Proyecto Base de Datos I

Hito 1

Base de datos para registrar los vuelos

Integrantes

Liang Corrales, Renzo Chen Heng
Quispe Monzón, Oswaldo Alejandro
Castillo Melchor, Michael Antonio

Javier Alejandro Castro Barreto

Universidad de Ingeniería y Tecnología

Curso: Base de Datos I

Ciclo: 2025.1

1. Requisitos

1.1. Introducción

Este proyecto presenta los requerimientos para el diseño e implementación de una base de datos orientada al análisis de datos históricos de vuelos comerciales en los Estados Unidos. Su propósito es proporcionar una estructura sólida y escalable que permita almacenar, gestionar y consultar eficientemente grandes volúmenes de información provenientes de distintas aerolíneas, aeropuertos y vuelos, facilitando el análisis de indicadores clave como retrasos, cancelaciones, rutas más transitadas y rendimiento por operador.

1.2. Descripción general del problema

La empresa requiere una solución tecnológica robusta y escalable que le permita almacenar, procesar y consultar eficientemente grandes volúmenes de datos históricos de vuelos provenientes de múltiples aerolíneas en los Estados Unidos. El objetivo principal es habilitar el análisis de métricas clave de rendimiento —como puntualidad, tiempos de retraso y frecuencia de vuelos—, así como identificar patrones de comportamiento que contribuyan a la toma de decisiones estratégicas y a la mejora continua de sus operaciones.

1.3. Necesidad/usos de la base de datos

La base de datos permitirá realizar análisis detallados sobre el comportamiento del transporte aéreo en Estados Unidos. Entre sus principales usos se encuentra el cálculo de retrasos promedio por aeropuerto, la determinación de frecuencia de vuelos por ruta, análisis de causas de cancelaciones y comparación del desempeño entre aerolíneas.

1.4. ¿Cómo se resuelve el problema hoy?

1.4.1. ¿Cómo se almacenan/procesan los datos hoy?

Actualmente, los datos se almacenan en hojas de cálculo y archivos CSV distribuidos localmente, sin una estructura relacional ni mecanismos que garanticen integridad o trazabilidad. Esto limita la capacidad de análisis avanzado.

1.4.2. Flujo de datos

El flujo de datos se basa en procesos manuales que implican descargas periódicas desde portales públicos. Posteriormente, se integran en hojas de cálculo, dificultando la estandarización y automatización de análisis.

1.5. Descripción detallada del sistema

1.5.1. Objetos de información actuales

Se manejan datos de vuelos (fechas, horarios, retrasos, cancelaciones, distancias), aerolíneas y aeropuertos (origen/destino).

1.5.2. Características y funcionalidades esperadas

Consultas rápidas, integridad referencial, actualizaciones mensuales, generación de reportes de rendimiento, escalabilidad y mantenimiento eficiente.

1.5.3. Tipos de usuarios existentes/necesarios

- Analistas de datos: para consultas agregadas y reportes.
- Personal operativo/logístico: monitoreo de retrasos/cancelaciones.
- Administradores del sistema: mantenimiento y carga de datos.

1.5.4. Tipos de consulta, actualizaciones

Promedio de retrasos, número de cancelaciones, frecuencia por ruta, rendimiento por aerolínea. Actualizaciones mensuales con integración de datos históricos.

1.5.5. Tamaño estimado de la base de datos

Proyección diaria del almacenamiento:

Reservaciones: 1000 x 26 = 26000 bytes
Boletos: 5000 x 63 = 315000 bytes
Vuelos: 200 x 86 = 17200 bytes
Datos de aeronaves: 50 x 64 = 3200 bytes
Datos de aeropuertos: 100 x 210 = 21000 bytes
Asientos: 3000 x 65 = 195000 bytes
Pasajeros: 10000 x 224 = 2240000 bytes
Tarjetas de embarque: 4800 x 48 = 230400 bytes
Boletos por vuelo: 5000 x 82 = 410000 bytes

En total:
3,457,800 bytes, es decir 3.30 megabytes (MB).

Proyección quincenal del almacenamiento:
3,457,800 x 15 = 51,867,000 bytes, es decir 49.44 megabytes (MB).

Proyección mensual del almacenamiento:
3,457,800 x 30 = 103,734,000 bytes, es decir 98.88 megabytes (MB).

Nota: Estas proyecciones se basan en estimaciones promedio de registros diarios y pueden variar según la carga real de la base de datos.

1.6. Objetivos del proyecto

Diseñar una base de datos relacional optimizada para almacenar y analizar datos históricos de vuelos comerciales en EE. UU., que permita realizar consultas eficientes y sea apta para aerolíneas pequeñas.

1.7. Referencias del proyecto

- Kaggle - *Airline Data Analysis*: `https://www.kaggle.com/datasets/melvinmarchena/airline-d`
- Bibliografía sobre diseño de bases de datos relacionales y normalización.

1.8. Eventualidades

1.8.1. Problemas que pudieran encontrarse en el proyecto

Datos incompletos, inconsistentes, diferencias de formato, ambigüedad en campos y necesidad de limpieza.

1.8.2. Límites y alcances del proyecto

Se enfoca en el análisis histórico. No incluye predicción, monitoreo en tiempo real ni integración con sistemas externos.

2. Modelo Relacional

2.1. Relaciones Semánticas

- Una aerolínea puede operar muchos vuelos, pero cada vuelo está asociado a una sola aerolínea.
- Un vuelo se realiza una sola vez en una fecha determinada. Está identificado por un código único para esa fecha.
- Un aeropuerto puede ser origen y/o destino de muchos vuelos.
- Un vuelo tiene una hora de salida y llegada programadas, así como horas reales que permiten calcular retrasos.
- Un vuelo puede o no estar cancelado. Si está cancelado, debe tener un código de cancelación válido (por clima, mantenimiento, etc.).
- Cada aeropuerto tiene un código único (como "JFK", "LAX"), y puede participar en muchos vuelos como origen o destino.
- Las causas de cancelación deben registrarse solo cuando el vuelo ha sido efectivamente cancelado.
- Se espera que las horas reales respeten la coherencia: la hora de llegada no puede ser anterior a la de salida, sin considerar zonas horarias (aunque para este modelo se asumirá la misma zona horaria en todos los registros).
- Cada vuelo registrado en la base de datos se identifica principalmente por su fecha de operación (`FlightDate`), y está asociado a una aerolínea específica (`Airline`), así como a un origen y destino (`Origin`, `Dest`), los cuales corresponden a códigos de aeropuertos.
- Un vuelo tiene asociados datos tanto de programación como de operación real. Los atributos `ScheduledDepTime` y `ScheduledArrTime` representan la hora programada de salida y llegada, mientras que `DepTime` y `ArrTime` indican las horas reales en que el vuelo despegó y aterrizó, respectivamente. Estas diferencias permiten calcular los

retrasos, capturados por los campos `DepDelay` (retraso en salida) y `ArrDelay` (retraso en llegada).

- El campo `Distance` indica la distancia total recorrida por el vuelo, medida en millas, y puede usarse tanto para análisis logísticos como para comparaciones por tipo de ruta.
- Cada vuelo puede haber sido cancelado o no, lo cual se representa mediante el atributo `Cancelled`, que toma valores booleanos. En caso de cancelación, se especifica una razón mediante el campo `CancellationCode`, que puede hacer referencia a condiciones meteorológicas, problemas operativos o causas relacionadas a la seguridad, entre otras.
- Las aerolíneas (`Airline`) se comportan como una entidad clave del sistema, ya que agrupan múltiples vuelos a lo largo del tiempo y permiten generar análisis agregados de desempeño, puntualidad o cancelación.
- Los aeropuertos (`Origin`, `Dest`) también deben considerarse entidades independientes, dado que un mismo aeropuerto puede ser origen de múltiples vuelos y destino de otros tantos. Esto permite estudiar métricas por ubicación geográfica, como retrasos promedio por aeropuerto.

2.2. Tablas

- `Reservaciones(Boletos.id_pasajero: int, fecha_reserva: date, monto_total: decimal)`
- `Boletos(numero_boleto: int, id_pasajero: int, referencia_de_reserva: varchar)`
- `Tarjetas_de_embarque(Boletos.Id_pasajero: int, numero_asiento: int, numero_embarque: int,Boletos.numero_del_boleto:int )`
- `Boletos_por_vuelo(Boletos.numero_boleto: int, Vuelos.id_vuelo: int, monto: decimal, condiciones_tarifarias: varchar)`
- `Vuelos(id_vuelo: int, numero_vuelo: varchar, salida_programada: datetime, llegada_programada: datetime, salida_real: datetime, llegada_real: datetime, estado_vuelo: varchar, aeropuerto_llegada: varchar)`
- `Datos_de_aeronaves(V.idVuelos:int,Ao.codigo_aeropuertos:int,codigo_aeronave: int, modelo: varchar, alcance: int)`
- `Asientos(Ae.codigo_aeronave: int, numero_asiento: int, condiciones_tarifarias: varchar)`
- `Datos_de_aeropuertos(V.idVuelos:int,Ae.codigo_aeronave: int, codigo_aeropuerto: int, ciudad: varchar, coordenadas: varchar, zona_horaria: varchar, nombre_del_aeropuerto: varchar)`
enditemize

2.3. Especificaciones y Recomendaciones por Entidad

1. **Reservaciones Especificaciones:**
   Registra cada compra de uno o más boletos por parte de un pasajero. Clave primaria: `referencia_reserva`.
   Recomendaciones:
   Asegura unicidad con un código alfanumérico (`VARCHAR(15)`). Relación 1:N con `Boletos`: una reservación puede tener múltiples boletos.
2. **Boletos Especificaciones:**
   Representa el derecho de un pasajero a abordar un vuelo. Clave primaria: `numero_boleto`.
   Recomendaciones:
   Asegura que `id_pasajero` y `id_vuelo` existan como claves foráneas válidas. Podrías fusionar `Boletos_por_vuelo` si no hay necesidad de separar condiciones tarifarias.
3. **Tarjetas de embarque Especificaciones:**
   Documento generado que autoriza el embarque del pasajero en un vuelo. Clave primaria sugerida: `id_tarjeta` o combinación de `numero_boleto` + `id_vuelo`.
   Recomendaciones:
   Elimina redundancia con `numero_asiento` si ya existe en `Boletos`. Define si es necesaria como entidad o si puede representarse como vista.
4. **Boletos por vuelo Especificaciones:**
   Tabla intermedia con datos adicionales como `monto` y `condiciones_tarifarias`.
   Recomendaciones:
   Úsala si un boleto puede estar vinculado a múltiples vuelos (caso raro). Si no se justifica, esta tabla puede eliminarse y migrar los atributos a `Boletos`.
5. **Vuelos Especificaciones:**
   Tabla principal que representa cada vuelo realizado. Clave primaria: `id_vuelo`.
   Recomendaciones:
   Añadir `aeropuerto_salida` además de `aeropuerto_llegada`. Verifica consistencia lógica entre `salida_programada`, `salida_real`, etc. Añade `cancelado` (booleano) y `codigo_cancelacion` si vas a analizar eventos operativos.
6. **Datos de aeronaves Especificaciones:**
   Registra detalles técnicos de cada avión usado.
   Recomendaciones:
   Elimina las columnas `V.idVuelos` y `Ao.codigo_aeropuertos` aquí. Esto rompe la normalización. Esta tabla debe estar relacionada 1:N con `Vuelos` mediante `codigo_aeronave`.
7. **Asientos Especificaciones:**
   Define los asientos disponibles en cada aeronave.
   Recomendaciones:
   Usa clave compuesta: `codigo_aeronave + numero_asiento`. Asegura integridad referencial con `Datos_de_aeronaves`.
8. **Datos de aeropuertos Especificaciones:**
   Describe cada aeropuerto por código, ciudad, zona horaria,

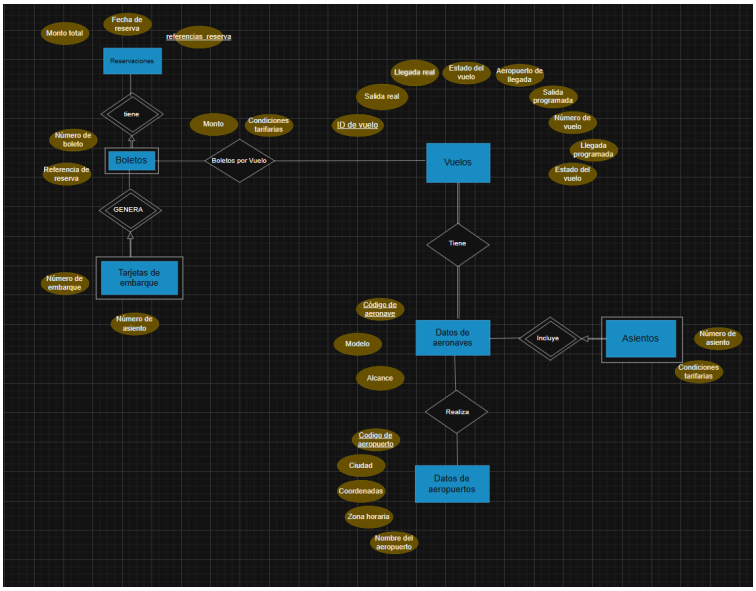


Figura 1: Modelo Entidad-Relación del sistema de reservaciones

etc.
Recomendaciones:
No debe tener `idVuelos` ni `codigo_aeronave`. Elimina esas FK mal ubicadas. Esta tabla es independiente. Clave primaria: `codigo_aeropuerto`. Relación 1:N con `Vuelos` desde `aeropuerto_salida` y `aeropuerto_llegada`.

3. Modelo Relacional

3.1. Modelo Relacional

A continuación se presenta el modelo relacional resultante de la transformación del modelo Entidad-Relación. Este modelo describe las tablas, sus atributos, claves primarias y foráneas, así como la forma en que se relacionan para representar la lógica del sistema de reservaciones y gestión de vuelos.

3.2. Especificaciones de transformación

3.2.1. Relaciones Semánticas

- **Reservaciones** (referencia_reserva: `varchar(15)`, fecha_reserva: `date`, monto_total: `numeric(10,2)`)
- **Boletos** (numero_boleto: `varchar(20)`, referencia_reserva: `varchar(15)`, id_pasajero: `int`, id_vuelo: `int`, numero_asiento: `varchar(5)`, numero_embarque: `varchar(15)`)
- **Tarjetas_de_embarque** (id_tarjeta: `int`, numero_boleto: `varchar(20)`, id_vuelo: `int`, numero_asiento: `varchar(5)`, numero_embarque: `varchar(15)`)

- **Boletos_por_vuelo** (numero_boleto: `varchar(20)`, id_vuelo: `int`, monto: `numeric(10,2)`, condiciones_tarifarias: `varchar(50)`)
- **Vuelos** (id_vuelo: `int`, numero_vuelo: `varchar(10)`, salida_programada: `timestamp`, llegada_programada: `timestamp`, salida_real: `timestamp`, llegada_real: `timestamp`, estado_vuelo: `varchar(20)`, aeropuerto_llegada: `varchar(10)`, codigo_aeronave: `varchar(10)`)
- **Datos_de_aeronaves** (codigo_aeronave: `varchar(10)`, modelo: `varchar(50)`, alcance: `int`)
- **Asientos** (codigo_aeronave: `varchar(10)`, numero_asiento: `varchar(5)`, condiciones_tarifarias: `varchar(50)`)
- **Datos_de_aeropuertos** (codigo_aeropuerto: `varchar(10)`, ciudad: `varchar(50)`, coordenadas: `varchar(30)`, zona_horaria: `varchar(20)`, nombre_del_aeropuerto: `varchar(100)`)

3.2.2. Entidades débiles

- **Tarjetas_de_embarque(id_tarjeta, numero_boleto, id_vuelo, numero_asiento, numero_embarque)** Depende de `Boletos` y `Vuelos`. No tiene sentido sin un boleto asociado.
- **Boletos_por_vuelo(numero_boleto, id_vuelo, monto, condiciones_tarifarias)** Es una relación intermedia con información adicional entre `Boletos` y `Vuelos`.

