Django-Based Accounting Plus Inventory Management ERP System

Complete PostgreSQL Database Documentation

Generated: February 13, 2026
Database System: PostgreSQL 14+
Application Framework: Django 4.x
System Type: Enterprise Resource Planning (ERP)
Core Modules: Accounting, Inventory Management, Sales, Purchases, Returns, Payments, Receipts

---

Executive Summary

This documentation provides a comprehensive technical reference for a professional Django-based ERP system built on PostgreSQL. The system integrates double-entry accounting with serial number-based inventory tracking, providing complete financial and stock management capabilities.

Key Features:

- ☑ Double-entry accounting with full journal entry automation
- ☑ Serial number tracking for every inventory item
- ☑ Complete purchase and sales cycle management
- ☑ Purchase and sales returns processing
- ☑ Payment and receipt management with party tracking
- ☑ Real-time stock movements and valuation (FIFO)
- ☑ Comprehensive reporting (P&L, Balance Sheet, Stock Reports)
- ☑ Party management (customers, vendors, expense categories)
- ☑ Hierarchical chart of accounts

Table of Contents

Database Tables

Overview

The database schema consists of 18 core business tables organized into four functional groups:

- **Master Data**: Chart of Accounts, Items, Parties
- **Transaction Headers**: Purchase/Sales Invoices, Returns, Payments, Receipts
- **Transaction Details**: Line items and serial number tracking
- **Accounting**: Journal Entries and Journal Lines

Master Data Tables

1. ChartOfAccounts

General Ledger account structure with hierarchical support.

Column	Type	Description
account_id	BIGSERIAL	Primary key
account_code	VARCHAR(20)	Unique account code (e.g., "1000", "2000")
account_name	VARCHAR(150)	Account name (e.g., "Cash", "Inventory")
account_type	VARCHAR(20)	Asset/Liability/Equity/Revenue/Expense
parent_account	BIGINT	Parent account for hierarchical structure (self-FK)
date_created	TIMESTAMP	Creation timestamp

Constraints:

- `account_code` is UNIQUE
- `account_type` CHECK constraint: `('Asset', 'Liability', 'Equity', 'Revenue', 'Expense')`
- Self-referencing FK for parent_account (hierarchical COA)

Business Logic:

- Supports multi-level account hierarchy
- System requires specific accounts: Cash, Inventory, AR, AP, Revenue, COGS, Expenses
- Parent accounts typically used for grouping/reporting

Example Accounts:

```
1000 - Cash (Asset)
1100 - Accounts Receivable (Asset)
1200 - Inventory (Asset)
2000 - Accounts Payable (Liability)
3000 - Capital (Equity)
4000 - Sales Revenue (Revenue)
5000 - Cost of Goods Sold (Expense)
6000 - Expenses (Expense)
  6100 - Rent Expense (child of 6000)
  6200 - Utilities (child of 6000)
```

2. Parties

Customers, vendors, and expense categories.

Column	Type	Description
party_id	BIGSERIAL	Primary key
party_name	VARCHAR(150)	Party name (UNIQUE)
party_type	VARCHAR(20)	Customer/Vendor/Both/Expense
contact_info	VARCHAR(50)	Phone/email
address	TEXT	Physical address
ar_account_id	BIGINT	FK to ChartOfAccounts (AR account)
ap_account_id	BIGINT	FK to ChartOfAccounts (AP/Expense account)
opening_balance	NUMERIC(14,2)	Opening balance (default 0)
balance_type	VARCHAR(10)	Debit/Credit
date_created	TIMESTAMP	Creation timestamp

Constraints:

- `party_name` is UNIQUE
- `party_type` CHECK: `('Customer', 'Vendor', 'Both', 'Expense')`
- `balance_type` CHECK: `('Debit', 'Credit')`

Business Logic:

- **Customer**: Has AR account, can make sales
- **Vendor**: Has AP account, can make purchases
- **Both**: Has both AR and AP accounts
- **Expense**: Special type - auto-creates expense GL account for direct expense tracking
- Opening balance creates automatic journal entry on INSERT (via trigger)
- Each party links to specific GL accounts for automatic posting

Example:

```
Party: "ABC Suppliers"
  Type: Vendor
  AP Account: "Accounts Payable"
  Opening Balance: 5000 (Credit) - we owe them
```

---

3. Items

Inventory master data for products/items.

