REPASO DE BASE DE DATOS

1) CONSULTAS JOIN

1.1) INNER JOIN

```
SELECT reservacion.id_reservacion,
       cliente.nombre,
       cliente.apellido,
       fecha_entrada,
       estado
FROM reservacion
INNER JOIN cliente ON reservacion.id_cliente = cliente.id_cliente;
```

Propósito:

- Obtenemos informacion completa de las reservaciones y tambien los datos de los clientes.
- Nos facilita las gestion de todas las reservaciones que estan activas.
- Nos permite la verificacion de datos de las reservas y tambien de los clientes.

Explicación del Código:

- **SELECT reservacion.id_reservacion**:
  - Seleccionamos el ID de la reservación.
- **cliente.nombre, cliente.apellido**:
  - Obtenemos los datos personales del cliente y asi tambien idetificamos al hospedado o huesped.
- **fecha_entrada, estado**:
  - Estos serian los campos clave de la reservacion.
- **FROM reservacion**:
  - Seria la tabla principal de la consulta.
- **INNER JOIN cliente**:
  - Seria la union que nos aseguraria que hay coincidencias entre las tablas.
- **ON reservacion.id_cliente = cliente.id_cliente**:
  - Aqui habria una condicion de unión que relaciona las reservaciones con los clientes.

Cómo Funciona:

1. La consulta comienza en la tabla `reservacion`, que busca coincidencias en la tabla `cliente`,ahora si encuentra coincidencias, va a incluir datos en el resultado, ahora si no hay coincidencias, se pasa o se omiten los registros.

1.2) LEFT JOIN

```
SELECT hotel.nombre,
       tipo_habitacion.nombre
FROM hotel
LEFT JOIN habitacion ON hotel.id_hotel = habitacion.id_hotel
LEFT JOIN tipo_habitacion ON habitacion.id_tipo = tipo_habitacion.id_tipo;
```

Propósito:

- Mostramos todos los hoteles con los tipos de habitaciones que tienen.
- Vemos información, tambien de hoteles que no tienen habitaciones.
- Generamos una lista completa de habitaciones por hotel.
- Podemos identificar hoteles que necesitan registro de habitaciones.

Explicación del Código:

- **SELECT hotel.nombre**:
  - Obtenemos el nombre de cada hotel.
- **tipo_habitacion.nombre**:
  - Aqui seleccionamos el nombre del tipo de habitación.
- **FROM hotel**:
  - Esta es la tabla principal de la consulta.
- **LEFT JOIN habitacion**:
  - Conectamos hoteles con habitaciones.
- **ON hotel.id_hotel = habitacion.id_hotel**:
  - Esto es una condición de unión entre hoteles y habitaciones.
- **LEFT JOIN tipo_habitacion**:
  - Conectamos la habitacion con tipos de habitación.
- **ON habitacion.id_tipo = tipo_habitacion.id_tipo**:
  - Aqui tambien hay una condición de unión entre habitaciones y sus tipos.

Cómo Funciona:

1. Parte de todos los hoteles en la tabla `hotel`. Luego conecta las habitaciones que se han asociado a cada hotel, tambien conecta tipos de habitaciones, tambien mantiene hoteles sin habitaciones, y nos va a mostrar`NULL`.

1.3) RIGHT JOIN

```
SELECT cadena_hotelera.nombre,
       hotel.nombre
FROM hotel
RIGHT JOIN cadena_hotelera ON hotel.id_cadena = cadena_hotelera.id_cadena;
```

Propósito:

- Mostrar todas las cadenas hoteleras y sus hoteles.
- Podemos ver cadenas incluso sin que los hoteles esten registrados.
- Nos permite ver una lista completa de cadenas hoteleras.

Explicación del Código:

- **SELECT cadena_hotelera.nombre**:
  - Seleccionamos el nombre de cada cadena hotelera.
- **hotel.nombre**:
  - Seleccionamos el nombre del hotel que esta asociado.
- **FROM hotel**:
  - Esta es la tabla inicial de nuestra consulta.
- **RIGHT JOIN cadena_hotelera**:
  - Aqui mantiene todas las cadenas hoteleras, tambien los que no tienen hoteles asociados.
- **ON hotel.id_cadena = cadena_hotelera.id_cadena**:
  - Aqui se relacionan las cadenas hoteleras con los hoteles.

