Juan Diego Rodriguez - 202221822

Juan Pablo Reyes - 202310852

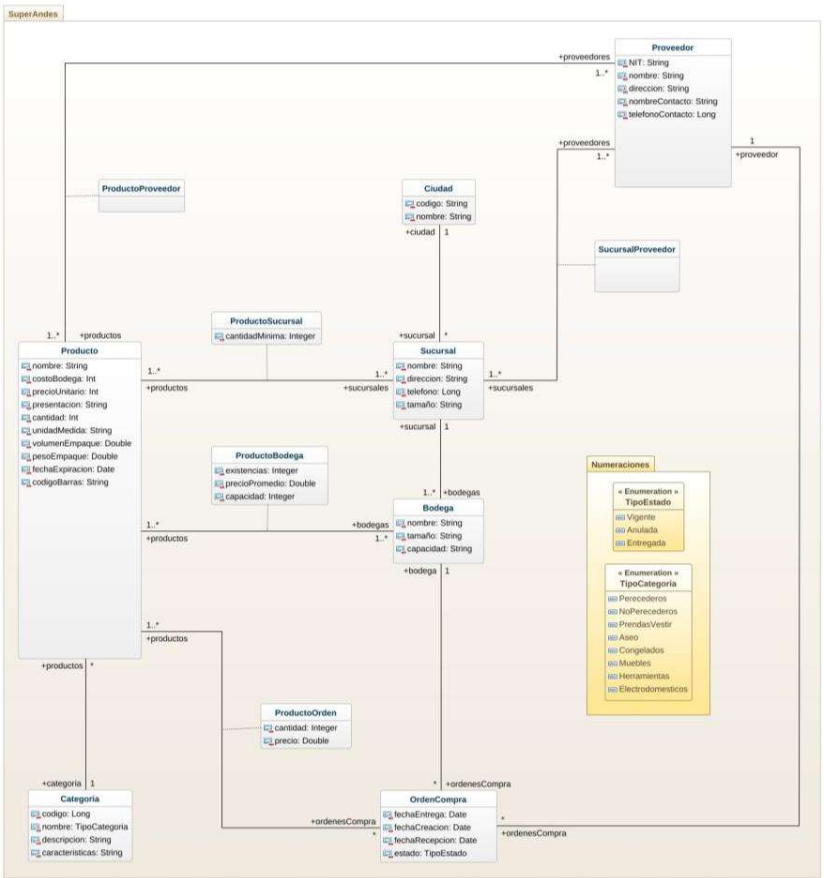
Daniela Munar – 202311392

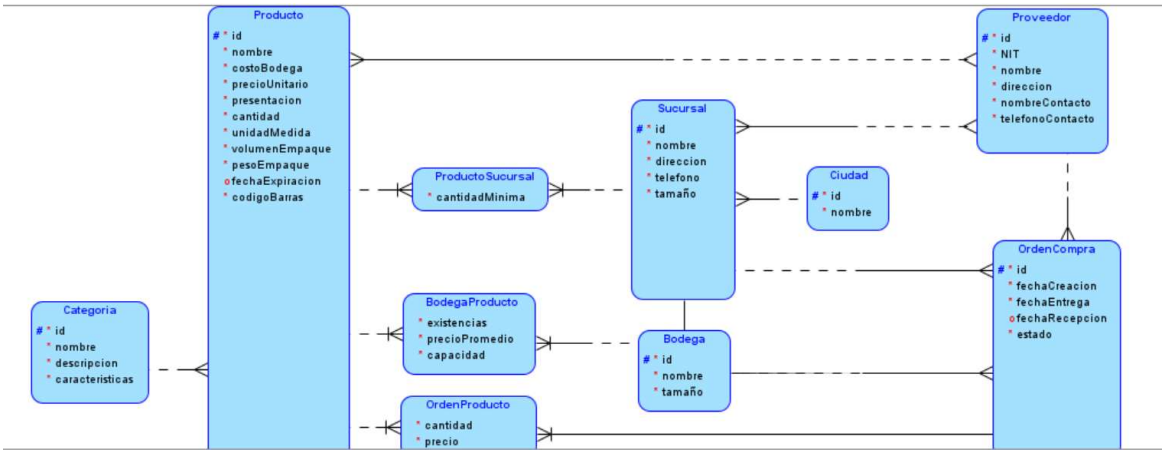
Documentación Modelo SuperAndes

Este documento tiene como objetivo explicar el modelo conceptual y el modelo Entidad-Relación que se utilizarán en la entrega del proyecto para Superandes. A continuación, abordaremos las clases definidas en el modelo, junto con sus atributos, y luego definiremos las relaciones entre las clases, especificando su cardinalidad y obligatoriedad.

1. Clases y atributos

A continuación, se presentarán el modelo conceptual como modelo Entidad-Relación representan todas las clases y entidades involucradas en SuperAndes, mostrando explícitamente sus atributos.





2. Relaciones entre clases

2.1 Categoría – Producto

Producto: La relación con categoría es obligatoria, debido a que no es posible la existencia de un producto sin estar asociado a una determinada categoría. Un producto solo puede tener un tipo de categoría.

Categoría: La relación con producto no es obligatoria, debido a que la categoría puede crearse sin tener ningún producto asociado. Una categoria puede tener varios productos asociados.

2.2 Sucursal - Producto

Producto: La relación con sucursal no es obligatoria, debido a que un producto puede crearse sin necesidad de estar registrado en alguna sucursal. Un producto puede estar registrado en varias sucursales.

Sucursal: La relación con producto no es obligatoria, debido a que cuando una sucursal se crea no tiene ningún producto registrado. Una sucursal puede tener varios productos registrados.

ProductoSucursal: Entidad intermedia que gestiona la relación muchos a muchos entre Producto y Sucursal. Se modela como entidad debido a que incluye un atributo adicional para la cantidad mínima de reorden que contiene una dependencia funcional de ambas entidades.

2.3 Producto – Bodega

