Universidad de San Carlos de Guatemala
Escuela de Ingeniería en Ciencias y Sistemas, Facultad de Ingeniería
Sistemas de Bases de Datos 1

Proyecto 1: Manual técnico

202111478 - José David Panaza Batres

Resumen

El proceso de migración de almacenamiento físico a entornos virtuales es un desafío común. Requiere una planificación cuidadosa y el diseño de una estructura eficiente para garantizar la persistencia y accesibilidad de la información.

Las pequeñas empresas suelen empezar con archivos de Excel para gestionar sus datos, pero a medida que crecen, la complejidad de manejar esos archivos aumenta. La falta de un modelado adecuado de bases de datos también puede conducir a sistemas ineficientes y operaciones tediosas.

Para superar estos desafíos, es esencial considerar soluciones que faciliten una transición efectiva hacia sistemas de información virtualizados y bases de datos optimizadas. Esto mejoraría la eficiencia y la productividad empresarial al garantizar la integridad y accesibilidad de los datos.

Palabras clave

migración, almacenamiento físico, entornos virtuales, planificación, eficiencia empresarial.

Abstract

The process of migrating from physical storage to virtual environments is a common challenge. It requires careful planning and designing an efficient structure to ensure persistence and accessibility of information.

Small businesses often start with Excel files to manage their data, but as they grow, the complexity of handling these files increases. The lack of proper database modeling can also lead to inefficient systems and tedious operations.

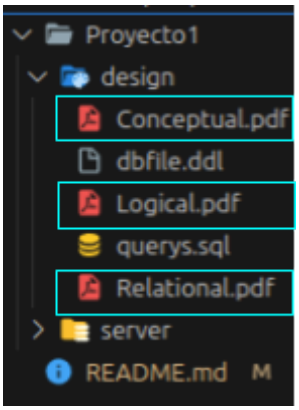
To overcome these challenges, it is essential to consider solutions that facilitate an effective transition to virtualized information systems and optimized databases. This would improve business efficiency and productivity by ensuring data integrity and accessibility.

Keywords:

migration, physical storage, virtual environments, planning, business efficiency.

Entregables

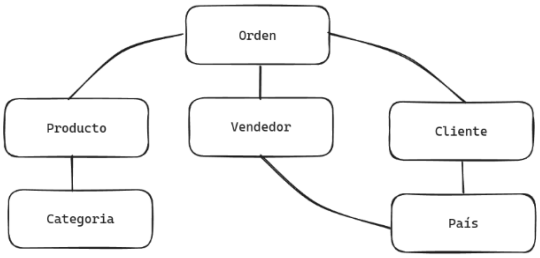
Ubicados en la carpeta “design”



Modelo conceptual

Se encuentra un pdf Conceptual, donde se aborda todo el diseño conceptual. Aquí se determinaron las entidades:

1. Orden
2. Producto
3. Cliente
4. Vendedor
5. País
6. Categoría



Modelo Lógico

En el modelo lógico ya se comienzan a ver las relaciones, las cuales son:

- CAT:PRO de uno a muchos
- PA:VEN de uno a muchos
- PA:CLI de uno a muchos
- PRO:ORD de uno a muchos
- VEN:ORD de uno a muchos
- CLI: ORD de uno a muchos

Esto quiere decir, por ejemplo, que en la entidad de orden pueden aparecer varios productos, pero que cada orden está asociada a solo un producto.

Al determinar las relaciones, es indispensable determinar las llaves primarias, para poder asociar estás mismas como llaves foráneas en las demás entidades. Las llaves primarias y foráneas para las entidades son:

1. Orden

Las llaves primarias son id_orden y linea_orden. Esto debido a que al tener una línea de productos se repiten los ids, pero la unión de estos dos genera un id único. Se pensó en utilizar una tabla maestro-detalle para mejor integridad, sin embargo, sólo la fecha y el id_cliente iban a ser únicos en está y eso no genera un mejor rendimiento ni integridad.

Las llaves foráneas son id_producto que hace referencia a la entidad producto, id_vendedor que hace referencia a la entidad de vendedor, id_cliente hace referencia a la entidad cliente.

