# Ampliando la caja de herramientas: *common table expression* y funciones analíticas

*y*

# El diablo está en los detalles: optimización de la base de datos en función de su uso.

**NOMBRE Y APELLIDOS: _____**

La empresa "UOC Salud", se ha puesto de nuevo en contacto con nosotros para que implementéis los requisitos que nos han propuesto.

Para la implementación de esta PEC, debéis de crear una base de datos nueva denominada **tx_pec4** y ejecutar el script adjunto **BBDD_Clinical_structure.sql**. El segundo script proporcionado (**BBDD_Clinical_data.sql**) os dará un conjunto de datos que os permitirá implementar los componentes requeridos en los diferentes apartados de esta PEC. **NOTA**: los datos proporcionados no se pueden modificar y los ejercicios han de realizarse en base a dichos datos.

Consideraciones para la entrega y realización de la PEC:

- Todo lo que se pide en esta PEC está explicado en los bloques didácticos 4 y 5.
- Sed fieles al enunciado. Si algo no está claro, por favor comentadlo en el foro o directamente al correo del profesor colaborador.
- Se recomienda la utilización de **pgAdmin** para la implementación de toda la PEC. Existe otra alternativa que es ***psql*** (línea de comandos), pero es preferible que utilicéis pgAdmin ya que es una interfaz gráfica que os permitirá editar y crear sentencias SQL (así como mostrar los resultados) de forma más sencilla que *psql*.
- Tal y como se indica en el enunciado, la respuesta de los ejercicios que se requiera ha de entregarse en un fichero *.sql* diferente, con el nombre correspondiente. Se evaluará el código entregado en estos ficheros *.sql* y **NO el código que aparezca en el documento o en los pantallazos adjuntos**.
- Las capturas de pantalla de los ejercicios (y explicaciones pertinentes) han de proporcionarse en un documento aparte (se proporciona una plantilla para el caso, **indicad vuestro nombre en el documento**, por favor).
- Se debe de realizar la entrega de todos los ficheros de la PEC (tanto los ficheros *.sql* como el documento con explicaciones y capturas de pantalla) en un fichero comprimido *.zip*.

Consideraciones para la evaluación del ejercicio:

- Se tendrá en cuenta la aplicación de las buenas prácticas de codificación en SQL, de consultas y de programación de procedimientos y disparadores. Es decir: código con sangrado, uso de cláusulas SQL de forma correcta, comentarios, cabeceras en el procedimiento, etc.
- Los *scripts* proporcionados por el estudiante con las soluciones de los ejercicios han de ejecutarse correctamente. El estudiante ha de asegurarse de que lanzando el *script* completo de cada ejercicio no produzca ningún error.
- **Importante**: Las sentencias SQL proporcionadas en los *scripts* han de ser creadas de forma manual y no mediante asistentes que PostgreSQL/pgAdmin puedan proporcionar. Se pretende aprender SQL y no la utilización de asistentes.
- Las sentencias SQL proporcionadas en los ejercicios han de ser **solamente** aquellas que pide el enunciado y ninguna otra más. Cualquier sentencia añadida a mayores, si está mal o provoca que el *script* no se ejecute correctamente a la hora de corregirlo, penalizará el ejercicio.

## EJERCICIO 1 (30%)

La empresa “UOC Salud” quiere realizar unas mejoras más en la base de datos. Para ello, nos han contactado nuevamente para pedirnos lo siguiente:

1) (10%) La tabla *tb_oders_catalog* contiene el catálogo de prestaciones clasificados según tres niveles, *category, subcategory y order_desc.* Ahora queremos modificar la estructura de esta tabla para permitir guardar el catálogo de prestaciones sin límite de niveles. Para resolverlo, nos piden:

   - Añadir un campo *parent_code* para referenciar a la categoría de prestación de nivel superior. Las prestaciones de nivel superior tendrán este campo a *null*.
   - Crear una secuencia llamada *seq_orders_catalog* que empiece por 1 y que se vaya incrementando en una unidad.
   - Insertar en la tabla *tb_orders_catalog* las distintas prestaciones de nivel 1 (campo *category*) y nivel 2 (campo *subcategory*) utilizando la secuencia anterior para asignar los distintos *order_code*. En las prestaciones de nivel 2 y 3 el campo *parent_code* tiene que apuntar a la categoría de nivel superior correspondiente.
   - Eliminar los campos redundantes *category* y *subcategory*.