Producto: La relación con bodega no es obligatoria, debido a que un producto puede crearse sin necesidad de estar almacenado en alguna bodega. Un producto puede ser almacenado por varias bodegas.

Bodega: La relación con producto no es obligatoria, debido a que cuando una bodega se crea no tiene ningún producto almacenado. Una bodega puede almacenar varios productos.

BodegaProdcuto: Entidad intermedia que gestiona la relación muchos a muchos entre Producto y Bodega. Se modela como entidad debido a que incluye atributos adicionales para el numero de existencias y el costo promedio de un producto que contienen una dependencia funcional de ambas entidades.

2.4 Producto – OrdenCompra

Producto: La relación con Orden de compra no es obligatoria, debido a que puede que un producto jamás sea comprado por alguna sucursal. Un producto puede tener varias ordenes de compra en el tiempo.

OrdenCompra: La relación con producto es obligatoria, debido a que una orden de compra al menos debe tener un producto a vender. Una orden de compra puede tener varios productos.

OrdenProducto: Entidad intermedia que gestiona la relación muchos a muchos entre Producto y OrdenCompra. Se modela como entidad debido a que incluye un atributo adicional para la cantidad de un producto a comprar que contiene una dependencia funcional de ambas entidades.

2.5 Producto – Proveedor

Producto: La relación con proveedor es obligatoria, debido a que un producto no se puede distribuir sin un proveedor. Un producto puede ser distribuido por varios proveedores.

Proveedor: La relación con producto no es obligatoria, debido a que cuando un proveedor se crea no tiene ningún producto registrado. Un proveedor puede distribuir varios productos.

2.6 Proveedor – Sucursal

Proveedor: La relación con sucursal no es obligatoria, debido a que cuando un proveedor se crea no tiene ninguna sucursal registrada. Un proveedor puede distribuir a varias sucursales.