3.2.3. Entidades superclase/subclases

En este modelo no se definen entidades con jerarquías de generalización/especialización. Todos los tipos de entidades (como pasajero o reserva) se representan con una sola tabla sin subtipos.

3.2.4. Relaciones binarias

- **Tiene**: Una `Reservación` puede tener varios `Boletos`, pero un `Boleto` solo pertenece a una `Reservación`.
- **Genera**: Cada `Boleto` genera exactamente una `Tarjeta_de_embarque`.
- **Pertenece**: Cada `Boleto` se asocia a un `Vuelo` mediante la relación directa o por medio de `Boletos_por_vuelo`.
- **Opera**: Cada `Vuelo` está asociado a una única `Aeronave` mediante `codigo_aeronave`.
- **Incluye**: Cada `Aeronave` tiene múltiples `Asientos`.
- **Ubicado en**: Cada `Vuelo` está asociado a un aeropuerto de llegada mediante `codigo_aeropuerto`.

3.2.5. Relaciones ternarias

El modelo actual no requiere relaciones ternarias, ya que todas las asociaciones entre entidades se pueden representar mediante relaciones binarias o entidades intermedias como `Boletos_por_vuelo` que contienen información adicional.

3.3. Diccionario de datos

Nombre campo	**Tipo de dato**	**PK**	**FK**	**Descripción**
referencia_reserva	VARCHAR(15)	X		Código único que identifica una reservación
fecha_reserva	DATE			Fecha en que se realizó la reservación
monto_total	NUMERIC(10,2)			Monto total pagado en la reservación

Cuadro 1: Diccionario de datos para la tabla `Reservaciones`

Nombre campo	**Tipo de dato**	**PK**	**FK**	**Descripción**
numero_boleto	VARCHAR(20)	X		Identificador único del boleto
referencia_reserva	VARCHAR(15)		X	Código de la reservación asociada
id_pasajero	INT		X	Identificador del pasajero que posee el boleto
id_vuelo	INT		X	Identificador del vuelo asignado al boleto
numero_asiento	VARCHAR(5)			Número del asiento asignado en el vuelo
numero_embarque	VARCHAR(15)			Número de embarque asignado

Cuadro 2: Diccionario de datos para la tabla `Boletos`

Nombre campo	Tipo de dato	PK	FK	Descripción
id_vuelo	INT	X		Identificador único del vuelo
numero_vuelo	VARCHAR(10)			Código o número oficial del vuelo
salida_programada	TIMESTAMP			Fecha y hora programada de salida
llegada_programada	TIMESTAMP			Fecha y hora programada de llegada
salida_real	TIMESTAMP			Fecha y hora real de salida
llegada_real	TIMESTAMP			Fecha y hora real de llegada
estado_vuelo	VARCHAR(20)			Estado actual del vuelo (programado, en vuelo, cancelado, etc.)
aeropuerto_llegada	VARCHAR(10)		X	Código del aeropuerto de llegada
codigo_aeronave	VARCHAR(10)		X	Código del avión asignado para el vuelo

Cuadro 3: Diccionario de datos para la tabla `Vuelos`

Nombre campo	Tipo de dato	PK	FK	Descripción
codigo_aeronave	VARCHAR(10)	X		Código único que identifica una aeronave
modelo	VARCHAR(50)			Modelo o tipo de aeronave (ej. Boeing 737)
alcance	INTEGER			Alcance máximo en kilómetros de la aeronave

Cuadro 4: Diccionario de datos para la tabla `Datos_de_aeronaves`

Nombre campo	Tipo de dato	PK	FK	Descripción
codigo_aeropuerto	VARCHAR(10)	X		Código único que identifica un aeropuerto (IATA o ICAO)
ciudad	VARCHAR(50)			Ciudad donde se encuentra el aeropuerto
coordenadas	VARCHAR(30)			Coordenadas geográficas del aeropuerto (latitud, longitud)
zona_horaria	VARCHAR(20)			Zona horaria del aeropuerto (ej. UTC-5)
nombre_del_aeropuerto	VARCHAR(100)			Nombre completo oficial del aeropuerto

Cuadro 5: Diccionario de datos para la tabla `Datos_de_aeropuertos`

Nombre campo	Tipo de dato	PK	FK	Descripción
codigo_aeronave	VARCHAR(10)	X		Código de la aeronave a la que pertenece el asiento
numero_asiento	VARCHAR(5)		X	Número o identificador del asiento dentro de la aeronave
condiciones_tarifarias	VARCHAR(50)			Condiciones o categoría tarifaria asociada al asiento (ej. Economy, Business)

Cuadro 6: Diccionario de datos para la tabla `Asientos`

Nombre campo	Tipo de dato	PK	FK	Descripción
id_tarjeta	INT	X		Identificador único de la tarjeta de embarque
numero_boleto	VARCHAR(20)		X	Número del boleto asociado
id_vuelo	INT		X	Identificador del vuelo asociado
numero_asiento	VARCHAR(5)			Número de asiento asignado en la tarjeta
numero_embarque	VARCHAR(15)			Número asignado para control de embarque

Cuadro 7: Diccionario de datos para la tabla `Tarjetas_de_embarque`

Nombre campo	Tipo de dato	PK	FK	Descripción
numero_boleto	VARCHAR(20)	X		Número del boleto asociado
id_vuelo	INT		X	Identificador del vuelo asociado
monto	NUMERIC(10,2)			Precio o monto pagado asociado a este boleto por vuelo
condiciones_tarifarias	VARCHAR(50)			Condiciones tarifarias específicas para este boleto en el vuelo

Cuadro 8: Diccionario de datos para la tabla `Boletos_por_vuelo`

4. Implementación de la base de datos

4.1. Creación de Tablas en PostgreSQL

Listing 1: Consulta SQL para crear tablas

```
1
2 -- Tabla aircrafts_data: almacena el modelo y alcance de
      cada avi n registrado
3
4 CREATE TABLE aircrafts_data (
5     aircraft_code VARCHAR(10) PRIMARY KEY,
```

```
    model TEXT NOT NULL,
    range INTEGER NOT NULL
);

-- Tabla airports_data: contiene informaci n sobre los aeropuertos registrados

CREATE TABLE airports_data (
    airport_code VARCHAR(10) PRIMARY KEY,
    airport_name TEXT NOT NULL,
    city TEXT NOT NULL,
    coordinates TEXT NOT NULL,
    timezone TEXT NOT NULL
);

-- Tabla boarding_passes: representa los pases de abordaje de los pasajeros en un vuelo

CREATE TABLE boarding_passes (
    ticket_no VARCHAR(20) NOT NULL,
    flight_id INTEGER NOT NULL,
    boarding_no INTEGER NOT NULL,
    seat_no VARCHAR(5) NOT NULL,
    PRIMARY KEY (ticket_no, flight_id)
);

-- Tabla bookings: almacena las reservas realizadas y el monto total pagado

CREATE TABLE bookings (
    book_ref VARCHAR(10) PRIMARY KEY,
    book_date TIMESTAMP NOT NULL,
    total_amount NUMERIC(10, 2) NOT NULL
);

-- Tabla flights: almacena informaci n detallada de cada vuelo, su estado y programaci n

CREATE TABLE flights (
    flight_id INTEGER PRIMARY KEY,
    flight_no VARCHAR(10) NOT NULL,
    scheduled_departure TIMESTAMP NOT NULL,
    scheduled_arrival TIMESTAMP NOT NULL,
    departure_airport VARCHAR(10) NOT NULL,
    arrival_airport VARCHAR(10) NOT NULL,
    status VARCHAR(20) NOT NULL,
    aircraft_code VARCHAR(10) NOT NULL,
    actual_departure TIMESTAMP,
    actual_arrival TIMESTAMP
);

-- Tabla seats: define los asientos y condiciones tarifarias asociadas a cada modelo de avi n

CREATE TABLE seats (
    aircraft_code VARCHAR(10) NOT NULL,
    seat_no VARCHAR(5) NOT NULL,
    fare_conditions VARCHAR(20) NOT NULL,
    PRIMARY KEY (aircraft_code, seat_no)
);

-- Tabla ticket_flights: vincula los tickets con vuelos espec ficos , tarifas y montos pagados

CREATE TABLE ticket_flights (
    ticket_no VARCHAR(20) NOT NULL,
    flight_id INTEGER NOT NULL,
```

```
    fare_conditions VARCHAR(20) NOT NULL,
    amount NUMERIC(10, 2) NOT NULL,
    PRIMARY KEY (ticket_no, flight_id)
);

-- Tabla tickets: relaciona cada ticket con su reserva y
     pasajero correspondiente

CREATE TABLE tickets (
    ticket_no VARCHAR(20) PRIMARY KEY,
    book_ref VARCHAR(10) NOT NULL,
    passenger_id VARCHAR(20) NOT NULL
);
```

4.2. Carga de Datos

La carga de datos se realizó mediante la ejecución de comandos `COPY` en PostgreSQL para insertar registros desde archivos `.csv` descargados de la plataforma Kaggle. Estos archivos contienen información real sobre vuelos, tickets, pasajeros, aeronaves, aeropuertos, entre otros. Cada tabla del modelo relacional fue poblada directamente desde su respectivo archivo `.csv`, utilizando scripts `.sql` preparados para ejecutar la carga de forma controlada y ordenada, respetando dependencias entre claves primarias y foráneas.

4.3. Simulación de Datos Faltantes

Dado que algunos archivos de Kaggle pueden contener registros incompletos o inconsistencias (por ejemplo, aeropuertos no registrados o asientos inexistentes), se implementaron funciones y triggers en PostgreSQL para validar y garantizar la integridad referencial durante la carga. Asimismo, para simular situaciones con datos faltantes, se configuraron pruebas específicas donde se eliminaron o corrompieron ciertas columnas temporalmente, permitiendo observar el comportamiento del sistema ante entradas defectuosas. Esto contribuyó a validar los mecanismos de control implementados y reforzar la persistencia de datos coherentes, sin depender de datos sensibles o privados.

4.4. Descripción del hardware utilizado

5. Optimización y Experimentación

En este apartado se llevarán a cabo varias pruebas, las cuales estarán basadas en una de las cuatro consultas seleccionadas, las cuales presentan un grado relevante de complejidad y representan escenarios frecuentes en sistemas reales de gestión de aerolíneas. Durante el proceso de experimentación, se trabajará con diferentes volúmenes de datos (1k, 10k, 100k y 1M), habilitando y deshabilitando los índices, con el objetivo de evaluar si la optimización propuesta mediante índices es realmente efectiva.

5.1. Consultas SQL para el experimento

5.1.1. Descripción del tipo de consultas seleccionadas

1. **¿Cuál es el gasto promedio por tipo de tarifa considerando el rango del avión en el año 2017?**
   Esta consulta filtra los vuelos realizados durante 2017 y analiza el gasto promedio por clase tarifaria (Economy, Business, etc.), cruzado con el rango de operación de cada avión. Permite identificar si existe relación entre el tipo de avión y el valor que los clientes están dispuestos a pagar, ideal para optimizar asignaciones de flota por ruta.
2. **¿Qué modelos de avión presentan mayor porcentaje de ocupación de asientos en vuelos realizados?**
   Esta consulta mide qué tan utilizados están los aviones, comparando la cantidad de asientos ocupados contra los disponibles, agrupados por modelo. Esto permite evaluar el rendimiento comercial de cada aeronave, optimizando la planificación de la flota y mejorando la eficiencia operativa según la demanda.
3. **¿Qué ciudades concentran más vuelos y pasajeros, y cómo varía su nivel de ocupación promedio?**
   Esta consulta agrupa los vuelos por ciudad de salida, calculando el número de vuelos, cantidad de pasajeros transportados, gasto promedio por pasajero y nivel promedio de ocupación. Esto permite identificar los hubs más relevantes en la red de operaciones y detectar oportunidades de expansión o mejora en rutas estratégicas.
4. **¿Qué rutas tienen los vuelos con mayor ocupación promedio de asientos?**
   Esta consulta analiza la ocupación promedio de asientos por ruta (origen y destino), permitiendo identificar las rutas con mayor demanda. Es fundamental para tomar decisiones sobre asignación de flota, frecuencia de vuelos y ajustes en la red de destinos, garantizando un uso eficiente de la capacidad disponible.

5.1.2. Implementación de consultas en SQL

Listing 2: Consulta 1: Aeropuertos con más vuelos programados entre 2017 y 2018

```sql
SELECT
    a.range,
    s.fare_conditions,
    ROUND(AVG(tf.amount), 2) AS avg_amount
FROM ticket_flights tf
JOIN flights f ON tf.flight_id = f.flight_id
JOIN seats s ON f.aircraft_code = s.aircraft_code
JOIN aircrafts_data a ON s.aircraft_code = a.
    aircraft_code
WHERE f.scheduled_departure BETWEEN '2017-01-01' AND '
    2017-12-31'
GROUP BY a.range, s.fare_conditions
ORDER BY a.range DESC, avg_amount DESC;
```

Listing 3: Consulta 2: Ingreso promedio por tipo de tarifa

```sql
SELECT
    a.aircraft_code,
    a.model,
    ROUND(
        (
            SELECT COUNT(DISTINCT bp.seat_no)
            FROM flights f
            JOIN ticket_flights tf ON f.flight_id = tf.flight_id
            JOIN boarding_passes bp ON tf.flight_id = bp.flight_id AND tf.ticket_no = bp.ticket_no
            WHERE f.aircraft_code = a.aircraft_code
        )::numeric / (
            SELECT COUNT(DISTINCT s.seat_no)
            FROM seats s
            WHERE s.aircraft_code = a.aircraft_code
        ) * 100,
        2
    ) AS porcentaje_ocupacion
FROM
    aircrafts_data a
ORDER BY
    porcentaje_ocupacion DESC;
```

Listing 4: Consulta 3: Vuelos con mayor número de pasajeros

```sql
SELECT
    ad.city,
    COUNT(DISTINCT f.flight_id) AS total_vuelos,
    COUNT(DISTINCT bp.ticket_no) AS total_pasajeros,
    ROUND(AVG(tf.amount), 2) AS gasto_promedio,
    ROUND(COUNT(DISTINCT bp.seat_no)::numeric / COUNT(DISTINCT s.seat_no) * 100, 2) AS ocupacion_promedio
FROM flights f
JOIN airports_data ad ON f.departure_airport = ad.airport_code
JOIN ticket_flights tf ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON tf.flight_id = bp.flight_id AND tf.ticket_no = bp.ticket_no
JOIN seats s ON f.aircraft_code = s.aircraft_code
GROUP BY ad.city
ORDER BY total_pasajeros DESC
LIMIT 10;
```

Listing 5: Consulta 4: Rutas con mayor mayor ocupacion de asientos por vuelos

```sql
SELECT
    f.departure_airport,
    f.arrival_airport,
    COUNT(DISTINCT bp.ticket_no)::float / COUNT(DISTINCT s.seat_no) AS promedio_ocupacion
FROM
    flights f
JOIN ticket_flights tf ON f.flight_id = tf.flight_id
JOIN boarding_passes bp ON tf.ticket_no = bp.ticket_no AND tf.flight_id = bp.flight_id
JOIN seats s ON f.aircraft_code = s.aircraft_code
GROUP BY
    f.departure_airport, f.arrival_airport
ORDER BY
    promedio_ocupacion DESC
LIMIT 10;
```

5.2. Metodología del experimento

El experimento se llevará a cabo siguiendo una secuencia estructurada basada en las tres consultas previamente presentadas. El objetivo es evaluar el impacto de los índices en el rendimiento de las consultas complejas sobre diferentes volúmenes de datos.