2) (10%) Define, mediante CTE recursivas, una consulta que devuelva todo el catálogo de prestaciones con el campo Descripción en forma de árbol (ver imagen).

| |
|---|
| Cardiologia |
| --->Cardioversión |
| --->Cardioversión electrica programada |
| --->Cardioversión electrica urgente |
| --->Ecocardiografía |
| --->Ecocardiografía transtorácica (AR) |
| --->Esfuerzo |
| --->Prueba de esfuerzo |
| --->Gestión |
| --->Solicitud Estudio hemodinámico Programad |
| --->Solicitud Estudio hemodinámico Urgente |

3) (10%) Define, mediante funciones analíticas, la sentencia SQL que obtenga todos los pacientes (*tb_patient*), indicando para cada uno de ellos:

   - El código, EHR y nombre del paciente.
   - Una columna que numere los episodios de cada paciente, ordenados cronológicamente.
   - Una columna que contenga la suma de los costes de las distintas prestaciones creadas para dicho episodio.
   - Una columna que contenga el acumulado de la columna anterior (en la última fila de un paciente veremos el total gastado por el paciente).

Las sentencias SQL de estos 3 apartados se tienen que entregar en un fichero llamado **pec4_ej1.sql**.

## EJERCICIO 2 (40%)

a) (5%) Cread la tabla tb_location para guardar información de las ubicaciones. Se quiere guardar información según la especificación proporcionada a continuación.

| **Tabla:** tb_location **Esquema:** clinical | | | | | |
|---|---|---|---|---|---|
| **Nombre columna** | **Tipo de datos** | **Acepta Nulos** | **Clave primaria** | **Clave foránea** | **Valor por defecto** |
| location_id | Numérico entero | No | Si | | |
| Name | Cadena de 50 caracteres variable | No | No | | |
| Description | Cadena de 50 caracteres variable | No | No | | |
| Type | Cadena de 50 caracteres variable | No | No | | |

Modificad la tabla tb_orders para añadir la ubicación y la fecha de la reserva. Para simplificar consideramos que una ubicación se reserva un día entero.

| **Tabla:** tb_orders **Esquema:** clinical | | | | | |
|---|---|---|---|---|---|
| **Nombre columna** | **Tipo de datos** | **Acepta Nulos** | **Clave primaria** | **Clave foránea** | **Valor por defecto** |
| location_id | Numérico entero | Si | No | tb_location | Null |
| location_dt | Fecha con tiempo | Si | No | | Null |

Añade las ubicaciones siguientes:

| | location_id [PK] integer | name character varying (50) | description character varying (50) | type character varying (50) |
|---|---|---|---|---|
| 1 | 1 | Sala1 | Sala general 1 | aa |
| 2 | 2 | Sala2 | Sala general 2 | General |
| 3 | 3 | Sala3 | Sala general 3 | General |

Las sentencias SQL de este apartado se tienen que entregar en un fichero llamado **pec4_ej2.sql**.

b) (25%) Para cada nivel de aislamiento, buscad una estrategia basada en transacciones para asegurar que no se puedan reservar la misma ubicación más de una vez en el mismo día. Se valorará que la estrategia sea óptima para cada nivel de aislamiento.