Sucursal: La relación con proveedor es obligatoria, debido a que la sucursal no sería capaz de operar sin tener al menos un proveedor que le venda productos. Una sucursal puede tener varios proveedores.

2.7 Sucursal – Ciudad

Sucursal: La relacion con ciudad es obligatoria, ya que, si la ciudad no existiese, no habría una sucursal en donde se puedan realizar las operaciones habituales. Una sucursal solo puede ser de una ciudad.

Ciudad: La relación con sucursal no es obligatoria, ya que, aunque no haya una sucursal, no quiere decir que la ciudad deje de existir. Una ciudad tiene varias sucursales.

2.8 Sucursal – Bodega

Sucursal: La relacion que tiene sucursal con bodega es de obligatoriedad sabiendo el hecho de que, sin las bodegas, una sucursal no tiene la posibilidad de almacenar los productos disponibles. Una sucursal puede tener varias bodegas.

Bodega: La relacion que tiene bodega con sucursal tambien es de obligatoriedad ya que, sin una sucursal, no podria existir una bodega. Una bodega solo puede pertenecer a una sucursal.

2.9 Proveedor– OrdenCompra

Proveedor: La relación con OrdenCompra no es obligatoria, ya que un proveedor puede existir sin tener órdenes de compra registradas. Un proveedor puede estar relacionado con varias órdenes de compra.

OrdenCompra: La relación con Proveedor es obligatoria, ya que cada orden de compra debe estar vinculada a un proveedor específico. Una orden de compra está asociada a un solo proveedor.

2.10 Bodega – OrdenCompra

Bodega: La relación con OrdenCompra no es obligatoria, ya que una bodega puede existir sin tener órdenes de compra asociadas. Una bodega puede estar asociada con múltiples órdenes de compra.

OrdenCompra: La relación con Bodega es obligatoria, ya que cada orden de compra debe ser registrada en una bodega específica. Una orden de compra está vinculada a una sola bodega.

Documentación Proyecto - Segunda Etapa

1. Requerimientos Funcionales

- **RF10**

```
// Actualización existencias y precio unitarios productos
@Modifying
@Transactional
@Query(value =

"UPDATE bodegaproducto
SET existencias = :existencias,
preciopromedio = :precioPromedio
WHERE bodega_id = :idBodega
AND producto_id = :idProducto",

nativeQuery = true)

void updateBodegaProducto(
@Param("idBodega") Long idBodega,
@Param("idProducto") Long idProducto,
@Param("existencias") Integer existencias,
@Param("precioPromedio") Double precioPromedio);

// Actualizacion estado orden compra
```

```java
@Modifying
@Transactional
@Query(value =

"UPDATE ordencompra
SET estado = :estado,
fecharecepcion = :fechaRecepcion
WHERE id = :idOrdenCompra",

nativeQuery = true)

void updateEstadoOrdenCompra(
@Param("idOrdenCompra") Long idOrdenCompra,
@Param("estado") String estado,
@Param("fechaRecepcion") LocalDate fechaRecepcion);
```

```java
lic class OrdenCompraService {
 public void updateEstadoOrdenCompra(Long idOrdenCompra, OrdenCompra ordenCompra) {
    if (ordenCompra.getEstado().name().equals(anObject:"Entregada")) {

        List<OrdenProducto> productosOrden = ordenProductoRepository.findProductosOrdenCompra(idOrdenCompra);

        for (OrdenProducto productoOrden : productosOrden) {
            Optional<BodegaProducto> bodegaProducto = bodegaProductoRepository.findBodegaProductoById(ordenCompraActual.getBodega().getId

            if (bodegaProducto.isEmpty()) {
                throw new ResponseStatusException(HttpStatus.NOT_FOUND, reason:"El producto no se encuentra registrado en la bodega");
            }

            BodegaProducto bodegaProductoActual = bodegaProducto.get();

            double NuevoPrecioPromedio = ((bodegaProductoActual.getPrecioPromedio() * bodegaProductoActual.getExistencias()) + (productoO
            Integer NuevasExistencias = bodegaProductoActual.getExistencias() + productoOrden.getCantidad();

            try {
                bodegaProductoRepository.updateBodegaProducto(ordenCompraActual.getBodega().getId(), productoOrden.getId().getProducto().
            }
            catch (Exception e) {
                throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, reason:"Error al actualizar el producto en la bodega"
            }
        }

        try {
            ordenCompraRepository.updateEstadoOrdenCompra(idOrdenCompra, ordenCompra.getEstado().name(), LocalDate.now());
        }
        catch (Exception e) {
            throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, reason:"Error al actualizar el estado de la orden de comp
        }
    }
```