1. Se creará una base de datos que contendrá cuatro esquemas distintos, cada uno con un volumen específico de datos: 1k, 10k, 100k y 1M filas.
2. Se desactivará temporalmente el uso de índices predeterminados en PostgreSQL para permitir la ejecución de las consultas sin optimización, de modo que se puedan medir los tiempos reales sin intervención del optimizador.
3. Se registrarán los tiempos de ejecución utilizando el comando `EXPLAIN ANALYZE` para cada consulta. Los resultados serán almacenados con el fin de construir una gráfica comparativa utilizando la herramienta *tatiyants*.
4. Antes de cada ejecución, se utilizará el comando `VACUUM FULL` para limpiar el caché de PostgreSQL y evitar resultados distorsionados debido a efectos de almacenamiento en memoria.
5. Finalmente, se aplicarán índices específicos diseñados para optimizar cada consulta. Luego, se reactivarán los índices previamente deshabilitados y se volverá a ejecutar cada consulta con `EXPLAIN ANALYZE`. Los nuevos resultados serán comparados con los anteriores para analizar el impacto de los índices en el rendimiento.

5.3. Optimización de consultas

Los siguientes índices fueron elaborados para demostrar su impacto en la mejora del tiempo de ejecución de las consultas complejas descritas anteriormente.

5.3.1. Planes de índices para Consulta 1

Listing 6: Índices para la Consulta 1

```
1
2  -- Optimiza JOIN entre ticket_flights y flights
3  CREATE INDEX idx_ticket_flights_flight_id ON
       ticket_flights(flight_id);
4
5  -- Optimiza JOIN entre flights y seats
6  CREATE INDEX idx_flights_aircraft_code ON flights(
       aircraft_code);
7
8  -- Optimiza JOIN entre seats y aircrafts_data
9  CREATE INDEX idx_seats_aircraft_code ON seats(
       aircraft_code);
10
11 -- Optimiza JOIN entre aircrafts_data y seats
12 CREATE INDEX idx_aircrafts_data_aircraft_code ON
       aircrafts_data(aircraft_code);
13
14 -- Optimiza filtro por fecha
```

```
15 CREATE INDEX idx_flights_scheduled_departure ON flights(
       scheduled_departure);
```

5.3.2. Planes de índices para Consulta 2

Listing 7: Índices para la Consulta 2

```
1
2  -- Para subconsulta de asientos ocupados
3  CREATE INDEX idx_boarding_passes_composite ON
       boarding_passes(flight_id, ticket_no, seat_no);
4
5  -- Optimiza JOIN con ticket_flights
6  CREATE INDEX idx_ticket_flights_flight_ticket ON
       ticket_flights(flight_id, ticket_no);
7
8  -- Optimiza JOIN en flights por aircraft_code
9  CREATE INDEX idx_flights_aircraft_code_2 ON flights(
       aircraft_code);
10
11 -- Para contar asientos disponibles en seats
12 CREATE INDEX idx_seats_aircraft_code_2 ON seats(
       aircraft_code);
13
14 -- Para filtrar el modelo y hacer SELECT principal
15 CREATE INDEX idx_aircrafts_data_model ON aircrafts_data(
       model);
```

5.3.3. Planes de índices para Consulta 3

Listing 8: Índices para la Consulta 3

```
1
2  -- Para JOIN de flights con airports_data
3  CREATE INDEX idx_flights_departure_airport ON flights(
       departure_airport);
4
5  -- Para airports_data (city lookup)
6  CREATE INDEX idx_airports_data_airport_code ON
       airports_data(airport_code);
7
8  -- JOIN entre ticket_flights y boarding_passes
9  CREATE INDEX idx_ticket_flights_flight_id_2 ON
       ticket_flights(flight_id);
10 CREATE INDEX idx_boarding_passes_composite_2 ON
       boarding_passes(flight_id, ticket_no);
11
12 -- JOIN con seats
13 CREATE INDEX idx_flights_aircraft_code_3 ON flights(
       aircraft_code);
14 CREATE INDEX idx_seats_aircraft_code_3 ON seats(
       aircraft_code);
```

5.3.4. Planes de índices para Consulta 4

Listing 9: Índices para la Consulta 4

```
1
2  -- JOIN entre ticket_flights y flights
3  CREATE INDEX idx_ticket_flights_flight_id_3 ON
       ticket_flights(flight_id);
```

```sql
CREATE INDEX idx_boarding_passes_composite_3 ON
    boarding_passes(ticket_no, flight_id);

-- JOIN con seats
CREATE INDEX idx_flights_aircraft_code_4 ON flights(
    aircraft_code);
CREATE INDEX idx_seats_aircraft_code_4 ON seats(
    aircraft_code);

-- GROUP BY y SELECT en rutas
CREATE INDEX idx_flights_departure_arrival_airport ON
    flights(departure_airport, arrival_airport);
```

5.4. Plataforma de Pruebas

Se describe la plataforma de pruebas en el cuadro 9.

Cuadro 9: Hardware para experimentos

Sistema Operativo	Fedora Linux 42(Workstation edition)
Tamaño Página (B)	4KB(1024)
RAM	32GB
CPU	Intel i9-13900H @ 5.2GHz (20 núcleos)
[HDD] Protocolo trans.	NVMe
[HDD] Capacidad	953.9GB
[HDD] RPM	N/A(SSD)
[HDD] Cache	HMB 64MB
[HDD] Taza trans.	3000 MB/sec
PostgreSQL	16.9.0

5.5. Medición de Tiempos

A continuación, se muestran las tablas con los tiempos de ejecución registrados para cada consulta. Los resultados, expresados en segundos, indican cuánto tardó cada consulta en completarse. Además, las mediciones están diferenciadas según si se utilizó o no un índice durante la ejecución.

5.5.1. Sin índices

Cuadro 10: Medición de tiempos (segundos) para 1000 tuplas (sin índices)

Cuadro 11: Medición de tiempos (segundos) para 10000 tuplas (sin índices)

Cuadro 12: Medición de tiempos (segundos) para 100000 tuplas (sin índices)

5.5.2. Con índices

Cuadro 13: Medición de tiempos (segundos) para 1000 tuplas (con índices)

Cuadro 14: Medición de tiempos (segundos) para 10000 tuplas (con índices)

Cuadro 13: Medición de tiempos (segundos) para 10000 tuplas (con índices)