La inserción de nuevas reservas la haremos con la sentencia siguiente:

```
UPDATE tb_orders SET location_id=A, location_dt=B WHERE order_id=C;
```

Para evitar reservar dos veces la misma ubicación a la misma fecha primero haremos una consulta para saber si ya existe alguna reserva en la misma ubicación y fecha:

```
SELECT * FROM tb_orders WHERE location_id=A AND location_dt=B;
```

Así pues, la idea básica de la transacción sería:

```
BEGIN TRANSACTION READ WRITE;
SELECT * FROM tb_orders WHERE location_id=A AND location_dt=B;
UPDATE tb_orders SET location_id=A, location_dt=B WHERE order_id=C;
COMMIT
```

Las posibles interferencias de este tipo de transacciones serían:

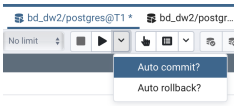
- Actualización perdida: si dos transacciones asignan una ubicación y fecha distintas a la misma prestación (tb_order).
- Análisis inconsistente: al final de la transacción tendríamos que asegurar que si ejecutásemos otra vez la consulta, daría sólo la nueva reserva que acabamos de hacer, sin aparecer reservas fantasmas creadas desde otra transacción.

Por tanto, el nivel de aislamiento mínimo para evitar la duplicidad de reservas sería SERIALIZABLE.

Para los niveles de aislamiento inferiores (READ UNCOMMITTED, READ COMMITTED y REPEATABLE READ) tendremos que buscar otra estrategia. Dado que no podemos controlar la aparición de reservas fantasmas, no nos queda más remedio que evitar reservar a la vez la misma ubicación. Y esto lo podríamos lograr haciendo un UPDATE de la ubicación al principio de la transacción. Así, sea cual sea el nivel de aislamiento, sólo la primera transacción podrá hacer la reserva. La segunda quedará bloqueada hasta que la primera no finalice.

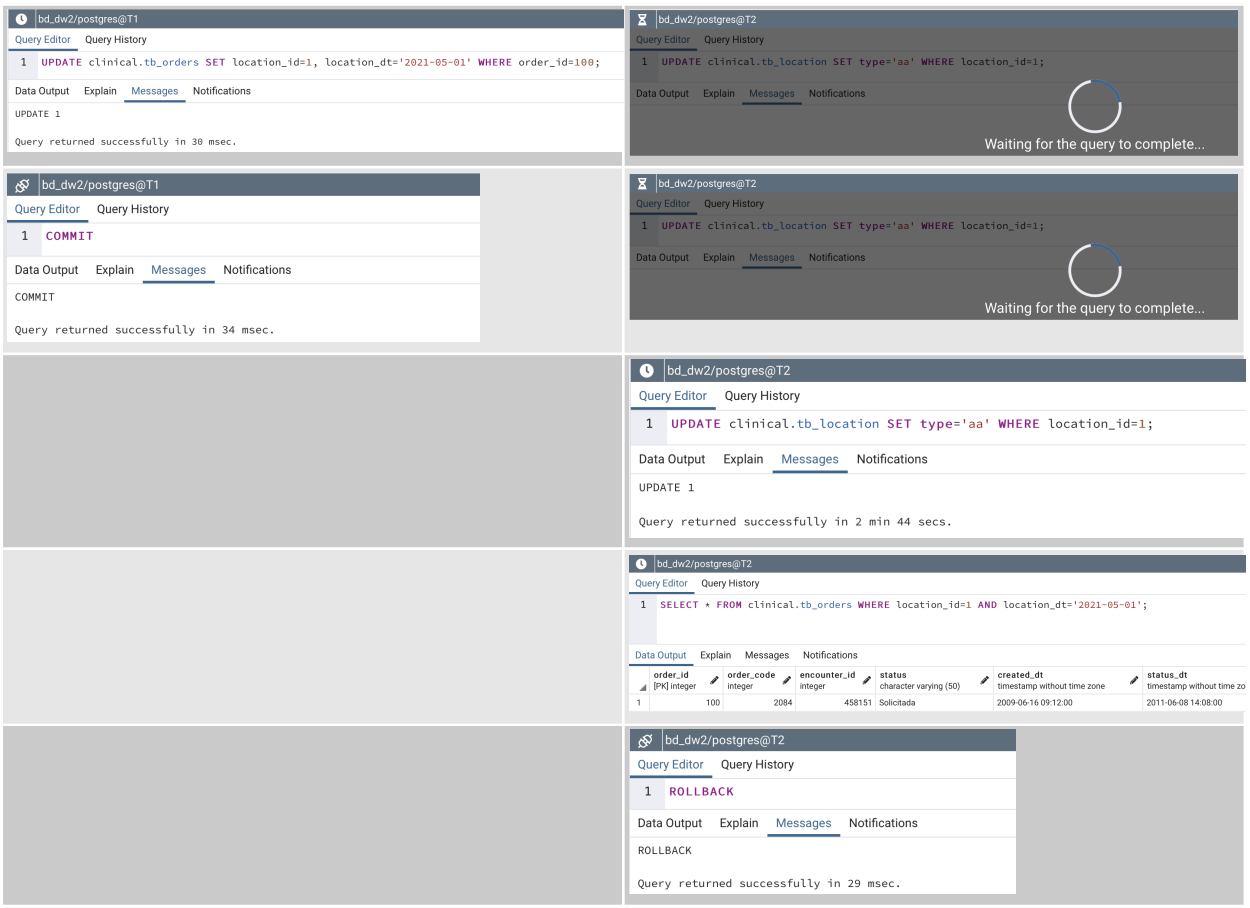
```
BEGIN TRANSACTION READ WRITE;
UPDATE tb_location SET type=type WHERE location_id=A;
SELECT * FROM tb_orders WHERE location_id=A AND location_dt=B;
UPDATE tb_orders SET location_id=A, location_dt=B WHERE order_id=C;
COMMIT
```