Para el requerimiento funcional de registrar la recepción de productos en la bodega mediante un Documento de Ingreso, se decidió no crear una entidad separada para este documento. En lugar de ello, se reutilizó la entidad Orden de Compra, ya que la información requerida en el documento es idéntica a la manejada en la orden de compra, evitando así duplicidad de datos.

- El ingreso de un documento se implementó como un servicio de actualización de la Orden de Compra, que realiza tres acciones clave:
- Actualiza la fecha de recepción con la fecha actual para reflejar el momento de ingreso de los productos.

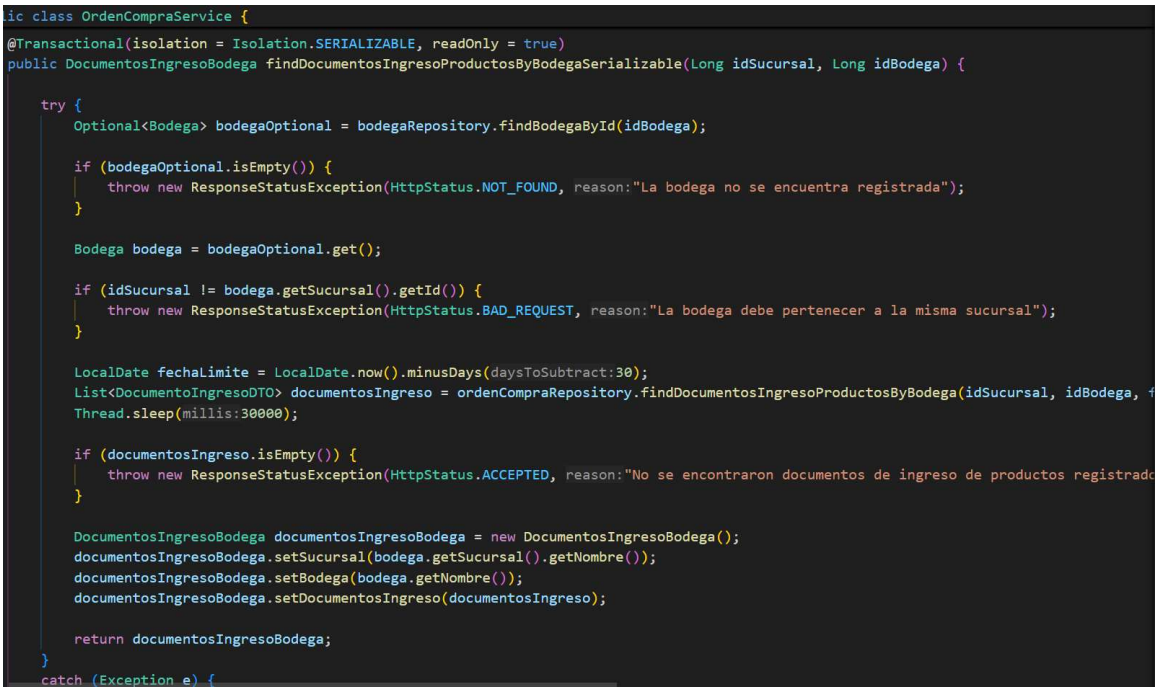
- Abastece los productos en la bodega correspondiente, incrementando el inventario según las cantidades especificadas en la orden.
- Cambia el estado de la orden a "entregada", indicando que el proceso de recepción se ha completado.