5.6. Resultados

5.6.1. Consulta 1

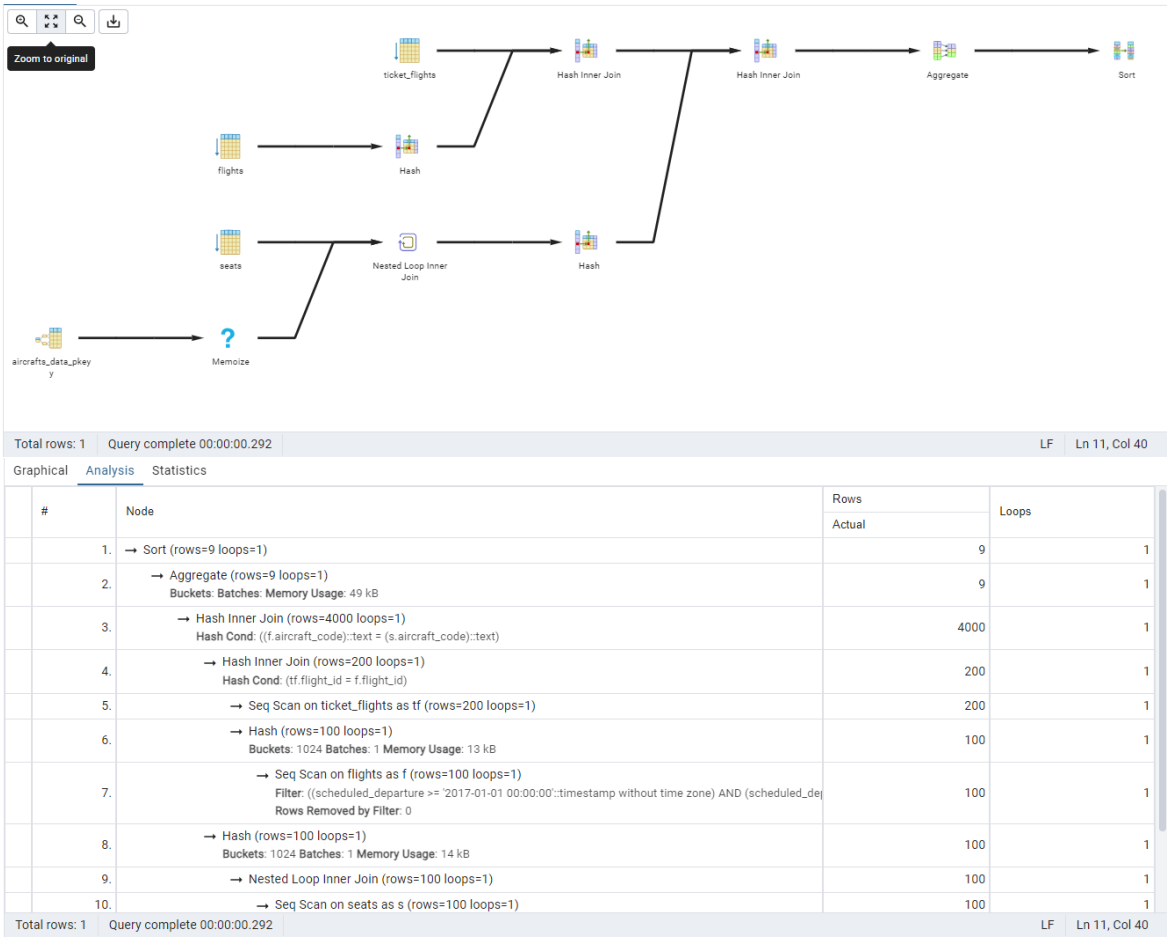


Figura 2: Consulta 1 de 1k de datos sin índice

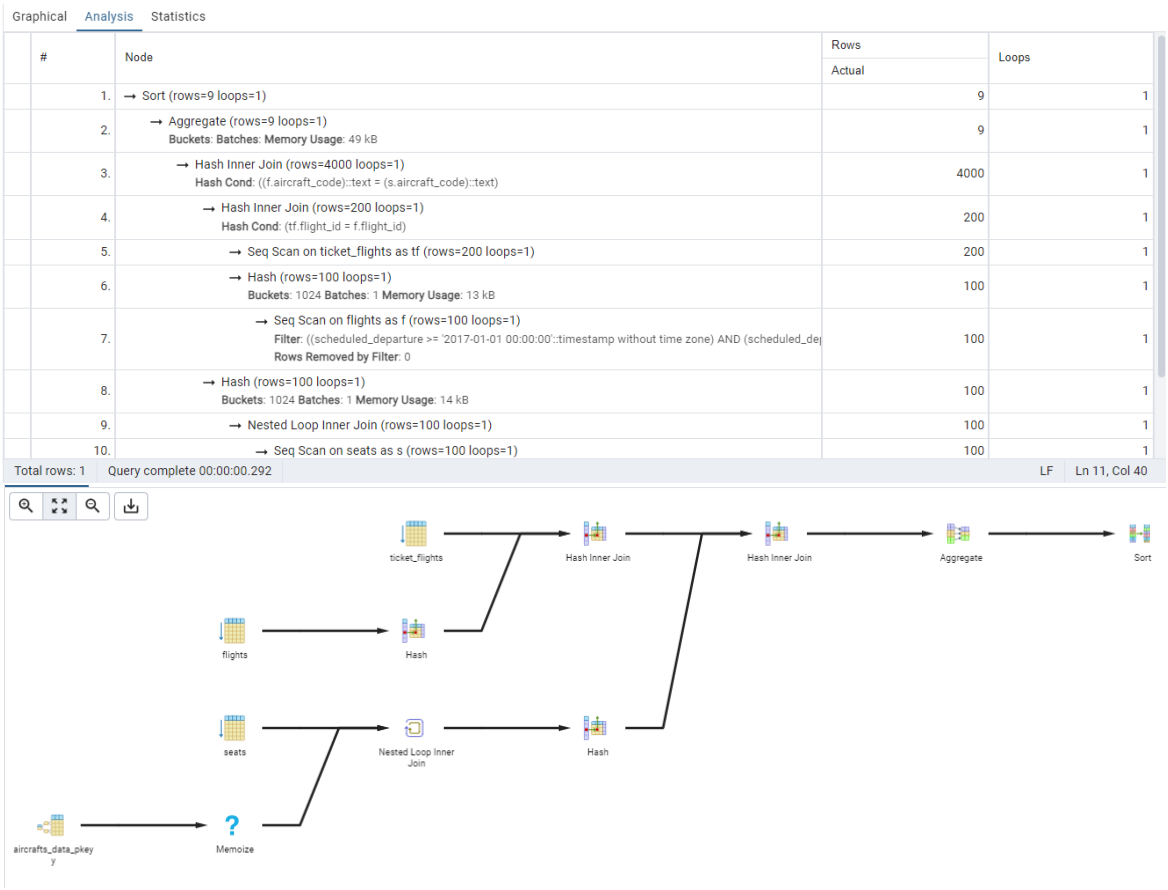


Figura 3: Consulta 1 de 1k de datos con índice

Ç

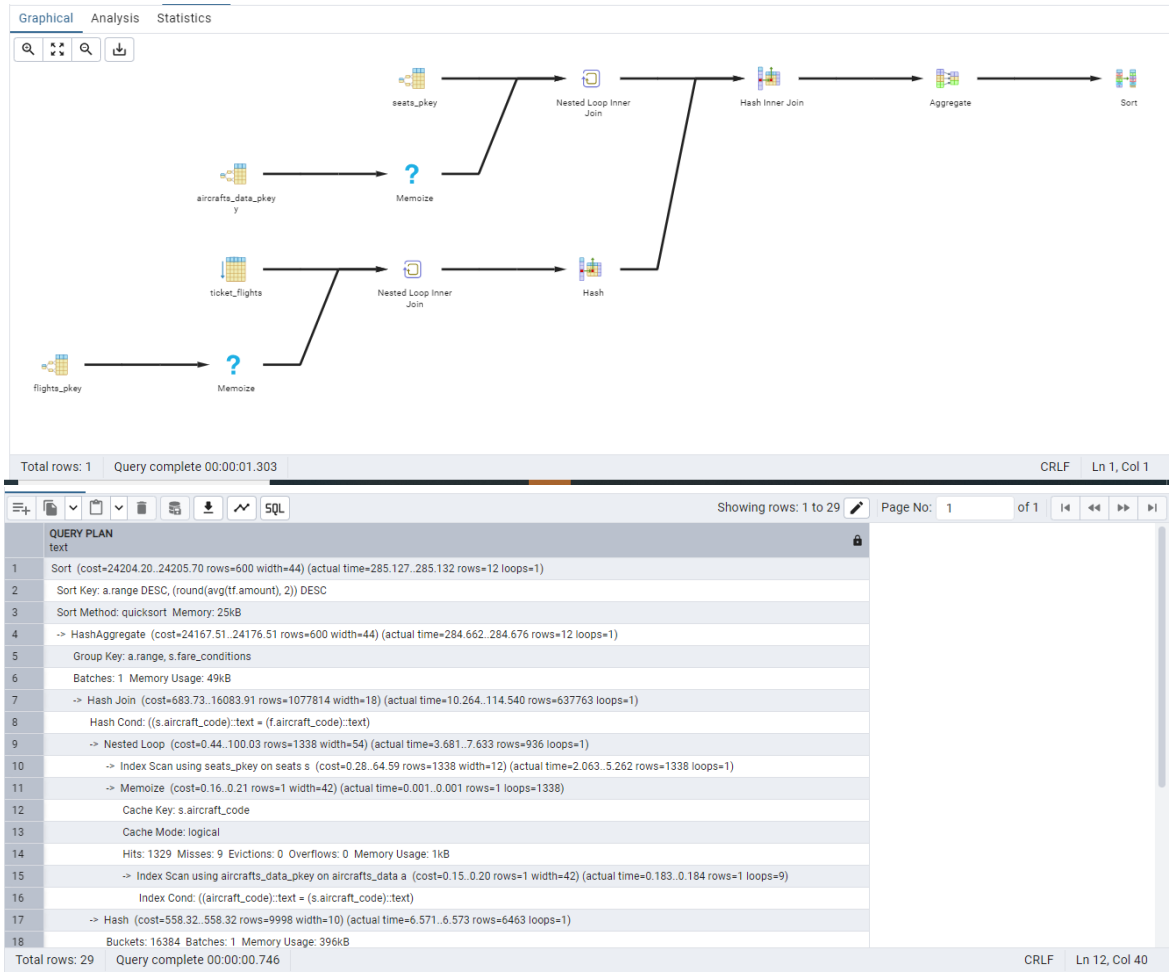


Figura 4: Consulta 1 de 10k de datos sin índice

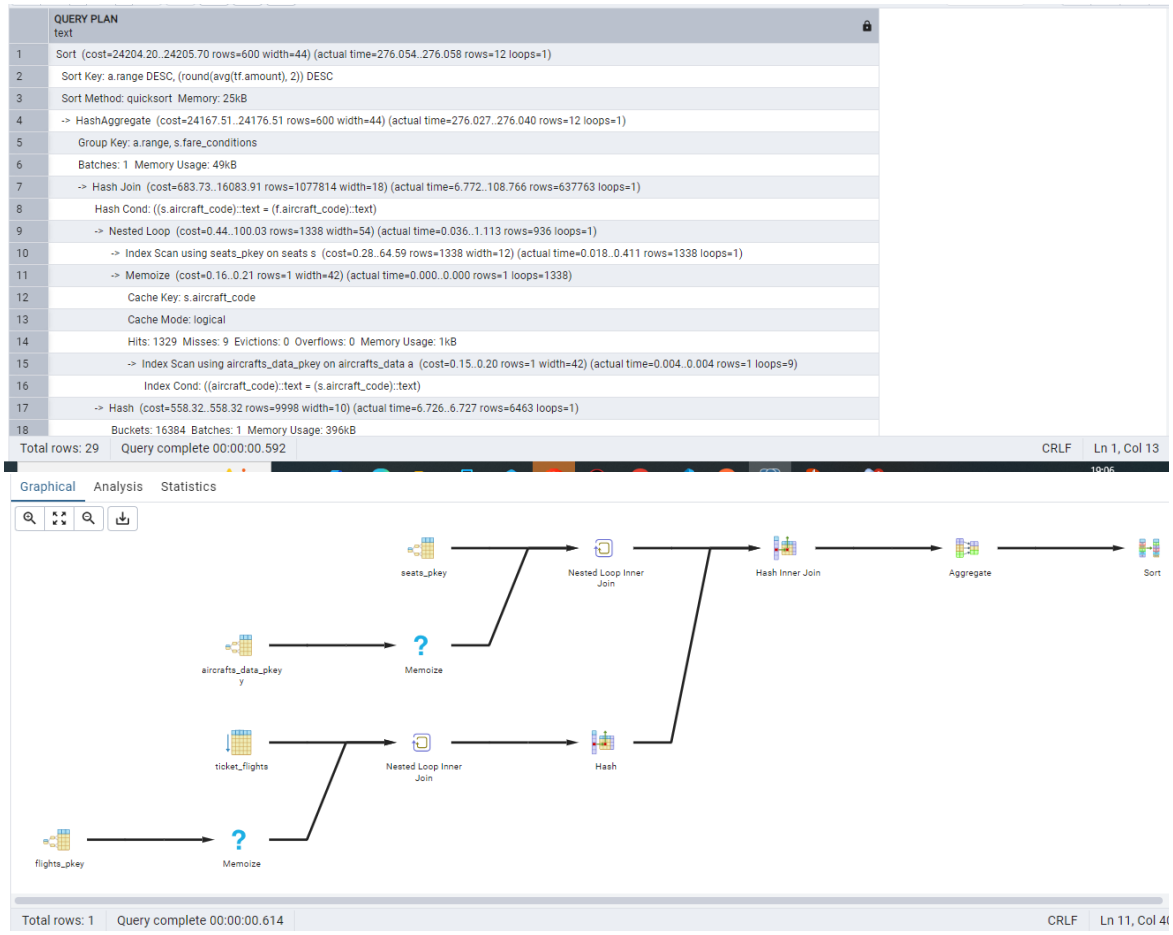


Figura 5: Consulta 1 de 10k de datos con índice

#	Node	Actual	Loops
1.	→ Sort (rows=12 loops=1)	12	1
2.	→ Aggregate (rows=12 loops=1) **Buckets**: **Batches**: **Memory Usage**: 49 kB	12	1
3.	→ Hash Inner Join (rows=10095102 loops=1) **Hash Cond**: ((s.aircraft_code)::text = (f.aircraft_code)::text)	10095102	1
4.	→ Hash Inner Join (rows=936 loops=1) **Hash Cond**: ((s.aircraft_code)::text = (a.aircraft_code)::text)	936	1
5.	→ Seq Scan on seats as s (rows=1338 loops=1)	1338	1
6.	→ Hash (rows=8 loops=1) **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 9 kB	8	1
7.	→ Seq Scan on aircrafts_data as a (rows=8 loops=1)	8	1
8.	→ Hash (rows=100000 loops=1) **Buckets**: 131072 **Batches**: 1 **Memory Usage**: 5157 kB	100000	1
9.	→ Hash Inner Join (rows=100000 loops=1) **Hash Cond**: (tf.flight_id = f.flight_id)	100000	1
10.	→ Seq Scan on ticket_flights as tf (rows=100000 loops=1)	100000	1
11.	→ Hash (rows=33120 loops=1) **Buckets**: 65536 **Batches**: 1 **Memory Usage**: 1806 kB	33120	1
12.	→ Seq Scan on flights as f (rows=33120 loops=1) **Filter**: ((scheduled_departure >= '2017-01-01 00:00:00'::timestamp without time zone) AND (scheduled_departure < **Rows Removed by Filter**: 0	33120	1

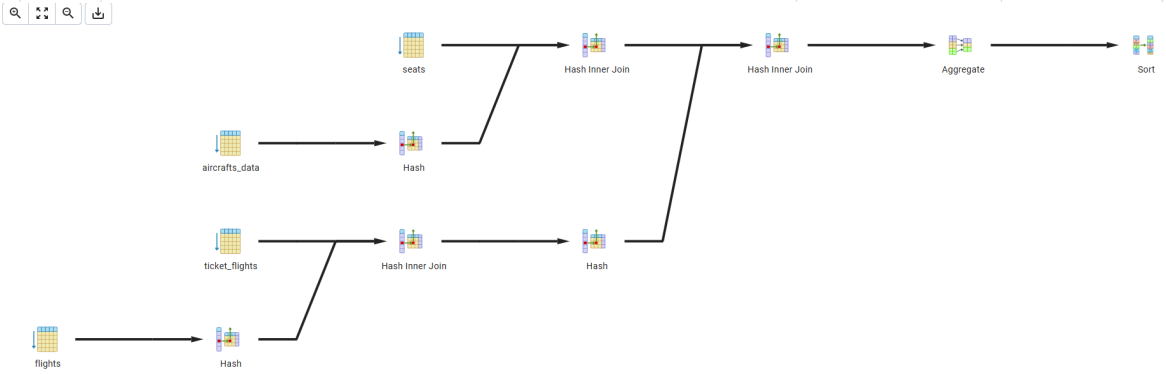


Figura 6: Consulta 1 de 100k de datos sin índice

#	Node	Actual	Loops
1.	→ Sort (rows=12 loops=1)	12	1
2.	→ Aggregate (rows=12 loops=1) **Buckets**: **Batches**: **Memory Usage**: 24 kB	12	1
3.	→ Hash Inner Join (rows=10095102 loops=1) **Hash Cond**: ((s.aircraft_code)::text = (f.aircraft_code)::text)	10095102	1
4.	→ Hash Inner Join (rows=936 loops=1) **Hash Cond**: ((s.aircraft_code)::text = (a.aircraft_code)::text)	936	1
5.	→ Seq Scan on seats as s (rows=1338 loops=1)	1338	1
6.	→ Hash (rows=8 loops=1) **Buckets**: 1024 **Batches**: 1 **Memory Usage**: 9 kB	8	1
7.	→ Seq Scan on aircrafts_data as a (rows=8 loops=1)	8	1
8.	→ Hash (rows=100000 loops=1) **Buckets**: 131072 **Batches**: 1 **Memory Usage**: 5157 kB	100000	1
9.	→ Hash Inner Join (rows=100000 loops=1) **Hash Cond**: (tf.flight_id = f.flight_id)	100000	1
10.	→ Seq Scan on ticket_flights as tf (rows=100000 loops=1)	100000	1
11.	→ Hash (rows=33120 loops=1) **Buckets**: 65536 **Batches**: 1 **Memory Usage**: 1806 kB	33120	1
12.	→ Seq Scan on flights as f (rows=33120 loops=1) **Filter**: ((scheduled_departure >= '2017-01-01 00:00:00'::timestamp without time zone) AND (scheduled_departure < **Rows Removed by Filter**: 0	33120	1

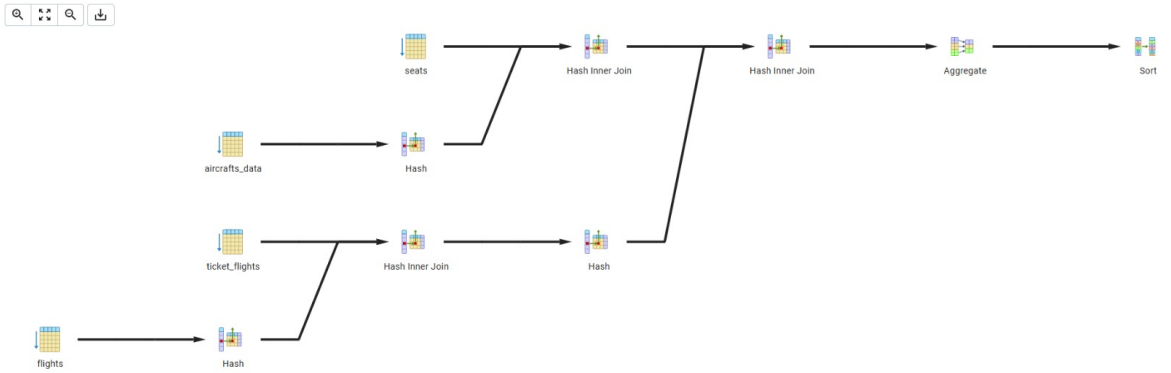
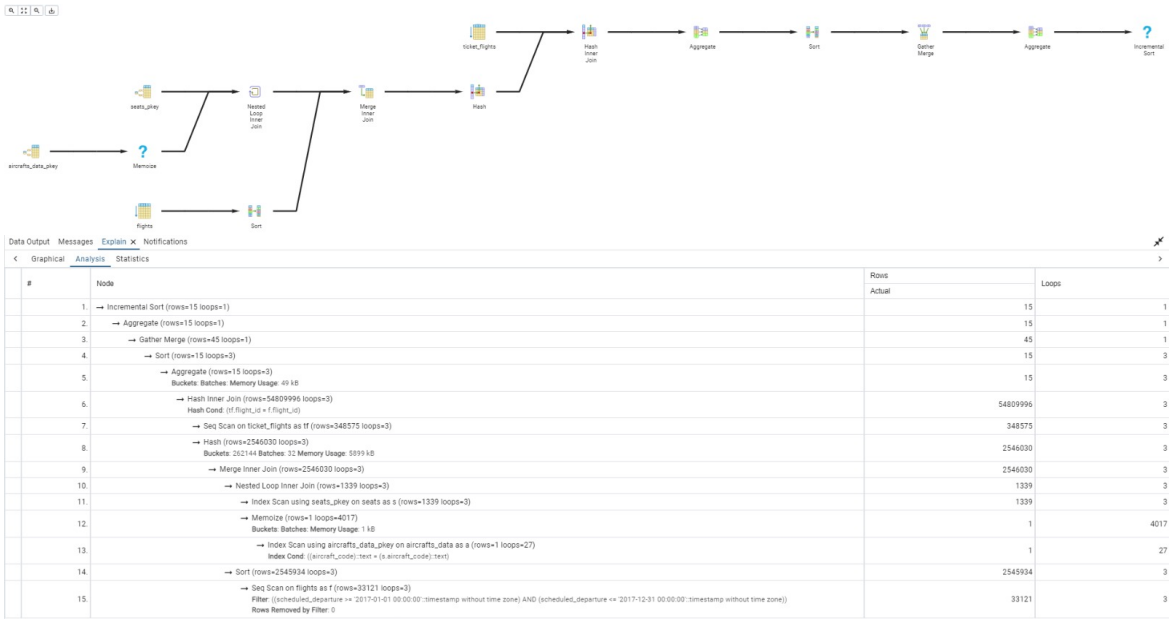


