Instituto Tecnológico de Buenos Aires

INFORME - TPE OBLIGATORIO

72.41 Bases de Datos II

Integrantes

62003 - MARCOS CASIRAGHI

62067 - MARCOS GRONDA

62353 - MAXIMO GUSTAVO ROJAS PELLICCIA

Capítulo I: Introducción

Este trabajo práctico tiene como objetivo implementar los conocimientos aprendidos durante la cursada de la materia Bases de Datos II del segundo cuatrimestre de 2023. Consta de dos partes, una implementación con una base de datos relacional y una no relacional, de esta manera cubriendo en su mayoría los temas vistos. La implementación relacional fue llevada a cabo en PostgreSQL y la no relacional fue hecha en MongoDB. Dentro del repositorio del trabajo, se encuentran en distintos directorios cada implementación con sus respectivas consultas respondiendo al enunciado. La inicialización de las bases de datos se puede llevar a cabo mediante un script PSQL y luego una migración automatizada a MongoDB, o directamente con un script MongoDB. Además, se implementó una API ABM (alta, baja y modificación) para ambas bases de datos. Finalmente, a lo largo y ancho de este informe se irá discutiendo las distintas implementaciones y decisiones tomadas para cada unidad.

Capítulo II: PostgreSQL

En esta sección hablaremos acerca de lo implementado en PostgreSQL, es decir la primera parte del trabajo. Se han implementado 10 consultas y 2 vistas dentro de un archivo llamado “tp-queries.sql” y “tp-views-queries.sql” respectivamente dentro del directorio “sql”. También se cuenta con el archivo para inicializar la base de datos llamado “ITBA_2023_esquema_facturacion.sql” provisto por la cátedra para utilizarse con PostgreSQL.

Para resolver las 10 queries, como equipo, decidimos cada uno realizar una implementación propia de cada una de ellas. Luego, realizando una puesta en común y utilizando la funcionalidad de explain-analyze, elegimos las que eran más eficientes en tiempo de ejecución. Esta función aprendida en clase resultó efectiva ya que fue nuestro lineamiento para decidir cuál de las distintas queries utilizar. En el caso de las vistas, no pudimos utilizarlo ya que los tres habíamos desarrollado la misma query para su creación. Se utilizaron para resolver tanto funciones de agregación como subqueries, dependiendo de qué era lo más adecuado para nosotros.

A continuación, se mostraran las consultas realizadas en SQL, junto a un breve explicación cuando se denomine necesario:

`-- 1. Obtener el teléfono y el número de cliente del cliente con nombre “Wanda” y apellido “Baker”.`
`SELECT T.nro_telefono, C.nro_cliente`
`FROM E01_TELEFONO T JOIN E01_CLIENTE C ON T.nro_cliente = C.nro_cliente`
`WHERE C.nombre = 'Wanda' AND C.apellido = 'Baker';`
	Se debió usar una unión entre las tablas: `E01_TELEFONO` y `E01_CLIENTE`, usando el `nro_cliente` como clave.

`-- 2. Seleccionar todos los clientes que tengan registrada al menos una factura.`
`SELECT *`
`FROM e01_cliente`
`WHERE nro_cliente IN (SELECT nro_cliente`
`FROM e01_factura);`
	Primero se seleccionó todos los `nro_cliente` presentas en la tabla `e01_factura`. Si un cliente tiene una factura con su `nro_cliente`, entonces tiene una factura registrada. Luego se usó el operador `IN` para determinar qué cliente efectivamente tenía una factura a su nombre.

`-- 3. Seleccionar todos los clientes que no tengan registrada una factura.`
`SELECT *`
`FROM e01_cliente`
`WHERE nro_cliente NOT IN (SELECT nro_cliente`
`FROM e01_factura);`
	Análogamente al ejercicio anterior, se obtuvo los `nro_cliente`, pero luego se usó el operador `NOT IN`.

