BASE DE DATOS

Universidad Politécnica de San Luis Potosí

Proyecto: Sistema de optimización de corte de materia prima en PostgreSQL - Manual Técnico

Integrantes:

179761 Domínguez García Coral Jazmín
182934 Gámez Contreras Bryan Yareth
182451 García Camacho Fernanda Daniela
181914 Quintanilla Ramírez Yael
182085 Ornelas Martínez Eric André
182829 Martínez Vázquez Uriel

Docente: Jesús Alberto Revilla Silva

24 de noviembre de 2025

Contenido

Manual Técnico — Sistema de Optimización de Corte de Materia Prima en PostgreSQL

Introducción Técnica

El Sistema de Optimización de Corte de Materia Prima tiene como objetivo mejorar la utilización de láminas de materiales laminados (madera, metal, tela, etc.), optimizando la ubicación de piezas dentro de la materia prima disponible.

El sistema está implementado íntegramente en PostgreSQL, utilizando:

- Tablas relacionales para la gestión de usuarios, roles, productos, piezas, geometrías y eventos.
- Procedimientos almacenados y funciones para automatizar el alta de materia prima, productos y cálculo de aprovechamiento.
- Funciones CRUD para la gestión de usuarios.
- JSON para el registro de eventos y configuraciones.
- Seguridad mediante roles de base de datos (`admin` y `operador`).
- Automatización de pruebas mediante Python (`pytest`) y un workflow en GitHub Actions.

La arquitectura está diseñada para ser modular y escalable, permitiendo futuras mejoras como la integración con interfaces de usuario gráficas.

Arquitectura del Sistema

Componente	**Función**
Tablas	Almacenan usuarios, roles, productos, piezas, geometrías, materia prima, eventos y cortes planificados.
Procedimientos almacenados	Automatizan el alta de productos, piezas y materia prima, así como la rotación y posicionamiento de figuras.
Funciones	Calculan áreas, porcentaje de utilización y permiten operaciones CRUD de usuarios.
Triggers	Validan parámetros automáticamente y recalculan el aprovechamiento tras cambios.
JSON	Permite registrar eventos complejos de rotación y posición de piezas.
GitHub Actions Workflow	Automatiza la configuración de la base de datos y pruebas unitarias.

Flujo general de datos:

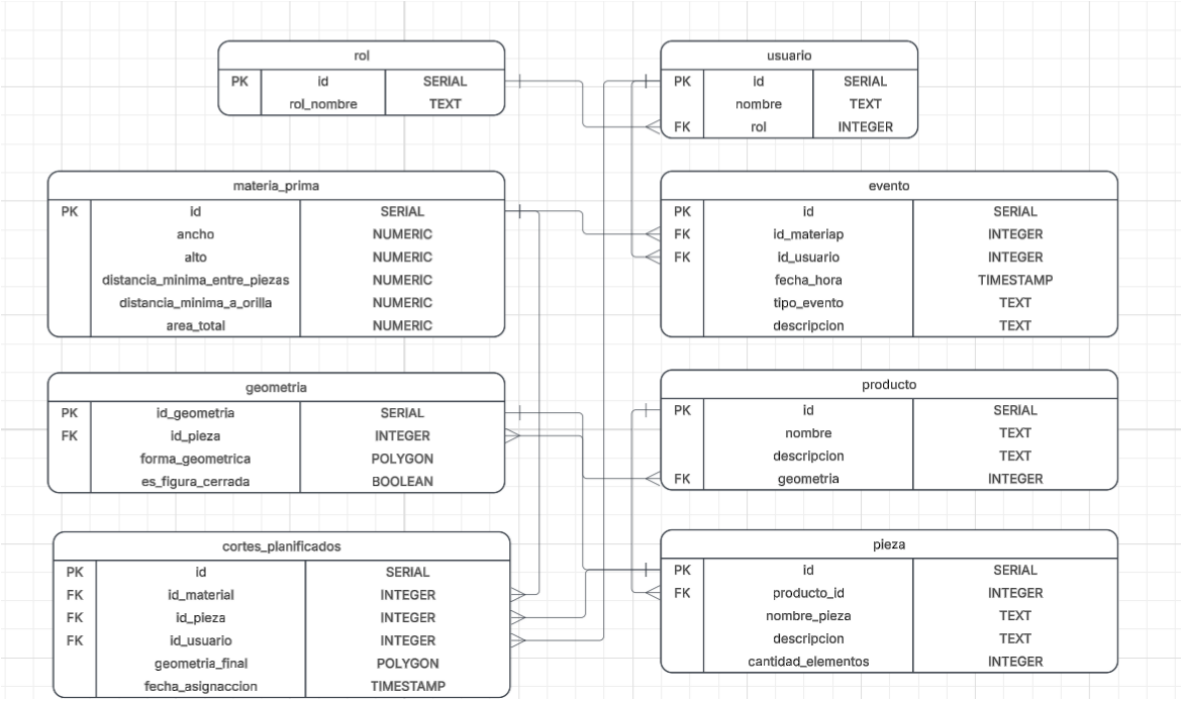
- Se crea materia prima y productos mediante procedimientos.
- Se generan piezas y geometrías asociadas.
- Se aplica rotación y posicionamiento de figuras mediante sp_rotar_posicionar_figuras.
- Se registran eventos JSON.
- Se calcula el porcentaje de utilización de la materia prima.