Figura 7: Consulta 1 de 100k de datos con índice

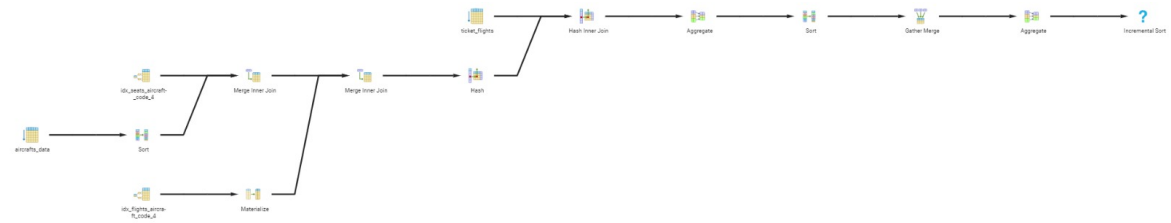


Data Output Messages Explain ✕ Notifications

Graphical Analysis Statistics

#	Node	Rows Actual	Loops
1.	→ Incremental Sort (rows=15 loops=1)	15	1
2.	→ Aggregate (rows=15 loops=1)	15	1
3.	→ Gather Merge (rows=45 loops=1)	45	1
4.	→ Sort (rows=15 loops=3)	15	3
5.	→ Aggregate (rows=15 loops=3) **Buckets**: **Batches**: **Memory Usage**: 49 kB	15	3
6.	→ Hash Inner Join (rows=54809996 loops=3) **Hash Cond**: (tf.flight_id = f.flight_id)	54809996	3
7.	→ Seq Scan on ticket_flights as tf (rows=348575 loops=3)	348575	3
8.	→ Hash (rows=2546030 loops=3) **Buckets**: 262144 **Batches**: 32 **Memory Usage**: 5899 kB	2546030	3
9.	→ Merge Inner Join (rows=2546030 loops=3)	2546030	3
10.	→ Nested Loop Inner Join (rows=1339 loops=3)	1339	3
11.	→ Index Scan using seats_pkey on seats as s (rows=1339 loops=3)	1339	3
12.	→ Memoize (rows=1 loops=4017) **Buckets**: **Batches**: **Memory Usage**: 1 kB	1	4017
13.	→ Index Scan using aircrafts_data_pkey on aircrafts_data as a (rows=1 loops=27) **Index Cond**: ((aircraft_code)::text = (s.aircraft_code)::text)	1	27
14.	→ Sort (rows=2545934 loops=3)	2545934	3
15.	→ Seq Scan on flights as f (rows=33121 loops=3) **Filter**: ((scheduled_departure >= '2017-01-01 00:00:00'::timestamp without time zone) AND (scheduled_departure <= '2017-12-31 00:00:00'::timestamp without time zone)) **Rows Removed by Filter**: 0	33121	3

Figura 8: Consulta 1 de 1 millón de datos sin índice



Data Output Messages Explain ✕ Notifications

Graphical Analysis Statistics

#	Node	Rows Actual	Loops
1.	→ Incremental Sort (rows=15 loops=1)	15	1
2.	→ Aggregate (rows=15 loops=1)	15	1
3.	→ Gather Merge (rows=45 loops=1)	45	1
4.	→ Sort (rows=15 loops=3)	15	3
5.	→ Aggregate (rows=15 loops=3) **Buckets**: **Batches**: **Memory Usage**: 24 kB	15	3
6.	→ Hash Inner Join (rows=54809996 loops=3) **Hash Cond**: (tf.flight_id = f.flight_id)	54809996	3
7.	→ Seq Scan on ticket_flights as tf (rows=348575 loops=3)	348575	3
8.	→ Hash (rows=2546030 loops=3) **Buckets**: 262144 **Batches**: 32 **Memory Usage**: 5936 kB	2546030	3
9.	→ Merge Inner Join (rows=2546030 loops=3)	2546030	3
10.	→ Merge Inner Join (rows=1339 loops=3)	1339	3
11.	→ Index Scan using idx_seats_aircraft_code_4 on seats as s (rows=1339 loops=3)	1339	3
12.	→ Sort (rows=9 loops=3)	9	3
13.	→ Seq Scan on aircrafts_data as a (rows=9 loops=3)	9	3
14.	→ Materialize (rows=2545934 loops=3)	2545934	3
15.	→ Index Scan using idx_flights_aircraft_code_4 on flights as f (rows=33121 loops=3) **Filter**: ((scheduled_departure >= '2017-01-01 00:00:00'::timestamp without time zone) AND (scheduled_departure <= '2 **Rows Removed by Filter**: 0	33121	3

Figura 9: Consulta 1 de 1 millón de datos con índice

5.6.2. Consulta 2

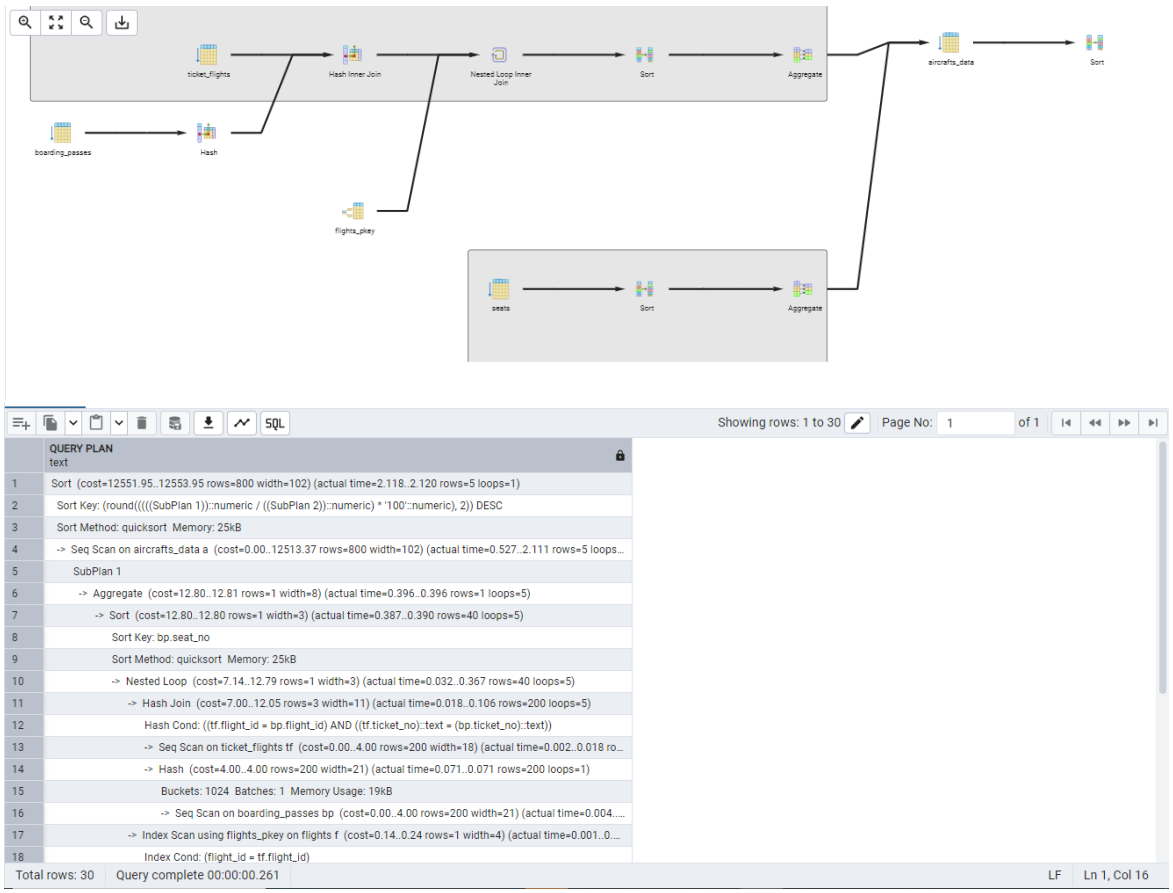


Figura 10: Consulta 2 de 1k de datos sin índice

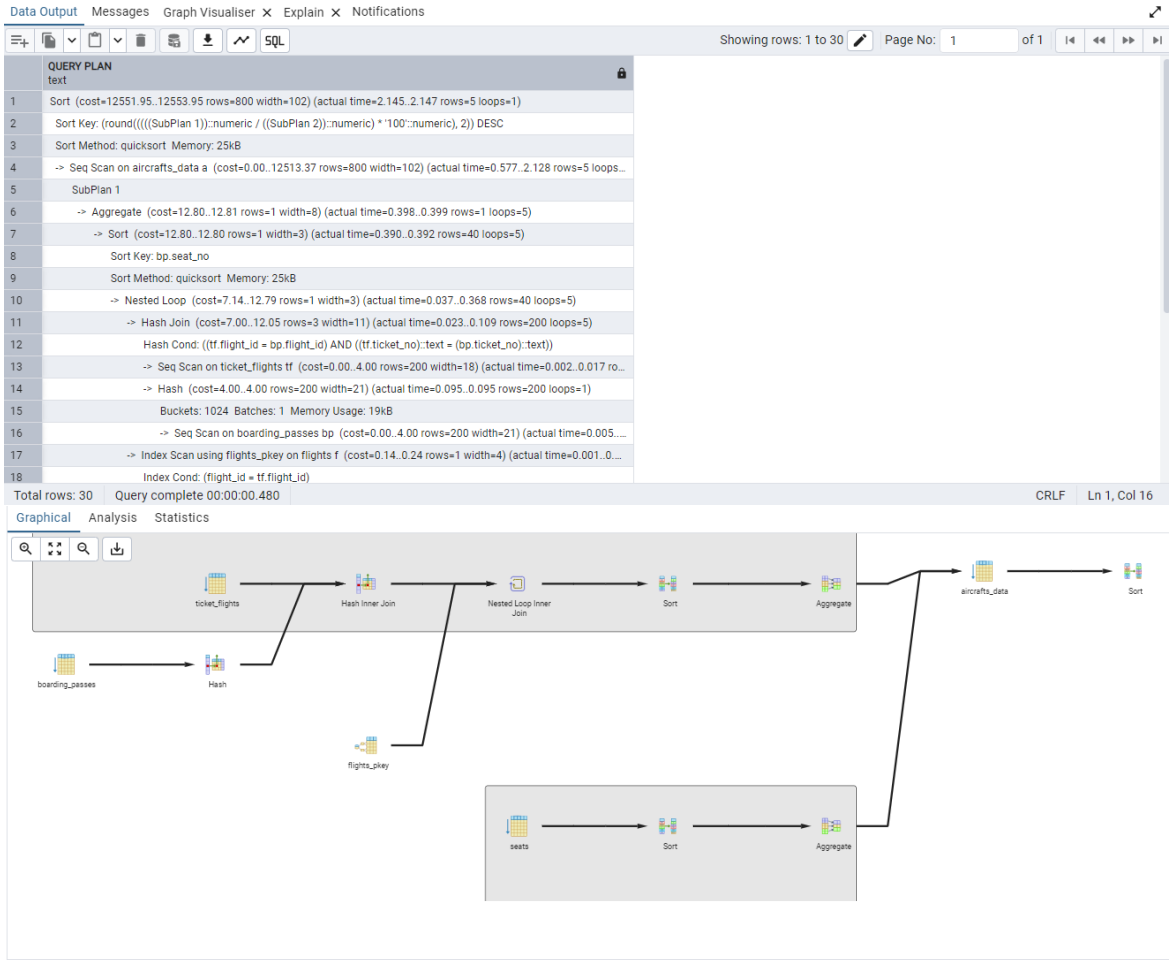


Figura 11: Consulta 1 de 1k de datos con índice

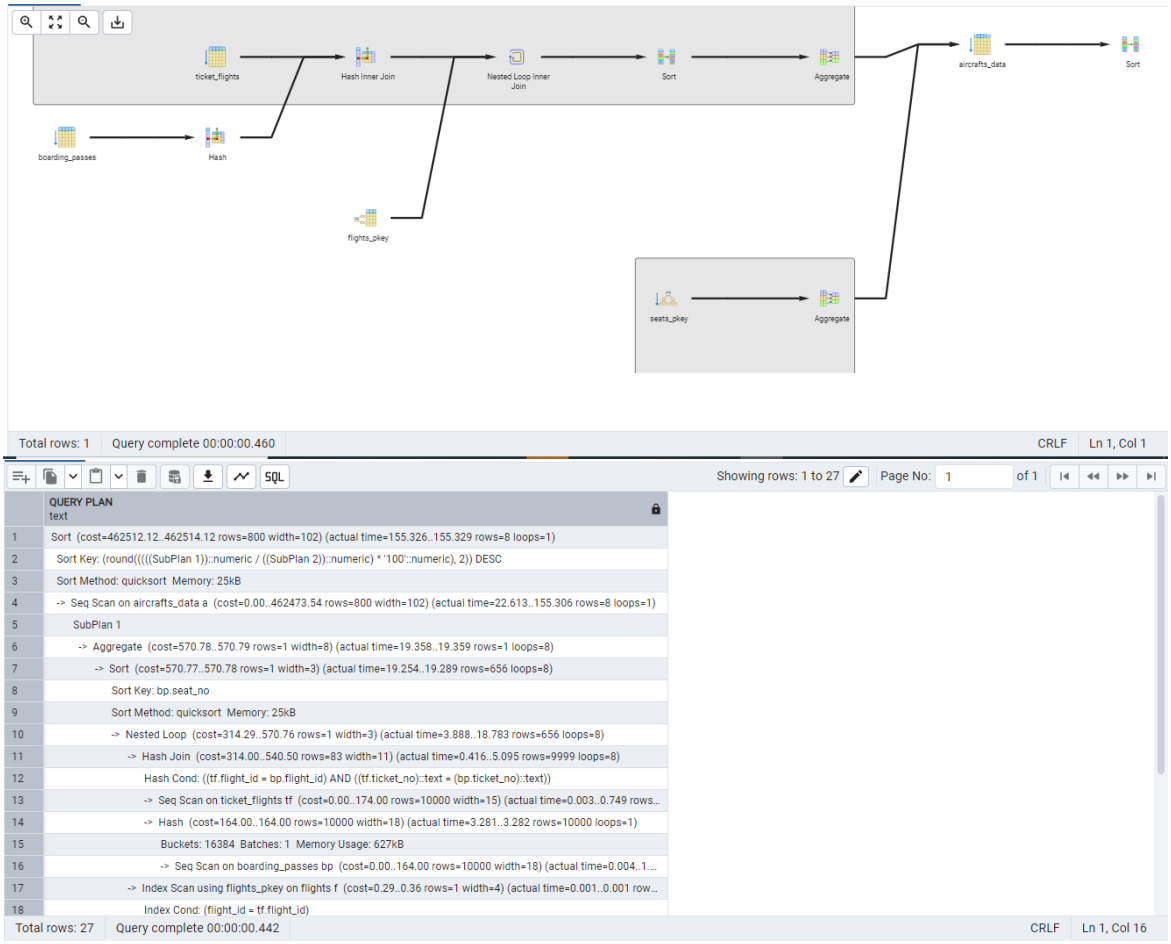
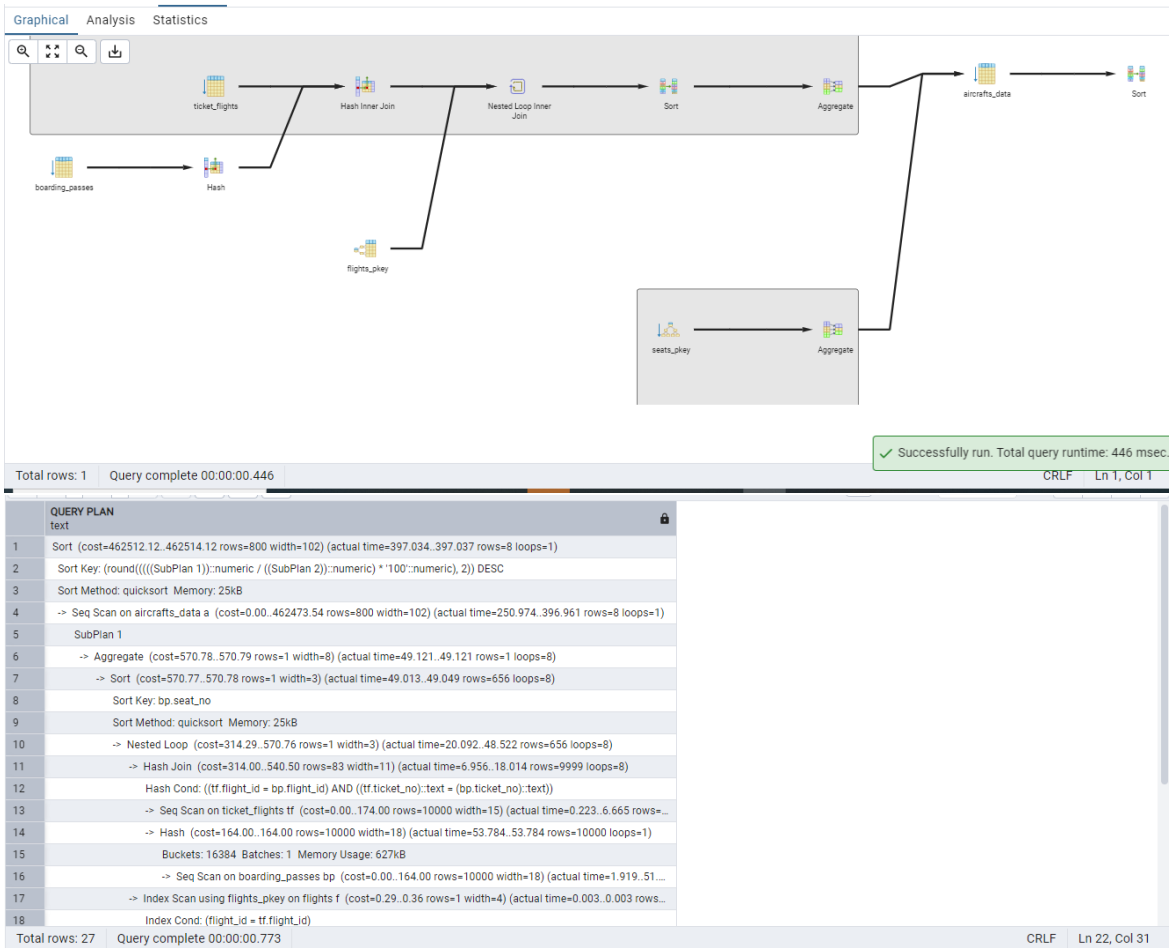