Este servicio encapsula todo el proceso de recepción y registro en la bodega sin requerir una entidad adicional, utilizando eficientemente la información existente en la Orden de Compra.

2. Requerimientos Funcionales de Consulta

- **RFC6**

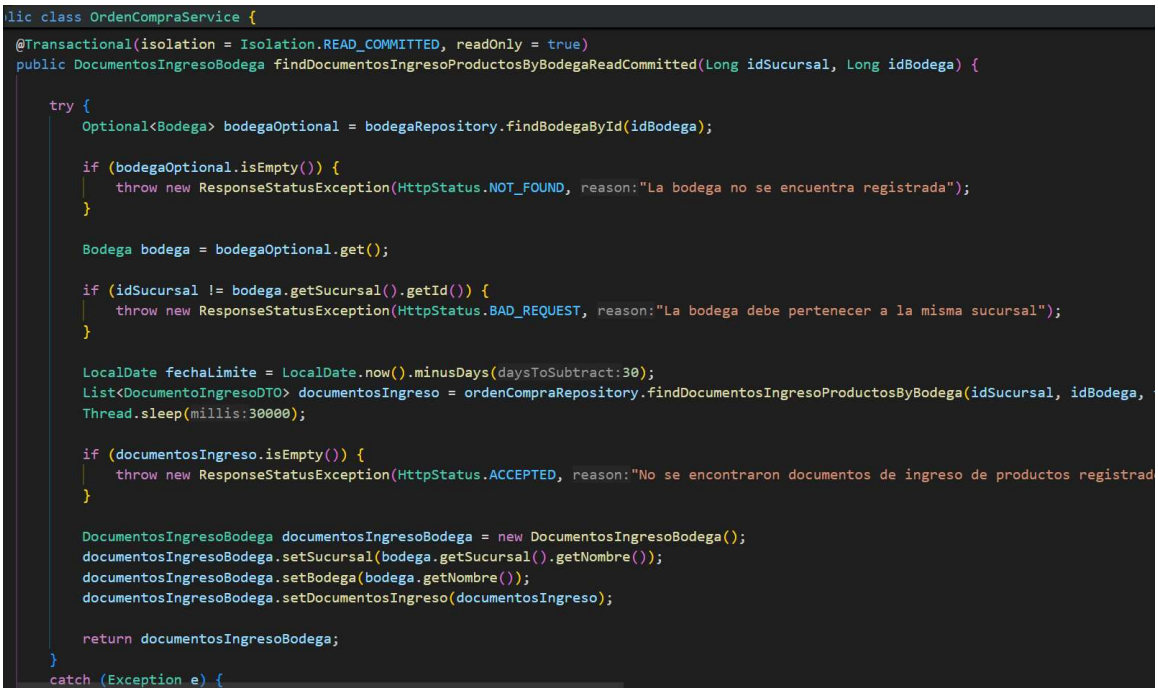
```
@Query(value =

"SELECT op.id, op.fecharecepcion, p.nombre FROM ordencompra op
INNER JOIN proveedor p
ON p.id = op.proveedor_id
WHERE sucursal_id = :idSucursal
AND bodega_id = :idBodega
AND fecharecepcion >= :fechaLimite
ORDER BY op.id",

nativeQuery = true)

List<DocumentoIngresoDTO>findDocumentosIngresoProductosByBodega(
@Param("idSucursal") Long idSucursal,
@Param("idBodega") Long idBodega,
@Param("fechaLimite") LocalDate fechaLimite);
```



```
lic class OrdenCompraService {
@Transactional(isolation = Isolation.SERIALIZABLE, readOnly = true)
public DocumentosIngresoBodega findDocumentosIngresoProductosByBodegaSerializable(Long idSucursal, Long idBodega) {

    try {
        Optional<Bodega> bodegaOptional = bodegaRepository.findBodegaById(idBodega);

        if (bodegaOptional.isEmpty()) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, reason:"La bodega no se encuentra registrada");
        }

        Bodega bodega = bodegaOptional.get();

        if (idSucursal != bodega.getSucursal().getId()) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, reason:"La bodega debe pertenecer a la misma sucursal");
        }

        LocalDate fechaLimite = LocalDate.now().minusDays(daysToSubtract:30);
        List<DocumentoIngresoDTO> documentosIngreso = ordenCompraRepository.findDocumentosIngresoProductosByBodega(idSucursal, idBodega, f
        Thread.sleep(millis:30000);

        if (documentosIngreso.isEmpty()) {
            throw new ResponseStatusException(HttpStatus.ACCEPTED, reason:"No se encontraron documentos de ingreso de productos registrad

        DocumentosIngresoBodega documentosIngresoBodega = new DocumentosIngresoBodega();
        documentosIngresoBodega.setSucursal(bodega.getSucursal().getNombre());
        documentosIngresoBodega.setBodega(bodega.getNombre());
        documentosIngresoBodega.setDocumentosIngreso(documentosIngreso);

        return documentosIngresoBodega;
    }
    catch (Exception e) {
```