Definición de Tablas

1. **rol**
    - Contiene los roles que pueden tener los usuarios, como *administrador* y *operador*.
    - Sirve para asignar permisos y controlar el acceso a distintas funcionalidades.
2. **usuario**
    - Guarda la información de los usuarios del sistema.
    - Cada usuario tiene un nombre y un rol asociado.
    - Relacionada con rol mediante una clave foránea.
3. **materia_prima**
    - Almacena los bloques de material disponibles para corte.
    - Cada registro incluye dimensiones, distancias mínimas entre piezas y a las orillas, y el área total calculada automáticamente.
4. **producto**
    - Representa los productos finales que se desean fabricar.
    - Contiene nombre, descripción y una geometría que sirve como referencia para el diseño de las piezas.
5. **pieza**
    - Cada producto puede tener varias piezas.
    - Contiene información de la pieza: nombre, descripción, cantidad de elementos, y referencia al producto al que pertenece.
6. **geometrias**
    - Guarda las formas geométricas de cada pieza.
    - Usamos el tipo POLYGON para poder calcular áreas y ubicar piezas sobre la materia prima.

- Si se elimina la pieza, su geometría también se elimina automáticamente (cascade).

7. **cortes_planificados**
   - Registra la posición final de cada pieza sobre la materia prima.
   - Incluye la geometría final de la pieza, referenciando tanto la pieza como la materia prima.
8. **evento**
   - Registra todas las acciones que modifican la colocación de piezas o ajustes geométricos.
   - Contiene información de usuario, fecha, tipo de evento y detalles en formato JSONB.



Relaciones clave

rol (1) —< usuario >—(n)

producto (1) —< pieza >—(n)

pieza (1) —< geometrias >—(1..n)

materia_prima (1) —< cortes_planificados >—(n)

usuario (1) —< evento >—(n)

cortes_planificados (1) —< evento >—(1)

Funciones y Procedimientos Clave

1. Funciones

1.1 fn_calcular_area_manual(p_geometria POLYGON)

Objetivo: Calcular el área de cualquier polígono definido como POLYGON.

Detalles del código:

- Convierte la geometría a texto y la limpia de paréntesis.
- Separa los puntos del polígono en un array (v_puntos).
- Aplica la **fórmula de shoelace** para calcular el área de manera iterativa.
- Retorna un valor decimal con la superficie total de la figura.

Uso: Se emplea en la validación de cortes y en triggers de auditoría (t_auditar_corte).

Algunas partes importantes son:

```
BEGIN
    -- 1. Convertir polígono a texto limpio: '((0,0),(10,0))' -> '0,0|10,0'
    -- Quitamos paréntesis dobles y simples
    v_texto := replace(p_geometria::text, '((', '');
    v_texto := replace(v_texto, '))', '');
    v_texto := replace(v_texto, '),(', '|');

    -- 2. Separar en array de puntos
    v_puntos := string_to_array(v_texto, '|');
    n := array_length(v_puntos, 1);
```