2. Producto

La llave primaria es id_produto, y tiene una llave foránea de categoría

3. Cliente

La llave primaria es id_cliente y tiene una llava foránea de país

4. Vendedor

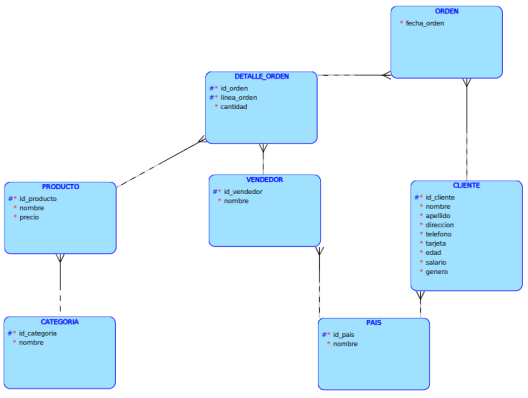
La llave primaria es id_vendedor y tiene una llave foránea de país

5. País

La llave primaria es id_país

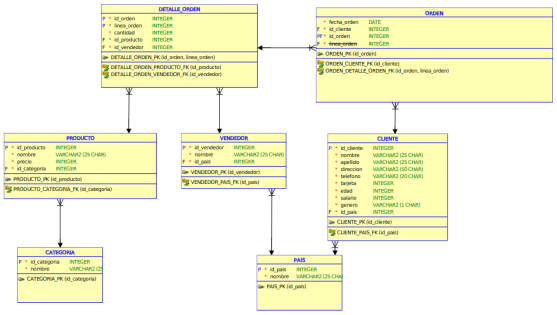
6. Categoría

La llave primaria es id_categoria



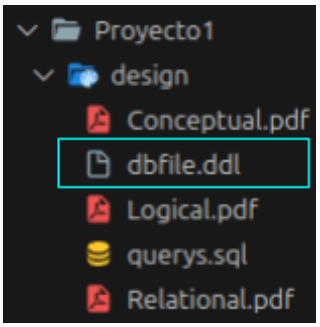
Modelo Relacional-Físico

Este se genera con la herramienta de data modeler, aquí se puede observar el tipo de dato a manejar, las llaves primarias y foráneas. Todas las llaves primarias las manejamos con un integer, esto para reducir el orden de complejidad de búsqueda. El precio del producto es el único dato que se toma como un valor decimal. Todo lo demás es varchar y la fecha se trabaja con data en un formato de DD-MM-YYYY.



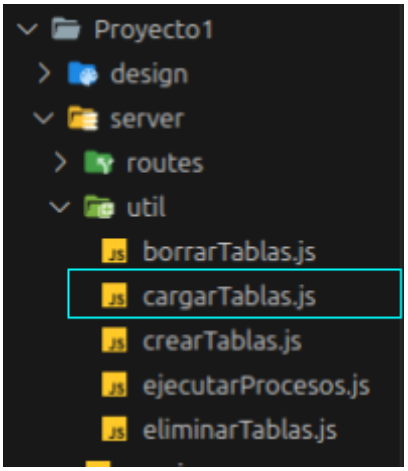
Script de la creación de la base de datos.

El script de la creación de la base de datos se generó con data modeler, es un archivo .ddl, el cual tiene la estructura, las llaves primarias y foráneas.



Carga del modelo

Se cuenta con 6 archivos CSV que cuentan con una data extensa, la manera de carga se pensó de varias maneras. Manejando el CSV, realizando una tabla temporal entre otras. Sin embargo, por términos de tiempo, se decidió manejar el CSV y al mismo tiempo ejecutar el comando sql. El archivo donde se cargan todos los modelos se encuentra aquí:



El CSV se lee con la librería csv-parser, un problema que surgió en el momento, es cómo el programa sabe cuál archivo leer primero y llenar las tablas, debido a que algunas tablas dependen de otras por la llaves foráneas y su integridad referencial, es por ello que se creó un array en un orden donde no se afecten las tablas:

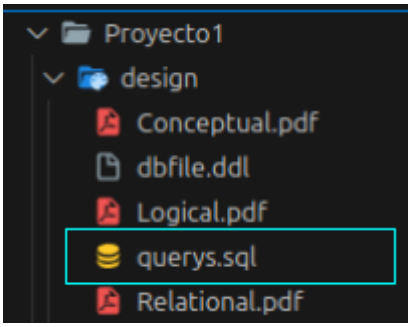
```
const tablasDeseadas = ['PAIS', 'CATEGORIA', 'CLIENTE', 'VENDEDOR', 'PRODUCTO', 'ORDEN'];
```

Después se leen los valores del csv para meterlos a las tablas correspondientes, realizando una sentencia según la tabla que se está recorriendo, y se insertan los valores. Después de recorrer todo el csv, se realiza un commit al final para ejecutar los cambios en la base de datos.