c) (10%) Para cada estrategia presentada mostrad una historia de dos transacciones intentando reservar la misma ubicación el mismo día. Esta historia se hará a base de capturas de pantallas de pgAdmin tras ejecutar cada sentencia SQL. Para evitar confusiones, en pgAdmin cread dos Servers T1 y T2 para poder diferenciar entre las dos transacciones. No olvidéis de desactivar "Auto commit" en el "Query Tool".

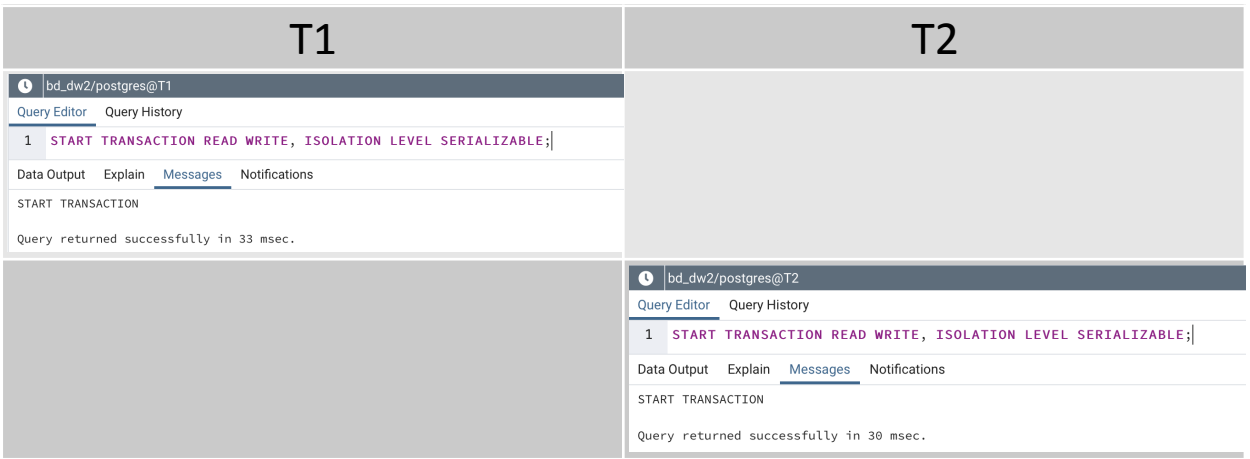



## Para los niveles de aislamiento READ UNCOMMITED, READ COMMITED y REPEATABLE READ:

| T1 | T2 |
|---|---|
| bd_dw2/postgres@T1<br>Query Editor Query History<br>`1 START TRANSACTION READ WRITE, ISOLATION LEVEL READ UNCOMMITTED;`<br>Data Output Explain Messages Notifications<br>START TRANSACTION<br>Query returned successfully in 35 msec. | |
| | bd_dw2/postgres@T2<br>Query Editor Query History<br>`1 START TRANSACTION READ WRITE, ISOLATION LEVEL READ UNCOMMITTED;`<br>Data Output Explain Messages Notifications<br>START TRANSACTION<br>Query returned successfully in 28 msec. |
| bd_dw2/postgres@T1<br>Query Editor Query History<br>`1 UPDATE clinical.tb_location SET type='aa' WHERE location_id=1;`<br>Data Output Explain Messages Notifications<br>UPDATE 1<br>Query returned successfully in 32 msec. | |
| | bd_dw2/postgres@T2<br>Query Editor Query History<br>`1 UPDATE clinical.tb_location SET type='aa' WHERE location_id=1;`<br>Data Output Explain Messages Notifications<br>Waiting for the query to complete... |
| bd_dw2/postgres@T1<br>Query Editor Query History<br>`1 SELECT * FROM clinical.tb_orders WHERE location_id=1 AND location_dt='2021-05-01';`<br>Data Output Explain Messages Notifications<br>order_id [PK] integer · order_code integer · encounter_id integer · status character varying (50) · created_dt timestamp without time zone · status_dt timestamp w | bd_dw2/postgres@T2<br>Query Editor Query History<br>`1 UPDATE clinical.tb_location SET type='aa' WHERE location_id=1;`<br>Data Output Explain Messages Notifications<br>Waiting for the query to complete... |