Explicación: Convierte la geometría POLYGON a un texto limpio, separando los puntos en un array v_puntos. Esto permite iterar sobre los vértices para aplicar la fórmula de shoelace.

```
25          FOR i IN 1..n LOOP
26              -- Extraer X e Y del punto actual
27              v_coord := string_to_array(v_puntos[i], ',');
28              x1 := CAST(v_coord[1] AS DECIMAL);
29              y1 := CAST(v_coord[2] AS DECIMAL);
30
31              -- Extraer X e Y del siguiente punto (o del primero si es el último)
32              IF i < n THEN
33                  v_coord := string_to_array(v_puntos[i+1], ',');
34              ELSE
35                  v_coord := string_to_array(v_puntos[1], ',');
36              END IF;
37              x2 := CAST(v_coord[1] AS DECIMAL);
38              y2 := CAST(v_coord[2] AS DECIMAL);
39
40              -- Sumas cruzadas
41              v_suma1 := v_suma1 + (x1 * y2);
42              v_suma2 := v_suma2 + (x2 * y1);
43          END LOOP;
44
45          -- 4. El área es la mitad de la diferencia absoluta
46          RETURN ABS(v_suma1 - v_suma2) / 2.0;
```

Explicación: Se calcula el área del polígono usando la **fórmula de shoelace** y retorna el resultado como decimal.

1.2 fn_calcular_utilizacion(p_id_materia INT)

Objetivo: Calcular el porcentaje de utilización de una lámina de materia prima.

Variables importantes:

- v_area_total_materia: área total disponible de la lámina.
- v_area_ocupada: suma de las áreas de las piezas colocadas (fn_calcular_area_manual).
- v_porcentaje: resultado final de la utilización.

Validaciones:

- Verifica que la materia prima exista.
- Maneja el caso donde no haya piezas (retorna 0% en lugar de NULL).

Salida: Porcentaje de utilización en formato decimal con 2 decimales.

Algunas partes importantes son:

```
SELECT area_total INTO v_area_total_materia
FROM materia_prima
WHERE id = p_id_materia;

-- Validación: Si la materia prima no existe
IF v_area_total_materia IS NULL THEN
    RAISE EXCEPTION 'La materia prima con ID % no existe', p_id_materia;
END IF;

-- 2. Sumar el área de todas las piezas colocadas (usando función nativa AREA de Postgres)
-- COALESCE asegura que si no hay piezas, devuelva 0 en lugar de NULL
SELECT COALESCE(SUM(fn_calcular_area_manual(geometria_final)), 0.0)
INTO v_area_ocupada
FROM cortes_planificados
WHERE id_materia = p_id_materia;
```

Explicación: Obtiene el área total de la materia prima y la suma de todas las áreas de las piezas colocadas (cortes_planificados).
COALESCE asegura que si no hay piezas colocadas, el área ocupada sea 0.

```
-- 3. Cálculo del porcentaje
IF v_area_total_materia > 0 THEN
    v_porcentaje := (v_area_ocupada / v_area_total_materia) * 100;
ELSE
    v_porcentaje := 0;
END IF;
```

Explicación: Calcula el porcentaje de aprovechamiento de la materia prima.

2. Procedimientos

2.1 alta_materia_prima(ancho, alto, dist_min_piezas, dist_min_orilla)

Objetivo: Crear un nuevo bloque de materia prima con número de parte automático.

Detalles del código:

- Usa la **secuencia num_materiap_aum** para generar un identificador incremental.
- Genera el num_parte con formato 'NUM-0001', 'NUM-0002', etc.
- Inserta la materia prima con dimensiones y distancias mínimas.
- Calcula automáticamente area_total a través de la columna generada en la tabla materia_prima.

Algunas partes importantes son:

```
12        num_nuevo := nextval('num_materiap_aum');
13
14        num_completo := 'NUM-' || LPAD(num_nuevo::TEXT, 4, '0');
15
16        INSERT INTO materia_prima (num_parte, ancho, alto,distancia_minima_entre_piezas,
17            distancia_minima_a_orilla)
18        VALUES (num_completo, ancho, alto,dist_min_piezas, dist_min_orilla);
```

Explicación: Usa una secuencia automática para generar el número de parte (NUM-0001, NUM-0002, ...).
Inserta la materia prima con sus dimensiones y distancias mínimas.
La columna area_total se calcula automáticamente al insertar el registro.

2.2 alta_producto(nombre, descripcion, geometria, piezas JSON)

Objetivo: Crear un producto y sus piezas asociadas con geometrías.

Flujo interno:

- Inserta el producto en la tabla producto.
- Recorre el array JSONB piezas para crear cada pieza:
    - Inserta la pieza en pieza.
    - Inserta la geometría de la pieza en geometrias.

