeropuerto}, '{codigo_IATA}', '{nombre_aeropuerto}', "
    f"'{ciudad}', '{estado}', "
    f"'{codigo_estado}', '{pais}', "
    f"{latitud}, {longitud}, {num_habitantes}, '{zona_horaria}')\n"
)
f_salida.write(insert)

print(f"Archivo generado exitosamente: {archivo_salida}")

except FileNotFoundError:
    print(f"Error: No se encontró el archivo '{archivo_entrada}'")
except KeyError as e:
    print(f"Error: Columna no encontrada en el CSV: {e}")
except Exception as e:
    print(f"Error inesperado: {e}")

if __name__ == "__main__":
    # Ejecutar la función
    generar_inserts()

----- DIMENSIÓN AVIÓN -----
import csv

def generar_inserts(archivo_entrada='avion.csv', archivo_salida='inserts_avion.sql'):
    """
    Lee un archivo CSV de aviones y genera un archivo SQL con sentencias INSERT.

    Args:
        archivo_entrada: Ruta al archivo CSV de entrada
        archivo_salida: Ruta al archivo SQL de salida
    """
    try:
        with open(archivo_entrada, 'r', encoding='utf-8') as f_entrada:
            # Leer el CSV
            reader = csv.DictReader(f_entrada)

            # Abrir archivo de salida
            with open(archivo_salida, 'w', encoding='utf-8') as f_salida:
                for row in reader:
                    # Extraer los valores del CSV (con espacios eliminados)
                    matricula = row['matricula'].strip()
                    agno = row['agno_fabricacion'].strip()
                    activo = row['esta_activo'].strip()
                    capacidad = row['capacidad_maxima'].strip()
                    fabricante = row['fabricante'].strip()
                    codigo_modelo = row['codigo_modelo'].strip()
                    nombre_modelo = row['nombre_modelo'].strip()

```

```

        id_avion = row['id_avion'].strip()

        # Generar la sentencia INSERT
        insert = (
            f"INSERT INTO AVION VALUES "
            f"({id_avion}, '{matricula}', '{codigo_modelo}', "
            f"'{nombre_modelo}', '{fabricante}', {capacidad}, "
            f"{agno}, '{activo}')\n"
        )

        f_salida.write(insert)

        print(f"Archivo generado exitosamente: {archivo_salida}")

    except FileNotFoundError:
        print(f"Error: No se encontró el archivo '{archivo_entrada}'")
    except KeyError as e:
        print(f"Error: Columna no encontrada en el CSV: {e}")
    except Exception as e:
        print(f"Error inesperado: {e}")

if __name__ == "__main__":
    # Ejecutar la función
    generar_inserts()

----- DIMENSIÓN OPERADORA -----
import csv

def generar_inserts(archivo_entrada='operadora.csv', archivo_salida='inserts_operadora.sql'):
    """
    Lee un archivo CSV de fechas y genera un archivo SQL con sentencias INSERT.

    Args:
        archivo_entrada: Ruta al archivo CSV de entrada
        archivo_salida: Ruta al archivo SQL de salida
    """
    try:
        with open(archivo_entrada, 'r', encoding='utf-8') as f_entrada:
            # Leer el CSV
            reader = csv.DictReader(f_entrada)

            # Abrir archivo de salida
            with open(archivo_salida, 'w', encoding='utf-8') as f_salida:
                for row in reader:
                    # Extraer los valores del CSV (con espacios eliminados)
                    nombre_operadora = row['nombre_operadora'].strip()
                    esta_activo = row['esta_activo'].strip()
                    codigo_IATA = row['codigo_IATA'].strip()
                    id_operadora = row['id_operadora'].strip()

                    # Generar la sentencia INSERT
                    insert = (
                        f"INSERT INTO OPERADORA VALUES "
                        f"({id_operadora}, '{codigo_IATA}', "
                        f"'{nombre_operadora}', '{esta_activo}')\n"
                    )

                    f_salida.write(insert)

                print(f"Archivo generado exitosamente: {archivo_salida}")

    except FileNotFoundError:
        print(f"Error: No se encontró el archivo '{archivo_entrada}'")
    except KeyError as e:
        print(f"Error: Columna no encontrada en el CSV: {e}")
    except Exception as e:
        print(f"Error inesperado: {e}")

```

```

except FileNotFoundError:
    print(f"Error: No se encontró el archivo '{archivo_entrada}'")
except KeyError as e:
    print(f"Error: Columna no encontrada en el CSV: {e}")
except Exception as e:
    print(f"Error inesperado: {e}")

if __name__ == "__main__":
    # Ejecutar la función
    generar_inserts()

----- DIMENSIÓN AEROLÍNEA -----
import csv

def generar_inserts(archivo_entrada='aerolinea.csv', archivo_salida='inserts_aerolinea.sql'):
    """
    Lee un archivo CSV de aerolíneas y genera un archivo SQL con sentencias INSERT.

    Args:
        archivo_entrada: Ruta al archivo CSV de entrada
        archivo_salida: Ruta al archivo SQL de salida
    """
    try:
        with open(archivo_entrada, 'r', encoding='utf-8') as f_entrada:
            # Leer el CSV
            reader = csv.DictReader(f_entrada)

            # Abrir archivo de salida
            with open(archivo_salida, 'w', encoding='utf-8') as f_salida:
                for row in reader:
                    # Extraer los valores del CSV (con espacios eliminados)
                    codigo_dot = row['codigo_DOT'].strip()
                    nombre = row['nombre_aerolinea'].strip()
                    activo = row['esta_activo'].strip()
                    id_aerolinea = row['id_aerolinea'].strip()

                    # Generar la sentencia INSERT con el orden solicitado
                    insert = (
                        f"INSERT INTO AEROLINEA VALUES "
                        f"({id_aerolinea}, '{codigo_dot}', '{nombre}', '{activo}');\n"
                    )

                    f_salida.write(insert)

    except FileNotFoundError:
        print(f"Error: No se encontró el archivo '{archivo_entrada}'")
    except KeyError as e:
        print(f"Error: Columna no encontrada en el CSV: {e}")
    except Exception as e:
        print(f"Error inesperado: {e}")

if __name__ == "__main__":
    # Ejecutar la función
    generar_inserts()

----- TABLA DE HECHOS VUELO -----
import csv

def generar_inserts(archivo_entrada='vuelos.csv', archivo_salida='inserts_vuelos.sql'):

```

```

"""
Lee un archivo CSV de vuelos y genera un archivo SQL con sentencias INSERT.
Solo inserta vuelos con datos completos.