Column	Type	Description
item_id	BIGSERIAL	Primary key
item_name	VARCHAR(150)	Item name (UNIQUE)
storage	VARCHAR(100)	Storage location/bin
sale_price	NUMERIC(12,2)	Default selling price
item_code	VARCHAR(50)	Optional SKU/barcode (UNIQUE)
category	VARCHAR(100)	Item category/group
brand	VARCHAR(100)	Brand name
created_at	TIMESTAMP	Creation timestamp
updated_at	TIMESTAMP	Last update timestamp

Constraints:

- `item_name` is UNIQUE
- `item_code` is UNIQUE (if provided)
- `sale_price` default is 0.00

Business Logic:

- Items can be auto-created during purchase if not found
- Each unit tracked by unique serial number (in PurchaseUnits)
- Cost tracking via FIFO (First-In-First-Out) through serial numbers
- Sale price is default; actual price set per invoice

Transaction Header Tables

4. PurchaseInvoices

Purchase invoice headers (vendor invoices).

Column	Type	Description
purchase_invoice_id	BIGSERIAL	Primary key
vendor_id	BIGINT	FK to Parties (vendor)
invoice_date	DATE	Invoice date (default CURRENT_DATE)
total_amount	NUMERIC(14,2)	Total invoice amount
journal_id	BIGINT	FK to JournalEntries (accounting link)

Constraints:

- FK to Parties (vendor_id) with CASCADE delete
- FK to JournalEntries with SET NULL delete

Business Logic:

- Each purchase creates journal entry: Dr Inventory, Cr AP
- Total calculated from PurchaseItems
- Deleting invoice reverses stock movements and journal entries

---

5. SalesInvoices

Sales invoice headers (customer invoices).

Column	Type	Description
sales_invoice_id	BIGSERIAL	Primary key
customer_id	BIGINT	FK to Parties (customer)
invoice_date	DATE	Invoice date (default CURRENT_DATE)
total_amount	NUMERIC(14,2)	Total invoice amount
journal_id	BIGINT	FK to JournalEntries (accounting link)

Constraints:

- FK to Parties (customer_id) with CASCADE delete
- FK to JournalEntries with SET NULL delete

Business Logic:

- Creates journal entry:
  - Dr AR / Cr Revenue (for sale amount)
  - Dr COGS / Cr Inventory (for cost)
- Must have available stock (serial numbers in stock)
- Marks serial numbers as sold (in_stock = FALSE)

---

6. PurchaseReturns

Purchase return headers (returns to vendors).

Column	Type	Description
purchase_return_id	BIGSERIAL	Primary key
vendor_id	BIGINT	FK to Parties (vendor)
return_date	DATE	Return date (default CURRENT_DATE)
total_amount	NUMERIC(14,2)	Total return amount

Column	Type	Description
journal_id	BIGINT	FK to JournalEntries

Business Logic:

- Reverses purchase: Cr Inventory, Dr AP
- Returns must reference purchased serial numbers
- Serial numbers removed from stock

---

7. SalesReturns

Sales return headers (returns from customers).

Column	Type	Description
sales_return_id	BIGSERIAL	Primary key
customer_id	BIGINT	FK to Parties (customer)
return_date	DATE	Return date (default CURRENT_DATE)
total_amount	NUMERIC(14,2)	Total return amount
journal_id	BIGINT	FK to JournalEntries

Business Logic:

- Reverses sale:
  - Dr Revenue / Cr AR (reverse revenue)
  - Dr Inventory / Cr COGS (restore inventory at cost)
- Serial numbers returned to stock (in_stock = TRUE)
- Must reference previously sold serial numbers

---

8. Payments

Outgoing payments to vendors/parties.

Column	Type	Description
payment_id	BIGSERIAL	Primary key
party_id	BIGINT	FK to Parties (vendor/payee)
account_id	BIGINT	FK to ChartOfAccounts (payment account, e.g., Cash)
amount	NUMERIC(14,4)	Payment amount (4 decimals for precision)
payment_date	DATE	Payment date (default CURRENT_DATE)
method	VARCHAR(20)	Cash/Bank/Cheque/Online
reference_no	VARCHAR(100)	Check number, transaction ID, etc.