Figura 12: Consulta 2 de 10k de datos sin índice



	QUERY PLAN text
1	Sort (cost=462512.12..462514.12 rows=800 width=102) (actual time=397.034..397.037 rows=8 loops=1)
2	Sort Key: (round(((((SubPlan 1))::numeric / ((SubPlan 2))::numeric) * '100'::numeric), 2)) DESC
3	Sort Method: quicksort Memory: 25kB
4	-> Seq Scan on aircrafts_data a (cost=0.00..462473.54 rows=800 width=102) (actual time=250.974..396.961 rows=8 loops=1)
5	SubPlan 1
6	-> Aggregate (cost=570.78..570.79 rows=1 width=8) (actual time=49.121..49.121 rows=1 loops=8)
7	-> Sort (cost=570.77..570.78 rows=1 width=3) (actual time=49.013..49.049 rows=656 loops=8)
8	Sort Key: bp.seat_no
9	Sort Method: quicksort Memory: 25kB
10	-> Nested Loop (cost=314.29..570.76 rows=1 width=3) (actual time=20.092..48.522 rows=656 loops=8)
11	-> Hash Join (cost=314.00..540.50 rows=83 width=11) (actual time=6.956..18.014 rows=9999 loops=8)
12	Hash Cond: ((tf.flight_id = bp.flight_id) AND ((tf.ticket_no)::text = (bp.ticket_no)::text))
13	-> Seq Scan on ticket_flights tf (cost=0.00..174.00 rows=10000 width=15) (actual time=0.223..6.665 rows=...
14	-> Hash (cost=164.00..164.00 rows=10000 width=18) (actual time=53.784..53.784 rows=10000 loops=1)
15	Buckets: 16384 Batches: 1 Memory Usage: 627kB
16	-> Seq Scan on boarding_passes bp (cost=0.00..164.00 rows=10000 width=18) (actual time=1.919..51....
17	-> Index Scan using flights_pkey on flights f (cost=0.29..0.36 rows=1 width=4) (actual time=0.003..0.003 rows...
18	Index Cond: (flight_id = tf.flight_id)

Total rows: 27 | Query complete 00:00:00.773 | CRLF | Ln 22, Col 31

Figura 13: Consulta 2 de 10k de datos con índice

#	Node	Actual	Loops
1.	→ Sort (rows=8 loops=1)	8	1
2.	→ Seq Scan on aircrafts_data as a (rows=8 loops=1)	8	1
3.	→ Aggregate (rows=1 loops=8)	1	8
4.	→ Sort (rows=10596 loops=8)	10596	8
5.	→ Hash Inner Join (rows=10596 loops=8) **Hash Cond**: ((bp.flight_id = f.flight_id) AND ((bp.ticket_no)::text = (tf.ticket_no)::text))	10596	8
6.	→ Seq Scan on boarding_passes as bp (rows=100000 loops=7)	100000	7
7.	→ Hash (rows=10596 loops=8) **Buckets**: 65536 **Batches**: 1 **Memory Usage**: 2195 kB	10596	8
8.	→ Hash Inner Join (rows=10596 loops=8) **Hash Cond**: (tf.flight_id = f.flight_id)	10596	8
9.	→ Seq Scan on ticket_flights as tf (rows=100000 loops=7)	100000	7
10.	→ Hash (rows=4064 loops=8) **Buckets**: 16384 **Batches**: 1 **Memory Usage**: 455 kB	4064	8
11.	→ Seq Scan on flights as f (rows=4064 loops=8) **Filter**: ((aircraft_code)::text = (a.aircraft_code)::text) **Rows Removed by Filter**: 29056	4064	8
12.	→ Aggregate (rows=1 loops=8)	1	8
13.	→ Index Only Scan using seats_pkey on seats as s (rows=117 loops=8) **Index Cond**: (aircraft_code = (a.aircraft_code)::text)	117	8

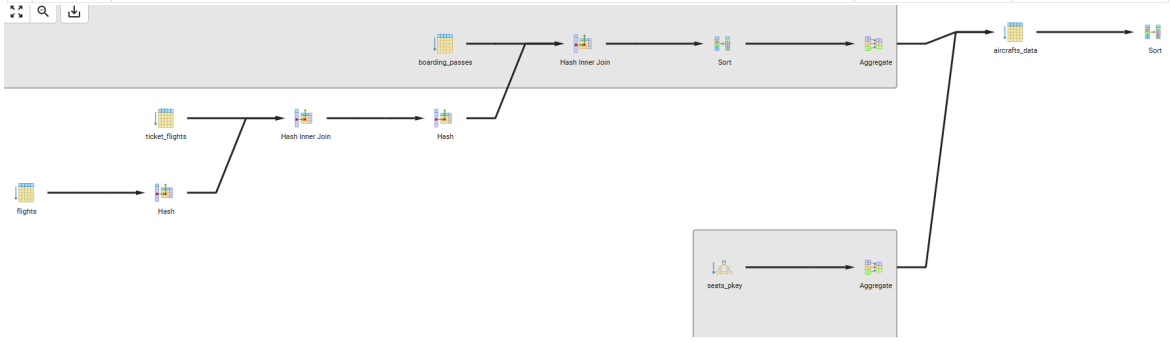


Figura 14: Consulta 2 de 100k de datos sin índice

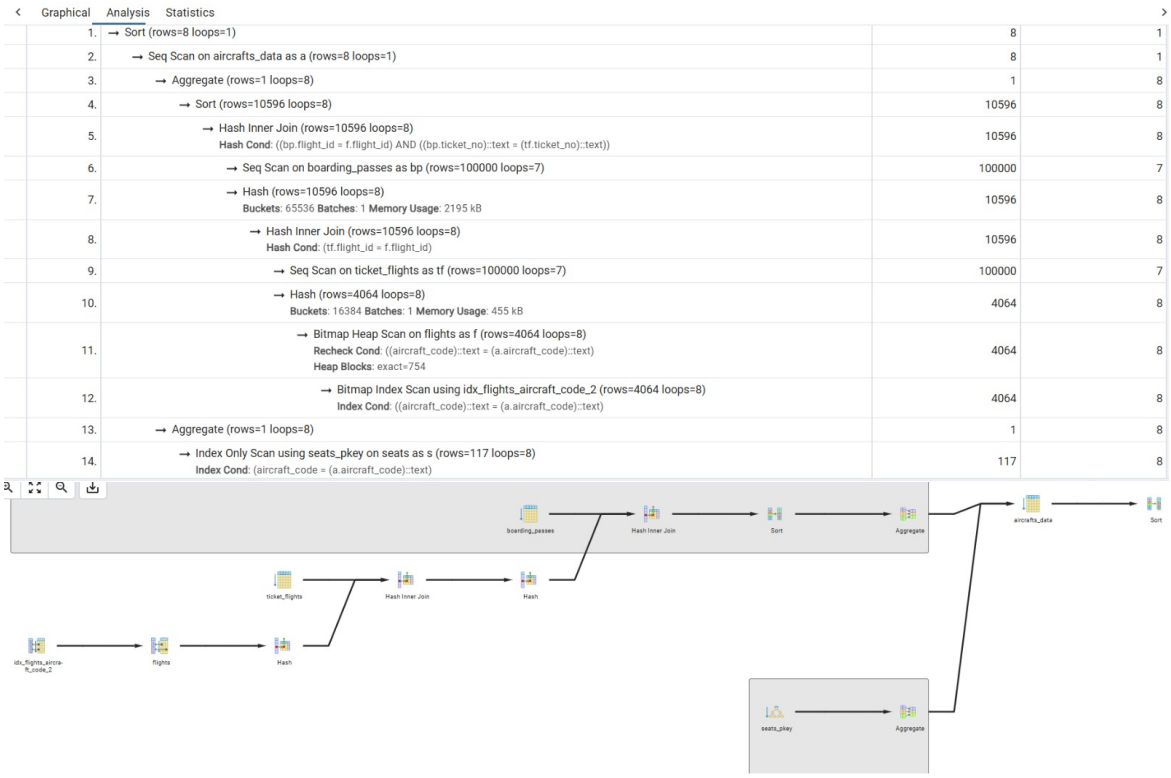


Figura 15: Consulta 2 de 100k de datos con índice

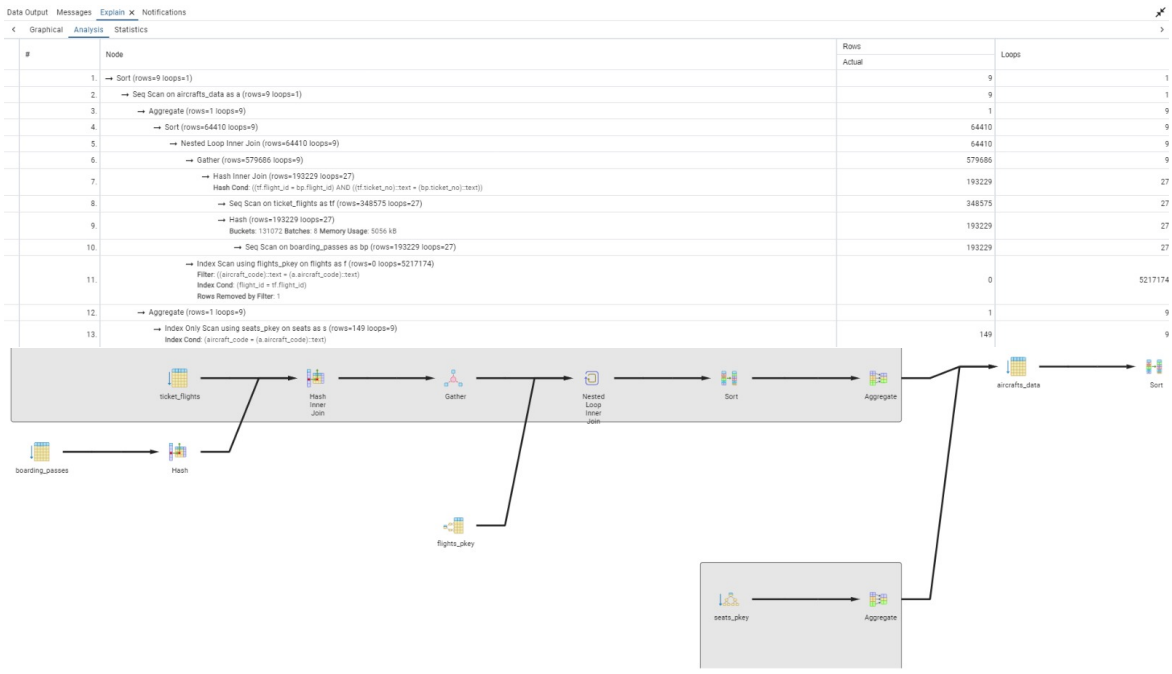


Figura 16: Consulta 2 de 1 millón de datos sin índice

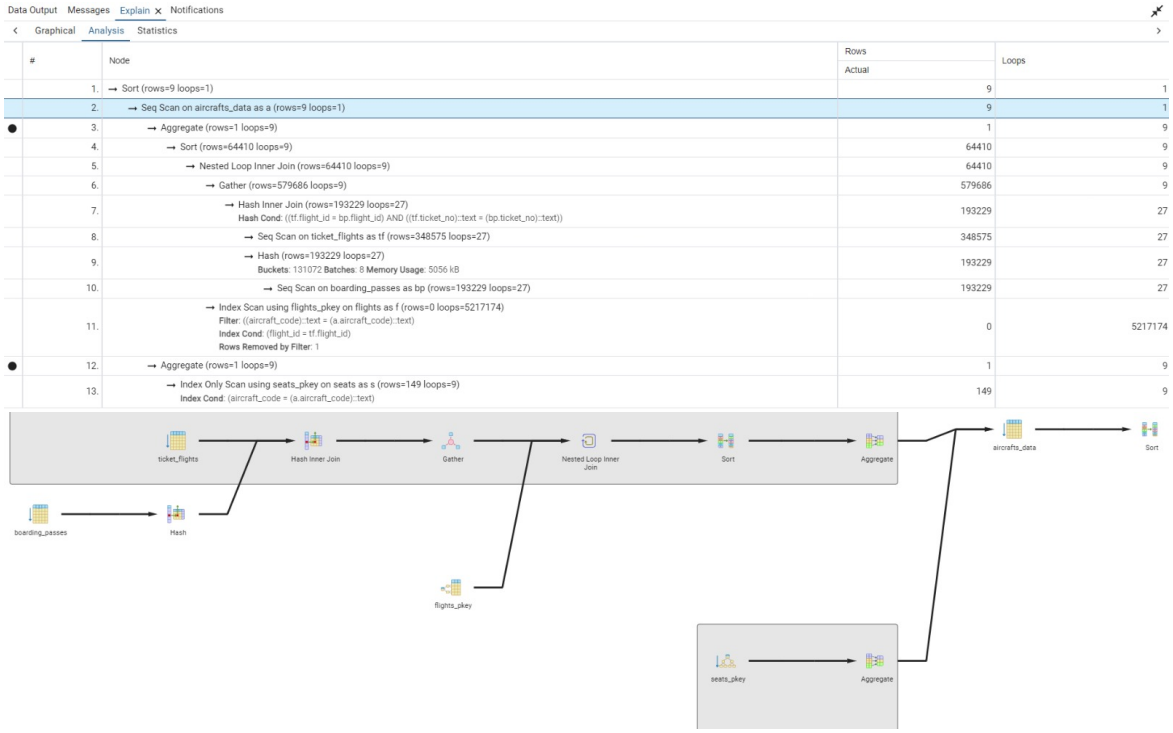
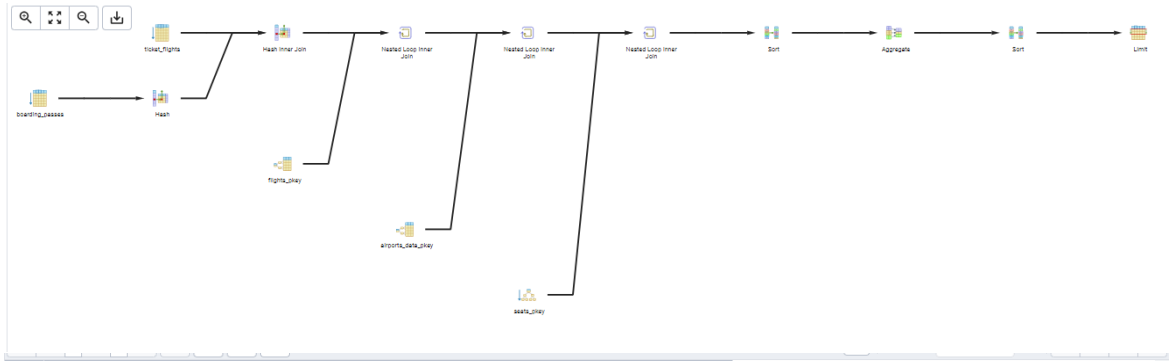