Punto clave: Permite **alta masiva de piezas** usando JSONB, lo que evita múltiples inserts individuales.

Algunas de las partes importantes son:

```
37        FOR pieza_arreglo IN SELECT * FROM jsonb_array_elements(piezas)
38        LOOP
39            --Insertar piezas
40            INSERT INTO pieza (producto_id, nombre_pieza, descripcion, cantidad_elementos)
41            VALUES ( producto_id, pieza_arreglo->>'nombre_pieza', pieza_arreglo->>'descripcion',
42            (pieza_arreglo->>'cantidad_elementos')::INT)
43            RETURNING id INTO pieza_id;
44
45            --Insertar geometría
46            INSERT INTO geometrias (id_pieza, forma_geometrica)
47            VALUES (pieza_id, (pieza_arreglo->>'geometria')::POLYGON);
48        END LOOP;
```

Explicación: Recorre el JSONB piezas y crea cada pieza y su geometría.
Esto permite crear **productos complejos con múltiples piezas en un solo procedimiento**, evitando inserciones individuales repetitivas.

2.3 fn_validar_colocacion_nativo(p_id_geometria, p_nueva_geometria_polygon, p_id_materia_prima)

Objetivo: Validar si una pieza se puede colocar sobre la materia prima sin violar restricciones.

Validaciones internas:

- Obtiene dimensiones y distancias mínimas de la materia prima.
- Calcula un **BOX límite** considerando el margen de orilla.
- Verifica que la nueva geometría esté dentro del BOX límite (@> operador).
- Revisa posibles **solapamientos** con otras piezas planificadas usando && (aproximación de colisión de cajas).

Salida: TRUE si es válida, FALSE si no.

Algunas de las partes importantes son:

```
79      -- 2. Validar Límites de la Materia Prima (MP) y Margen de Orilla
80      v_limite_caja := BOX(
81          POINT(v_min_distancia_orilla, v_min_distancia_orilla),
82          POINT(v_mp_ancho - v_min_distancia_orilla, v_mp_alto - v_min_distancia_orilla)
83      );
84
85      -- Comprobación: La caja límite debe contener a la caja de la pieza (@> operador nativo)
86      IF NOT (v_limite_caja @> v_nueva_caja) THEN
87          RETURN FALSE;
88      END IF;
```

Explicación: Crea un BOX límite que representa la zona segura dentro de la materia prima. Usa el operador @> para verificar que la geometría de la pieza esté completamente dentro de este límite.

```
93          SELECT COUNT(cp.id)
94          INTO v_solapamiento_conteo
95          FROM cortes_planificados cp
96          WHERE cp.id_materia = p_id_materia_prima
97            AND cp.geometria_final::BOX && v_nueva_caja;
98
99          -- Si hay solapamiento, asumimos que también vi
100         IF v_solapamiento_conteo > 0 THEN
101             RETURN FALSE;
102         END IF;
```

Explicación: Comprueba si la nueva pieza se solapa con otras ya colocadas usando el operador && (intersección de cajas).

5.4 sp_rotar_posicionar_figuras(p_id_geometria, p_id_materia_prima, p_angulo_rot, p_pos_x, p_pos_y, p_evento_json, p_id_usuario)

Objetivo: Rotar y posicionar piezas sobre la materia prima de forma segura.

Detalles técnicos importantes:

- Permite rotaciones en múltiplos de 90° únicamente.
- Calcula dimensiones de la figura usando width() y height() del BOX convertido.
- Genera la **geometría final** (POLYGON) con rotación y traslación.
- Llama a fn_validar_colocacion_nativo para asegurar que la colocación sea válida.
- Inserta el corte en cortes_planificados solo si es válido.
- Registra el evento completo en JSONB, incluyendo:
    - ID de geometría base
    - Validez
    - Ángulo de rotación
    - Posición X/Y
    - ID del corte generado

Beneficio: Este procedimiento integra cálculo geométrico, validación de restricciones y auditoría, todo en una sola operación segura.