Args:
    archivo_entrada: Ruta al archivo CSV de entrada
    archivo_salida: Ruta al archivo SQL de salida
"""

try:
    with open(archivo_entrada, 'r', encoding='utf-8') as f_entrada:
        # Leer el CSV
        reader = csv.DictReader(f_entrada)

        # Contadores
        total_filas = 0
        filas_insertadas = 0
        filas OMITIDAS = 0

        # Abrir archivo de salida
        with open(archivo_salida, 'w', encoding='utf-8') as f_salida:
            for row in reader:
                total_filas += 1

                # Extraer los valores del CSV (con espacios eliminados)
                codigo_vuelo = row['codigo_vuelo'].strip().replace(" ", " ")
                id_fecha = row['id_fecha'].strip()
                id_hora = row['id_hora'].strip()
                id_avion = row['id_avion'].strip()
                id_aeropuerto_origen = row['id_aeropuerto_origen'].strip()
                id_aeropuerto_destino = row['id_aeropuerto_destino'].strip()
                id_aerolinea = row['id_aerolinea'].strip()
                id_operadora = row['id_operadora'].strip()
                tiempo_estimado_minutos = row['tiempo_estimado_minutos'].strip()
                tiempo_real_minutos = row['tiempo_real_minutos'].strip()
                retardo_minutos = row['retardo_minutos'].strip()
                distancia_km = row['distancia_km'].strip()

                # Verificar que todos los campos obligatorios estén completos
                if not all([codigo_vuelo, id_fecha, id_hora, id_avion,
                           id_aeropuerto_origen, id_aeropuerto_destino,
                           id_aerolinea, id_operadora, tiempo_estimado_minutos,
                           tiempo_real_minutos, retardo_minutos, distancia_km]):
                    filas OMITIDAS += 1
                    print(f"Fila {total_filas} OMITIDA - Datos incompletos: {codigo_vuelo}")
                    continue

                # Generar la sentencia INSERT siguiendo el orden del CREATE TABLE
                insert = (
                    f"INSERT INTO VUELO VALUES "
                    f"({codigo_vuelo}, {id_fecha}, {id_hora}, "
                    f"{id_avion}, {id_aeropuerto_origen}, {id_aeropuerto_destino}, "
                    f"{id_aerolinea}, {id_operadora}, "
                    f"{tiempo_estimado_minutos}, {tiempo_real_minutos}, "
                    f"{retardo_minutos}, {distancia_km});\n"
                )

                f_salida.write(insert)
                filas_insertadas += 1

            print(f"\n{'='*50}")
            print(f"Archivo generado exitosamente: {archivo_salida}")

```

```

        print(f"Total de filas procesadas: {total_filas}")
        print(f"Filas insertadas: {filas_insertadas}")
        print(f"Filas omitidas (datos incompletos): {filas OMITIDAS}")
        print(f'{'*50}')

    except FileNotFoundError:
        print(f"Error: No se encontró el archivo '{archivo_entrada}'")
    except KeyError as e:
        print(f"Error: Columna no encontrada en el CSV: {e}")
    except Exception as e:
        print(f"Error inesperado: {e}")

if __name__ == "__main__":
    # Ejecutar la función
    generar_inserts()

```

### 6.3. Script de consultas

**Consulta 1 (Aerolíneas con más de 1 vuelo):** Evalúa la actividad y puntualidad de las aerolíneas para aquellas que hayan hecho más de 1 vuelo.

SQL

```

SELECT
    a.nombre_aerolinea,
    COUNT(*) as total_vuelos,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(AVG(v.tiempo_real_minutos), 2) as duracion_promedio,
    ROUND(SUM(v.distancia_km), 2) as distancia_total_km
FROM VUELO v
JOIN AEROLINEA a ON v.id_aerolinea = a.id_aerolinea
GROUP BY a.nombre_aerolinea
HAVING COUNT(*) > 1
ORDER BY total_vuelos DESC;

```

NOMBRE_AEROLINEA	TOTAL_VUELOS	RETRASO_PROMEDIO	DURACION_PROMEDIO	DISTANCIA_TOTAL_KM
Republic Airline	168	4.91	117.93	147879.03
PSA Airlines Inc.	153	5.95	102.76	105785.14
SkyWest Airlines Inc.	142	2.82	97.11	90337.08
Mesa Airlines Inc.	139	2.91	122.35	136121.2
Spirit Air Lines	131	2.83	153.03	194788.08
Envoy Air	128	6.39	129.1	137403.84
Southwest Airlines Co.	124	1.03	131.74	166180.45
Delta Air Lines Inc.	117	3.89	166.24	191992.65
GoJet Airlines LLC d/b/a United Express	115	3.04	88.93	60591.65
Frontier Airlines Inc.	108	4.31	159.67	161270.35
Alaska Airlines Inc.	93	3.61	205.11	201783.88
United Air Lines Inc.	93	3.71	149.86	130474.02

American Airlines Inc.	75	4.08	349.37	302048.98
Hawaiian Airlines Inc.	66	4.45	332.08	265985.28
JetBlue Airways	10	1	181.2	19466.58

Republic Airline tiene más vuelos (168) pero Southwest es la más puntual con solo 1 minuto de retraso. Hawaiian y American hacen los vuelos más largos.

**Consulta 2 (Análisis jerárquico por aerolínea y operadora):** Muestra quién vende el billete (aerolínea) y quién realmente hace volar el avión (operadora), utilizando *ROLLUP* para generar subtotales automáticos.

SQL