Para el requerimiento funcional de consultar los documentos de ingreso de una bodega en los últimos 30 días, y dado que se modeló con la entidad Orden de Compra, se realiza la verificación de que la fecha de recepción sea mayor a los 30 días anteriores. Si este atributo es distinto de null, significa que la orden ya fue entregada, es decir, existe el "supuesto" Documento de Ingreso. Esto permite identificar las órdenes que cumplen con el criterio de ingreso sin necesidad de una entidad adicional. Este servicio fue implementado con un nivel de aislamiento SERIALIZABLE para garantizar la consistencia de los datos y evitar anomalías en la consulta, como lecturas sucias, no repetibles y lecturas fantasma, asegurando que solo se incluyan registros que cumplan con el criterio de ingreso dentro del período especificado.

- **RFC7**



```
lic class OrdenCompraService {
@Transactional(isolation = Isolation.READ_COMMITTED, readOnly = true)
public DocumentosIngresoBodega findDocumentosIngresoProductosByBodegaReadCommitted(Long idSucursal, Long idBodega) {

    try {
        Optional<Bodega> bodegaOptional = bodegaRepository.findBodegaById(idBodega);

        if (bodegaOptional.isEmpty()) {
            throw new ResponseStatusException(HttpStatus.NOT_FOUND, reason:"La bodega no se encuentra registrada");
        }

        Bodega bodega = bodegaOptional.get();

        if (idSucursal != bodega.getSucursal().getId()) {
            throw new ResponseStatusException(HttpStatus.BAD_REQUEST, reason:"La bodega debe pertenecer a la misma sucursal");
        }

        LocalDate fechaLimite = LocalDate.now().minusDays(daysToSubtract:30);
        List<DocumentoIngresoDTO> documentosIngreso = ordenCompraRepository.findDocumentosIngresoProductosByBodega(idSucursal, idBodega,
        Thread.sleep(millis:30000);

        if (documentosIngreso.isEmpty()) {
            throw new ResponseStatusException(HttpStatus.ACCEPTED, reason:"No se encontraron documentos de ingreso de productos registrad
        }

        DocumentosIngresoBodega documentosIngresoBodega = new DocumentosIngresoBodega();
        documentosIngresoBodega.setSucursal(bodega.getSucursal().getNombre());
        documentosIngresoBodega.setBodega(bodega.getNombre());
        documentosIngresoBodega.setDocumentosIngreso(documentosIngreso);

        return documentosIngresoBodega;
    }
    catch (Exception e) {
```