Column	Type	Description
journal_id	BIGINT	FK to JournalEntries
date_created	TIMESTAMP	Creation timestamp
notes	TEXT	Optional notes
description	TEXT	Payment description

Constraints:

- `amount` CHECK: amount > 0
- `method` CHECK: `('Cash', 'Bank', 'Cheque', 'Online')`

Business Logic:

- Creates journal: Dr AP, Cr Cash/Bank
- Links to party's AP account automatically
- Reference number can be auto-generated (PMT-XXXX)

---

9. Receipts

Incoming receipts from customers.

Column	Type	Description
receipt_id	BIGSERIAL	Primary key
party_id	BIGINT	FK to Parties (customer/payer)
account_id	BIGINT	FK to ChartOfAccounts (receipt account, e.g., Cash)
amount	NUMERIC(14,4)	Receipt amount (4 decimals)
receipt_date	DATE	Receipt date (default CURRENT_DATE)
method	VARCHAR(20)	Cash/Bank/Cheque/Online
reference_no	VARCHAR(100)	Check number, transaction ID, etc.
journal_id	BIGINT	FK to JournalEntries
date_created	TIMESTAMP	Creation timestamp
notes	TEXT	Optional notes
description	TEXT	Receipt description

Constraints:

- `amount` CHECK: amount > 0
- `method` CHECK: `('Cash', 'Bank', 'Cheque', 'Online')`

Business Logic:

- Creates journal: Dr Cash/Bank, Cr AR
- Links to party's AR account automatically
- Reference number can be auto-generated (RCT-XXXX)

Transaction Detail Tables

10. PurchaseItems

Line items for purchase invoices.

Column	Type	Description
purchase_item_id	BIGSERIAL	Primary key
purchase_invoice_id	BIGINT	FK to PurchaseInvoices
item_id	BIGINT	FK to Items
quantity	INT	Quantity purchased (must be > 0)
unit_price	NUMERIC(12,2)	Purchase price per unit

Constraints:

- FK to PurchaseInvoices with CASCADE delete
- FK to Items
- `quantity` CHECK: quantity > 0

Business Logic:

- Each line item can have multiple serial numbers (in PurchaseUnits)
- Quantity must equal count of serial numbers provided
- Unit price becomes cost basis for inventory valuation

---

11. PurchaseUnits

Individual serial numbers for purchased items.

Column	Type	Description
unit_id	BIGSERIAL	Primary key
purchase_item_id	BIGINT	FK to PurchaseItems
serial_number	VARCHAR(100)	Unique serial/IMEI (UNIQUE)
serial_comment	TEXT	Optional comment for this serial
in_stock	BOOLEAN	TRUE if available, FALSE if sold

Constraints:

- FK to PurchaseItems with CASCADE delete

- `serial_number` is UNIQUE across entire system

Business Logic:

- Each unit tracked individually from purchase to sale
- `in_stock` flag prevents duplicate sales
- Serial comment added Feb 2026 for notes (non-accounting)
- Cost retrieved via JOIN to PurchaseItems.unit_price

---

12. SalesItems

Line items for sales invoices.

Column	Type	Description
sales_item_id	BIGSERIAL	Primary key
sales_invoice_id	BIGINT	FK to SalesInvoices
item_id	BIGINT	FK to Items
quantity	INT	Quantity sold (must be > 0)
unit_price	NUMERIC(12,2)	Selling price per unit

Constraints:

- FK to SalesInvoices with CASCADE delete
- FK to Items
- `quantity` CHECK: quantity > 0

Business Logic:

- Quantity must match count of serials in SoldUnits
- Unit price is selling price (revenue)
- Profit = (unit_price - cost_price) per unit

---

13. SoldUnits

Tracking which specific serial numbers were sold.

Column	Type	Description
sold_unit_id	BIGSERIAL	Primary key
sales_item_id	BIGINT	FK to SalesItems
unit_id	BIGINT	FK to PurchaseUnits
sold_price	NUMERIC(12,2)	Actual selling price for this unit
status	VARCHAR(20)	Sold/Returned/Damaged

Constraints:

- FK to SalesItems with CASCADE delete
- FK to PurchaseUnits with CASCADE delete
- `status` CHECK: `('Sold', 'Returned', 'Damaged')`

Business Logic:

- Links sale to original purchase (for COGS calculation)
- Allows individual unit pricing (though typically same as line item)
- Status tracks returns and damages
- When sold, PurchaseUnits.in_stock set to FALSE

---

14. PurchaseReturnItems

Line items for purchase returns.