```

SELECT
    a.nombre_aerolinea as aerolinea,
    o.nombre_operadora as operadora,
    COUNT(*) as num_vuelos,
    ROUND(SUM(v.distancia_km), 2) as distancia_total_km,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(AVG(v.tiempo_real_minutos), 2) as duracion_promedio
FROM VUELO v
JOIN AEROLINEA a ON v.id_aerolinea = a.id_aerolinea
JOIN OPERADORA o ON v.id_operadora = o.id_operadora
GROUP BY ROLLUP(a.nombre_aerolinea, o.nombre_operadora)
ORDER BY a.nombre_aerolinea NULLS LAST, o.nombre_operadora NULLS LAST;

```

AEROLÍNEA	OPERADORA	NUM_VUELOS	DISTANCIA_TOTAL_KM	RETRASO_PROMEDIO	DURACION_PROMEDIO
Alaska Airlines Inc.	Alaska Airlines Inc.	93	201,783.88	3.61	205.11
Alaska Airlines Inc.	Subtotal	93	201,783.88	3.61	205.11
American Airlines Inc.	American Airlines Inc.	75	302,048.98	4.08	349.37
American Airlines Inc.	Subtotal	75	302,048.98	4.08	349.37
Delta Air Lines Inc.	Delta Air Lines Inc.	117	191,992.65	3.89	166.24
Delta Air Lines Inc.	Subtotal	117	191,992.65	3.89	166.24
Envoy Air	Envoy Air	128	137,403.84	6.39	129.1
Envoy Air	Subtotal	128	137,403.84	6.39	129.1
Frontier Airlines Inc.	Frontier Airlines Inc.	108	161,270.35	4.31	159.67
Frontier Airlines Inc.	Subtotal	108	161,270.35	4.31	159.67
GoJet Airlines LLC d/b/a United Express	SkyWest Airlines Inc.	115	60,591.65	3.04	88.93
GoJet Airlines LLC d/b/a United Express	Subtotal	115	60,591.65	3.04	88.93
Hawaiian Airlines Inc.	Hawaiian Airlines Inc.	66	265,985.28	4.45	332.08
Hawaiian Airlines Inc.	Subtotal	66	265,985.28	4.45	332.08
JetBlue Airways	JetBlue Airways	10	19,466.58	1	181.2
JetBlue Airways	Subtotal	10	19,466.58	1	181.2
Mesa Airlines Inc.	SkyWest Airlines Inc.	139	136,121.2	2.91	122.35

Mesa Airlines Inc.	Subtotal	139	136,121.2	2.91	122.35
PSA Airlines Inc.	PSA Airlines Inc.	153	105,785.14	5.95	102.76
PSA Airlines Inc.	Subtotal	153	105,785.14	5.95	102.76
Republic Airline	Republic Airline	168	147,879.03	4.91	117.93
Republic Airline	Subtotal	168	147,879.03	4.91	117.93
SkyWest Airlines Inc.	SkyWest Airlines Inc.	142	90,337.08	2.82	97.11
SkyWest Airlines Inc.	Subtotal	142	90,337.08	2.82	97.11
Southwest Airlines Co.	Southwest Airlines Co.	124	166,180.45	1.03	131.74
Southwest Airlines Co.	Subtotal	124	166,180.45	1.03	131.74
Spirit Air Lines	Spirit Air Lines	131	194,788.08	2.83	153.03
Spirit Air Lines	Subtotal	131	194,788.08	2.83	153.03
United Air Lines Inc.	United Air Lines Inc.	93	130,474.02	3.71	149.86
United Air Lines Inc.	Subtotal	93	130,474.02	3.71	149.86
<b>TOTAL GENERAL</b>		<b>1,662</b>	<b>2,312,108.2</b>	<b>3.86</b>	<b>149.88</b>

La mayoría de aerolíneas operan sus propios vuelos, excepto GoJet y Mesa que vuelan bajo la marca de otras compañías (operadas por SkyWest). Southwest sigue siendo la más puntual con 1.03 minutos de retraso.

**Consulta 3 (Análisis por temporada y fin de semana):** Analiza la puntualidad según la época del año y si es fin de semana o no, usando CUBE para mostrar todas las combinaciones.

SQL