Cómo Funciona:

1. Esto se inicia con todas las cadenas hoteleras, luego busca los hoteles que estan asociados a cada cadena hotelera, ahora si una cadena hotelera no tiene hoteles, nos va a mortrar NULL, luego ya en el resultado nos va a mostrar una lista completa de cadenas y hoteles.

2) CONSULTAS DE DISTINCT

2.1) Hoteles por Ciudad y Cadena

```
SELECT DISTINCT ciudad, cadena_hotelera.nombre AS cadena
FROM hotel
JOIN cadena_hotelera ON hotel.id_cadena = cadena_hotelera.id_cadena
ORDER BY ciudad;
```

Propósito:

- Aqui vamos a obtener una lista unica de ciudades junto con las cadenas hoteleras.
- Aqui evitamos duplicados para cada combinación de ciudad y cadena.
- Tambien se facilita el análisis de cadenas hoteleras.

Explicación del Código:

- **SELECT DISTINCT:** Nos aseguramos que las combinaciones ciudad-cadena sean únicas.
- **ciudad:** Este campo de la tabla `hotel`, representa las ubicaciones.
- **cadena_hotelera.nombre AS cadena:** Nos muestra el nombre de la cadena hotelera.

- **FROM hotel:** Esta es la tabla base de la consulta.
- **JOIN cadena_hotelera:** Relacionamos las tablas `hotel` y `cadena_hotelera`.
- **ON hotel.id_cadena = cadena_hotelera.id_cadena:** Esta seria una condición que conecta las dos tablas.
- **ORDER BY ciudad:** Ordenamos los resultados por nombre de ciudad.

Cómo Funciona:

1. Selecciona todas las combinaciones de ciudad y cadena, luego eliminamos duplicados con `DISTINCT`.
2. Luego relacionamos los hoteles con cadenas hoteleras.
3. Y por ultimo ordena los resultados alfabéticamente por ciudad.

2.2) Tipos de Habitación por Hotel

Código SQL:

```sql
SELECT DISTINCT hotel.nombre AS hotel,
       tipo_habitacion.nombre AS tipo,
       tipo_habitacion.capacidad
FROM hotel
JOIN habitacion ON hotel.id_hotel = habitacion.id_hotel
JOIN tipo_habitacion ON habitacion.id_tipo = tipo_habitacion.id_tipo
ORDER BY hotel.nombre;
```

Propósito:

- Listamos los tipos únicos de habitaciones disponibles en cada hotel.
- Tambien evitamos duplicados de registros redundantes oq ue ya existen.
- Facilitamos la gestión de la lista de habitaciones.

Explicación del Código:

- **SELECT DISTINCT:** Aqui nos garantiza combinaciones únicas de hotel, tipo y capacidad.
- **hotel.nombre AS hotel:** Cambiamos var. que nos muestra el nombre del hotel.
- **tipo_habitacion.nombre AS tipo:** Nos muestra las categorías de habitación con una var..
- **tipo_habitacion.capacidad:** Aqui incluye la capacidad como detalle importante.
- **FROM hotel:** Esta es la tabla base o principal de la consulta.
- **JOIN habitacion:** Aqui conecta `hotel` con `habitacion`.
- **JOIN tipo_habitacion:** Relacionamos las habitaciones con sus categorías.
- **ORDER BY hotel.nombre:** Ordenamos los resultados alfabéticamente por hotel.

Cómo Funciona:

1. Primero relacionamos las tablas `hotel`, `habitacion` y `tipo_habitacion`, luego eliminamos duplicados en la combinación de hotel, tipo y capacidad, tambien nos muestra los tipos únicos de habitaciones junto con su capacidad.
2. Y por ultimo ordena los resultados por el nombre del hotel.

3) CONSULTAS GROUP BY

3.1) Total de habitaciones por hotel

```
SELECT hotel.nombre,
       COUNT(habitacion.id_habitacion) as total_habitaciones
FROM hotel
JOIN habitacion ON hotel.id_hotel = habitacion.id_hotel
GROUP BY hotel.nombre;
```

Propósito:

- Contamos el número total de habitaciones por hotel.
- Facilitamos la gestión de listado.
- Nos permite ver la capacidad.