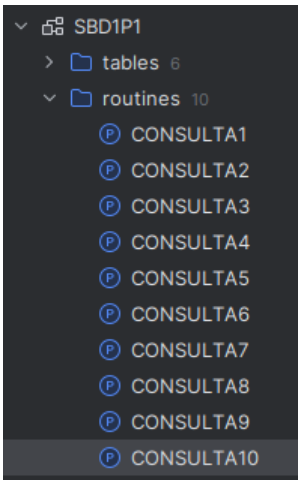
```
const stream = fs.createReadStream(tablaFilePath);
await new Promise((resolve, reject) => {
    stream.pipe(csv({ separator: ';' }))
        .on('data', async (row) => {
            try {
                //const keys = Object.keys(row).map(limpiarCadena);
                const values = Object.values(row).map(value => {
                    // Envuelve todos los valores en comillas simples
                    const cleanedValue = value.replace(/'/g, '').replace(/\//g, '-');
                    //const cleanedValue = value.replace(/'/g, "''");

                    //si viene un valor de fecha tipo 20-02-2024 o 1-03-2024, devolverlo de
                    if (cleanedValue.match(/\d{2}-\d{2}-\d{4}/)) {
                        return `TO_DATE('${cleanedValue}', 'DD-MM-YYYY')`;
                    } else if (cleanedValue.match(/\d{1}-\d{2}-\d{4}/)) {
                        //agregar el 0 al dia
                        return `TO_DATE('0${cleanedValue}', 'DD-MM-YYYY')`;
                    }
                    return `'${cleanedValue}'`;
                });
                const sql = `INSERT INTO "SBD1P1"."${tabla}" VALUES (${values.join(',')})`;
                await connection.execute(sql);
```

Consultas creadas

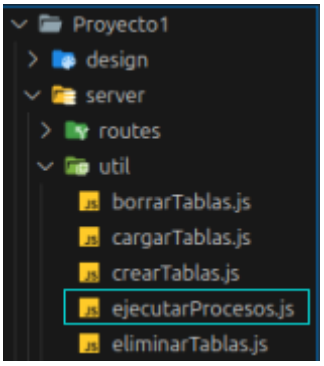
Estas están en un archivo .sql, en este archivo sql están todas las consultas que se realizan, desde la 1 a la 10.



Sin embargo, las consultas no se realizaron desde la API. Se decidió utilizar un método seguro para aprender a utilizar procedures-routines en bases de datos. Lo que nos ayuda a tener un mejor control sobre los querys, debido a que solo mandamos a llamar los procesos, recopilamos el output, lo transformamos y lo devolvemos.



Estas se llaman a través del archivo ejecutarProcesos.js, donde se manda a ejecutar una query que dice begin y el nombre del proceso. Después se obtiene el output del dbms. Se recopila cada línea, se concatena y se transforma al final en un json para poder desplegarla.



Código fuente

El código fuente se basa en una API desarrollada en Nodejs, la cual cuenta con una App.js, donde se encuentra el servidor desarrollado con express. Esta cuenta con 3 rutas principales:

- Conexión
- Procesos
- Consultas

Estas 3 rutas hacen referencia a 3 archivos situados en la carpeta routes. Estos archivo proveen directamente los endpoint:

```
// Usar las rutas de conexion
app.use('/', conexionRouter);
app.use('/',procesosdb);
app.use('/',consultasdb);
```

La otra carpeta llamada util, están todas las funciones que mandan a llamar en los endpoint. Estas son:

- borrartablas: esta se llama en el endpoint borrarinfodb, la cual borra la información de todas las tablas.
- cargarTablas: esta se llama en el endpoint cargarmodelo, carga todo lo que está en los csv.
- crearTablas: este crea la base de datos con SQL, en el endpoint crearmodelo
- ejecutarProcesos: esta función manda a llamar los procedures dentro de la base de datos y devuelve la salida en formato json.
- eliminarTablas: está en el endpoint de eliminarmodelo y elimina la base de datos.