```

SELECT
    f.temporada,
    f.es_fin_semana,
    COUNT(*) as total_vuelos,
    ROUND(AVG(v.tiempo_real_minutos), 2) as duracion_promedio,
    ROUND(AVG(v.retraso_minutos), 2) as retraso_promedio,
    ROUND(SUM(v.distancia_km), 2) as distancia_total,
    ROUND(AVG(v.tiempo_real_minutos - v.tiempo_estimado_minutos), 2) as diferencia_tiempo
FROM VUELO v
JOIN FECHA f ON v.id_fecha = f.id_fecha
GROUP BY CUBE(f.temporada, f.es_fin_semana)
ORDER BY f.temporada NULLS LAST, f.es_fin_semana NULLS LAST;

```

TEMPORADA	ES_FIN_SEMANA	TOTAL_VUELOS	DURACION_PROMEDIO	RETRASO_PROMEDIO	DISTANCIA_TOTAL	DIFERENCIA_TIEMPO
Invierno	Fin de semana	315	154.33	4.18	457,652.84	-4.71
Invierno	Entre semana	713	147.52	3.63	968,151.59	-5.69
Invierno	Subtotal	1,028	149.61	3.8	1,425,804.43	-5.39
Primavera	Fin de semana	219	150.25	5.04	301,437.43	-4.15
Primavera	Entre semana	415	150.36	3.39	584,866.34	-5.54
Primavera	Subtotal	634	150.32	3.96	886,303.77	-5.06
Total por tipo	Fin de semana	534	152.66	4.53	759,090.27	-4.48

Total por tipo	Entre semana	1,128	148.57	3.55	1,553,017.93	-5.63
<b>TOTAL GENERAL</b>		<b>1,662</b>	<b>149.88</b>	<b>3.86</b>	<b>2,312,108.2</b>	<b>-5.26</b>

Hay más vuelos en invierno (1028) que en primavera (634), y los fines de semana tienen más retrasos (4.53 min) que entre semana (3.55 min). Los vuelos entre semana son ligeramente más rápidos que los de fin de semana.

**Consulta 4 (Aeropuertos con distancia promedio mayor a 600km):** Identifica los aeropuertos que operan rutas con una distancia media superior a 600km.

SQL

```

SELECT
    ap.ciudad,
    ap.nombre_aeropuerto,
    COUNT(*) as num_vuelos,
    ROUND(AVG(v.distancia_km), 2) as distancia_promedio_km,
    ROUND(MIN(v.distancia_km), 2) as distancia_minima,
    ROUND(MAX(v.distancia_km), 2) as distancia_maxima,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio
FROM VUELO v
JOIN AEROPUERTO ap ON v.id_aeropuerto_origen = ap.id_aeropuerto
GROUP BY ap.ciudad, ap.nombre_aeropuerto
HAVING AVG(v.distancia_km) > 600
ORDER BY distancia_promedio_km DESC;

```

CIUDAD	NOMBRE_AEROPUERTO	NUM_VUELOS	DISTANCIA_PROMEDIO_KM	DISTANCIA_MINIMA	DISTANCIA_MAXIMA	RETRASO_PROMEDIO
Salt Lake City, UT	Salt Lake City International	8	3,661.45	629.25	4,818.36	6.88
Washington, DC	Washington Dulles International	1	3,625.84	3,625.84	3,625.84	2
Los Angeles, CA	Ontario International	4	3,526.47	1,538.53	4,189.11	16.5
Los Angeles, CA	Orange County	6	3,518.02	1,361.5	3,949.32	3.5
New York City, NY	John F. Kennedy International	58	3,065.85	300.95	4,161.75	4.33
Kona, HI	Keahole	7	3,022.34	262.32	4,372.58	0.71
San Francisco, CA	Metropolitan Oakland International	12	2,706.24	542.35	3,954.15	0.75
Anchorage, AK	Anchorage International	4	2,674.72	2,330.32	3,707.92	6.25
Los Angeles, CA	Long Beach Daugherty Field	5	2,668.61	371.76	4,134.39	0
San Juan, Puerto Rico	Puerto Rico International	4	2,530.69	2,489.65	2,571.73	0.25
Fairbanks, AK	Fairbanks International	1	2,467.12	2,467.12	2,467.12	0
Washington, DC	Friendship International	6	2,453.17	606.72	3,748.15	0
Los Angeles, CA	Los Angeles International	74	2,371.73	175.42	4,208.42	2.09
San Francisco, CA	San Jose International	8	2,243.22	495.68	3,889.77	1.75
Honolulu, HI	Honolulu International	48	2,235.91	160.93	4,818.36	1.98
... (111 aeropuertos más)						

Salt Lake City tiene los vuelos más largos en promedio (3661 km), mientras que Ontario (LA) tiene los peores retrasos con 16.5 minutos.

**Consulta 5, 6 y 7 (Análisis por niveles geográficos):** Estas tres consultas implementan un análisis jerárquico (*DRILL DOWN*), que permite ver la información a diferentes niveles de detalle geográfico: desde estados de origen hasta los aeropuertos específicos.

## SQL

```
-- 5. Análisis por estado de ORIGEN (DRILL DOWN Nivel 1)
SELECT
    ap.estado,
    COUNT(*) as total_vuelos,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(SUM(v.distancia_km), 2) as distancia_total,
    ROUND(AVG(v.tiempo_real_minutos), 2) as duracion_promedio
FROM VUELO v
JOIN AEROPUERTO ap ON v.id_aeropuerto_origen = ap.id_aeropuerto
GROUP BY ap.estado
ORDER BY total_vuelos DESC;

-- 6. Análisis por estado → ciudad de ORIGEN (DRILL DOWN Nivel 2)
SELECT
    ap.estado,
    ap.ciudad,
    COUNT(*) as total_vuelos,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(SUM(v.distancia_km), 2) as distancia_total,
    ROUND(AVG(v.tiempo_real_minutos), 2) as duracion_promedio
FROM VUELO v
JOIN AEROPUERTO ap ON v.id_aeropuerto_origen = ap.id_aeropuerto
GROUP BY ap.estado, ap.ciudad
ORDER BY ap.estado, total_vuelos DESC;