Figura 17: Consulta 2 de 1 millón de datos con índice

5.6.3. Consulta 3



	QUERY PLAN text
1	Limit (cost=19.42..19.44 rows=10 width=112) (actual time=10.425..10.431 rows=10 loops=1)
2	-> Sort (cost=19.42..19.52 rows=40 width=112) (actual time=10.424..10.428 rows=10 loops=1)
3	Sort Key: (count(DISTINCT bp.ticket_no)) DESC
4	Sort Method: quicksort Memory: 25kB
5	-> GroupAggregate (cost=16.75..18.55 rows=40 width=112) (actual time=5.698..10.420 rows=10 loops=1)
6	Group Key: ad.city
7	-> Sort (cost=16.75..16.85 rows=40 width=62) (actual time=5.301..5.518 rows=4000 loops=1)
8	Sort Key: ad.city, f.flight_id
9	Sort Method: quicksort Memory: 315kB
10	-> Nested Loop (cost=7.43..15.69 rows=40 width=62) (actual time=0.118..2.748 rows=4000 loops=1)
11	-> Nested Loop (cost=7.29..14.06 rows=2 width=64) (actual time=0.108..0.841 rows=200 loops=1)
12	-> Nested Loop (cost=7.14..12.77 rows=2 width=38) (actual time=0.101..0.478 rows=200 loops=1)
13	-> Hash Join (cost=7.00..12.05 rows=3 width=31) (actual time=0.093..0.203 rows=200 loops=1)
14	Hash Cond: ((tf.flight_id = bp.flight_id) AND ((tf.ticket_no)::text = (bp.ticket_no)::text))
15	-> Seq Scan on ticket_flights tf (cost=0.00..4.00 rows=200 width=24) (actual time=0.014..0.030 rows=200 loops=...
16	-> Hash (cost=4.00..4.00 rows=200 width=21) (actual time=0.072..0.073 rows=200 loops=1)
17	Buckets: 1024 Batches: 1 Memory Usage: 19kB
18	-> Seq Scan on boarding_passes bp (cost=0.00..4.00 rows=200 width=21) (actual time=0.006..0.031 rows=20

Total rows: 27 | Query complete 00:00:00.377 | LF | Ln 1, Col 16

Figura 18: Consulta 3 de 1k de datos sin índice

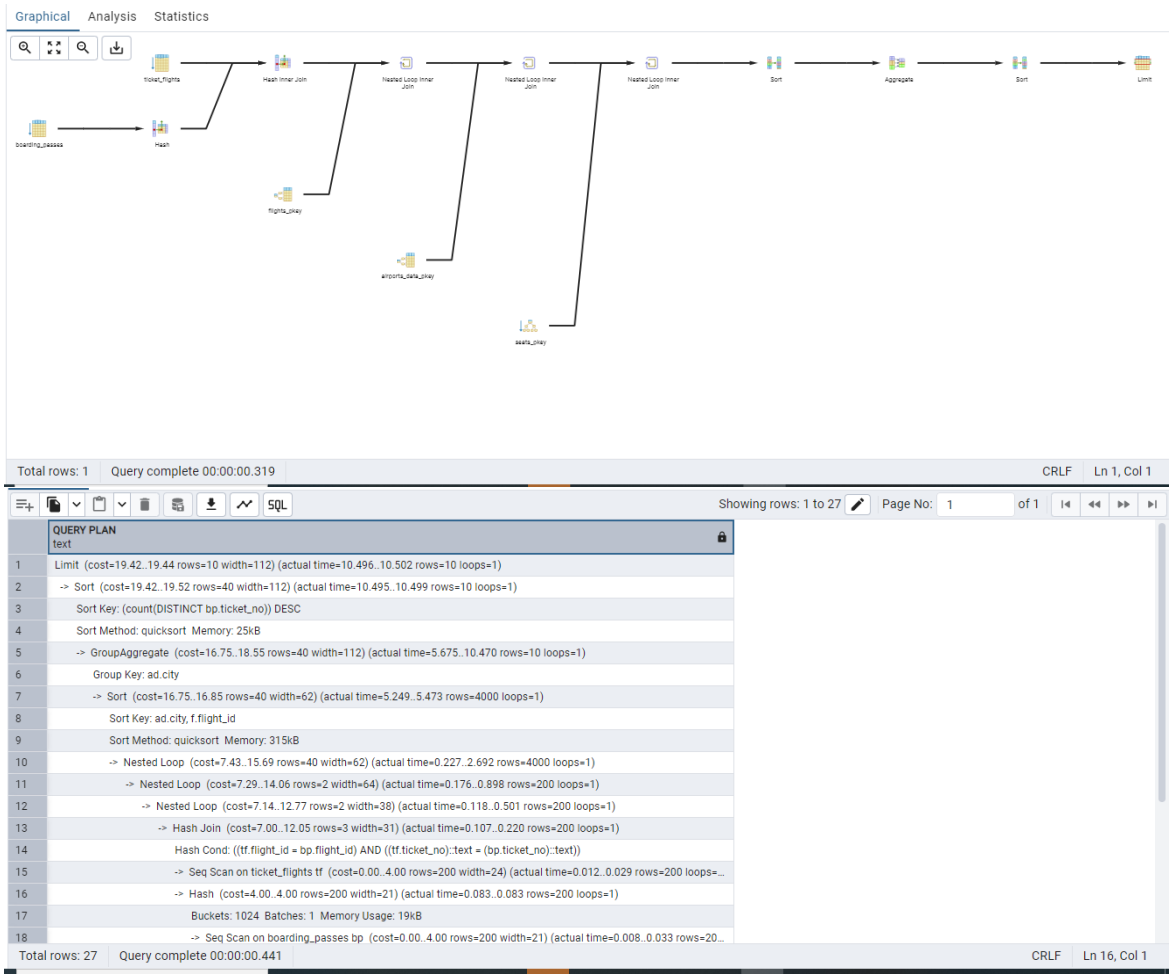


Figura 19: Consulta 3 de 1k de datos CON índice

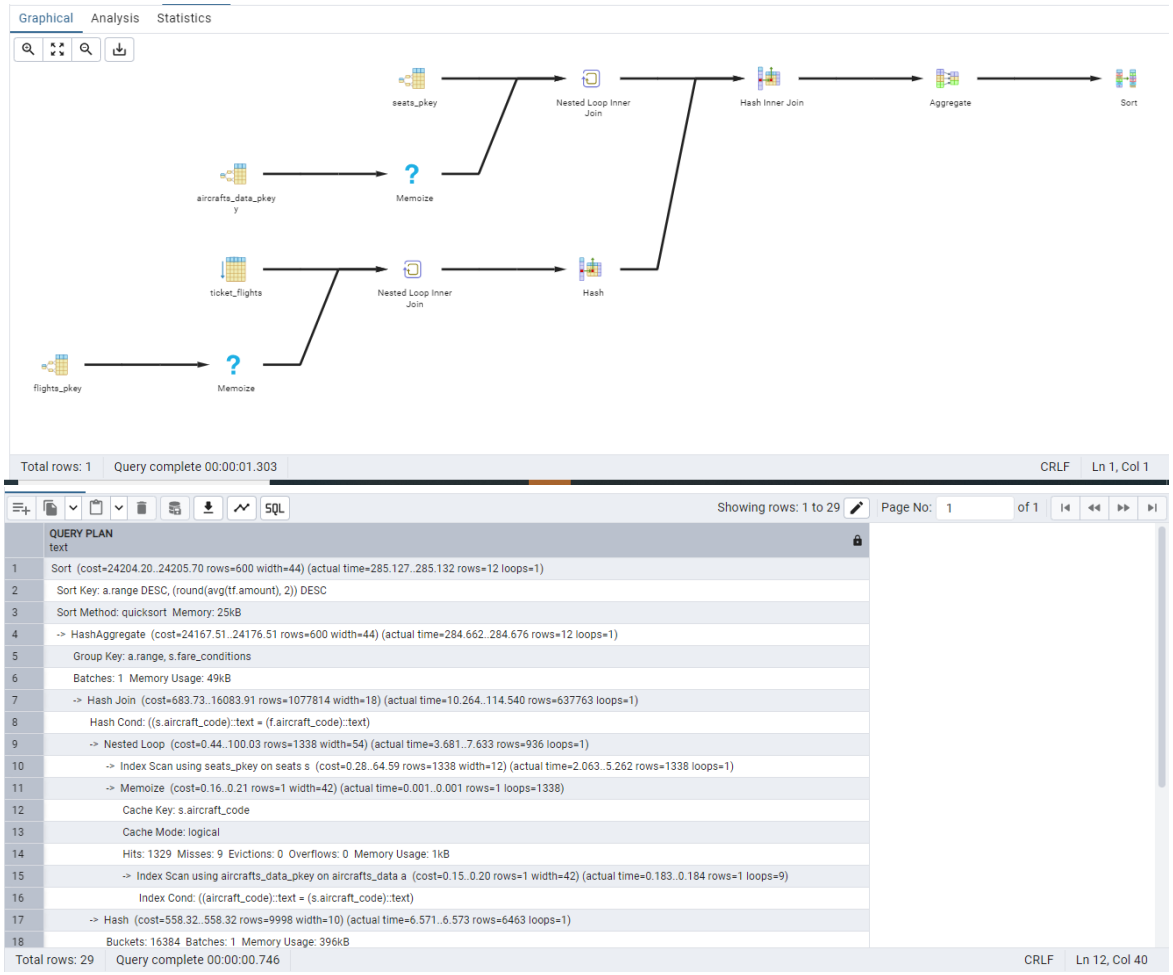


Figura 20: Consulta 3 de 1k de datos sin índice

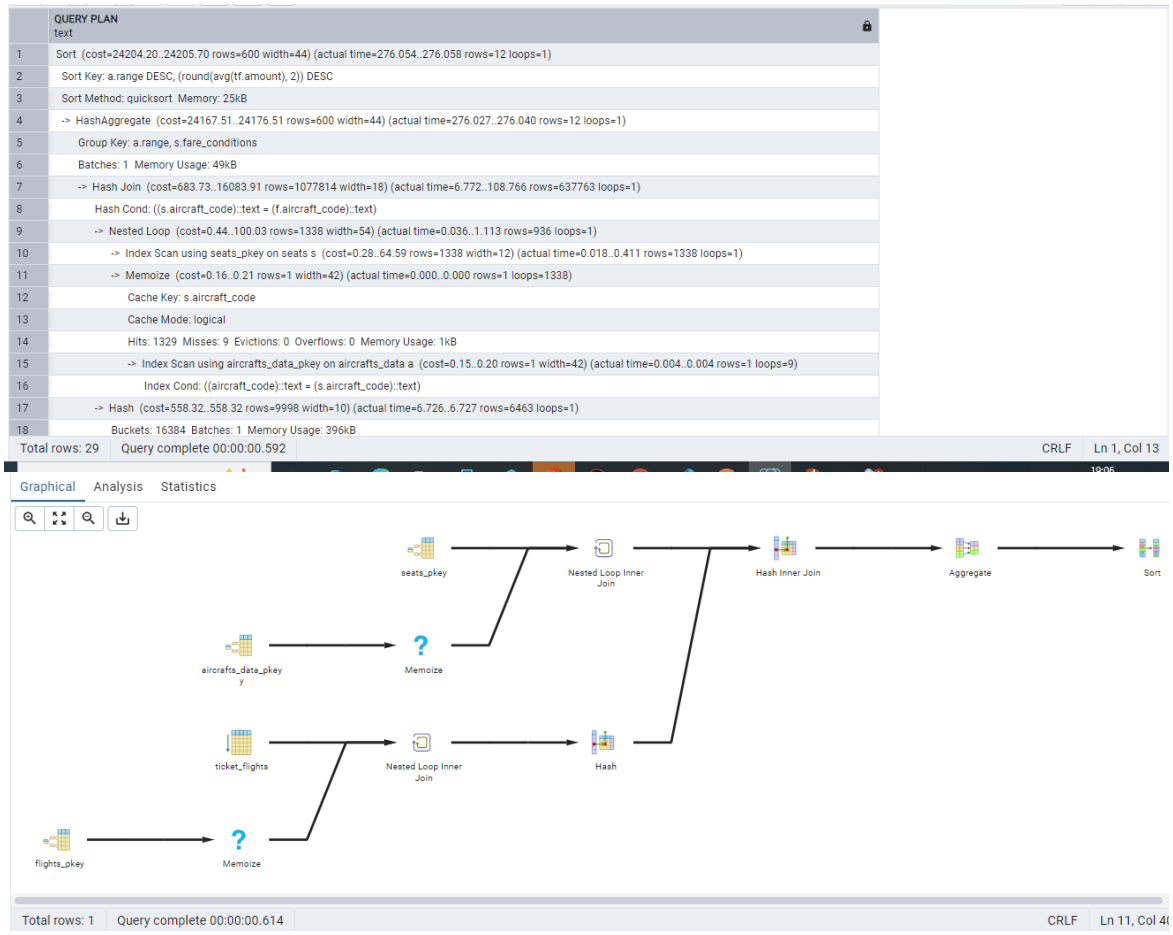


Figura 21: Consulta 3 de 1k de datos con índice

1.	→ Limit (rows=10 loops=1)	10	1
2.	→ Sort (rows=10 loops=1)	10	1
3.	→ Aggregate (rows=85 loops=1)	85	1
4.	→ Sort (rows=16203887 loops=1)	16203887	1
5.	→ Nested Loop Inner Join (rows=16203887 loops=1)	16203887	1
6.	→ Nested Loop Inner Join (rows=99762 loops=1) **Join Filter**: ((ad.airport_code)::text = (f.departure_airport)::text)	99762	1
7.	→ Seq Scan on airports_data as ad (rows=103 loops=1)	103	1
8.	→ Materialize (rows=99999 loops=103)	99999	103
9.	→ Nested Loop Inner Join (rows=99999 loops=1)	99999	1
10.	→ Hash Inner Join (rows=99999 loops=1) **Hash Cond**: ((tf.flight_id = bp.flight_id) AND ((tf.ticket_no)::text = (bp.ticket_no)::text))	99999	1
11.	→ Seq Scan on ticket_flights as tf (rows=100000 loops=1)	100000	1
12.	→ Hash (rows=100000 loops=1) **Buckets**: 131072 **Batches**: 1 **Memory Usage**: 6103 kB	100000	1
13.	→ Seq Scan on boarding_passes as bp (rows=100000 loops=1)	100000	1
14.	→ Index Scan using flights_pkey on flights as f (rows=1 loops=99999) **Index Cond**: (flight_id = tf.flight_id)	1	99999
15.	→ Index Only Scan using seats_pkey on seats as s (rows=162 loops=99762) **Index Cond**: (aircraft_code = (f.aircraft_code)::text)	162	99762

