Practica 4 BD1

Esquema final

> La clave primaria está en negrita

PATIENT (**patient_id** , patient_name, patient_address, patient_city, primary_phone, secondary_phone)

DOCTOR (**doctor_id**, doctor_name, doctor_address, doctor_city, doctor_speciality)

APPOINTMENT (patient_id, appointment_date, appointment_duration, contact_phone, observations, payment_card)

MEDICAL_REVIEW (**patient_id, appointment_date, doctor_id**)

PRESCRIBED_MEDICATION (**patient_id, appointment_date, medication_name**)

Ejercicio 1

1. Crea un usuario para las bases de datos usando el nombre '*appointments_user'*. Asigne a estos todos los permisos sobre sus respectivas tablas. Habiendo creado este usuario evitaremos el uso de '*root'* para el resto del trabajo práctico.
   Adicionalmente, con respecto a esta base de datos:
   1. Cree un usuario sólo con permisos para realizar consultas de selección, es decir que no puedan realizar cambios en la base. Use el nombre 'appointments_select'.
   2. Cree un usuario que pueda realizar consultas de selección, inserción, actualización y eliminación a nivel de filas, pero que no puedan modificar el esquema. Use el nombre 'appointments_update'.
   3. Cree un usuario que tenga los permisos de los anteriores, pero que además pueda modificar el esquema de la base de datos. Use el nombre 'appointments_schema'.

Resolución

Inciso 1

```
GRANT SELECT, INSERT, UPDATE, DELETE, CREATE, DROP, ALTER, CREATE ROUTINE,
ALTER ROUTINE, EXECUTE
ON appointments.* TO 'appointments_user'@'localhost';
```

Inciso 1.1

```
CREATE USER 'appointments_select'@'localhost' IDENTIFIED BY 'bd1_2025';
GRANT SELECT ON appointments.* TO 'appointments_select'@'localhost';
```

Inciso 1.2

```
CREATE USER 'appointments_update'@'localhost' IDENTIFIED BY 'bd1_2025';
GRANT SELECT, INSERT, UPDATE, DELETE ON appointments.* TO
'appointments_update'@'localhost';
```

Inciso 1.3

```
CREATE USER 'appointments_schema'@'localhost' IDENTIFIED BY 'bd1_2025';
GRANT ALL PRIVILEGES ON appointments.* TO
'appointments_schema'@'localhost';
```

Ejercicio 2

Hallar aquellos pacientes que para todas sus consultas médicas siempre hayan dejado su número de teléfono primario (nunca el teléfono secundario).

Solución 1

```
SELECT p.patient_id, p.patient_name
FROM patient p INNER JOIN appointment a ON p.patient_id = a.patient_id
GROUP BY p.patient_id, p.primary_phone, p.patient_name
HAVING COUNT(*) = SUM(a.contact_phone = p.primary_phone);
```

Solución 2

```
SELECT p.patient_id, p.patient_name
FROM PATIENT p
WHERE NOT EXISTS (
    SELECT *
    FROM APPOINTMENT a
    WHERE a.patient_id = p.patient_id
      AND a.contact_phone <> p.primary_phone
);
```

Ejercicio 3

Crear una vista llamada ‘doctors_per_patients’ que muestre los id de los pacientes y los id de doctores de la ciudad donde vive el paciente.

Resolución

```
CREATE VIEW appointments.doctors_per_patients AS
SELECT p.patient_id, d.doctor_id
FROM appointments.patient p, appointments.doctor d
ON p.patient_city = d.doctor_city;
```

Ejercicio 4

Utiliza la vista generada en el ejercicio anterior para resolver las siguientes consultas:

1. Obtener la cantidad de doctores por cada paciente que tiene disponible en su ciudad
2. Obtener los nombres de los pacientes sin doctores en su ciudad
3. Obtener los doctores que comparten ciudad con más de cinco pacientes.