`-- 4. Seleccionar los productos que han sido facturados al menos 1 vez.`
`SELECT *`
`FROM e01_producto`
`WHERE codigo_producto IN (SELECT codigo_producto`
`FROM e01_detalle_factura);`
	Nuevamente se usó la misma estrategia de primero obtener los `codigo_producto` que cumplen lo requerido (en este caso que hayan sido facturados)

	y luego se seleccionó los productos que tienen su `codigo_producto` en esa lista.

`-- 5. Seleccionar los datos de los clientes junto con sus teléfonos`
`SELECT *`
`FROM e01_cliente c LEFT JOIN e01_telefono t ON c.nro_cliente = t.nro_cliente`
`ORDER BY c.nro_cliente;`
	Se aprovechó el uso del *`LEFT JOIN`* para obtener todos los clientes junto a sus teléfonos, *si tienen*.

`-- 6. Devolver todos los clientes, con la cantidad de facturas que tienen registradas (admitir nulos en valores de Clientes)`
`SELECT C.nro_cliente, nombre, apellido, direccion, activo, COUNT(F.nro_factura) AS cantidad_facturas`
`FROM E01_CLIENTE C LEFT JOIN E01_FACTURA F ON C.nro_cliente = F.nro_cliente`
`GROUP BY C.nro_cliente, nombre, apellido, direccion, activo`
`ORDER BY C.nro_cliente;`
	En esta consulta se tuvo que combinar el uso del `LEFT JOIN`, junto a la sentencia `GROUP BY` que permite agrupar por clientes para luego obtener la cantidad total de facturas a su nombre.

`-- 7. Listar todas las Facturas que hayan sido compradas por el cliente de`
`-- nombre "Pandora" y apellido "Tate"`
`SELECT *`
`FROM e01_factura`
`WHERE nro_cliente IN (SELECT nro_cliente`
`FROM e01_cliente`
`WHERE nombre = 'Pandora' AND apellido = 'Tate');`
	Nuevamente se usó la estrategia de obtener, en una subconsulta, el `nro_cliente` de el cliente apropiado y luego obtener las facturas que le corresponden con el operador `IN`.

`-- 8. Listar todas las Facturas que contengan productos de la marca “In Faucibus Inc.”`
`SELECT *`
`FROM e01_factura`
`WHERE nro_factura IN ( SELECT nro_factura`
`FROM e01_detalle_factura NATURAL JOIN e01_producto`
`WHERE marca = 'In Faucibus Inc.');`
	En esta consulta se usó una estrategia similar pero, se debió unir la tabla `e01_producto` con la tabla `e01_detalle_factura`, lo cual presenta los productos que realmente han sido facturados.

`-- 9. Mostrar cada teléfono junto con los datos del cliente.`
`SELECT nro_telefono, nro_cliente, nombre, apellido, direccion, activo`
`FROM e01_telefono NATURAL JOIN e01_cliente;`
	A diferencia de la consulta 5, se usó `NATURAL JOIN`, lo cual descarta a los clientes que no tienen un teléfono registrado.

`-- 10. Mostrar nombre y apellido de cada cliente junto con lo que gastó en total (con IVA incluido).`
`SELECT nombre, apellido, sum(coalesce(total_con_iva, 0)) as gasto_total_con_IVA`
`FROM e01_cliente c LEFT JOIN e01_factura f ON c.nro_cliente = f.nro_cliente`
`GROUP BY c.nro_cliente, nombre, apellido;`
	Para esta consulta, se unieron las tablas `e01_cliente` y `e01_factura`, mediante un `LEFT JOIN` (el cual permite que el caso en cual el cliente no tenía facturas a su nombre). Teniendo

	esto en cuenta, se debió también usar la función `coalesce` para lidiar con el caso que no se tenga una factura.