Algunas de las partes importantes son:

```
IF MOD(p_angulo_rot / 90, 2) = 0 THEN
    -- Rotación 0, 180, 360: Ejes paralelos (Ancho sigue siendo Ancho, Alto sigue siendo
    v_geometria_final := format('((%s,%s), (%s,%s), (%s,%s), (%s,%s), (%s,%s))',
        p_pos_x, p_pos_y,
        p_pos_x + v_ancho_base, p_pos_y,
        p_pos_x + v_ancho_base, p_pos_y + v_alto_base,
        p_pos_x, p_pos_y + v_alto_base,
        p_pos_x, p_pos_y
    )::POLYGON;
```

Explicación: Calcula la geometría final de la pieza considerando rotación (0°, 180°) y traslación a la posición deseada.

```
184        v_es_valido := fn_validar_colocacion_nativo(
185            p_id_geometria,
186            v_geometria_final,
187            p_id_materia_prima
188        );
189
190        IF NOT v_es_valido THEN
191            RAISE NOTICE 'Advertencia: Colocación no válida detectada.';
192        END IF;
193
194    -- 5. Insertar la tabla de cortes_planificados (SOLO SI ES VÁLIDO)
195    IF v_es_valido THEN
196
197        -- Insertar un nuevo corte planificado
198        INSERT INTO cortes_planificados (
199            id_materia,
200            id_pieza,
201            id_usuario,
202            geometria_final
203        )
204        VALUES (
205            p_id_materia_prima,
206            v_id_pieza_maestra,
207            p_id_usuario,
208            v_geometria_final
209        )
210        RETURNING id INTO v_corte_id; -- Usamos RETURNING para capturar el ID del corte
```

Explicación: Valida que la colocación sea correcta antes de insertar el corte.
Si no es válido, no se inserta y se genera un aviso en la tabla evento en formato JSON.

Triggers

Trigger de validación de cortes: tr_validar_antes_de_corte

Objetivo:
Evitar que se inserten cortes que violen las restricciones de colocación de piezas sobre la materia prima, como límites de lámina o solapamiento con otras piezas.

Detalles técnicos importantes:

- Se ejecuta **antes de insertar un registro** en la tabla cortes_planificados.
- Utiliza la función fn_validar_colocacion_nativo para:
    - Verificar que la pieza esté dentro de los límites de la lámina.
    - Comprobar que no haya colisiones con otras piezas ya planificadas.
- Si la validación falla, el trigger cancela la inserción y muestra un mensaje de error claro.

Algunas partes importantes son:

```
6         v_es_valido := fn_validar_colocacion_nativo(
7             NEW.id_pieza,
8             NEW.geometria_final,
9             NEW.id_materia
10        );
11
12        IF NOT v_es_valido THEN
13            RAISE EXCEPTION 'ALERTA DE TRIGGER: El corte viola las reglas de colisión o lí
14            USING HINT = 'Verifique que la pieza no se salga de la lámina ni se encime con
15        END IF;
```

Explicación: Este trigger se ejecuta **antes de insertar un corte planificado**.
Llama a la función fn_validar_colocacion_nativo para comprobar límites y solapamientos.
Si la validación falla, **se cancela la inserción** con un mensaje de alerta, evitando errores de colocación.

```
21    CREATE TRIGGER t_validar_corte_insert
22    BEFORE INSERT ON cortes_planificados
23    FOR EACH ROW
24    EXECUTE FUNCTION tr_validar_antes_de_corte();
```

Explicación: Vincula la función a la tabla cortes_planificados para que cada inserción se valide automáticamente.

Trigger de auditoría de cortes: tr_auditoria_cortes

Objetivo:
Registrar automáticamente eventos de cortes planificados para **llevar un historial completo**, incluyendo información clave como área de corte y la pieza involucrada.

Detalles técnicos importantes:

- Se ejecuta después de insertar un registro en cortes_planificados.
- Genera un registro en la tabla evento con:
    - ID de materia prima y usuario.
    - Fecha y hora.
    - Tipo de evento (Corte Planificado).
    - Detalles en JSONB (id_pieza y area_corte).

Algunas partes importantes son:

```
31        INSERT INTO evento (id_materiap, id_usuario, fecha_hora, tipo_evento, descripcion)
32        VALUES (
33            NEW.id_materia,
34            NEW.id_usuario,
35            NOW(),
36            'Corte Planificado',
37            jsonb_build_object('id_pieza', NEW.id_pieza, 'area_corte',
38        fn_calcular_area_manual(NEW.geometria_final))
39        );
40        RETURN NEW;
41    END;
```

Explicación: Este trigger registra automáticamente un evento cuando se inserta un corte. Usa JSONB para almacenar información como el ID de la pieza y su área, permitiendo historial detallado de acciones sin afectar el diseño de las tablas.

```
44    CREATE TRIGGER t_auditar_corte
45    AFTER INSERT ON cortes_planificados
46    FOR EACH ROW
47    EXECUTE FUNCTION tr_auditoria_cortes();
```

Explicación: Se ejecuta después de la inserción, asegurando que el corte se registrará solo si se ha insertado correctamente.