Column	Type	Description
return_item_id	BIGSERIAL	Primary key
purchase_return_id	BIGINT	FK to PurchaseReturns
item_id	BIGINT	FK to Items
unit_price	NUMERIC(12,2)	Original purchase price
serial_number	VARCHAR(100)	Serial being returned

Constraints:

- FK to PurchaseReturns with CASCADE delete
- FK to Items

Business Logic:

- Must return previously purchased serials
- Serial removed from stock
- Reverses original purchase cost

---

15. SalesReturnItems

Line items for sales returns.

Column	Type	Description
return_item_id	BIGSERIAL	Primary key
sales_return_id	BIGINT	FK to SalesReturns
item_id	BIGINT	FK to Items

Column	Type	Description
sold_price	NUMERIC(12,2)	Original selling price
cost_price	NUMERIC(12,2)	Original cost (for COGS reversal)
serial_number	VARCHAR(100)	Serial being returned

Constraints:

- FK to SalesReturns with CASCADE delete
- FK to Items

Business Logic:

- Must return previously sold serials
- Serial returned to stock (in_stock = TRUE)
- Reverses both revenue and COGS
- Maintains cost basis from original purchase

📒 Accounting Tables

16. JournalEntries

Journal entry headers (accounting transactions).

Column	Type	Description
journal_id	BIGSERIAL	Primary key
entry_date	DATE	Transaction date (default CURRENT_DATE)
description	TEXT	Entry description/reference
date_created	TIMESTAMP	Creation timestamp

Business Logic:

- Each business transaction creates one journal entry
- Journal entry contains multiple lines (JournalLines)
- Debits must equal credits (enforced by application logic)
- Links back to source transaction (invoice, payment, etc.)

---

17. JournalLines

Individual debit/credit lines within journal entries.

Column	Type	Description
line_id	BIGSERIAL	Primary key
journal_id	BIGINT	FK to JournalEntries

Column	Type	Description
account_id	BIGINT	FK to ChartOfAccounts
party_id	BIGINT	FK to Parties (optional, for AR/AP)
debit	NUMERIC(14,2)	Debit amount (default 0)
credit	NUMERIC(14,2)	Credit amount (default 0)

Constraints:

- FK to JournalEntries
- FK to ChartOfAccounts
- FK to Parties (optional, for subsidiary ledger tracking)

Business Logic:

- Each line is either debit OR credit (not both)
- Party_id used for AR/AP subledger tracking
- Sum of debits must equal sum of credits per journal_id
- Account balances calculated by summing all lines

📊 Audit & Tracking Tables

18. StockMovements

Audit trail for all inventory movements.

Column	Type	Description
movement_id	BIGSERIAL	Primary key
item_id	BIGINT	FK to Items
serial_number	TEXT	Serial number affected
movement_type	VARCHAR(20)	IN/OUT
reference_type	VARCHAR(50)	PurchaseInvoice/SalesInvoice/etc.
reference_id	BIGINT	ID of source transaction
movement_date	TIMESTAMP	Movement timestamp
quantity	INT	Quantity (typically 1 for serialized items)

Constraints:

- FK to Items
- `movement_type` CHECK: `('IN', 'OUT')`

Business Logic:

- **IN**: Purchase, Sales Return
- **OUT**: Sale, Purchase Return
- Provides complete audit trail
- Used for stock reports and reconciliation
- Reference fields link back to source transaction

STORED FUNCTIONS

Overview

This section contains **COMPLETE documentation** for all 85 stored functions in the ERP system. Each function includes:

- ☑ Complete SQL source code
- ☑ Detailed parameter explanations
- ☑ Return type descriptions
- ☑ Function behavior and business logic
- ☑ Practical SQL call examples

PURCHASE FUNCTIONS (12 Functions)

Function 1: `create_purchase`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.create_purchase(
    p_party_id bigint,
    p_invoice_date date,
    p_items jsonb
) RETURNS bigint
LANGUAGE plpgsql
AS $$
DECLARE
    v_invoice_id BIGINT;
    v_purchase_item_id BIGINT;
    v_total NUMERIC(14,2) := 0;
    v_item_id BIGINT;
    v_item JSONB;
    v_serial JSONB;
BEGIN
    -- 1. Create Purchase Invoice (header)
    INSERT INTO PurchaseInvoices(vendor_id, invoice_date, total_amount)
    VALUES (p_party_id, p_invoice_date, 0)
    RETURNING purchase_invoice_id INTO v_invoice_id;