Capítulo III: MongoDB

Inicialmente, comenzamos pensando cómo íbamos a implementar la estructura de los documentos ya que, a diferencia de la primera parte (Capítulo I) no fue provisto por la cátedra. Decidimos incluir las tablas, previamente independientes, e01_detalle_factura y e01_telefonos como documentos embebidos dentro de factura y cliente respectivamente.

```
"cliente": {
"nro_cliente": …,
"nombre": …,
"apellido": …,
"direccion": …,
"activo": … ,
"telefonos": [
      {
          "codigo_area": …,
          "nro_telefono": …,
          "tipo": …
      },
      …
]
}
```

```
"factura": {
      "nro_factura": … ,
      "fecha": … ,
      "total_sin_iva": … ,
      "iva": … ,
      "total_con_iva": …,
      "nro_cliente": …,
      "detalles": [
      {
            "cantidad": … ,
            "codigo_producto": … ,
            "nro_item": …
      },
      …
]
}
```

Esto se debió a que entre sí, estas relaciones son 1:N y pertenecen unas a las otras. Es decir, cada cliente puede tener varios teléfonos relacionados a él y cada factura varios productos. Esta de-normalización de los datos nos pareció oportuna ya que nos permitió realizar algunas queries de manera más sencilla y además no creemos que provoque complicaciones ya que suponemos que ninguno de los dos documentos tendrá una gran cantidad de datos generando que los arrays mutables provoquen complicaciones.

`// 1. Obtener el teléfono y el número de cliente del cliente con nombre "Wanda" y apellido "Baker".`
 	
`db.cliente.findOne({"nombre":"Wanda", "apellido": "Baker"}, {"nro_cliente": 1, "telefonos": 1 })`
	En esta consulta, se usó `findOne` y se especificó las condiciones del nombre y apellido, y luego se seleccionó `nro_cliente` y `telefonos` como valores que se debían retornar.

`// 2. Seleccionar todos los clientes que tengan registrada al menos una factura.`
`db.cliente.find({nro_cliente: {$in: db.factura.distinct("nro_cliente")}}, {_id: 0}`
`)`
	Primero se hizo una subconsulta, en donde se obtuvo los `nro_cliente` presentes en `factura` y con el operador `$in`, se filtró solo aquellos clientes que tenían facturas.

`// 3. Seleccionar todos los clientes que no tengan registrada una factura.` `db.cliente.find({nro_cliente: {$nin: db.factura.distinct("nro_cliente")}}, {_id: 0})`	Se usó la misma estrategia que el ejercicio anterior, pero en este caso usando `$nin`.
`// 4. Seleccionar los productos que han sido facturados al menos 1 vez.` `db.producto.find({codigo_producto: {$in: db.factura.distinct("detalles.codigo_producto")}}, {_id: 0})`	Se hizo un subconsulta para obtener los productos facturados, pero en este caso cabe destacar que se usó los valores de un arreglo: `detalles.codigo_producto`
`// 5. Seleccionar los datos de los clientes junto con sus teléfonos` `db.cliente.find()`	Al tener tener en la colección de clientes un arreglo con los teléfonos, simplemente se deben obtener todos estos con: `find`.

`// 6. Devolver todos los clientes, con la cantidad de facturas que tienen registradas (admitir nulos en valores de Clientes)`
`db.cliente.aggregate([`
`{`
`$lookup: {`
`from: "factura",`
`localField: "nro_cliente",`
`foreignField: "nro_cliente",`
`as: "facturas"`
`}`
`},`
`{`
`$project: {`
`_id: 0,`
`nro_cliente: 1,`
`nombre: 1,`
`apellido: 1,`
`direccion: 1,`
`activo: 1,`
`cantidad_facturas: { $size: "$facturas" }`
`}`
`}`
`])`
	En esta consulta se debió usar `aggregate`, junto a `$lookup`, para unir las colecciones de `cliente` y `factura`. Esta operación es equivalente a `LEFT JOIN` en PSQL. Luego se seleccionaron los atributos necesarios con el operador `$project`. Cabe destacar que para conseguir la cantidad de facturas, se usó `$size: "$facturas"`. Importantísimo el `$`.