Uso de JSON para eventos

Objetivo:
Permitir registrar **información compleja y dinámica** sobre operaciones (rotación, traslación, validación de piezas, etc.) sin modificar la estructura de las tablas.

Detalles técnicos importantes:

- JSONB se utiliza en la columna descripcion de la tabla evento.
- Permite concatenar datos de entrada con información generada automáticamente.
- Facilita auditoría, reporting y trazabilidad.

Ejemplo en procedimiento sp_rotar:posicionar_figuras:

```
        -- 6. Registrar evento
        INSERT INTO evento (id_materiap, id_usuario, fecha_hora, tipo_evento, descripcion)
        VALUES (
            p_id_materia_prima,
            p_id_usuario,
            NOW(),
            'Ajuste Geométrico',
            p_evento_json || jsonb_build_object(
                'id_geometria_base', p_id_geometria,
                'validez', v_es_valido,
                'angulo_deg', p_angulo_rot,
                'pos_x', p_pos_x,
                'pos_y', p_pos_y,
                'id_corte',v_corte_id
            )
        );
    END;
```

Explicación:

- p_evento_json: JSON de entrada con información adicional proporcionada por el usuario o sistema.
- jsonb_build_object: Construye un objeto JSON con información generada por el procedimiento.
- ||: Operador de concatenación JSONB para unir ambos objetos en un solo registro.
- Resultado: cada evento queda documentado de manera estructurada y completa, listo para auditoría.

Workflow de Automatización

Explicación resumida:
El workflow ejecuta automáticamente pruebas unitarias cada vez que se hace push o pull request en la rama main.
Incluye pasos para:

1. Levantar un contenedor PostgreSQL.
2. Configurar la base de datos y usuarios.
3. Ejecutar scripts SQL para crear tablas, insertar datos y cargar funciones/procedimientos.
4. Ejecutar tests con pytest para validar el correcto funcionamiento de las funciones y procedimientos.

Beneficio:
Garantiza que cualquier cambio en el código SQL se pruebe automáticamente antes de integrarse, asegurando consistencia y seguridad en la base de datos.

Pruebas Unitarias (Python / pytest)

Objetivo:
Garantizar la correcta operación de todas las funciones, procedimientos, triggers y la lógica de negocio del sistema sin afectar la base de datos de producción.

Detalles técnicos importantes:

- Se usa pytest como framework de pruebas.
- La conexión a la base de datos se realiza con psycopg2, usando RealDictCursor para obtener resultados como diccionarios.
- Cada prueba se ejecuta dentro de una transacción y se hace rollback al final para mantener la base de datos limpia.
- Se prueban:
    1. CRUD de usuarios.
    2. Alta de materia prima y productos con JSON.
    3. Lógica de posicionamiento y validación de cortes.
    4. Triggers que bloquean solapamientos.
    5. Cálculo de porcentaje de utilización.
    6. Seguridad y permisos de usuarios.

Pruebas CRUD de usuarios

```python
def test_crud_usuarios(cursor):
    """Prueba la creación, modificación y baja de usuarios."""
    # 1. Alta
    cursor.execute("SELECT alta_usuario(%s, %s)", ('Admin', 1))
    user_id = cursor.fetchone()['alta_usuario']
    assert user_id is not None
    # 2. Verificar existencia
    cursor.execute("SELECT * FROM usuario WHERE id = %s", (user_id,))
    user = cursor.fetchone()
    assert user['nombre'] == 'Admin'
    # 3. Modificación
    cursor.execute("SELECT modificar_usuario(%s, %s, %s)", (user_id, 'Operador', 2))
    cursor.execute("SELECT * FROM usuario WHERE id = %s", (user_id,))
    user = cursor.fetchone()
    assert user['nombre'] == 'Operador'
    assert user['rol'] == 2
    # 4. Baja
    cursor.execute("SELECT baja_usuario(%s)", (user_id,))
    cursor.execute("SELECT * FROM usuario WHERE id = %s", (user_id,))
    assert cursor.fetchone() is None

```

Explicación:

- Verifica alta, lectura, actualización y baja de usuarios.
- Asegura que las funciones CRUD funcionen según lo esperado.