-- 7. Análisis por estado → ciudad → aeropuerto de ORIGEN (DRILL DOWN Nivel 3)
SELECT
    ap.estado,
    ap.ciudad,
    ap.nombre_aeropuerto,
    COUNT(*) as total_vuelos,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(SUM(v.distancia_km), 2) as distancia_total
FROM VUELO v
JOIN AEROPUERTO ap ON v.id_aeropuerto_origen = ap.id_aeropuerto
GROUP BY ap.estado, ap.ciudad, ap.nombre_aeropuerto
ORDER BY ap.estado, ap.ciudad, total_vuelos DESC;
```

ESTADO	TOTAL_VUELOS	RETRASO_PROMEDIO	DISTANCIA_TOTAL	DURACION_PROMEDIO
California	217	2.93	432,413.56	185.86
Colorado	171	2.53	155,407.53	112.01
Florida	122	7.87	181,266.4	165.45
Illinois	117	2.76	95,665.61	106.53
New York	116	4.81	238,546.03	217.36
Virginia	93	5.16	75,283.32	112.68
Hawaii	90	1.26	184,792.47	172.84

Texas	82	3.91	106,733.04	138.76
Georgia	60	3.68	91,676.05	163.58
North Carolina	47	5.7	33,551.52	106.04
Michigan	43	1.98	54,892.98	142.65
Washington	37	2.51	73,482.46	185.16
Massachusetts	34	3.38	50,295.09	175.29
Arizona	33	5.09	52,169.97	162.52
Kentucky	33	8.61	38,810.84	137.55
... (33 estados más)				

**NIVEL 1** - California tiene más vuelos (217) pero Florida tiene peores retrasos (7.87 min). Idaho y Louisiana son los más puntuales con menos de 1 minuto de retraso.

ESTADO	CIUDAD	TOTAL_VUELOS	RETRASO_PROMEDIO	DISTANCIA_TOTAL	DURACION_PROMEDIO
California	Los Angeles, CA (Metropolitan Area)	91	2.74	225,904.67	217.99
California	San Francisco, CA (Metropolitan Area)	81	3.25	150,938.39	177.28
California	San Diego, CA	21	3.67	41,334.29	190.62
Colorado	Denver, CO	121	2.24	128,330.38	118.54
Colorado	Aspen, CO	37	2.84	18,813.18	94.59
Florida	Miami, FL (Metropolitan Area)	47	8.38	72,145.1	170.62
Florida	Orlando, FL	26	5.96	38,400.46	161.69
Illinois	Chicago, IL	115	2.7	93,472.08	105.89
New York	New York City, NY (Metropolitan Area)	107	4.65	232,189.14	226.4
Virginia	Washington, DC (Metropolitan Area)	87	4.98	71,190.76	112.69
Georgia	Atlanta, GA (Metropolitan Area)	55	3.98	86,878.61	167.29
Hawaii	Honolulu, HI	48	1.98	107,323.67	187.17
Michigan	Detroit, MI	35	0.8	51,682.34	153.94
Massachusetts	Boston, MA (Metropolitan Area)	34	3.38	50,295.09	175.29
Washington	Seattle, WA	33	2.24	67,918.98	188.79
... (129 combinaciones estado-ciudad más)					

**NIVEL 2** - Los Ángeles domina California con 91 vuelos. Miami tiene los peores retrasos de Florida (8.38 min). Denver lidera Colorado con 121 vuelos.

ESTADO	CIUDAD	AEROPUERTO	TOTAL_VUELOS	RETRASO_PROMEDIO	DISTANCIA_TOTAL
Colorado	Denver, CO	Stapleton International	121	2.24	128,330.38
Illinois	Chicago, IL	Chicago O'Hare International	114	2.72	92,454.97
Virginia	Washington, DC (Metropolitan Area)	Ronald Reagan Washington National	86	5.01	67,564.92

California	Los Angeles, CA (Metropolitan Area)	Los Angeles International	74	2.09	175,508.18
California	San Francisco, CA (Metropolitan Area)	San Francisco International	61	3.93	100,517.77
New York	New York City, NY (Metropolitan Area)	John F. Kennedy International	58	4.33	177,819.2
Georgia	Atlanta, GA (Metropolitan Area)	Atlanta Municipal	55	3.98	86,878.61
Hawaii	Honolulu, HI	Honolulu International	48	1.98	107,323.67
New York	New York City, NY (Metropolitan Area)	LaGuardia	43	5.05	49,121.88
Colorado	Aspen, CO	Aspen Pitkin County Sardy Field	37	2.84	18,813.18
Michigan	Detroit, MI	Detroit Metro Wayne County	35	0.8	51,682.34
Massachusetts	Boston, MA (Metropolitan Area)	Logan International	34	3.38	50,295.09
Washington	Seattle, WA	Seattle International	33	2.24	67,918.98
Texas	Dallas/Fort Worth, TX	Dallas Fort Worth Regional	32	1.56	35,046.6
Florida	Miami, FL (Metropolitan Area)	Miami International	30	12.7	43,877.05
... (144 aeropuertos más)					

**NIVEL 3** - Los Angeles International es el aeropuerto más activo (74 vuelos). Ontario International tiene los peores retrasos (16.5 min). Chicago O'Hare es el segundo más activo con 114 vuelos.

**Consulta 8 (Análisis por año, trimestre y mes):** Muestra cómo se distribuyen los vuelos a lo largo del año, con totales automáticos por trimestre y año completo usando *ROLLUP*.

SQL

```

SELECT
    f.agno,
    f.trimestre,
    f.nombre_mes,
    COUNT(*) as num_vuelos,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(SUM(v.distancia_km), 2) as km_totales,
    ROUND(AVG(v.tiempo_real_minutos), 2) as duracion_promedio
FROM VUELO v
JOIN FECHA f ON v.id_fecha = f.id_fecha
GROUP BY ROLLUP(f.agno, f.trimestre, f.nombre_mes)