Tenemos un archivo config.js el cual tiene los datos del usuario de la base de datos:

```
// config.js

module.exports = {
    dbConfig: {
      user: "sbd1p1",
      password: "sbd1p1",
      connectString: "localhost:1521/FREE"
    }
  };
```

Este se llama en todos los archivos donde se hace uso de la librería de oracle.

```
const oracledb = require('oracledb');
const { dbConfig } = require('../config');
```

En el archivo de conexión prueba si existe conexión con la base de datos.

```
const express = require('express');
const oracledb = require('oracledb');
const { dbConfig } = require('../config');
const router = express.Router();

// Endoint para probar la conexión a la base de datos
router.get('/conexion', async (req, res) => {
    try{
        //Realizar la conexión a la base de datos
        const connection = await oracledb.getConnection(dbConfig);

        //Realizar una consulta de prueba
        const result = await connection.execute(
            `SELECT 'Conexion exitosa' AS message FROM dual`
        );

        //Cerrar la conexión
        await connection.close();

        //Enviar la respuesta
        res.status(200).json({conexion: result.rows[0][0]});

    } catch (error){
        console.error('Error en la conexión a la base de datos', error);
        res.status(500).json({error: 'Error en la conexión a la base de datos'
    }
});

module.exports = router;
```

En el archivo consultas, se tienen todos los endpoints de consultas y se llama la función para llamar los procesos según la consulta.

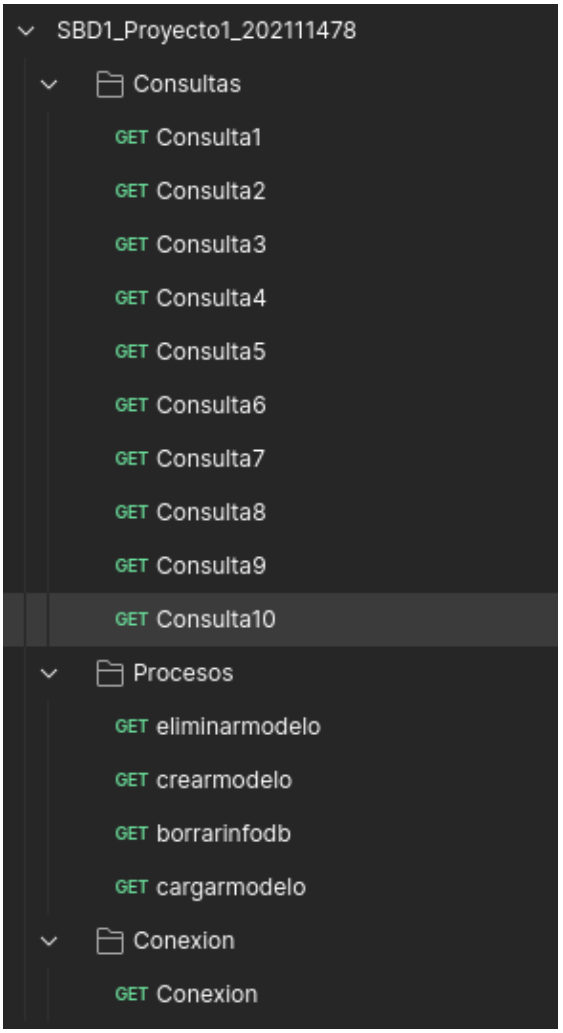
```
// ============================================================ CONSULTA 1

router.get('/consulta1', async (req, res) => {
    try{
        // Llamar la funcion para ejecutar el proceso
        const resultado = await ejecutarProcesos('CONSULTA1');

        if (resultado.success){
            return res.status(200).json(resultado.data);
        }
        return res.status(500).json({error: 'Error en la consulta 1'});

    } catch (error){
        console.error('Error en la consulta 1', error);
        return res.status(500).json({error: 'Error en la consulta 1'});
    }
});
```

En el archivo procesosdb, se tienen todos los procesos que tienen que ver con la base de datos. Como creación, eliminación, cargar la data y borrarla. Aquí de igual manera, se llaman las funciones a utilizar.

```
// ============================================================ ELIMINAR MO

router.get('/eliminarmodelo', async (req, res) => {
    try{
        // Llamar a la funcion para eliminar las tablas
        const resultado = await eliminarTablas();

        if (resultado.success){
            res.status(200).json({message: resultado.message});
        } else{
            res.status(500).json({error: resultado.message});
        }

    } catch (error){
        res.status(500).json({error: 'Error eliminando las tablas'});
    }
});
```

Verificación

Para poder verificar los endpoints de la API, se decidió utilizar postman, donde están todas las consultas guardadas con su debido resultado.



Manejo de la Base de datos

- Contenedor de docker sobre Oracle
- container-registry.oracle.com/database/free:latest
- data modeler
- oracle sqldeveloper
- datagrid