Explicación del código:

- `SELECT hotel.nombre`: Seleccionamos el nombre del hotel, campo de agrupación y su identificador principal.
- `COUNT(habitacion.id habitacion) as total habitaciones`: A qui es la función agregada `COUNT`, cuenta habitaciones por hotel.
- `FROM hotel`: Esta es la tabla principal, base de datos de hoteles, inicio de la consulta.
- `JOIN habitacion`: Une con la tabla `habitacion`, que seeriaa necesario para contar, ya tamien tiene relación de uno a muchos.
- `GROUP BY hotel.nombre`: Agrupamos los resultados por hotel, base para el conteo, tambien organiza la información y mas.

  Cómo funciona:

  – Primero selecciona todos los hoteles, luego cuenta habitaciones de cada hotel, tambien agrupa los resultados por hotel y por ultimo nos muestra el total por cada hotel.

3.2) Hoteles con más de 2 reservaciones

```
SELECT hotel.nombre,
       COUNT(reservacion.id_reservacion) as total_reservaciones
FROM hotel
JOIN habitacion ON hotel.id_hotel = habitacion.id_hotel
JOIN reservacion ON habitacion.id_habitacion = reservacion.id_habitacion
GROUP BY hotel.nombre
HAVING COUNT(reservacion.id_reservacion) > 2;
```

Propósito:

– Identificamos hoteles con alta demanda.

– Filtramos por el número de reservaciones.

Explicación del código:

– `SELECT hotel.nombre`: Seleccionamos nombre del hotel, campo principal y base de agrupación.

– `COUNT(reservacion.id`$_r eservacion)astotal_r eservaciones$: $Cuentalasreservaciones, asignaunavar$..`FROM hotel` $Estaeslatablainicial, contienehoteles, puntodepartida.$

- `JOIN habitacion`: La primera unión, conecta con habitaciones, esto es necesario para las reservaciones.
- `JOIN reservacion`: La segunda unión, accede a reservaciones, nos permite el conteo.
- `GROUP BY hotel.nombre`: Lo agrupa por hotel, base para el conteo.

– `HAVING COUNT(reservacion.id`$_r eservacion) > 2 : Filtramoslosgrupos, condiciónpost-agrupación, ysolopuedehaberr$

Cómo funciona:

- Unimos las tablas necesarias, luego agrupamos por hotel contamos las reservaciones, filtramos con HAVING y por ultimos mostramos los hoteles que cumplen esta condicion.

4) PROCEDIMIENTOS ALMACENADOS

4.1) Nueva Reservación

```
DELIMITER //
CREATE PROCEDURE nueva_reservacion(
    IN cliente_id INT,
    IN habitacion_id INT,
    IN fecha_in DATE,
    IN fecha_out DATE,
    IN num_huespedes INT,
    IN metodo_pago VARCHAR(50)
)
BEGIN
    INSERT INTO reservacion(id_cliente, id_habitacion, fecha_entrada,
        fecha_salida,
    estado, numero_huespedes, metodo_pago, fecha_reservacion)
    VALUES (cliente_id, habitacion_id, fecha_in, fecha_out,
    'confirmada', num_huespedes, metodo_pago, CURDATE());

    UPDATE habitacion SET estado_actual = 'ocupada'
    WHERE id_habitacion = habitacion_id;
END //
DELIMITER ;
```

Propósito:

- Creamos nuevas reservaciones en el sistema.
- Actualizamos automáticamente el estado de la habitación.
- Mantenemos la privacidad de los datos.
- Tambien simplificamos el proceso de reserva.

Explicación línea por línea:

- `CREATE PROCEDURE nueva_reservacion`: Iniciamos la creación del procedimiento almacenado, que su nombre será `nueva_reservacion`.
- `IN cliente_id INT`: Tenemos un parametro de entrada `cliente_id` de tipo `INT`, que nos indica el ID del cliente que realiza la reserva.
- `INSERT INTO reservacion(...)`: Insertamos una nueva fila en la tabla `reservacion`.
- `UPDATE habitacion SET estado_actual = 'ocupada'`: Se actualiza el estado de la habitación a `habitacion_id` a `ocupada`.

Cómo funciona: Recibe los datos de la reservación, inserta la nueva reservación en la base de datos y actualiza el estado de la habitación a `ocupada`.