`// 7. Listar todas las Facturas que hayan sido compradas por el cliente de`
`-- nombre "Pandora" y apellido "Tate"`
`db.factura.find({ nro_cliente:`
`db.cliente.findOne({"nombre": "Pandora",`
`"apellido": "Tate"}, {"nro_cliente":`
`1`
`}).nro_cliente})`
	Se usó una subconsulta nuevamente, pero en este caso, cabe aclarar que se tuvo que obtener el atributo `nro_cliente` por fuera haciendo `.nro_cliente`.

`// 8. Listar todas las Facturas que contengan productos de la marca "In Faucibus Inc."`
`db.factura.aggregate([`
`{`
`$lookup: {`
`from: "producto",`
`localField: "detalles.codigo_producto",`
`foreignField: "codigo_producto",`
`as: "productInfo"`
`}`
`},`
`{`
`$match: {"productInfo.marca": "In Faucibus Inc."}`
`},`
`{`
`$project: {_id: 0, detalles: 1, fecha: 1, iva: 1, nro_cliente: 1, nro_factura: 1, total_con_iva: 1, total_sin_iva: 1}`
`}`
`])`
	Nuevamente se usó `$lookup` pero en este caso, de un subarreglo: `detalles`. Luego se usó la sentencia `$match`, para seleccionar solo el producto adecuado. Finalmente se seleccionaron los atributos para retornar usando `$project`.

`// 9. Mostrar cada teléfono junto con los datos del cliente.`
`db.cliente.find({"telefonos":{`
`$exists: true,`
`$ne: []`
`}`
`})`
	Al tener los teléfonos embebidos en `cliente`, se consulta si el arreglo "telefonos" existe y que no esté vacío.

`// 10. Mostrar nombre y apellido de cada cliente junto con lo que gastó en total (con IVA incluido).`
`db.cliente.aggregate([`
`{`
`$lookup: {`
`from: "factura",`
`localField: "nro_cliente",`
`foreignField: "nro_cliente",`
`as: "cliente_facturas"`
`}`
`},`
`{`
`$project: {`
`_id: 0,`
`nombre: 1,`
`apellido: 1,`
`total_con_iva: { $sum: "$cliente_facturas.total_con_iva" }`
`}`
`}`
`])`
	Se usó `$lookup` para unir las colecciones `cliente` y `factura` luego se proyectó. Se debe destacar nuevamente que al hacerse la suma total del gasto, se usó la función `$sum`, obteniendo los valores de la colección unida: `cliente_facturas`.

Capítulo IV: Migración

Este capítulo hablará de cómo realizamos la migración de las tablas de PostgreSQL a MongoDB. Se desarrollaron 2 métodos para hacer la migración:

1. Con un archivo estático (encontrado en “/migration/migration.mongodb”) que tiene las sentencias para insertar los valores.
2. Un programa que corre en NodeJS que hace la migración en el momento.

La opción 1. simplemente utiliza las sentencias propias de MongoDB para insertar todos los mismos valores iniciales que el archivo “ITBA_2023_esquema_facturacion.sql”. No se recomienda utilizar esta opción luego de haber modificado las tablas de PostgreSQL ya que los cambios se perderán y no se migrarán a MongoDB.

La opción 2. va generando los objetos JSON a partir de la toma de datos de las tablas de PSQL, inicialmente conectándose a la base de datos y luego mediante queries recuperando los datos. Primero se generan las estructuras para los clientes, luego las facturas y por último los productos. Luego, mediante sentencias de MongoDB se insertan las listas con las estructuras JSON a cada documento. Esta opción es la recomendada para utilizar ya que mantendrá los datos consistentes entre ambas bases de datos excepto que durante la ejecución de la migración se realicen cambios sobre las tablas de PSQL.

Para utilizar la opción (1) simplemente ejecutar el documento en una consola de mongodb y se irán introduciendo en las tablas los valores automáticamente. Para esta solución no se utilizó ninguna tecnología de migración y no es recomendada ya que no realiza una migración consistente de los datos de las tablas de PSQL.