Para el nivel de aislamiento SERIALIZABLE: en este caso podemos ver como T2 no se bloquea al comprobar si hay reservas hechas, pero tampoco ve los cambios introducidos por T1. Es al intentar añadir la reserva que postgreSql detecta la aparición del fantasma y lanza un error.



bd_dw2/postgres@T1

Query Editor | Query History

```
1  SELECT * FROM clinical.tb_orders WHERE location_id=1 AND location_dt='2021-05-01';
```

Data Output | Explain | Messages | Notifications

| order_id [PK] integer | order_code integer | encounter_id integer | status character varying (50) | created_dt timestamp without time zone | status_dt timestamp without time zone |
|---|---|---|---|---|---|

bd_dw2/postgres@T1

Query Editor | Query History

```
1  UPDATE clinical.tb_orders SET location_id=1, location_dt='2021-05-01' WHERE order_id=100;
```

Data Output | Explain | Messages | Notifications

```
UPDATE 1

Query returned successfully in 31 msec.
```

bd_dw2/postgres@T2

Query Editor | Query History

```
1  SELECT * FROM clinical.tb_orders WHERE location_id=1 AND location_dt='2021-05-01';
```

Data Output | Explain | Messages | Notifications

| order_id [PK] integer | order_code integer | encounter_id integer | status character varying (50) | created_dt timestamp without time zone | status_dt timestamp without time zone |
|---|---|---|---|---|---|

bd_dw2/postgres@T1

Query Editor | Query History

```
1  COMMIT
```

Data Output | Explain | Messages | Notifications

```
COMMIT

Query returned successfully in 34 msec.
```

bd_dw2/postgres@T2

Query Editor | Query History

```
1  UPDATE clinical.tb_orders SET location_id=1, location_dt='2021-05-01' WHERE order_id=100;
```

Data Output | Explain | Messages | Notifications

```
ERROR:  could not serialize access due to concurrent update
SQL state: 40001
```

bd_dw2/postgres@T2

Query Editor | Query History

```
1  ROLLBACK
```

Data Output | Explain | Messages | Notifications

```
ROLLBACK

Query returned successfully in 28 msec.
```

## EJERCICIO 3 (20%)

Proporcionar las sentencias SQL necesarias para cada uno de los apartados, proporcionando un pantallazo del resultado y la explicación pertinente.