Pruebas de alta de materia prima y productos

```python
def test_alta_materia_y_producto(cursor):
    """Prueba los Procedures de alta con JSON."""

    # 1. Alta Materia Prima
    cursor.execute("CALL alta_materia_prima(%s, %s, %s, %s)", (200, 300, 5, 10))
    cursor.execute("SELECT * FROM materia_prima WHERE ancho = 200 AND alto = 300")
    materia = cursor.fetchone()
    assert materia is not None

    # 2. Alta Producto
    piezas_json = [
        {
            "nombre_pieza": "RelojTest",
            "descripcion": "Mecánico",
            "cantidad_elementos": 1,
            "geometria": "((0,0),(4,0),(4,2),(0,2))"
        }
    ]

    cursor.execute(
        "CALL alta_producto(%s, %s, box(point(0,0), point(10,5)), %s)",
        ('Reloj', 'Reloj Mecanico', json.dumps(piezas_json))
    )
    # Verificar que se crearon la pieza y la geometría
    cursor.execute(
        "SELECT p.id, g.id_geometria FROM pieza p JOIN geometrias g ON p.id = g.id_pieza "
        "JOIN producto pr ON p.producto_id = pr.id WHERE pr.nombre = 'Reloj'"
    )
    resultado = cursor.fetchone()
    assert resultado is not None, "No se generaron las piezas o geometrías correctamente"
```

Explicación:

- Valida que los procedimientos de alta inserten correctamente la materia prima y sus productos asociados con piezas y geometrías.

Pruebas de lógica de corte y triggers

```python
def test_logica_corte_valido(cursor):
    """Prueba de un corte que si debe entrar."""
    cursor.execute("CALL alta_materia_prima(100, 100, 0, 0)")
    cursor.execute("SELECT id FROM materia_prima ORDER BY id DESC LIMIT 1")
    id_mp = cursor.fetchone()['id']
    cursor.execute("CALL alta_producto('Valido', 'Producto Valido', box(point(0,0), point(10,1
                   (json.dumps([{"nombre_pieza": "P1", "descripcion": "D", "cantidad_elementos
    cursor.execute("SELECT g.id_geometria FROM geometrias g JOIN pieza p ON g.id_pieza = p.id
    id_geo = cursor.fetchone()['id_geometria']

    # Crear usuario
    cursor.execute("INSERT INTO usuario (nombre, rol) VALUES ('Tester', 1) RETURNING id")
    id_user = cursor.fetchone()['id']
    # --- EJECUCIÓN DEL TEST ---
    cursor.execute(
        "CALL sp_rotar_posicionar_figuras(%s, %s, %s, %s, %s, %s, %s)",
        (id_geo, id_mp, 0, 10, 10, json.dumps({'test': 'ok'}), id_user)
    )
    # Validación
    cursor.execute("SELECT COUNT(*) as total FROM cortes_planificados WHERE id_materia = %s",
    assert cursor.fetchone()['total'] == 1
```

```python
def test_logica_colision_trigger(cursor):
    """Prueba que el TRIGGER bloquee una colisión (Insert directo)."""
    # 1. SETUP (Igual que antes)
    cursor.execute("CALL alta_materia_prima(100, 100, 0, 0)")
    cursor.execute("SELECT id FROM materia_prima ORDER BY id DESC LIMIT 1")
    id_mp = cursor.fetchone()['id']
    cursor.execute("CALL alta_producto('Colision', 'D', box(point(0,0), point(10,10)), %s)",
                   (json.dumps([{"nombre_pieza": "P1", "descripcion": "D", "cantidad_elementos
    # Necesitamos el ID de la PIEZA para el insert directo
    cursor.execute("""
        SELECT g.id_geometria, p.id as id_pieza
        FROM geometrias g
        JOIN pieza p ON g.id_pieza = p.id
        JOIN producto pr ON p.producto_id = pr.id
        WHERE pr.nombre = 'Colision' LIMIT 1
    """)
    datos = cursor.fetchone()
    id_geo = datos['id_geometria']
    id_pieza = datos['id_pieza']
    cursor.execute("INSERT INTO usuario (nombre, rol) VALUES ('Tester', 1) RETURNING id")
    id_user = cursor.fetchone()['id']
    # 2. Insertar primera pieza BIEN
    cursor.execute("CALL sp_rotar_posicionar_figuras(%s, %s, 0, 10, 10, '{}', %s)", (id_geo, i

    with pytest.raises(psycopg2.errors.RaiseException) as excinfo:
        cursor.execute("""
            INSERT INTO cortes_planificados (id_materia, id_pieza, id_usuario, geometria_final
            VALUES (%s, %s, %s, '((11,11),(21,11),(21,21),(11,21),(11,11))')
        """, (id_mp, id_pieza, id_user))
```