4.2) Actualizar Estado Habitación

```
DELIMITER //
CREATE PROCEDURE actualizar_estado_habitacion(
   IN habitacion_id INT,
   IN nuevo_estado ENUM('disponible','ocupada','mantenimiento')
)
BEGIN
   UPDATE habitacion
   SET estado_actual = nuevo_estado
   WHERE id_habitacion = habitacion_id;
END //
DELIMITER ;
```

Propósito:

- Actualizamos el estado de cualquier habitación.
- Se gestiona la disponibilidad de habitaciones.
- Mantenemos el control de mantenimiento de habitaciones.

Explicación línea por línea:

- `CREATE PROCEDURE actualizar_estado_habitacion`: Creamos el procedimiento almacenado para actualizar el estado de una habitación, aqui lo llamamos `actualizar_estado_habitacion`.
- `IN habitacion_id INT`: Este es un parámetro de entrada `habitacion_id` que nos indica la habitación a que tenemos que actualizar.
- `UPDATE habitacion SET estado_actual = nuevo_estado`: Realiza la actualización del estado de la habitación con un valor que haya recibido en el parámetro `nuevo_estado`.

Cómo funciona: Recibe el `id_habitacion` y el `nuevo_estado`, luego actualiza el estado de la habitación en la base de datos.

4.3) Cancelar Reservación

```
DELIMITER //
CREATE PROCEDURE cancelar_reservacion(
   IN reservacion_id INT
)
BEGIN
   UPDATE reservacion
   SET estado = 'cancelada'
   WHERE id_reservacion = reservacion_id;

   UPDATE habitacion
   SET estado_actual = 'disponible'
   WHERE id_habitacion = (SELECT id_habitacion
   FROM reservacion WHERE id_reservacion = reservacion_id);
END //
DELIMITER ;
```

Propósito:

- Cancelamos una reservación existente.
- Tambien liberamos la habitación automáticamente.
- Mantenemos un registro de las cancelaciones.

- Y se asegura la privacidad de los datos.

Explicación línea por línea:

- `DELIMITER //`: Segun vi aqui se cambia el delimitador a `//` para permitirnos la definición completa del procedimiento que hagamos.
- `CREATE PROCEDURE cancelar_reservacion`: Creamos el procedimiento almacenado que se llamara `cancelar_reservacion`.
- `IN reservacion_id INT`: El parámetro de entrada `reservacion_id` de tipo `INT`, que nos identifica la reservación que queremos cancelarr.
- `UPDATE reservacion SET estado = 'cancelada'`: Aqui actualizamos el estado de la reservación a `cancelada`.
- `UPDATE habitacion SET estado_actual = 'disponible'`: Actualiza el estado de la habitación a `disponible` luego de que cancelemos la reservación.
- `WHERE id_habitacion = (SELECT id_habitacion FROM reservacion...)`: Utilizamos una subconsulta para obtener el `id_habitacion` de la reservación que hemos cancelado y liberamos (o se vuelve disponible) la habitación.

Cómo funciona: Recibe el `id_reservacion`, cambia su estado a `cancelada`, y luego se actualiza el estado de la habitación a `disponible`.

5) FUNCIONES PERSONALIZADAS

5.1) Calcular Precio Total

```
DELIMITER //
CREATE FUNCTION calcular_precio_total(dias INT, precio_base DECIMAL(10,2))
RETURNS DECIMAL(10,2)
DETERMINISTIC
BEGIN
    RETURN dias * precio_base;
END //
DELIMITER ;
```

Propósito:

- Calculamos el precio total de una reservación.
- Tambien automatizamos cálculos de costos.
- Aseguramos la consistencia en precios.
- Tambien facilitamos las operaciones financieras.

Explicación línea por línea:

- `CREATE FUNCTION calcular_precio_total`: Declaramos la función personalizada con un nombre descriptivo.
- `(dias INT, precio_base DECIMAL(10,2))`: Definimos dos parámetros de entrada: `dias` como un número entero y `precio_base` como un decimal con dos decimales.
- `RETURNS DECIMAL(10,2)`: Especificamos el tipo de retorno de la función como `DECIMAL` con precisión de dos decimales.
- `DETERMINISTIC`: Nos indica que la función es determinista, es decir, siempre devuelve el mismo resultado para los mismos parámetros.

- `BEGIN ... END`: Tambien contiene la lógica de la función, que multiplica `dias` por `precio_base` y nos retorna el resultado.

Cómo funciona: La función recibe los parámetros `dias` y `precio_base`, realiza la multiplicación y retorna el resultado con dos decimales.

