



Transacciones y API REST

Transacciones

Implementar operaciones transaccionales en una base de datos para mantener la consistencia de los datos utilizando el entorno Node.js.

Implementar la capa de acceso a datos utilizando un ORM con entidades no relacionadas para realizar operaciones CRUD.

Utilizar asociaciones uno a uno, uno a muchos y muchos a muchos en la definición de las relaciones entre las entidades de un modelo que resuelven un problema.

{desafío}
latam_

- Unidad 1:
Implementación y gestión de una base de datos
- Unidad 2:
Transacciones y API REST
- Unidad 3:
Trabajo práctico



Te encuentras
aquí



¿Qué aprenderás en esta sesión?

- *Describe conceptos básicos de transaccionalidad y sus beneficios para la implementación de un aplicativo que interactúa con una base de datos.*
- *Utiliza sentencias para el inicio, confirmación y vuelta atrás de una transacción, dando solución a un problema.*
- *Utiliza sentencias para la captura y manejo de errores de operación SQL utilizando el entorno Node.js.*

Comenta con tus
palabras:

¿Qué es una
transacción?



/* Transacciones */

Transacciones

Una transacción en base de datos corresponde a una o más operaciones SQL que se deben ejecutar de manera íntegra, es decir, en un 100%, como si se tratase de una sola operación. En el caso de que alguna sentencia falle, todos los cambios aplicados hasta ese punto son revertidos.

En la imagen, podemos ver cómo se realiza el flujo de operaciones o algorítmica aplicada en una transacción de base de datos, ya sea si se realizó exitosamente o



/* Importancia de las transacciones en SQL */

Importancia de las transacciones en SQL

Las transacciones en SQL, resuelven un problema de alta importancia que puede suceder en el caso de necesitar realizar varias consultas cuyos impactos tienen alguna relación, el ejemplo más típico y académico es el de una transacción bancaria, donde el saldo de una persona disminuye una cantidad determinada y otra persona recibe este monto.



Importancia de las transacciones en SQL

La pregunta que resuelve el concepto de transacción es ¿Dónde puede ocurrir el problema? Y la respuesta es que las consultas SQL podrían experimentar algún problema en su ejecución, por ejemplo:

- Error de comunicación con la base de datos.
- Error de sintaxis en la instrucción SQL.
- Error por una restricción definida en la creación de un campo.
- Error por no encontrar el registro específico.
- Entre otros.



**/* Preparando un escenario para las
transacciones */**

Preparando un escenario para las transacciones

1. Para iniciar con la programación de esta aplicación, deberás tener una base de datos ya creada o crear una nueva. Puedes usar la siguiente instrucción para crear una base de datos llamada **dinero_azul**.

```
CREATE DATABASE dinero_azul;
```

Preparando un escenario para las transacciones

2. Ahora que tienes la base de datos creada, deberás crear una tabla **usuarios** con los campos "first_name", "last_name", "email" y "saldo". ¿No recuerdas exactamente cómo se escribe la instrucción SQL para crear la tabla? No te preocupes, a continuación te mostramos el código para hacerlo:

```
CREATE TABLE usuarios (first_name VARCHAR(100), last_name VARCHAR(100),  
email VARCHAR(100), saldo INT CHECK (saldo >= 0) );
```

Preparando un escenario para las transacciones

Una vez creada la tabla, necesitarás importar el csv con los datos de los clientes y para eso puedes ejecutar la siguiente instrucción en la terminal de PostgreSQL.

```
\copy usuarios from '<dirección del archivo csv>' DELIMITER ',' CSV  
HEADER;
```

Recuerda cambiar la dirección del archivo csv según el árbol de archivos de tu sistema operativo.

¡Ahora sí!, estamos listos para ir manos al código y empezar a programar nuestras transacciones SQL desde una aplicación en Node.