1) (5%) Cread un espacio de tablas mediante una sentencia SQL. Este *tablespace* ha de llamarse **ts_clinical** y se ha de almacenar en la ruta "x**:\ts_clinical**", donde ***x*** es la unidad de Windows donde se deben de almacenar los ficheros del *tablespace*. Por ejemplo, *x* puede ser la unidad *C:*, la unidad *F:*, etc. (**Nota**: aquellos que utilicéis Linux utilizad la ruta /clinical/ en el directorio raíz, o en Mac la ruta /Users/<vuestro usuario>/ts_clinical). Adjuntad una captura de pantalla de lo que PostgreSQL genera y explicad si habéis tenido que realizar alguna operación extra en el sistema operativo.

***Nota***: Es posible que los que utilicéis MAC y/o Linux tengáis además que asignar el permiso de "*owner*" al usuario "postgres". (Linux: sudo chown postgres /ts_clinical  Mac: sudo chown postgres /Users/<vuestro usuario/ts_clinical) .

Se debe crear la carpeta ts_*clinical*, y luego ya se puede ejecutar la instrucción para crear el tablespace:

```
1  CREATE TABLESPACE ts_clinical LOCATION '/ts_clinical';
```

Data Output  Explain  Messages  Notifications

```
CREATE TABLESPACE

Query returned successfully in 43 msec.
```

2) (5%) En el esquema clinical cread una tabla tb_dummy indicando que se guarde en el nuevo espacio de tablas creado en el apartado anterior. Esta tabla tendrá un único campo "description" de tipo carácter de longitud 1024 no nulo. Proporcionar la sentencia SQL.

```
2  CREATE TABLE clinical.ts_dummy (
3      description CHARACTER(1024) NOT NULL
4      ) TABLESPACE ts_clinical;
5
```

Data Output  Explain  Messages  Notifications

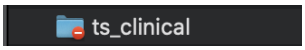
```
CREATE TABLE

Query returned successfully in 33 msec.
```

3) (10%) En este apartado vamos a investigar cómo se guarda la tabla tb_dummy en el espacio de tablas.

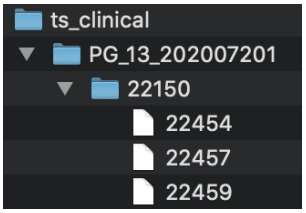
a) Haced una captura de la estructura de ficheros que se ha creado para almacenar la tabla tb_dummy.

***Nota***: Es posible que los que utilicéis MAC y/o Linux tengáis problemas de permisos para acceder al directorio del espacio de tablas. Podéis abrir el acceso con sudo chmod -R 777 /clinical en Linux, o sudo chmod -R 777 /Users/<vuestro usuario>/clinical en Mac.

## Solución:




```
sudo chmod -R 777 ts_clinical
```




b) Insertad un registro con en tb_dummy con la descripción “valor1”. ¿En qué fichero se ha guardado este registro? ¿Cuál es el tamaño de este fichero? ¿Por qué es tan grande este fichero si solo tiene que guardar un registro? ¿Cuántos registros caben en este fichero?

**Nota**: Postgresql guarda las tablas en ficheros que contienen códigos especiales y que no se

```
000017B0  20 20 20 20 20 20 20 20 20 20 20 20 00 00 00 00                    ....
000017C0  C9 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ╔...............
000017D0  02 00 01 00 02 09 18 00 10 10 00 00 76 61 6C 75  ............valu
000017E0  65 20 32 20 20 20 20 20 20 20 20 20 20 20 20 20  e 2
000017F0  20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
```

pueden ver del todo bien en un editor normal. Aconsejamos usar un editor hexadecimal online como `https://hexed.it` para abrir estos ficheros y ver su contenido.

## Solución:

```
6  INSERT INTO clinical.ts_dummy(description) VALUES('value 1');
```

Data Output | Explain | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 30 msec.
```

Los registros de la tabla tb_dummy se encuentran en el final del primer fichero, en este caso el fichero 22454.