Resolución

Inciso 1

```
SELECT patient_id, COUNT(doctor_id) AS cantidad_doctores
FROM appointments.doctors_per_patients
GROUP BY patient_id;
```

Inciso 2

```
SELECT p.patient_name
FROM patient p
LEFT JOIN appointments.doctors_per_patients dp
  ON p.patient_id = dp.patient_id
WHERE dp.doctor_id IS NULL;
```

Inciso 3

```
SELECT doctor_id
FROM appointments.doctors_per_patients
GROUP BY doctor_id
HAVING COUNT(patient_id) > 5;
```

Ejercicio 5

Resolución

```
CREATE TABLE appointments_per_patient (
    idApP INT(11) NOT NULL AUTO_INCREMENT,
```

```
    id_patient INT(11),
    count_appointments INT(11),
    last_update DATETIME,
    user VARCHAR(16),
    PRIMARY KEY (idApP)
);
```

Ejercicio 6

Crear un Stored Procedure que realice los siguientes pasos dentro de una transacción:

1. Realizar la siguiente consulta: cada *pacient* (identificado por *id_patient*), calcule la cantidad de appointments que tiene registradas. Registrar la fecha en la que se realiza esta carga y además del usuario con el se realiza.
2. Guardar el resultado de la consulta en un cursor.
3. Iterar el cursor e insertar los valores correspondientes en la tabla APPOINTMENTS PER PATIENT. Tenga en cuenta que last_update es la fecha en que se realiza esta carga, es decir la fecha actual, mientras que user es el usuario logueado actualmente, utilizar las correspondientes funciones para esto.

Resolución

```
DELIMITER //

CREATE PROCEDURE appointments_patient()
BEGIN
    DECLARE aux_id INT(11);
    DECLARE aux_count INT(11);
    DECLARE aux_last DATETIME;
    DECLARE aux_user VARCHAR(16);
    DECLARE fin INT DEFAULT 0;

    DECLARE cursor_appointments CURSOR FOR
        SELECT a.patient_id,
               COUNT(*) AS count_appointments,
               NOW() AS last_update
        FROM appointment a
        GROUP BY a.patient_id;

    DECLARE CONTINUE HANDLER FOR NOT FOUND SET fin = 1;

    SET aux_user = LEFT(CURRENT_USER(), 16);

    START TRANSACTION;

    OPEN cursor_appointments;
```

```
    loop_cursor: LOOP
        FETCH cursor_appointments INTO aux_id, aux_count, aux_last;

        IF fin = 1 THEN
            LEAVE loop_cursor;
        END IF;

        INSERT INTO appointments_per_patient (id_patient,
count_appointments, last_update, user)
        VALUES (aux_id, aux_count, aux_last, aux_user);
    END LOOP;

    CLOSE cursor_appointments;

    COMMIT;
END //

DELIMITER ;
```

Ejercicio 7

1. Indique si las siguientes afirmaciones sobre triggers son verdaderas o falsas. Justifique las falsas.
    1. Un trigger se ejecuta únicamente cuando se inserta una fila en una tabla.
    2. Un trigger puede ejecutarse antes o después de la operación, esto es definido automáticamente según el tipo de la operación (UPDATE, INSERT o DELETE)
    3. Todo trigger debe asociarse a una tabla en concreto.
    4. NEW y OLD son palabras clave que permiten acceder a los valores de las filas afectadas y se pueden usar ambos independientemente de la operación utilizada.
    5. FOR EACH ROW en un trigger se usa para indicar que el trigger se ejecutará una vez por cada fila afectada por la operación.

Resolución

Inciso 1

Falso, puede ejecutarse en otros eventos también, no solo cuando se inserta en una tabla. Como en update y delete.

Inciso 2

Falso, esto es definido al crear el trigger con BEFORE o AFTER

Inciso 3

Verdadero

Inciso 4

Falso, **no se pueden usar ambos indistintamente en cualquier operación**, y dependiendo del tipo de trigger algunos no estarán disponibles.