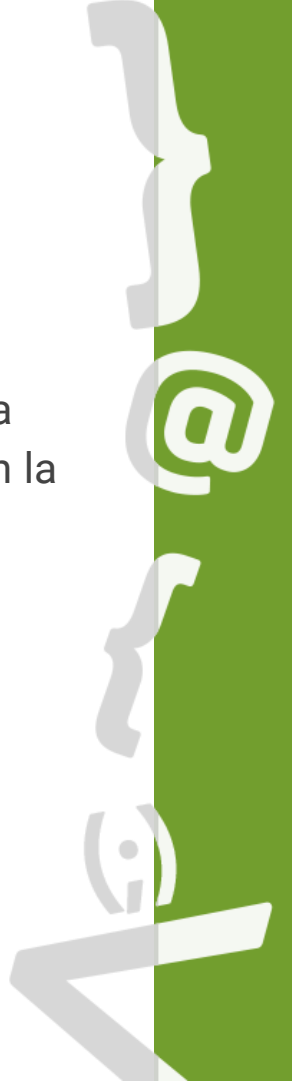
Ejercicio guiado: Usando transacciones SQL en Node, violando restricciones



Ejercicio guiado

Usando transacciones SQL en Node, violando restricciones

Desarrollar una aplicación en Node, que al ser ejecutada realice consultas SQL, usando las transacciones para transferir \$25.000 desde el cliente registrado en la tabla **usuarios** de correo **yuki_whobrey@aol.com**, al cliente también registrado en la tabla con el correo **fletcher.flosi@yahoo.com**, ambos son clientes del banco llamado “Dinero Azul”. Debemos esperar que esta transacción no se termine, puesto que el primer usuario no tiene esa cantidad en su saldo.

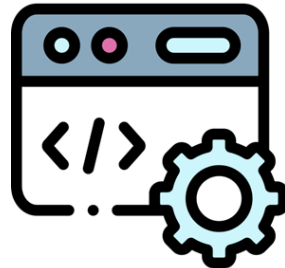


Ejercicio guiado

Usando transacciones SQL en Node, violando restricciones

Prosigue con los siguientes pasos para el desarrollo de este ejercicio:

- **Paso 1:** Realizar una consulta SQL con el comando BEGIN para iniciar la transacción.
- **Paso 2:** Realizar una consulta SQL para actualizar el saldo del usuario de correo yuki_whobrey@aol.com descontando \$25.000.
- **Paso 3:** Realizar una consulta SQL para actualizar el saldo del usuario de correo fletcher.flosi@yahoo.com acreditando \$25.000.



Ejercicio guiado

Usando transacciones SQL en Node, violando restricciones

- **Paso 4:** Realizar una consulta SQL con el comando COMMIT para terminar la transacción.

```
const { Pool } = require("pg");

const pool = new Pool({
  user: "postgres",
  host: "localhost",
  password: "postgres",
  database: "dinero_azul",
  port: 5432,
```



Ejercicio guiado

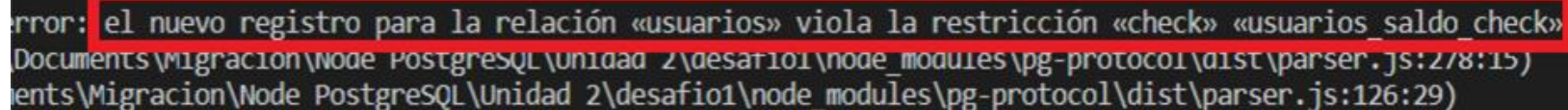
Usando transacciones SQL en Node, violando restricciones

```
});  
(async () => {  
  // Paso 1  
  await pool.query("BEGIN");  
  
  // Paso 2  
  const descontar =  
    "UPDATE usuarios SET saldo = saldo - 25000 WHERE email = 'yuki_whobrey@aol.com' ";  
  await pool.query(descontar);  
  
  // Paso 3  
  const acreditar =  
    "UPDATE usuarios SET saldo = saldo + 25000 WHERE email = 'fletcher.flosi@yahoo.com'";  
  await pool.query(acreditar);  
  
  // Paso 4  
  await pool.query("COMMIT");  
})();
```