5.2) Está Disponible

```
DELIMITER //
CREATE FUNCTION esta_disponible(habitacion_id INT)
RETURNS BOOLEAN
DETERMINISTIC
BEGIN
    DECLARE estado VARCHAR(20);
    SELECT estado_actual INTO estado
    FROM habitacion
    WHERE id_habitacion = habitacion_id;
    RETURN estado = 'disponible';
END //
DELIMITER ;
```

Propósito:

- Verificamos la disponibilidad de una habitación.
- Nos facilita las consultas del estado.
- Prevenimos las reservas duplicadas.
- Controlamos la ocupación de habitaciones si estan disponible o tal.

Explicación línea por línea:

- `CREATE FUNCTION esta_disponible`: Declaramos la función con un nombre que indica su propósito.
- `(habitacion_id INT)`: Definimos un parámetro de entrada para la ID de la habitación.
- `RETURNS BOOLEAN`: Definimos que la función retornará un valor booleano (verdadero o falso).
- `DETERMINISTIC`: Bueno aqui la funcion es determinista.
- `BEGIN ... END`: Contiene la lógica que consulta el estado de la habitación y retorna `TRUE` si está disponible, sino ps `FALSE`.

Cómo funciona: Recibe el ID de la habitación, consulta su estado y retorna `TRUE` ahora si está disponible, sino ps `FALSE`.

5.3) Contar Reservaciones Activas

```
DELIMITER //
CREATE FUNCTION contar_reservaciones_activas(cliente_id INT)
RETURNS INT
DETERMINISTIC
BEGIN
    DECLARE total INT;
    SELECT COUNT(*) INTO total
    FROM reservacion
    WHERE id_cliente = cliente_id
    AND estado = 'confirmada';
    RETURN total;
END //
DELIMITER ;
```

Propósito:

- Contamos las reservaciones activas por cliente.
- Tambien controlamos las reservaciones simultáneas.
- Se realiza el seguimiento de clientes.
- Gestionamos la ocupación de habitaciones.

Explicación línea por línea:

- `CREATE FUNCTION contar_reservaciones_activas`: Declaramos la función que contará las reservaciones activas de un cliente.
- `(cliente_id INT)`: Definimos un parámetro de entrada para la ID del cliente.
- `RETURNS INT`: Luego la función nos va a retornar un valor entero, que es el número de reservaciones.
- `DETERMINISTIC`: Bueno aqui la fun. es determinista.
- `BEGIN ... END`: Contiene la lógica que cuenta las reservaciones activas del cliente con el estado "confirmada" y nos retorna el total.

Cómo funciona: Recibe el ID del cliente, cuenta las reservaciones con el estado "confirmada" y retorna el número total de reservaciones activas.

6) TRIGGERS

El DELIMITER cambia el delimitador estándar para que soporte los bloques de código.

6.1) Trigger para Actualizar Estado de Habitación Después de una Reserva

```
DELIMITER //
CREATE TRIGGER actualizar_estado_despues_reserva
AFTER INSERT ON reservacion
FOR EACH ROW
BEGIN
    UPDATE habitacion
    SET estado_actual = 'ocupada'
    WHERE id_habitacion = NEW.id_habitacion;
END //
DELIMITER ;
```

Propósito:

- Actualizar automáticamente el estado de la habitación a `ocupada` después de nosotros haber realizado una reservación.
- Mantenemos la permanencia entre las reservaciones y los estados de las habitaciones.
- Evitamos las reservas duplicadas.
- Aseguramos la seguridad o privacidad de los datos.

Explicación línea por línea:

- **CREATE TRIGGER actualizar_estado_despues_reserva**: Definimos el trigger con un nombre que nos es mejor para la funcion.
- **AFTER INSERT ON reservacion**: Especificamos que es lo que se ejecutará después de una inserción en la tabla `reservacion`.
- **FOR EACH ROW**: Aplicamos la acción a cada fila afectada.
- **BEGIN**: Tambien marcamos el inicio del bloque de instrucciones.
- **UPDATE habitacion SET estado_actual = 'ocupada'**: Actualizamos el estado de la habitación asociada a la reservación.
- **WHERE id_habitacion = NEW.id_habitacion**: Identificamos la habitación específica basada en la nueva reservación.
- **END //**: Finalizamos el bloque de código del trigger.