Explicación:

- Valida la lógica de colocación y el trigger tr_validar_antes_de_corte.
- Asegura que no se permitan solapamientos.

Prueba de cálculo de utilización

```python
def test_calculo_utilizacion(cursor):
    """Verifica que la función matemática de utilización no de cero."""
    # Insertamos datos manuales para controlar el cálculo
    cursor.execute("INSERT INTO materia_prima (num_parte, ancho, alto, distancia_minima_entre_
    id_mp = cursor.fetchone()['id']

    # Insertamos un corte manual de 50x50 (Area 2500). Total 10000. Utilización esperada 25%.
    cursor.execute("INSERT INTO usuario (nombre, rol) VALUES ('U', 1) RETURNING id")
    u_id = cursor.fetchone()['id']

    # Necesitamos una pieza y producto
    cursor.execute("INSERT INTO producto (nombre, descripcion, geometria) VALUES ('D', 'D', bo
    prod_id = cursor.fetchone()['id']
    cursor.execute("INSERT INTO pieza (producto_id, nombre_pieza, descripcion, cantidad_elemen
    pieza_id = cursor.fetchone()['id']

    cursor.execute("INSERT INTO cortes_planificados (id_materia, id_pieza, id_usuario, geometr

    # Llamar función
    cursor.execute("SELECT fn_calcular_utilizacion(%s) as util", (id_mp,))
    resultado = cursor.fetchone()['util']

    assert resultado == 25.00
```

Explicación:

- Verifica que la función fn_calcular_utilizacion calcule correctamente el porcentaje de aprovechamiento.

Prueba de seguridad y permisos

```python
def test_seguridad_permisos():
    """Prueba que el usuario 'operador' no pueda borrar datos."""
    # Configurar conexión como OPERADOR
    config_operador = DB_CONFIG.copy()
    config_operador['user'] = 'operador'
    config_operador['password'] = '456789'
    try:
        conn_op = psycopg2.connect(**config_operador)
        cur_op = conn_op.cursor()

        # Intentar borrar (Debe fallar)
        with pytest.raises(psycopg2.errors.InsufficientPrivilege):
            cur_op.execute("DELETE FROM materia_prima")

        conn_op.rollback()
        conn_op.close()

    except psycopg2.OperationalError:
        pytest.skip("No se pudo conectar como usuario 'operador'. Verifica que el usuario exis
```

Explicación:

- Comprueba que los permisos de roles estén correctamente definidos (operador no puede eliminar datos críticos).

Conclusión

El Sistema de Optimización de Corte de Materia Prima desarrollado en PostgreSQL ofrece una solución robusta, segura y escalable para la gestión eficiente de materiales laminados y la planificación de cortes. Gracias a la combinación de tablas relacionales, procedimientos almacenados, funciones, triggers y JSON, el sistema permite automatizar tareas complejas como la creación de productos, el posicionamiento y rotación de piezas, así como la validación de colisiones y límites de la materia prima.

La implementación de pruebas unitarias con Python y pytest garantiza la confiabilidad del sistema, asegurando que cada función, trigger y procedimiento se comporte correctamente bajo diferentes escenarios, mientras que la seguridad se mantiene mediante roles de base de datos.

Este enfoque modular y documentado no solo facilita el mantenimiento y la expansión futura (como la integración con interfaces gráficas o nuevas reglas de optimización), sino que también proporciona una base sólida para la toma de decisiones eficientes en entornos productivos, maximizando el aprovechamiento de materiales y minimizando desperdicios.

En resumen, el sistema combina eficiencia operativa, seguridad y facilidad de mantenimiento, constituyéndose en una herramienta integral para la planificación inteligente de cortes de materia prima.