Ejercicio guiado

Usando transacciones SQL en Node, violando restricciones

Ahora ejecuta la aplicación y deberás recibir algo como lo que te mostramos en la siguiente imagen:



```
error: el nuevo registro para la relación «usuarios» viola la restricción «check» «usuarios saldo check»  
Documents\Migracion\Node PostgreSQL\Unidad 2\desafio1\node_modules\pg-protocol\dist\parser.js:278:15)  
Documents\Migracion\Node PostgreSQL\Unidad 2\desafio1\node_modules\pg-protocol\dist\parser.js:126:29)
```

Como puedes notar hemos recibido un error indicando que el cambio que intentamos realizar viola la restricción del campo saldo.

Ejercicio guiado

Usando transacciones SQL en Node, violando restricciones

¿Pero esto realmente funcionó? Utiliza tu consola de PostgreSQL y ejecuta la siguiente instrucción para revisar el saldo actual del usuario al que intentamos descontar saldo.

```
select * from usuarios where email = 'yuki_whobrey@aol.com';
```

Y deberás recibir lo que te mostramos en la siguiente imagen:

```
clientes=# select * from usuarios where email = 'yuki_whobrey@aol.com';
               clientes
first_name | last_name | email | saldo
-----+-----+-----+-----
Yuki      | Whobrey   | yuki_whobrey@aol.com | 20000
(1 fila)
```



Ejercicio guiado

Usando transacciones SQL en Node, violando restricciones

La restricción ha funcionado sin problema y no tenemos un saldo negativo en este usuario, pero ¿Qué pasó con el saldo del usuario acreditado en la transacción? Pues debería ser también el mismo previo a la transacción, pero si quieres comprobarlo ocupa la siguiente instrucción:

```
select * from usuarios where email = 'fletcher.flosi@yahoo.com';
```

Y deberás recibir lo que te mostramos en la siguiente imagen:

```
clientes=# select * from usuarios where email = 'fletcher.flosi@yahoo.com';
          clientes
first_name | last_name | email | saldo
-----+-----+-----+-----
Fletcher  | Flosi    | fletcher.flosi@yahoo.com | 20000
(1 fila)
```

{desafío}
latam_

Ahí lo tenemos, la transacción devolvió exactamente lo que esperábamos y hemos comprobado que ambos usuarios mantuvieron sus saldos.

Ejercicio guiado: Usando transacciones SQL en Node, caso de éxito



Ejercicio guiado

Usando transacciones SQL en Node, caso de éxito

Al comprobar que las transacciones con resultados fallidos no afectarán la consistencia de nuestra base de datos.

Sigue los pasos para hacer unas modificaciones al código anterior y realizar un caso de éxito, donde podamos comprobar que si se intenta transferir un saldo que sí está disponible en la cuenta del emisor, esta llegará sin problema a la cuenta receptora:

- **Paso 1:** Modificar la consulta SQL para actualizar el saldo del usuario de correo yuki_whobrey@aol.com descontando en este caso \$20.000. Agrega el comando RETURNING * al final de la consulta y almacena el resultado en una constante.
- **Paso 2:** Modificar la consulta SQL para actualizar el saldo del usuario de correo fletcher.flosi@yahoo.com acreditando en este caso \$20.000. Agrega el comando RETURNING * al final de la consulta y almacena el resultado en una constante.