Inciso 5

Verdadero.

Ejercicio 8

1. Crear un Trigger de modo que al insertar un dato en la tabla Appointment, se actualice la cantidad de appointments del paciente, la fecha de actualización y el usuario responsable de la misma (actualiza la tabla APPOINTMENTS PER PATIENT).

Resolución

```
USE appointments;

DELIMITER $$
CREATE TRIGGER update_appointments_per_patient
AFTER INSERT ON appointment
for each row
BEGIN
    UPDATE appointments_per_patient
    SET idApP
        count_appointments = count_appointments + 1,
        last_update = NOW(),
        user = CURRENT_USER()
    WHERE id_patient = NEW.patient_id;
END$$
DELIMITER ;
```

Ejercicio 9

Crear un stored procedure que sirva para agregar un *appointment*, junto el registro de un doctor que lo atendió (*medical_review*) y un medicamento que se le recetó (*prescribed_medication*), dentro de una sola transacción. El stored procedure debe recibir los siguientes parámetros: patient_id, doctor_id, appointment_duration, contact_phone, appointment_address, medication_name. El appointment_date será la fecha actual. Los atributos restantes deben ser obtenidos de la tabla Patient (o dejarse en NULL).

Resolución

```
DELIMITER //
CREATE PROCEDURE `new_appointment_patient` (IN pat_id INTEGER, IN doct_id
INTEGER, IN appoint_duration INTEGER,IN cont_phone VARCHAR(255), IN
```

```
appoint_adress VARCHAR(255), IN med_name VARCHAR(30))
BEGIN
    START TRANSACTION;
        INSERT INTO appointments.appointment (patient_id,
appointment_date, appointment_duration, contact_phone)
        VALUES (pat_id,NOW(),appoint_duration,cont_phone);
        INSERT INTO appointments.medical_review (patient_id,
appointment_date, doctor_id)
        VALUES (pat_id,NOW(),doct_id);
        INSERT INTO appointment.prescribed_medication (patient_id,
appointment_date, medication_name)
        VALUES (pat_id,NOW(), med_name);
    COMMIT;
END

DELIMITER ;
```

Ejercicio 10

1. Ejecutar el stored procedure del punto 9 con los siguientes datos:
   patient_id: 10004427
   doctor_id: 1003
   appointment_duration: 30
   contact_phone: +54 15 2913 9963
   appointment_address: ‘Hospital Italiano’
   medication_name: ‘Paracetamol’

Resolución

```
CALL appointments.new_appointment_patient(10004427,1003,30,'+54 15 2913
9963','Hospital Italiano','Paracetamol');
```

Ejercicio 11

Considerando la siguiente consulta:

```
SELECT
    COUNT(a.patient_id)
FROM
    appointment a,
    patient p,
    doctor d,
    medical_review mr
WHERE
    a.patient_id = p.patient_id
    AND a.patient_id = mr.patient_id
    AND a.appointment_date = mr.appointment_date
```

```
    AND mr.doctor_id = d.doctor_id
    AND d.doctor_specialty = 'Cardiology'
    AND p.patient_city = 'Rosario';
```

Analice su plan de ejecución mediante el uso de la sentencia EXPLAIN.

1. ¿Qué atributos del plan de ejecución encuentra relevantes para evaluar la performance de la consulta?
2. Observe en particular el atributo type ¿cómo se están aplicando los JOIN entre las tablas involucradas?
3. Según lo que observó en los puntos anteriores, ¿qué mejoras se pueden realizar para optimizar la consulta?
4. Aplique las mejoras propuestas y vuelva a analizar el plan de ejecución. ¿Qué cambios observa?