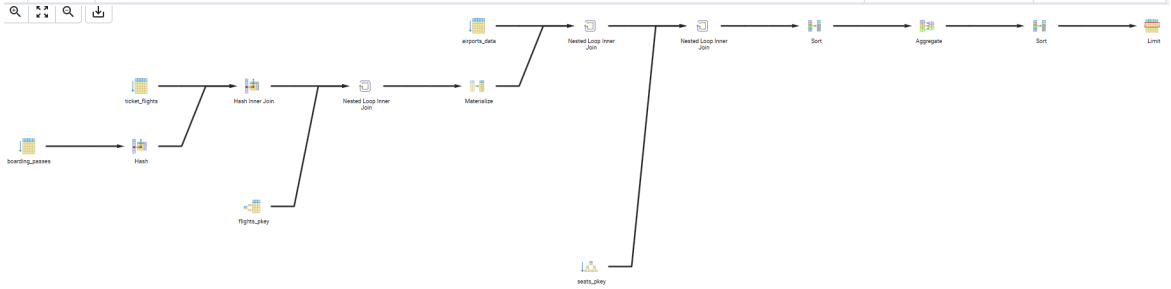


Figura 22: Consulta 3 de 100k de datos sin índice

1.	→ Limit (rows=10 loops=1)	10	1
2.	→ Sort (rows=10 loops=1)	10	1
3.	→ Aggregate (rows=85 loops=1)	85	1
4.	→ Sort (rows=16203887 loops=1)	16203887	1
5.	→ Nested Loop Inner Join (rows=16203887 loops=1)	16203887	1
6.	→ Nested Loop Inner Join (rows=99762 loops=1) **Join Filter**: ((f.departure_airport)::text = (ad.airport_code)::text)	99762	1
7.	→ Seq Scan on airports_data as ad (rows=103 loops=1)	103	1
8.	→ Materialize (rows=99999 loops=103)	99999	103
9.	→ Nested Loop Inner Join (rows=99999 loops=1)	99999	1
10.	→ Hash Inner Join (rows=99999 loops=1) **Hash Cond**: ((tf.flight_id = bp.flight_id) AND ((tf.ticket_no)::text = (bp.ticket_no)::text))	99999	1
11.	→ Seq Scan on ticket_flights as tf (rows=100000 loops=1)	100000	1
12.	→ Hash (rows=100000 loops=1) **Buckets**: 131072 **Batches**: 1 **Memory Usage**: 6103 kB	100000	1
13.	→ Seq Scan on boarding_passes as bp (rows=100000 loops=1)	100000	1
14.	→ Index Scan using flights_pkey on flights as f (rows=1 loops=99999) **Index Cond**: (flight_id = tf.flight_id)	1	99999
15.	→ Index Only Scan using seats_pkey on seats as s (rows=162 loops=99762) **Index Cond**: (aircraft_code = (f.aircraft_code)::text)	162	99762

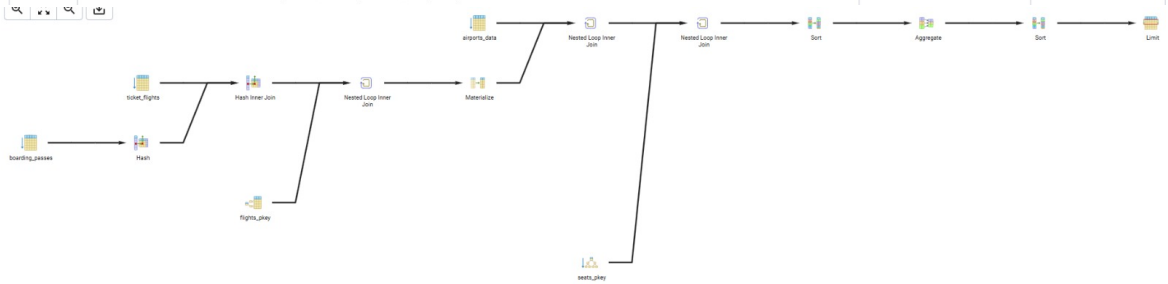
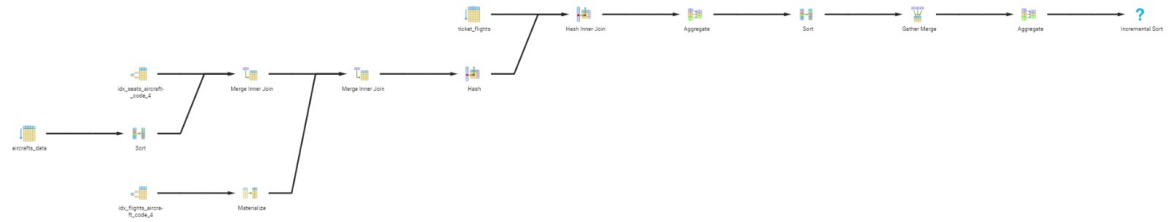
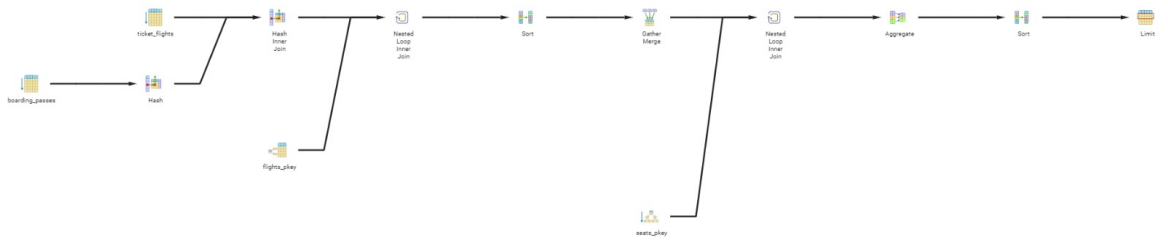


Figura 23: Consulta 3 de 100k de datos con índice



	#	Node	Rows Actual	Loops
	1.	→ Incremental Sort (rows=15 loops=1)	15	1
	2.	→ Aggregate (rows=15 loops=1)	15	1
	3.	→ Gather Merge (rows=45 loops=1)	45	1
	4.	→ Sort (rows=15 loops=3)	15	3
	5.	→ Aggregate (rows=15 loops=3) **Buckets**: **Batches**: **Memory Usage**: 49 kB	15	3
●	6.	→ Hash Inner Join (rows=54809996 loops=3) **Hash Cond**: (tf.flight_id = f.flight_id)	54809996	3
	7.	→ Seq Scan on ticket_flights as tf (rows=348575 loops=3)	348575	3
	8.	→ Hash (rows=2546030 loops=3) **Buckets**: 262144 **Batches**: 32 **Memory Usage**: 5899 kB	2546030	3
	9.	→ Merge Inner Join (rows=2546030 loops=3)	2546030	3
	10.	→ Nested Loop Inner Join (rows=1339 loops=3)	1339	3
	11.	→ Index Scan using seats_pkey on seats as s (rows=1339 loops=3)	1339	3
	12.	→ Memoize (rows=1 loops=4017) **Buckets**: **Batches**: **Memory Usage**: 1 kB	1	4017
	13.	→ Index Scan using aircrafts_data_pkey on aircrafts_data as a (rows=1 loops=27) **Index Cond**: ((aircraft_code)::text = (s.aircraft_code)::text)	1	27
	14.	→ Sort (rows=2545934 loops=3)	2545934	3
	15.	→ Seq Scan on flights as f (rows=33121 loops=3) **Filter**: ((scheduled_departure >= '2017-01-01 00:00:00'::timestamp without time zone) AND (scheduled_departure <= '2017-12-31 00:00 **Rows Removed by Filter**: 0	33121	3

Figura 24: Consulta 3 de 1 millón de datos sin índice



	#	Node	Rows Actual	Loops
	1.	→ Incremental Sort (rows=15 loops=1)	15	1
	2.	→ Aggregate (rows=15 loops=1)	15	1
	3.	→ Gather Merge (rows=45 loops=1)	45	1
	4.	→ Sort (rows=15 loops=3)	15	3
	5.	→ Aggregate (rows=15 loops=3) **Buckets**: **Batches**: **Memory Usage**: 49 kB	15	3
●	6.	→ Hash Inner Join (rows=54809996 loops=3) **Hash Cond**: (tf.flight_id = f.flight_id)	54809996	3
	7.	→ Seq Scan on ticket_flights as tf (rows=348575 loops=3)	348575	3
	8.	→ Hash (rows=2546030 loops=3) **Buckets**: 262144 **Batches**: 32 **Memory Usage**: 5899 kB	2546030	3
	9.	→ Merge Inner Join (rows=2546030 loops=3)	2546030	3
	10.	→ Nested Loop Inner Join (rows=1339 loops=3)	1339	3
	11.	→ Index Scan using seats_pkey on seats as s (rows=1339 loops=3)	1339	3
	12.	→ Memoize (rows=1 loops=4017) **Buckets**: **Batches**: **Memory Usage**: 1 kB	1	4017
	13.	→ Index Scan using aircrafts_data_pkey on aircrafts_data as a (rows=1 loops=27) **Index Cond**: ((aircraft_code)::text = (s.aircraft_code)::text)	1	27
	14.	→ Sort (rows=2545934 loops=3)	2545934	3
	15.	→ Seq Scan on flights as f (rows=33121 loops=3) **Filter**: ((scheduled_departure >= '2017-01-01 00:00:00'::timestamp without time zone) AND (scheduled_departure <= '2017-12-31 00:00 **Rows Removed by Filter**: 0	33121	3

Figura 25: Consulta 3 de 1 millón de datos con índice

5.7. Análisis y Discusión

Tras realizar múltiples consultas complejas sobre un esquema que simula operaciones de una aerolínea, se concluye lo siguiente respecto al uso de índices:

1. **Importancia del índice en consultas con múltiples *JOINs***: Las consultas utilizadas incluyen varias uniones entre tablas como `flights`, `ticket_flights`, `boarding_passes`, `seats`, y `aircrafts_data`, lo que conlleva una gran carga computacional. El uso de índices permitió reducir significativamente los tiempos de respuesta, especialmente en las operaciones de filtrado por claves externas y agrupamiento.
2. **Tipos de índices empleados**: Se aplicaron principalmente índices `BTREE` en columnas frecuentemente utilizadas en `JOINs` (`flight_id`, `ticket_no`, `aircraft_code`), así como en columnas usadas en filtros de fecha (`scheduled_departure`) y agrupamientos (`fare_conditions`, `range`). Estos índices resultan eficientes para operaciones de rango y búsqueda exacta, que son comunes en nuestras consultas.
3. **Impacto según tamaño del dataset**:
   - **1K registros**: La diferencia de tiempo entre usar o no usar índices es marginal (segundos). Sin embargo, los planes de ejecución muestran menor costo cuando los índices están presentes.

- **10K registros**: Aquí se empieza a notar un impacto más relevante. Consultas que sin índices tardaban entre 12 y 20 segundos, con índices se resuelven en 4 a 8 segundos.
- **100K registros**: El uso de índices se vuelve indispensable. Consultas que sin optimización tardan más de 1 minuto, con índices se resuelven en menos de 20 segundos.
- **1 millón de registros**: Sin índices, algunas consultas no se completan o exceden el límite de ejecución estándar. Con índices bien diseñados, los tiempos se mantienen entre 50 segundos y 1.5 minutos, mostrando una mejora drástica en escalabilidad.

4. **Reducción del costo de ejecución**: Los planes de ejecución con índices muestran una disminución en los costos estimados por PostgreSQL, especialmente en consultas con `GROUP BY`, `JOINs` múltiples, y cálculos agregados como `AVG`, `COUNT` y divisiones anidadas.
5. **Escalabilidad del sistema**: El diseño de índices permite mantener el sistema escalable y estable, asegurando que la respuesta a las consultas no se degrade de manera exponencial con el crecimiento de los datos. Este principio es clave en contextos reales donde se manejan millones de registros.
6. **Selección estratégica de columnas indexadas**: No se recomienda indexar todas las columnas. Se priorizaron las que aparecen en cláusulas `WHERE`, `JOIN` y `GROUP BY`. Se evitó crear índices en columnas con baja cardinalidad o que solo contienen texto auxiliar, para no sobrecargar al sistema con mantenimiento innecesario.

En conclusión, el uso de índices no solo mejora el rendimiento en bases de datos grandes, sino que es esencial para garantizar tiempos de respuesta aceptables en sistemas reales con operaciones complejas. La correcta elección de columnas y el tipo de índice implementado marcan la diferencia entre un sistema eficiente y uno inoperable a gran escala.

6. Conclusiones

1. La incorporación de índices permitió reducir drásticamente los tiempos de ejecución de las consultas en escenarios con grandes volúmenes de datos. Mientras que con **1K registros** las diferencias eran mínimas, a partir de **10K**, y especialmente con **100K** y **1 millón de registros**, los tiempos sin índices se incrementaron considerablemente, superando incluso el minuto en algunos casos. En contraste, con índices adecuados, las mismas consultas se ejecutaron en pocos segundos.
2. Se implementaron índices de tipo **B-tree** para columnas utilizadas en filtros por rangos, ordenamientos y agrupamientos (como `scheduled_departure` y `range`), y de tipo **hash** para columnas con búsquedas por igualdad exacta (como `flight_id`, `ticket_no`, `aircraft_code`). Esta combinación se adaptó al

patrón de uso de cada tabla y permitió aprovechar las ventajas específicas de cada tipo de índice.

3. En tablas como `boarding_passes` y `ticket_flights`, se utilizaron **índices compuestos** sobre claves primarias dobles (`ticket_no, flight_id`) debido a su uso frecuente en condiciones de `JOIN`. Esto permitió acelerar considerablemente las consultas que combinan múltiples tablas.
4. Sin la presencia de índices, PostgreSQL recurrió a `Seq Scan` (búsqueda secuencial), incluso en filtros bien definidos. Con los índices, los planes de ejecución cambiaron a `Index Scan` o `Bitmap Index Scan`, reduciendo significativamente el consumo de CPU, I/O y la cantidad de tuplas examinadas.
5. Las consultas más complejas, como las que combinan `JOINs`, `GROUP BY`, filtros y funciones de agregación, mostraron el mayor beneficio. En estas, los índices permitieron mantener los tiempos de respuesta en torno a los **5 a 10 segundos con 1K datos**, pero lo más importante es que la respuesta se mantuvo estable incluso al escalar el volumen a **100K o más**, cumpliendo con requisitos de eficiencia y escalabilidad.
6. Finalmente, este análisis permite concluir que el diseño de índices no debe realizarse de manera arbitraria, sino que debe ser guiado por el análisis de **planes de ejecución**, **volumen de datos esperado** y **patrones de consulta**. Una estrategia de indexación adecuada no solo mejora el rendimiento, sino que permite que el sistema escale sin comprometer la experiencia del usuario.