ORDER BY f.agno NULLS LAST, f.trimestre NULLS LAST, f.nombre_mes NULLS LAST;

```

AGNO	TRIMESTRE	NOMBRE_MES	NUM_VUELOS	RETRASO_PROMEDIO	KM_TOTALES	DURACION_PROMEDIO
2025	1	Marzo	1,662	3.86	2,312,108.2	149.88
2025	1	Subtotal Trimestre 1	1,662	3.86	2,312,108.2	149.88
2025	Subtotal Año 2025		1,662	3.86	2,312,108.2	149.88
<b>TOTAL GENERAL</b>			<b>1,662</b>	<b>3.86</b>	<b>2,312,108.2</b>	<b>149.88</b>

Todos los vuelos son de marzo de 2025 (1662 vuelos totales), con un retraso promedio de 3.86 minutos y una distancia total de 2.3 millones de km. El ROLLUP genera subtotales por año, trimestre y mes.

**Consulta 9 (Análisis por hora del día y fines de semana):** Compara la puntualidad según la hora del día (madrugada, mañana, tarde y noche) y si es fin de semana o no, usando *CUBE* para obtener todas las combinaciones.

SQL

```

SELECT
    h.franja_horaria,
    f.es_fin_semana,
    COUNT(*) as total_vuelos,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(AVG(v.tiempo_real_minutos - v.tiempo_estimado_minutos), 2) as diferencia_tiempo,
    ROUND(SUM(v.distancia_km), 2) as distancia_total,
    ROUND(AVG(v.distancia_km), 2) as distancia_promedio
FROM VUELO v
JOIN HORA h ON v.id_hora = h.id_hora
JOIN FECHA f ON v.id_fecha = f.id_fecha
GROUP BY CUBE(h.franja_horaria, f.es_fin_semana)
ORDER BY h.franja_horaria NULLS LAST, f.es_fin_semana NULLS LAST;

```

FRANJA HORARIA	ES_FIN_SEMANA	TOTAL_VUELOS	RETRASO PROMEDIO	DIFERENCIA TIEMPO	DISTANCIA TOTAL	DISTANCIA PROMEDIO
00:00 - 04:59	Fin de semana	4	4.75	-3.25	9,067.02	2,266.76
00:00 - 04:59	Entre semana	4	0	-19.5	9,046.1	2,261.53
00:00 - 04:59	Subtotal	8	2.38	-11.38	18,113.12	2,264.14
05:00 - 11:59	Fin de semana	233	3.86	-4.96	348,436.59	1,495.44
05:00 - 11:59	Entre semana	482	2.93	-6.01	695,997.71	1,443.98
05:00 - 11:59	Subtotal	715	3.23	-5.67	1,044,434.3	1,460.75
12:00 - 19:59	Fin de semana	247	5.34	-4.19	313,718.3	1,270.11
12:00 - 19:59	Entre semana	529	4.29	-4.76	670,983.74	1,268.4
12:00 - 19:59	Subtotal	776	4.63	-4.58	984,702.04	1,268.95
20:00 - 23:59	Fin de semana	50	3.68	-3.76	87,868.35	1,757.37
20:00 - 23:59	Entre semana	113	2.8	-7.63	176,990.39	1,566.29
20:00 - 23:59	Subtotal	163	3.07	-6.44	264,858.74	1,624.9
Total por tipo	Fin de semana	534	4.53	-4.48	759,090.27	1,421.52
Total por tipo	Entre semana	1,128	3.55	-5.63	1,553,017.93	1,376.79
<b>TOTAL GENERAL</b>		<b>1,662</b>	<b>3.86</b>	<b>-5.26</b>	<b>2,312,108.2</b>	<b>1,391.16</b>

Los vuelos de tarde (12:00-19:59) tienen más retrasos (4.63 min), mientras que la madrugada (00:00-04:59) es la más puntual (2.38 min). Los fines de semana siempre tienen más retrasos que entre semana en todas las franjas horarias.

**Consulta 10 (Modelos de avión modernos y usados):** Muestra los modelos de avión que se usan con más frecuencia (más de 1 vuelo), calculando su velocidad efectiva promedio en base a la distancia recorrida y tiempo real de vuelo, usando HAVING para filtrar por frecuencia de uso.

## SQL