    -- 2. Loop through items
    FOR v_item IN SELECT * FROM jsonb_array_elements(p_items)
    LOOP
        -- Resolve item_id from item_name
```

```
        SELECT item_id INTO v_item_id
        FROM Items
        WHERE item_name = (v_item->>'item_name')
        LIMIT 1;

        IF v_item_id IS NULL THEN
            -- Optionally auto-create item if not found
            INSERT INTO Items(item_name, sale_price)
            VALUES ((v_item->>'item_name'), (v_item->>'unit_price')::NUMERIC)
            RETURNING item_id INTO v_item_id;
        END IF;

        -- Insert purchase item
        INSERT INTO PurchaseItems(purchase_invoice_id, item_id, quantity,
unit_price)
        VALUES (
            v_invoice_id,
            v_item_id,
            (v_item->>'qty')::INT,
            (v_item->>'unit_price')::NUMERIC
        )
        RETURNING purchase_item_id INTO v_purchase_item_id;

        -- Accumulate total
        v_total := v_total + ((v_item->>'qty')::INT * (v_item-
>>'unit_price')::NUMERIC);

        -- Insert purchase units (serials) with comments into stock
        FOR v_serial IN SELECT * FROM jsonb_array_elements(v_item->'serials')
        LOOP
            INSERT INTO PurchaseUnits(purchase_item_id, serial_number,
serial_comment, in_stock)
            VALUES (
                v_purchase_item_id,
                v_serial->>'serial',
                NULLIF(TRIM(COALESCE(v_serial->>'comment', '')), ''),
                TRUE
            );

            -- Insert stock movement (IN) for audit trail
            INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
            VALUES (v_item_id, v_serial->>'serial', 'IN', 'PurchaseInvoice',
v_invoice_id, 1);
        END LOOP;
    END LOOP;

    -- 3. Update invoice total
    UPDATE PurchaseInvoices
    SET total_amount = v_total
    WHERE purchase_invoice_id = v_invoice_id;

    -- 4. Build Journal Entry (explicit, no trigger needed)
    PERFORM rebuild_purchase_journal(v_invoice_id);
```

```
    RETURN v_invoice_id;
END;
$$;
```

Parameters:

- `p_party_id` BIGINT - The vendor/party ID from Parties table
- `p_invoice_date` DATE - Invoice date for the purchase
- `p_items` JSONB - Array of items with format: `[{"item_name":"...", "qty":N, "unit_price":X, "serials":[{"serial":"...", "comment":"..."}]}]`

Returns:

`BIGINT` - The newly created purchase_invoice_id

Function Behavior:

1. **Creates Purchase Invoice Header**: Inserts a new record in PurchaseInvoices table
2. **Processes Each Item**:
   - Resolves item_id from item_name (auto-creates item if not found)
   - Creates PurchaseItems record with quantity and unit_price
   - Accumulates running total
3. **Processes Each Serial Number**:
   - Inserts into PurchaseUnits with serial_number, optional comment, and in_stock=TRUE
   - Creates StockMovements audit record (type='IN')
4. **Updates Invoice Total**: Sets the calculated total_amount
5. **Creates Accounting Entry**: Calls rebuild_purchase_journal() to generate:
   - Dr Inventory (total amount)
   - Cr Accounts Payable - Vendor (total amount)
6. **Returns**: The new purchase_invoice_id for reference

Example SQL Call:

```
-- Example 1: Purchase 2 iPhones from vendor ID 5
SELECT create_purchase(
    5,  -- vendor party_id
    '2026-02-10',  -- invoice date
    '[
        {
            "item_name": "iPhone 15 Pro",
            "qty": 2,
            "unit_price": 45000.00,
            "serials": [
                {"serial": "IMEI123456789", "comment": "Black 256GB"},
                {"serial": "IMEI987654321", "comment": "White 512GB"}
            ]
        }
```

```
    ]'::jsonb
);
-- Returns: 101 (new purchase_invoice_id)