Ejercicio guiado

Usando transacciones SQL en Node, caso de éxito

- **Paso 3:** Imprimir ambas constantes creadas que representarán los registros afectados en ambas actualizaciones. Esto lo hacemos para visualizar los saldos actuales de ambos usuarios.

```
(async () => {  
    await pool.query("BEGIN");  
  
    // Paso 1  
    const descontar =  
        "UPDATE usuarios SET saldo = saldo - 20000 WHERE email =  
        'yuki_whobrey@aol.com' RETURNING *";  
    const descuento = await pool.query(descontar);  
  
    // Paso 2  
    const acreditar =  
        "UPDATE usuarios SET saldo = saldo + 20000 WHERE email =  
        'fletcher.flosi@yahoo.com' RETURNING *";  
    const acreditacion = await pool.query(acreditar);  
  
    // Paso 3  
    console.log("Descuento realizado con éxito: ", descuento.rows[0]);  
    console.log("Acreditación realizada con éxito: ",  
    acreditacion.rows[0]);  
  
    await pool.query("COMMIT");  
})
```



Ejercicio guiado

Usando transacciones SQL en Node, caso de éxito

Ahora ejecuta la aplicación y deberás recibir algo como lo que te mostramos en la siguiente imagen:

```
Descuento realizado con éxito: {  
  first_name: 'Yuki',  
  last_name: 'Whobrey',  
  email: 'yuki_whobrey@aol.com',  
  saldo: '0'  
}  
Acreditación realizada con éxito: {  
  first_name: 'Fletcher',  
  last_name: 'Flosi',  
  email: 'fletcher.flosi@yahoo.com',  
  saldo: '40000'  
}
```

¡Excelente! El saldo del usuario Yuki se redujo a \$0 puesto que inicialmente se le asignó la misma cantidad recientemente transferida, y el usuario Fletcher ahora tiene \$40.000, comprobando que la transacción hizo su trabajo sin problemas.



/* Transacciones en las bases de datos */

Transacciones en las bases de datos

Primero, para iniciar una transacción en base de datos, siempre es necesario indicarle a la base de datos que deseamos realizar dicho proceso. Para esto, existe una instrucción que se conoce como Begin. Mediante esta instrucción, la base de datos tendrá en consideración que a continuación todo lo que viene se necesita ejecutar de manera íntegra.

Luego de marcar el inicio mediante la instrucción Begin, podemos iniciar el flujo de operaciones SQL, ya sea si es 1, 2 ó más sentencias SQL las que necesitemos ejecutar. En el caso de la imagen anterior, tenemos un conjunto de 3 operaciones que por un lado se resuelven de manera correcta y por otro no:

- En el caso de la transacción exitosa, vemos que cada una de las 3 operaciones SQL se ejecutan una tras otra sin problemas. En este caso, cuando se terminan de ejecutar todas las operaciones requeridas, se le debe indicar a la base de datos que la transacción debe finalizar. Para realizar esta tarea, se utiliza una instrucción llamada Commit.

Transacciones en las bases de datos

- Por el lado de la transacción fallida, podemos apreciar que la primera operación SQL se realiza correctamente, pero en cambio, la segunda no se pudo procesar. Ya que todo el conjunto de operaciones, en su 100% no se pudo completar de manera exitosa, se deben revertir los cambios realizados hasta el momento, en este caso, revertir la primera operación SQL. Para realizar esta tarea, se utiliza la instrucción llamada Rollback.

Con lo anterior dicho, revisemos un caso de uso donde se aplique este concepto. Para esto, revisaremos el proceso de agendamiento de horas médicas de un sistema que tiene los siguientes requisitos para reservar una hora, tomando las siguientes consideraciones:

- Cada registro de agendamiento, debe estar ligado a un paciente, un profesional y una especialidad médica.

Transacciones en las bases de datos

- Una vez que se realice un agendamiento, se le debe enviar un correo al paciente con los datos de su cita médica.

Para poder llevar a cabo estas tareas debe realizar los siguientes procesos asociados a la base de datos:

- Insertar en una tabla llamada agendamiento los siguientes datos:
 - Rut del paciente
 - Nombre del paciente
 - Especialidad médica
 - Nombre del médico
 - Fecha junto con la hora.

Transacciones en las bases de datos

- Apenas se realice la inserción en la tabla agendamiento, se debe insertar en una tabla **cola_correo** los siguientes datos:
 - Id del agendamiento
 - Correo del paciente
 - Fecha y hora en que se realizó esta inserción.