Para utilizar la opción (2) se deben seguir los siguientes pasos:

1) Tener corriendo una base de datos Postgres y una MongoDB.
2) Ingresar por línea de comandos al directorio “/migration”
3) En el archivo “config.json” se deben completar las configuraciones necesarias de ambas bases de datos.
4) Estando en “/migration”, se debe ejecutar:
    a) `npm install`
    b) `node .`

Habiendo ejecutado estos comandos, se migran los datos automáticamente.

Para programar la migración, decidimos también utilizar node.js ya que cuando encaramos esta parte del enunciado ya teníamos realizada la API para la parte relacional del trabajo por lo cual ya estábamos familiarizados con la tecnología.

Capítulo V: API

Para la implementación de la API, al igual que la migración, utilizamos NodeJS y los archivos correspondientes se encuentran bajo el directorio “/api”.

Para ejecutar la API, se debe:

1) Tener corriendo la base de datos Postgres o la MongoDB.
2) Ingresar por línea de comandos al directorio “/api”
3) En el archivo “config.json” se deben completar las configuraciones necesarias de ambas bases de datos.
    a) En el campo `"database"`, se debe especificar qué base de datos se quiere usar. Para elegir Postgres debe tener como valor `"postgres"` y para MongoDB, debe ser `"mongodb"`.
4) Estando en “/api”, se debe ejecutar:
    a) `npm install`
    b) `node .`

    Una vez ejecutado `node .` la api estará corriendo sobre el puerto 8080 en localhost.

Se implementaron los llamados indicados por el enunciado y además consultar los datos de un solo cliente tanto en PostgreSQL como en MongoDB. En los llamados “POST”, “PUT” y “DELETE”, se chequea que no se infrinjan las reglas impuestas por el schema de cada valor de cada tabla, devolviendo un error adecuado en cada caso.

Endpoints	Descripción
/clientes	GET: recuperar todos los clientes POST: crea un nuevo cliente
/clientes/:id	GET: recupera un solo cliente PUT: actualiza un cliente DELETE: elimina un cliente
/productos	POST: crea un nuevo producto
/productos/:id	PUT: actualiza un producto

En el caso de los POST y PUT, se debe acompañar el pedido con una estructura del tipo JSON para almacenar. A continuación detalles de las estructuras tanto como para el endpoint /clientes y /productos. Se utiliza la misma estructura para PostgreSQL como para MongoDB

-	POST	PUT

/clientes	{
	"nro_cliente": ...,
	"nombre": "...",
	"apellido": "...",
	"direccion": "...",
	"activo": …,
	“telefonos”: [
	{
	“codigo_area”: …,
	“nro_telefono”: …,
	“tipo”:”...”
	},...
	]
	}
	*Observación: el elemento “telefonos” es opcional en PSQL
		-

/clientes/:id	-	{
		"nombre": "...",
		"apellido": "...",
		"direccion": "...",
		"activo": …,
		“telefonos”: [
		{
		“codigo_area”: …,
		“nro_telefono”: …,
		“tipo”:”...”
		},...
		]
		}
		*Observación: el elemento “telefonos” es opcional en PSQL

/productos	{
	"marca": "...",
	"nombre": "...",
	"descripcion": "...",
	"precio": …,
	"stock": …
	}
		-

/productos/:id	-	{
		"marca": "...",
		"nombre": "...",
		"descripcion": "...",
		"precio": ...,
		"stock": ...
		}

Capítulo VI: Conclusión

Durante el progreso de ese informe, se habló acerca de la implementación de dos bases de datos, una relacional (PostgreSQL) y una no relacional (MongoDB), una migración de datos de la primera a la segunda y una implementación de una API para realizar pedidos de datos sin necesidad de utilizar queries. Para el desarrollo de la API y el proceso de migración decidimos arbitrariamente la tecnología a utilizar lo cual nos permitió, además de interiorizar conceptos vistos en clase, aprender otras tecnologías elegidas como grupo. Finalmente, este proyecto nos ayudó a integrar los conceptos aprendidos, tanto en clase como por nuestra cuenta, en un entorno de uso real y no solo de estudio. Estamos contentos con el resultado que alcanzamos y nuestro desempeño como grupo.

Marcos, Marcos y Máximo. (M^3)