```

SELECT
    av.fabricante,
    av.nombre_modelo,
    av.capacidad_maxima,
    av.agno_fabricacion,
    COUNT(*) as vuelos_realizados,
    ROUND(AVG(v.distancia_km), 2) as distancia_promedio,
    ROUND(AVG(v.tiempo_real_minutos), 2) as duracion_real_promedio,
    ROUND(AVG(v.retardo_minutos), 2) as retraso_promedio,
    ROUND(AVG(v.distancia_km / v.tiempo_real_minutos * 60), 2) as velocidad_efectiva_kmh,
    ROUND(SUM(v.distancia_km), 2) as distancia_total_recorrida
FROM VUELO v
JOIN AVION av ON v.id_avion = av.id_avion
GROUP BY av.fabricante, av.nombre_modelo, av.capacidad_maxima, av.agno_fabricacion
HAVING COUNT(*) > 1
ORDER BY vuelos_realizados DESC, distancia_promedio DESC;

```

FABRICANTE	NOMBRE MODELÓ	CAPACIDAD MAXIMA	AGNO FABRICACION	VUELOS REALIZADOS	DISTANCIA PROMEDIO	DURACION REAL PROMEDIO	RETRASO PROMEDIO	VELOCIDAD EFECTIVA_KMH	DISTANCIA TOTAL RECORRIDA
Embraer	Embraer ERJ170-200LR	76	2007	168	880.23	117.93	4.91	425.82	147,879.03
GE	GE CL-600-2C10	70	2001	153	691.41	102.76	5.95	395.2	105,785.14
Embraer	Embraer ERJ170-200LR	76	2014	142	636.18	97.11	2.82	351.24	90,337.08
Embraer	Embraer ERJ170-200LL	70	2020	139	979.29	122.35	2.91	439.82	136,121.2
AirbusIndustries	AirbusIndustries A320-232	176	2010	131	1,486.93	153.03	2.83	571.22	194,788.08
Embraer	Embraer ERJ170-200LR	76	2015	128	1,073.47	129.1	6.39	474.81	137,403.84
THEBOEINGCO	THEBOEINGCO 737-8	175	2022	124	1,340.16	131.74	1.03	477.51	166,180.45
AirbusIndustries	AirbusIndustries A321-211	191	2020	117	1,640.96	166.24	3.89	581.51	191,992.65
BombardierAerospace	BombardierAero space CL-600-2C11	50	2005	115	526.88	88.93	3.04	334.67	60,591.65
Airbus	Airbus 320-214	180	2015	108	1,493.24	159.67	4.31	556.4	161,270.35
TheBoeingCompany	TheBoeingCompany 737-900ER	181	2019	93	2,169.72	205.11	3.61	611.67	201,783.88
BoeingCo	BoeingCo 737-824	154	2000	93	1,402.95	149.86	3.71	547.03	130,474.02
Airbus	Airbus A321-231	102	2013	75	4,027.32	349.37	4.08	697.88	302,048.98
Airbus	Airbus A321-271N	189	2017	66	4,030.08	332.08	4.45	723.35	265,985.28
AirbusIndustrie	AirbusIndustrie A321-271NX	200	2019	10	1,946.66	181.2	1	592.13	19,466.58

El Embraer ERJ170-200LR de 2007 es el avión más usado (168 vuelos). Los Airbus A321 hacen los vuelos más largos (más de 4000 km de promedio). El Boeing 737-8 de 2022 es el más puntual con solo 1.03 minutos de retraso, y el Airbus A321-271N tiene la mayor velocidad efectiva (723 km/h).

## 7. Análisis de cargas incrementales

La carga realizada con los ficheros mostrados anteriormente corresponde a un snapshot con los datos de un mes. Sin embargo, si el objetivo es disponer de un data mart actualizado de forma continua, es necesario implementar cargas incrementales.

Aunque en el trabajo realizado no se ha implementado este tipo de cargas con pruebas reales, se ha realizado un análisis exhaustivo de la estrategia que se seguiría en el caso de llevarlas a cabo. La estrategia elegida para conseguirlo consiste en identificar los nuevos registros o los modificados desde la última ejecución, utilizando como referencia la fecha y hora de la última carga.

Para esta gestión, es importante tratar de manera diferente la tabla de hechos y las dimensiones. En el primer caso, se asume que los vuelos pasados no cambian, por lo que el proceso ETL únicamente consulta los vuelos con fecha posterior a la última ejecución. Tras su procesamiento, estos datos se añaden a la tabla de hechos sin alterar los registros ya existentes.

En cuanto a las dimensiones, que sí pueden cambiar a lo largo del tiempo, se diferencian dos tipos. Por un lado, las dimensiones estáticas, como **FECHA** y **HORA**, que se cargan una única vez y no requieren actualizaciones. Por otro, las dimensiones cambiantes, en las que se distinguen dos situaciones: si el cambio tiene carácter histórico, se aplicaría un modelo de tipo SCD 2, cerrando el registro antiguo con una fecha de fin de validez y creando un nuevo registro válido desde la fecha actual. Mientras que, si se trata de una corrección menor, como un error de escritura, se aplicaría un modelo de tipo SCD 1, actualizando directamente el registro existente.

Para facilitar este proceso, se ha considerado que el sistema debe mantener una marca temporal de la última ejecución en una tabla de control, lo que permite que las cargas posteriores filtren automáticamente los datos fuente según esa fecha. De este modo, el data mart se mantiene actualizado de forma eficiente, reduciendo los tiempos de carga y evitando reprocesar información que no ha cambiado.

## 8. Análisis del rendimiento

En la implementación actual, se procesaron los datos correspondientes a un único mes, marzo de 2025. Este conjunto de datos, a pesar de ser realista, resultó en un volumen manejable de aproximadamente 1.700 registros de vuelos tras terminar la etapa de limpieza del proceso ETL. El tiempo de ejecución completo del *workflow* en **KNIME**, desde la lectura de los ficheros Excel hasta la generación de los ficheros CSV finales para las dimensiones y la tabla de hechos, fue inferior al minuto. La posterior carga de estos ficheros en Oracle Database mediante los *scripts* SQL también fue prácticamente instantánea.

Aunque no se llegaron a hacer pruebas empíricas con volúmenes de datos mayores debido a la complejidad de procesar múltiples meses o incluso años de datos, se realizó un análisis profundo de los posibles cuellos de botella que podrían producirse a la hora de escalar la solución. Considerando que un mes generó aproximadamente 1.700 vuelos, un año produciría alrededor de 20.000 registros y un histórico de una década podría alcanzar los

200.000 vuelos o más. Por ello, si se consideran volúmenes de datos de tal orden de magnitud los principales puntos de congestión en del flujo de **KNIME** desarrollado se encontrarían en los nodos *Joiner* y *GroupBy*, ya que se encargan de hacer cruces y agregaciones de grandes volúmenes de datos en memoria. Además, la lectura de archivos **Excel** y **CSV** con tal cantidad de datos sería muy lenta, al igual que para el proceso de escritura de los CSVs. Basándonos en la complejidad algorítmica de estas operaciones, se estima que procesar un año podría tomar entre 5 y 10 minutos, y una década de históricos podría requerir varias horas.

Se podrían adoptar varias estrategias para poder mitigar estas congestiones:

1. **Optimización de memoria:** Incrementar la memoria RAM asignada a la JVM de **KNIME**.
2. **Procesamiento en base de datos:** Delegar las operaciones de *join* y agregación al motor de la base de datos, mediante los nodos *Database Connection* y *DB Query*, aprovechando así su capacidad de procesamiento optimizado en disco.
3. **Procesamiento por lotes (Streaming):** Configurar el flujo de **KNIME** para que procese los datos en chunks en lugar de intentar cargar el conjunto de datos completo en memoria de una sola vez.

## 9. Distribución del trabajo

En el desarrollo de esta práctica, todas las tareas se llevaron a cabo de forma conjunta por los integrantes del equipo. La colaboración fue continua en todas las fases del proceso ETL, combinando esfuerzos en el análisis de fuentes de datos, el diseño e implementación de las transformaciones, la configuración de las herramientas y la elaboración de la memoria.

Las actividades desarrolladas conjuntamente incluyen:

- **Análisis y preparación de fuentes de datos**, explorando los conjuntos de datos de **GeoNames** y **RITA**, identificando los datos disponibles, y definiendo las estrategias de extracción.
- **Modificación del modelo dimensional**, ajustando el diseño de la base de datos analítica para adaptar el esquema a los datos realmente disponibles.
- **Diseño e implementación del proceso ETL**, desarrollando las fases de extracción, transformación y carga, incluyendo la limpieza de datos, el manejo de inconsistencias y la implementación de las transformaciones necesarias mediante **KNIME**.
- **Relación con los subsistemas ETL de Kimball**, analizando y documentando el proceso implementado y relacionándolo con los subsistemas propuestos en la metodología de Kimball.
- **Elaboración de todos los ficheros asociados a los vuelos**, incluyendo los scripts de creación, generación de inserciones automáticas, inserción de datos y consultas.

- **Análisis del proceso de carga incremental**, reflexionando sobre las estrategias para realizar cargas incrementales periódicas y realizando pruebas demostrativas a pequeña escala.
- **Análisis de rendimiento**, realizando mediciones de tiempos de carga, identificando cuellos de botella y optimizando procesos.
- **Redacción de los apartados correspondientes en la memoria**, integrando de manera coherente los resultados obtenidos y las decisiones de diseño adoptadas.

En conjunto, el proyecto fue resultado de una **colaboración equitativa y coordinada** entre todos los miembros del grupo, sin asignaciones individuales diferenciadas.

## 10. Problemas encontrados

Durante el proceso de carga de datos se identificó un problema crítico relacionado con la definición de la clave primaria de la tabla de hechos **VUELO**. El esquema dimensional establecía como clave primaria compuesta la combinación de *codigo\_vuelo* e *id\_fecha*, asumiendo que un mismo código de vuelo solo operaría una vez por día.

Sin embargo, al analizar los datos reales de marzo de 2025, se detectó que un mismo vuelo puede operar múltiples veces en el mismo día, especialmente en rutas de alta frecuencia. Esta situación generaba violaciones de la restricción de clave primaria, ya que existían múltiples registros que compartían los valores de la clave primaria, pero diferían en el *id\_hora*.

Para resolver este problema se implementó una solución mediante **KNIME** para garantizar la unicidad de *codigo\_vuelo* cuando existen múltiples operaciones del mismo vuelo en un día. El proceso seguido fue el siguiente:

1. **Ordenación de datos (Nodo Sorter)**: primero se utilizó el nodo *Sorter* para ordenar todos los datos por *codigo\_vuelo* e *id\_fecha*, y de este modo agrupamos físicamente los registros duplicados (según la clave primaria).
2. **Generación de Ranking Secuencial (Nodo Rank)**: una vez ordenados los datos, se aplicó el nodo *Rank* para asignar un ranking secuencial a cada registro. El ranking se reiniciaba por cada grupo único de *codigo\_vuelo* e *id\_fecha*, generalmente secuencias 1, 2, 3, 4....
3. **Ajuste del Contador (Nodo Math Formula)**: para generar sufijos que comienzan desde "0" en vez de "1", se utilizó este nodo para restar 1 al ranking obtenido en el paso anterior.
4. **Conversión de Tipo de Dato (Nodo Number to String)**: cambiamos de tipo int a string la columna Rank generada.
5. **Creación del Identificador Único (Nodo String Manipulation)**: se creó el nuevo identificador *codigo\_vuelo* concatenando *codigo\_vuelo* junto con la columna *Rank*.

Esta solución permitió mantener la estructura original de la tabla de hechos sin necesidad de modificar la definición de la clave primaria ni añadir columnas adicionales, preservando la legibilidad de *codigo\_vuelo*.

## 11. Conclusiones

En esta práctica se ha podido aplicar de forma práctica el proceso ETL completo, lo que ha permitido comprender la complejidad de integrar datos reales provenientes de fuentes heterogéneas y transformarlos en un modelo dimensional coherente y funcional. El trabajo con datos reales de vuelos comerciales ha facilitado entender los desafíos inherentes a la necesidad de limpieza y validación, y la importancia de diseñar procesos robustos y reproducibles.

La implementación del workflow ETL mediante **KNIME** ha permitido consolidar conocimientos sobre las fases de extracción y transformación, comprendiendo la relevancia de cada etapa y su impacto en el data mart final. El diseño de transformaciones complejas, como la generación de atributos derivados, la gestión de valores nulos y la consolidación de múltiples fuentes, ha resultado especialmente formativo.

La automatización de las inserciones mediante scripts Python demostró ser una estrategia eficaz para optimizar el proceso de carga y minimizar errores.

El análisis de la carga incremental y su prueba práctica permitió reflexionar sobre la sostenibilidad a largo plazo del data mart, comprendiendo la importancia de diseñar desde el inicio mecanismos que faciliten actualizaciones periódicas sin comprometer la integridad de los datos históricos.

Apartados del trabajo realizados	Lucía	Irene
Análisis y preparación de fuentes de datos	4	7
Modificación del modelo dimensional	1.5	0
Diseño e implementación del proceso ETL	1	4.5
Creación, implementación y consultas vuelos sin pasajeros	3	2
Documentación	7.5	3.5
	17	17