Estos datos luego serán procesados por otro sistema que se encargará de enviar el correo.

Entonces, para realizar todo el proceso de agendamiento de horas necesitaríamos realizar 2 inserciones en la base de datos. La segunda inserción, depende de la primera, ya que necesita el id de agendamiento, por lo que llevar a cabo una transacción en esto es de carácter obligatorio.

/* Por qué se necesita control sobre las transacciones */

Por qué se necesita control sobre las transacciones

El no disponer de operaciones transaccionales en un sistema de software supone una desventaja importante en muchos aspectos. Uno de estos recae en el hecho de deshacer cada una de las operaciones que se desean llevar a cabo, lo cual aumenta su complejidad en función de la cantidad de operaciones. En el ejemplo visto anteriormente donde se realizaba un agendamiento a través de 2 inserciones, pudimos ver que en un caso la transacción fallaba al tratar de realizar la segunda inserción, de no utilizar transacciones y desear revertir los cambios, el desarrollador tendría que generar una rutina adicional para eliminar los datos insertados y restablecer la secuencia **agendamiento_id_seq** generada mediante un campo serial. Sin embargo, nada de eso fue necesario, solo marcar el inicio de la transacción mediante **begin** y el fin de esta a través de **commit**; el resto de la lógica transaccional la maneja el motor de base de datos, lo cual se traduce finalmente en menores costos de desarrollo y asegurar una mayor consistencia en relación a los datos de la base de datos.

`/* Cuando utilizar transacciones */`

Cuándo utilizar transacciones

Con el enfoque de las transacciones es posible que se genere la duda de cuándo es correcto o deberían ser utilizadas este tipo de operaciones. Uno de los principios más básicos de cara a determinar el uso de estas, yace en sí necesitamos que un conjunto o lote de operaciones se ejecuten como una sola operación. Esto quiere decir, que si las operaciones a ejecutar deben ser ejecutadas todas de manera exitosa o fallar en su conjunto.

La certeza del uso de estas operaciones puede, en ocasiones, no ser clara del todo. Sin embargo, todo dependerá netamente de las variables del contexto o proyecto, como el diseño de la base de datos (el cómo está desarrollada) y las reglas/lógicas del negocio (requerimientos).

A final de cuentas, siempre se debe tener en mente lo siguiente: ¿Necesito ejecutar un conjunto o lote de operaciones en la base de datos como si se tratase de una sola operación? ¿Si no se ejecutan todas las operaciones debería fallar el proceso completo?. Si la respuesta es sí a ambas, entonces las transacciones son el camino a seguir.

Cuándo utilizar transacciones

Ahora, ¿En dónde podemos encontrar transacciones? Se encuentran presentes en innumerables sistemas de información de todo orden, desde la compra de productos, el agendamiento de horas médicas en una clínica, la creación de una cuenta en una red social, transferencias bancarias, etc.



/* La integridad de datos */

La integridad de datos

Una de las propiedades que todo motor de base de datos debería garantizar es asegurar la integridad con la que los datos son tratados. Esto se refiere, principalmente, a que cada dato sea **consistente, preciso y se encuentre completo** de acuerdo a una serie de reglas, estándares o métodos preestablecidos de acuerdo a un diseño de datos previo.

Dentro de estas reglas y en el marco lógico de la estructura de datos, podemos encontrar algunos elementos tales como las relaciones que existen entre una tabla y otra, si un dato puede tener un valor nulo o no, qué tipo de dato tiene asignado un campo de una tabla, entre otros elementos comunes y particulares de cada motor de base de datos.