```
22454 ×
00001B90   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00001BA0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00001BB0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00001BC0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00001BD0   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00001BE0   D4 05 00 00 00 00 00 00 00 00 00 00 00 00 00 00   └...............
00001BF0   01 00 01 00 02 08 18 00 10 10 00 00 76 61 6C 75   ............valu
00001C00   65 20 31 20 20 20 20 20 20 20 20 20 20 20 20 20   e 1
```

El tamaño del fichero és de 8192 bytes, equivalente a 8KB, que es el tamaño de página por defecto de postgreSQL. Los gestores de bases de datos utilizan un tamaño de página grande para minimizar los accesos a disco y así acelerar las operaciones.

```
-rwxrwxrwx  1 postgres  staff  8192 May  1 23:14 22454
```

En el caso de la tabla tb_dummy, un registro ocupa exactamente 1024 caracteres o bytes. Así pues, desde un punto de vista teórico en una página cabrían exactamente 8 registros. Pero, como veremos en el apartado siguiente, esto no será así.

c) Insertad tantos registros como hayáis calculado en el apartado b) y comprobad que, justo en el último registro añadido, incrementa el tamaño del fichero. Si el tamaño incrementa antes de lo esperado, buscad una explicación.

## Solución:

```
6  INSERT INTO clinical.ts_dummy(description) VALUES('value2');
7  INSERT INTO clinical.ts_dummy(description) VALUES('value3');
8  INSERT INTO clinical.ts_dummy(description) VALUES('value4');
9  INSERT INTO clinical.ts_dummy(description) VALUES('value5');
10 INSERT INTO clinical.ts_dummy(description) VALUES('value6');
11 INSERT INTO clinical.ts_dummy(description) VALUES('value7');
12
```

Data Output | Explain | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 30 msec.
```

Podemos añadir 7 registros sin sobrepasar el tamaño de página. Pero al añadir el octavo registro se añade otra página. Esto se debe a que postgreSql no se limita a guardar los datos del registro sino que añade bytes con información del registro, como por ejemplo el número de registro (que se puede consultar con la función ROWNUMBER()).

```
14 INSERT INTO clinical.ts_dummy(description) VALUES('value8');
15
```

Data Output | Explain | Messages | Notifications

```
INSERT 0 1

Query returned successfully in 32 msec.
```

```
-rwxrwxrwx  1 postgres  staff  16384 May  1 23:21 22454
```

Al añadir el octavo registro, se añade una nueva página y el tamaño del fichero aumenta a 16KB.

## EJERCICIO 4 (10%)

Dadas las siguientes consultas SQL sobre la base de datos clinical:

```
SELECT ehr_number, name,  encounter_id
FROM clinical.tb_patient p
JOIN clinical.tb_encounter e ON e.patient_id=p.patient_id
```

Y

```
SELECT ehr_number, name,  encounter_id
FROM clinical.tb_patient p
JOIN clinical.tb_encounter e ON e.patient_id<=p.patient_id
```

1) (5%) Identifica cual es el plan de ejecución que utiliza PostgreSQL para resolver dichas consultas. Haz captura de pantalla de dicho plan y descríbelo con tus propias palabras.

El plan de la consulta se obtiene ejecutando la sentencia con EXPLAIN:

```
EXPLAIN SELECT ehr_number, name,  encounter_id
FROM clinical.tb_patient p
JOIN clinical.tb_encounter e ON e.patient_id=p.patient_id
```

| | QUERY PLAN text |
|---|---|
| 1 | Hash Join  (cost=13.60..19.14 rows=200 width=126) |
| 2 | Hash Cond: (e.patient_id = p.patient_id) |
| 3 | -> Seq Scan on tb_encounter e  (cost=0.00..5.00 rows=200 width=8) |
| 4 | -> Hash  (cost=11.60..11.60 rows=160 width=126) |
| 5 | -> Seq Scan on tb_patient p  (cost=0.00..11.60 rows=160 width=126) |