Cómo funciona: Esto see activa automáticamente luego de hacer una nueva reservación, luego se indentifica la habitación asociada a la reservación tambien se actualiza su estado a `ocupada` y por utlimo se asegura que la habitacion no pueda ser reservada nuevamente hasta que esté disponible.

6.2) Trigger para Verificar Capacidad Antes de Insertar Reservación

```
DELIMITER //
CREATE TRIGGER verificar_capacidad_antes_reserva
BEFORE INSERT ON reservacion
FOR EACH ROW
BEGIN
    DECLARE max_capacidad INT;
    SELECT capacidad INTO max_capacidad
    FROM habitacion h
    JOIN tipo_habitacion t ON h.id_tipo = t.id_tipo
    WHERE h.id_habitacion = NEW.id_habitacion;

    IF NEW.numero_huespedes > max_capacidad THEN
        SIGNAL SQLSTATE '45000'
        SET MESSAGE_TEXT = 'N mero de hu spedes excede la capacidad de la
            habitaci n';
    END IF;
END //
DELIMITER ;
```

Propósito:

- Validamos que el número de huéspedes no exceda la capacidad de la habitación antes de insertar una nueva reservación.
- Prevenimos los errores y conflictos en las reservaciones.
- Nos aseguramos que las habitaciones cumplan con las restricciones de capacidad.

Explicación línea por línea:

- **CREATE TRIGGER verificar_capacidad_antes_reserva**: Definimos un trigger que va a validar la capacidad antes de insertar.

- **BEFORE INSERT ON reservacion**: Esto se va a ejecutar antes de realizar una inserción en la tabla `reservacion`.
- **DECLARE max_capacidad INT**: Declaramos una variable para almacenar la capacidad máxima.
- **SELECT capacidad INTO max_capacidad**: Obtenemos la capacidad máxima de la habitación especificada.
- **IF NEW.numero_huespedes ¿ max_capacidad THEN**: Vamos a verificar si el número de huéspedes excede la capacidad.
- **SIGNAL SQLSTATE '45000'**: Generamos un error personalizado si se excede la capacidad.
- **END IF;**: Se finaliza la condición.
- **END //**: Y terminamos el bloque del trigger.

Cómo funciona: Primero esto, antes de insertar una reservación, el trigger se va a activar, luego vamos a obtener la capacidad máxima de la habitación asociada, comparamos el número de huéspedes con la capacidad máxima, ahora si se excede la capacidad, generara un error y se detendra la operación.

6.3) Trigger para Liberar Habitación Después de Cancelar una Reservación

```
DELIMITER //
CREATE TRIGGER liberar_habitacion_despues_cancelar
AFTER UPDATE ON reservacion
FOR EACH ROW
BEGIN
    IF NEW.estado = 'cancelada' AND OLD.estado = 'confirmada' THEN
        UPDATE habitacion
        SET estado_actual = 'disponible'
        WHERE id_habitacion = NEW.id_habitacion;
    END IF;
END //
DELIMITER ;
```

Propósito:

- Cambiamos automáticamente el estado de una habitación a `disponible` cuando una reservación confirmada sea cancelada.
- Mantenemos la disponibilidad de las habitaciones actualizada.
- Garantizamos la consistencia entre las reservaciones y la disponibilidad.

Explicación línea por línea:

- **CREATE TRIGGER liberar_habitacion_despues_cancelar**: Definimos un trigger para liberar habitaciones tras cancelar reservaciones.
- **AFTER UPDATE ON reservacion**: Aqui se ejecuta después de actualizar una fila en la tabla `reservacion`.
- **IF NEW.estado = 'cancelada' AND OLD.estado = 'confirmada' THEN**: Verificamos que el estado cambió de `confirmada` a `cancelada`.
- **UPDATE habitacion SET estado_actual = 'disponible'**: Actualizamos el estado de la habitación asociada.
- **WHERE id_habitacion = NEW.id_habitacion**: Identificamos la habitación a actualizar.
- **END IF;**: Se finaliza la condición.
- **END //**: Y aqui se cierra el bloque del trigger.

Cómo funciona: Primero el trigger se va a activar tras nosotros actualizar una reservación, luego se va a evaluar si la reservación fue cancelada después de estar confirmada, si ese es el caso la condición se cumple, cambia el estado de la habitación a `disponible` y por ultimo nos permite que la habitación sea reservada nuevamente.