La integridad de datos

Para tener una idea más concreta del concepto de integridad, analicemos el pequeño modelo de datos que creamos anteriormente, donde teníamos una tabla llamada `agendamiento` y otra tabla llamada `cola_correo`. Dentro de las reglas o mecanismos de validación que tenemos en dicho modelo de datos, podemos contar con que:

- Los datos se encuentran separados en distintas tablas o unidades lógicas.
- Los datos almacenados en la base de datos pueden ser consultados, modificados, creados y eliminados mediante instrucciones específicas.
- Los datos persisten en el tiempo hasta que son modificados o eliminados.
- Cada una de las tablas, tanto `agendamiento` como `cola_correo`, poseen un dato único llamado `id` que identifica un conjunto de datos y los diferencia de los demás. Además, cada uno de estos datos corresponde a un número entero o `integer` y se crea mediante un objeto en la base de datos llamado `secuencia`.

La integridad de datos

- La tabla de agendamiento se relaciona con la tabla cola_correo mediante su campo una clave foránea (id en la tabla agendamiento, agendamiento_id en la tabla cola_correo).
- Los datos que contienen caracteres, como una combinación de letras, números y símbolos, se definen mediante un tipo de dato varchar con un largo o precisión específico. Por ejemplo, los campos rut_paciente y nombre_paciente dentro de la tabla agendamiento pueden contener una cantidad de caracteres distintos.

Cada vez que un dato se ingresa dentro de la base de datos, mediante una instrucción insert, esta es tratada y almacenada de acuerdo a las condicionantes nombradas anteriormente y, eventualmente, a otras reglas que pueda disponer la instancia del motor de base de datos en cuestión.

La integridad de datos

Es así, que los datos, cuando son consultados, se obtienen de manera consistente (no se deforma en el proceso), precisa (aquello que se espera obtener) y completa (en su totalidad), tal como se puede ver en la ilustración anterior. Cuando estos 3 principios o conceptos nombrados anteriormente están garantizados, podemos hablar de que los datos se mantienen íntegros.