La consulta hace primero un recorrido secuencial sobre *tb_patient* **(5)** para generar su índice de dispersión **(4)** y facilitar su posterior combinación con *tb_encounter***.** Para ello, hace un recorrido secuencial sobre *tb_encounter* (**3**) y, mediante la condición definida en la combinación (igualdad), combina ambas tablas (**1,2**).

```
EXPLAIN SELECT ehr_number, name,  encounter_id
FROM clinical.tb_patient p
JOIN clinical.tb_encounter e ON e.patient_id<=p.patient_id
```

| | QUERY PLAN text |
|---|---|
| 1 | Nested Loop  (cost=0.14..373.50 rows=10667 width=126) |
| 2 | -> Seq Scan on tb_encounter e  (cost=0.00..5.00 rows=200 width=8) |
| 3 | -> Index Scan using pk_tb_patient on tb_patient p  (cost=0.14..1.31 rows=53 width=126) |
| 4 | Index Cond: (patient_id >= e.patient_id) |

Combinamos (1) las filas de las tablas tb_encounter y tb_patient usando el algoritmo Nested Loop. Para ello hacemos un recorrido secuencial de la tabla tb_encounter (2) y, para cada episodio, hacemos un recorrido secuencial del índice pk_tb_patient(3) para obtener los pacientes con un identificador igual o mayor que el del paciente del episodio (condición de la join).

2) (5%) Explicad el porqué de los diferentes algoritmos aplicados para resolver las consultas del apartado anterior.

Generalmente postgreSql utiliza el algoritmo HashJoin para combinar tablas. Para ello primero construye los índices de dispersión de cada tabla y, en un segundo paso, combina las tablas. Incluso ya habiendo creado los índices fk_encounter_patient y pk_tb_patient para combinar las tablas, posgreSql determina que es más eficiente crear los índices de dispersión de nuevo que no aprovechar los ya creados. El motivo es que el acceso directo de los índices de dispersión es más rápido y compensa el coste de creación de éstos.

En la segunda consulta el cambio es radical. PostgeSql deja de utilizar el algoritmo HashJoin a favor de Nested Loop. El motivo es que la condición de la join ya no es una igualdad (=) sino una desigualdad (<=) y en este caso los índices de dispersión no sirven, ya que no permiten acceder ordenadamente a los registros de la tabla. Por tanto, en esta segunda consulta ya no se crean índices nuevos sino que aprovechamos uno existente, el pk_tb_patient de tipo árbol B+, para acceder a los pacientes con un id igual o superior que el del paciente del episodio. Recordamos que postgreSql crea índices árbol B+ por defecto.

## Criterios de valoración

En el enunciado se indica el peso/valoración de cada ejercicio.

Para conseguir la puntuación máxima en los ejercicios, es necesario explicar con claridad la solución que se propone.

## Formato y fecha de entrega

Tenéis que enviar la PEC al buzón de Entrega y registro de EC disponible en el aula (apartado Evaluación). El formato del archivo que contiene vuestra solución puede ser **.pdf, .doc y .docx**. **Para otras opciones, por favor, contactar previamente con vuestro consultor**. El nombre del fichero debe contener el código de la asignatura, vuestro apellido y vuestro nombre, así como el número de actividad (PEC4).

La fecha límite para entregar la PEC4 es el **11/06/2021**.

Nota: **Propiedad intelectual**

Al presentar una práctica o PEC que haga uso de recursos ajenos, se tiene que presentar junto con ella un documento en que se detallen todos ellos, especificando el nombre de cada recurso, su autor, el lugar donde se obtuvo y su estatus legal: si la obra está protegida por el copyright o se acoge a alguna otra licencia de uso (Creative Commons, licencia GNU, GPL etc.). El estudiante tendrá que asegurarse que la licencia que sea no impide específicamente su uso en el marco de la práctica o PEC. En caso de no encontrar la información correspondiente tendrá que asumir que la obra está protegida por el copyright.

Será necesario, además, adjuntar los ficheros originales cuando las obras utilizadas sean digitales, y su código fuente, si así corresponde.