Respuestas

1. Atributos relevantes

Los atributos más importantes del resultado del `EXPLAIN` para evaluar la performance de esta consulta son:

- `type` : Indica el **método de acceso** a los datos de la tabla. Es el indicador más importante de eficiencia. Valores como `ALL` (escaneo de tabla completa) son muy ineficientes, mientras que `eq_ref` , `ref` , o `const` son eficientes.
- `rows` : Muestra la cantidad **estimada de filas** que MySQL debe examinar (leer) de la tabla. Un valor alto, especialmente combinado con un `type` de `ALL` , indica una operación costosa.
- `key` : Muestra el **índice real** que el optimizador ha seleccionado para utilizar. Si este campo es `NULL` , la tabla está siendo escaneada completamente.
- `Extra` :: proporciona información adicional sobre como se ejecuta la consulta.Los posibles valores son: distinct, not exists, range checked for each record (index map: #), Using filesort, Using index, Using temporary, Using where, Using sort_union(...), Using union(...), Using ntersect(...), Using index for group-by

---

2. Atributo type

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	p	NULL	ALL	PRIMARY	NULL	NULL	NULL	1000	10.00	Using where
2	1	SIMPLE	mr	NULL	ref	PRIMARY,doctor_id	PRIMARY	4	appointments.p.patient_id	29	100.00	Using index
3	1	SIMPLE	d	NULL	eq_ref	PRIMARY	PRIMARY	4	appointments.mr.doctor_id	1	10.00	Using where
4	1	SIMPLE	a	NULL	eq_ref	PRIMARY	PRIMARY	9	appointments.p.patient_id,appointm…	1	100.00	Using index

Tabla	type Esperado	Clave Usada (key)	Observación de la Unión (JOIN)
d (`doctor`)	**eq_ref**	`PRIMARY`	La unión es **eficiente**, ya que el acceso se realiza por la clave primaria (`doctor_id`), lo que garantiza la unicidad de las filas.
mr (`medical_review`)	`eq_ref`	`PRIMARY`	La unión con `appointment` es **muy eficiente**, usando la clave primaria compuesta (`patient_id` , `appointment_date`).
a (`appointment`)	`eq_ref`	`PRIMARY`	Se une de forma eficiente con `medical_review` por la clave compuesta.
p (`patient`)	**ALL**	NULL	Se realiza un **escaneo de tabla completa** (`ALL`) porque no hay un índice creado en la columna de filtro (`patient_city`). Esta es la **principal ineficiencia**.

3. Aspectos a mejorar

El aspecto a mejorar corresponde principalmente a lo que ocurre con la tabla paciente, que esta escaneando toda la tabla.

- **Crear un índice en** `patient(patient_city)` : Esto permite a MySQL encontrar a los pacientes de 'Rosario' directamente, sin tener que leer toda la tabla.

4. Aplicar mejoras

```
CREATE INDEX idx_patient_city ON patient (patient_city);
```

#	id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	1	SIMPLE	p	NULL	ref	PRIMARY,idx_patient_city	idx_patient_city	768	const	82	100.00	Using index
2	1	SIMPLE	mr	NULL	ref	PRIMARY,doctor_id	PRIMARY	4	appointments.p.patient_id	29	100.00	Using index
3	1	SIMPLE	d	NULL	eq_ref	PRIMARY	PRIMARY	4	appointments.mr.doctor_id	1	10.00	Using where
4	1	SIMPLE	a	NULL	eq_ref	PRIMARY	PRIMARY	9	appointments.p.patient_id,appointm…	1	100.00	Using index

Al volver a ejecutar el `EXPLAIN` después de aplicar la mejora, los cambios observados son:

- **En la tabla** `patient` **:**
  - El `type` cambia de `ALL` a `ref` (o `range`). Esto confirma que el optimizador ahora está usando el índice.
  - La columna `key` muestra el nombre del nuevo índice: `idx_patient_city` .

- El valor en la columna `rows` para la tabla `patient` se **reduce drásticamente**, ya que la consulta solo necesita examinar las filas que coinciden con `'Rosario'`, en lugar de la tabla completa.

Conclusión: La creación del índice elimina el problema mas grande, reduciendo el costo de la consulta y optimizando significativamente su performance.