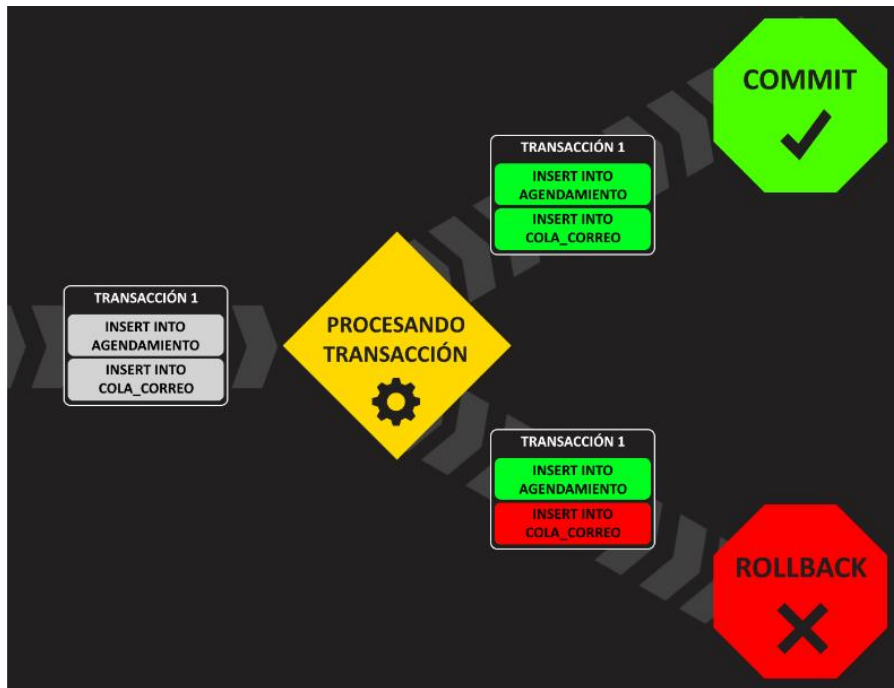


**/* La propiedad ACID en las
transacciones: Atomicidad,
Consistencia, Aislamiento, Durabilidad
*/**

La propiedad ACID en las transacciones

Atomicidad

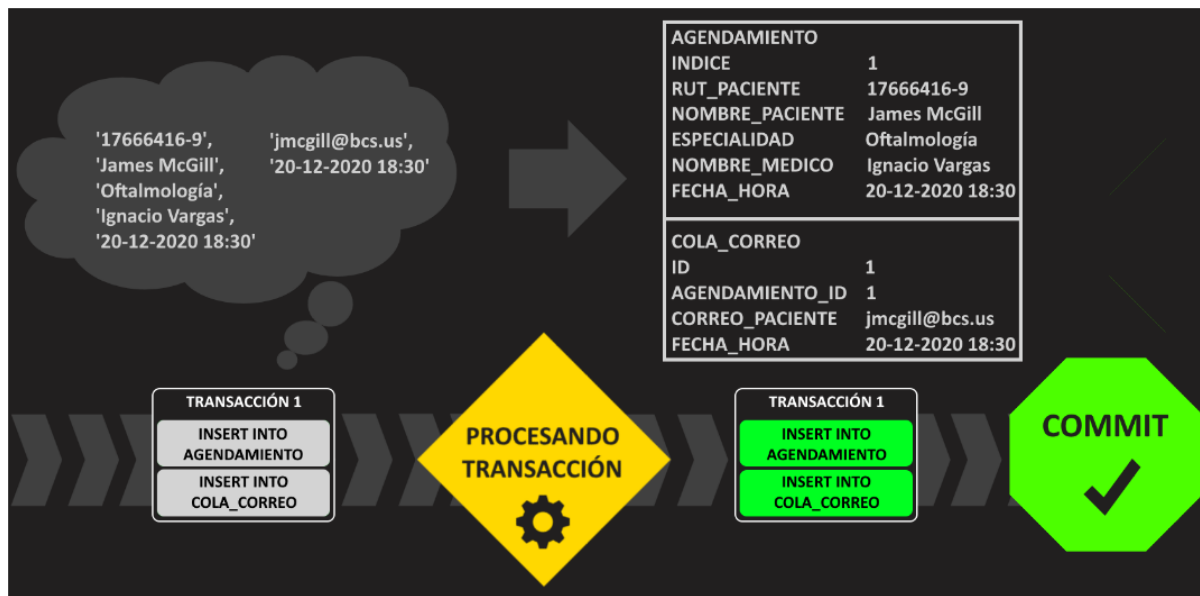
Del inglés Atomicity, se refiere al hecho de que cada lote de operaciones SQL dentro de una transacción debe ser ejecutada como si se tratase de 1 sola operación, limitando el resultado de la ejecución a un valor binario, es decir, se ejecutan correctamente todas las operaciones de la transacción de manera correcta (commit) y se aplican los cambios esperados o se deshacen todas las operaciones y los cambios son revertidos (rollback).



La propiedad ACID en las transacciones

Consistencia

Del inglés Consistency, se refiere a que los datos se mantengan en un estado consistente desde que inicia una transacción hasta que ésta termina, es decir, que sean exactos y coherentes durante el proceso transaccional y no sean deformados.



La propiedad ACID en las transacciones

Aislamiento

Del inglés Isolation, se refiere a que las transacciones no se ven afectadas entre sí, ya que cada una de estas actúa de manera independiente a otra, es decir, el estado intermedio de una transacción (la ejecución de la misma) es invisible para otra transacción.



La propiedad ACID en las transacciones

Durabilidad

Del inglés Durability, se refiere al hecho de que cuando una transacción es ejecutada completamente y con éxito, los cambios realizados por ésta se mantienen y perduran en el tiempo, independiente si se manifiestan fallas en el sistema.