Para el requerimiento funcional de consultar los documentos de ingreso de una bodega en los últimos 30 días, y dado que se modeló con la entidad Orden de Compra, se implementó un servicio que verifica que la fecha de recepción sea mayor a los 30 días anteriores. Si este atributo es distinto de null, significa que la orden fue entregada y, por lo tanto, existe el "supuesto" Documento de Ingreso. Este servicio fue implementado con un nivel de aislamiento READ COMMITTED, el cual evita lecturas sucias, pero permite lecturas no repetibles y lecturas fantasma, proporcionando un equilibrio entre consistencia y rendimiento.

3. Escenarios de Pruebas

- **RFC6**

Tiempo	Usuario 1 (RFC6)	**Usuario 2 (RF10)**
T = 0 seg	Inicia la ejecución de RFC6 con nivel de aislamiento SERIALIZABLE. Bloquea las filas consultadas para evitar modificaciones concurrentes.	
T = 5 seg		Inicia la ejecución de RF10 para registrar el ingreso de productos en la bodega. Debido al nivel SERIALIZABLE de RFC6, RF10 queda bloqueado esperando a que termine RFC6.

T = 30 seg	Finaliza la ejecución de RFC6 y libera los bloqueos en las filas consultadas.	
T = 30+ seg		RF10 continúa su ejecución tras la finalización de RFC6, registrando el ingreso de productos y actualizando la fecha de recepción de la orden de compra.

En este escenario, aunque RFC6 se ejecutó con el nivel de aislamiento SERIALIZABLE, el request al API del RF10 fue exitoso en su ejecución sin necesidad de esperar a que RFC6 terminara. Esto indica que RF10 pudo completar el proceso de actualización de la orden de compra, marcándola como "entregada" y registrando el ingreso de productos. Sin embargo, RFC6 no mostró el registro recién agregado en sus resultados. Esto confirma que no hubo lecturas fantasmas, ya que el nivel de aislamiento SERIALIZABLE en RFC6 asegura que los datos leídos no cambien durante su ejecución. Por lo tanto, los cambios realizados por RF10 no se reflejaron en la consulta de RFC6, manteniendo la consistencia de los datos leídos originalmente

- **RFC7**

Tiempo	Usuario 1 (RFC6)	**Usuario 2 (RF10)**
T = 0 seg	Inicia la ejecución de RFC6 con nivel de aislamiento READ COMMITTED. Realiza una consulta de documentos de ingreso con datos confirmados en la base de datos.	
T = 5 seg	RFC6 sigue en ejecución, manteniendo el nivel de READ COMMITTED, lo cual permite modificaciones concurrentes.	Inicia la ejecución de RF10 para registrar el ingreso de productos en la bodega. Dado que READ COMMITTED no bloquea otras transacciones, RF10 puede ejecutar sin esperar a RFC6.
T = 8 seg		RF10 finaliza exitosamente el registro del ingreso de productos en la bodega y actualiza la orden de compra.
T = 30	RFC6 finaliza su ejecución y, dado que está en READ COMMITTED, puede ver los datos confirmados por RF10 en sus resultados.	

En este escenario, como RFC6 se ejecutó con el nivel de aislamiento READ COMMITTED, el request al API de RF10 fue exitoso en su ejecución sin necesidad de esperar a que RFC6 terminara. Esto indica que RF10 pudo completar el proceso de actualización de la orden de compra, marcándola como "entregada" y registrando el ingreso de productos. Debido a que RFC6 utilizó READ COMMITTED, sí mostró el registro recién agregado en sus resultados. Esto confirma que hubo una lectura fantasma, ya que este nivel de aislamiento permite que los datos leídos puedan cambiar si otra transacción realiza una modificación concurrente. Por lo tanto, los cambios realizados por RF10 se reflejaron en la consulta de RFC6, mostrando los datos más recientes en lugar de mantener la consistencia de los datos leídos originalmente.