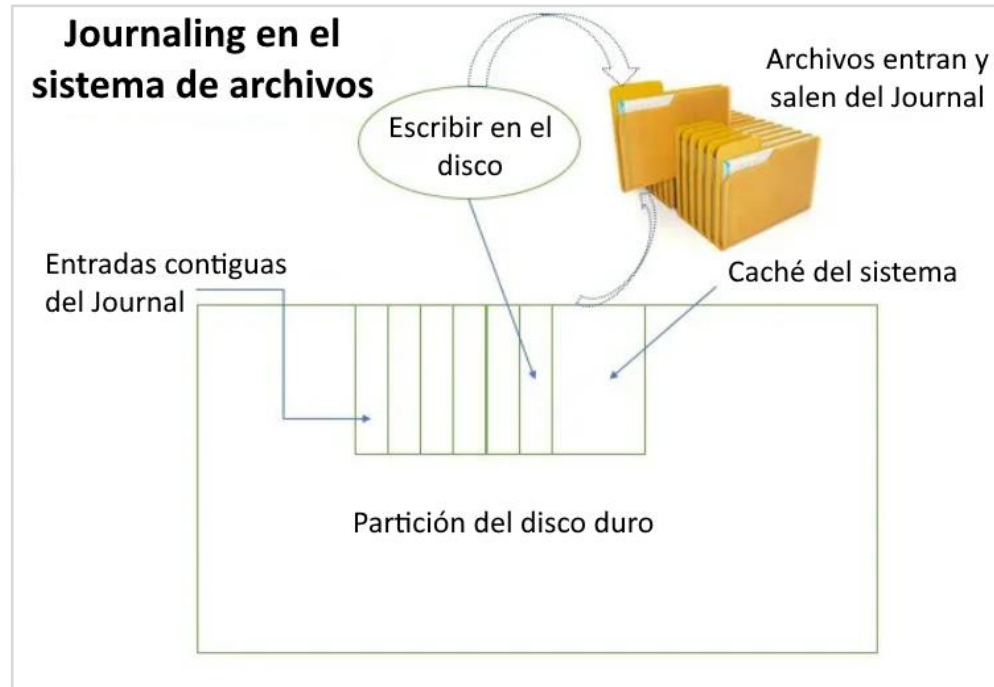
/* Journaling en bases de datos */

Journaling en bases de datos

Una de las técnicas para asegurar que los datos manejados por transacciones, ya sea aplicadas a un sistema de archivos o una base de datos, es el journaling. Esto consiste básicamente en un mecanismo que permite garantizar que las transacciones ejecutadas se manifiestan en resultados completos donde la integridad de los datos es primordial, por lo que se evita (en la medida de lo posible) que estos se vean corrompidos por fallos inesperados, como cortes de luz o errores del sistema.

Este concepto viene del término **journal** que significa diario o periódico en español y se debe a que cada transacción es registrada en una unidad que funciona como un catálogo o diario de registros. Cada uno de estos registros es clasificado y así además se permite un acceso, tanto de lectura como de escritura, mucho más rápido a los datos.

Journaling en bases de datos



/* Errores en transacciones SQL desde Node */

Errores en transacciones SQL desde Node

En temas anteriores, pudimos realizar transacciones SQL desde Node y comprobar su efectividad, no obstante, cuando la transacción falló, la lectura del código se detuvo y el error se manifestó por terminal, ¿Cuál es el problema con esto? El problema es que el error no está siendo impreso por consola de forma controlada, sino que nuestra aplicación se estrelló con el error (conocido en inglés como crashed) y esto puede traer varias consecuencias, siendo la más importante el hecho de que al ser detenida la lectura del código, no se alcanza a ejecutar la instrucción del método “end()”, por lo que no cierra la conexión con PostgreSQL, dejando un tiempo muerto de espera, es decir, que todo quedará detenido y solo nos queda acabar con el proceso cancelando en la terminal.



¿Puedes resumir qué
entendiste de la clase de
hoy?





Próxima sesión...

- *Reconoce las características del lenguaje Javascript ES6 para la definición de clases y objetos.*

{desafío}
latam_

*Academia de
talentos digitales*