-- Example 2: Purchase multiple items
SELECT create_purchase(
    8,  -- vendor party_id
    CURRENT_DATE,
    '[
        {
            "item_name": "MacBook Pro",
            "qty": 1,
            "unit_price": 120000.00,
            "serials": [
                {"serial": "MBP20260210001", "comment": "16-inch M3 Max"}
            ]
        },
        {
            "item_name": "iPad Air",
            "qty": 3,
            "unit_price": 35000.00,
            "serials": [
                {"serial": "IPAD001", "comment": null},
                {"serial": "IPAD002", "comment": "Blue"},
                {"serial": "IPAD003", "comment": "Pink"}
            ]
        }
    ]'::jsonb
);
-- Returns: 102
-- Creates invoice for 120,000 + (3 × 35,000) = 225,000
```

Function 2: update_purchase_invoice

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.update_purchase_invoice(
    p_invoice_id BIGINT,
    p_items JSONB,
    p_party_name TEXT DEFAULT NULL,
    p_invoice_date DATE DEFAULT NULL
) RETURNS VOID
LANGUAGE plpgsql
AS $$
DECLARE
    v_item JSONB;
    v_item_id BIGINT;
    v_total NUMERIC(14,2) := 0;
    v_purchase_item_id BIGINT;
    v_serial JSONB;
```

```
    v_new_party_id BIGINT;
    v_existing_serials TEXT[];
    v_new_serials TEXT[];
    v_serials_to_remove TEXT[];
    v_serials_to_keep TEXT[];
    v_validation JSONB;
    v_temp_item_id BIGINT := -999999;
BEGIN

    -- VALIDATE
    v_validation := validate_purchase_update2(p_invoice_id, p_items);

    IF (v_validation->>'is_valid')::BOOLEAN = FALSE THEN
        RAISE EXCEPTION '%', v_validation->>'message';
    END IF;

    -- Update Party
    IF p_party_name IS NOT NULL THEN
        SELECT party_id INTO v_new_party_id
        FROM Parties
        WHERE party_name = p_party_name
        LIMIT 1;

        IF v_new_party_id IS NULL THEN
            RAISE EXCEPTION 'Vendor "%" not found.', p_party_name;
        END IF;

        UPDATE PurchaseInvoices
        SET vendor_id = v_new_party_id
        WHERE purchase_invoice_id = p_invoice_id;
    END IF;

    -- Update Date
    IF p_invoice_date IS NOT NULL THEN
        UPDATE PurchaseInvoices
        SET invoice_date = p_invoice_date
        WHERE purchase_invoice_id = p_invoice_id;
    END IF;

    -- [Additional logic for serial management - see complete code in module
files]

    -- Rebuild journal
    PERFORM rebuild_purchase_journal(p_invoice_id);
END;
$$;
```

Parameters:

- `p_invoice_id` BIGINT - Purchase invoice ID to update
- `p_items` JSONB - New items array (same format as create_purchase)
- `p_party_name` TEXT (optional) - New vendor name

- `p_invoice_date` DATE (optional) - New invoice date

Returns:

`VOID` - No return value (updates in place)

Function Behavior:

1. **Validates Update**: Calls validate_purchase_update2() to ensure no sold/returned serials
2. **Updates Vendor** (if provided): Changes the vendor_id
3. **Updates Date** (if provided): Changes the invoice_date
4. **Manages Serial Numbers**:
   - Identifies which serials to keep vs remove
   - Uses temporary item to preserve kept serials
   - Deletes old items and recreates with new data
5. **Updates Comments**: Can update serial_comment for existing serials
6. **Recalculates Total**: Updates total_amount
7. **Rebuilds Journal Entry**: Regenerates accounting entries

Example SQL Call:

```
-- Example: Update purchase invoice 101
CALL update_purchase_invoice(
    101,  -- purchase_invoice_id
    '[
        {
            "item_name": "iPhone 15 Pro",
            "qty": 3,  -- Changed from 2 to 3
            "unit_price": 44500.00,  -- Price updated
            "serials": [
                {"serial": "IMEI123456789", "comment": "Updated - Black 256GB"},
                {"serial": "IMEI987654321", "comment": "White 512GB"},
                {"serial": "IMEI111222333", "comment": "New unit - Blue 128GB"}
            ]
        }
    ]'::jsonb,
    'New Vendor Name',  -- Optional: change vendor
    '2026-02-11'  -- Optional: change date
);
```

Function 3: `delete_purchase`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.delete_purchase(p_invoice_id bigint)
RETURNS void
LANGUAGE plpgsql
```

```
AS $$
DECLARE
    v_journal_id BIGINT;
    v_validation JSONB;
BEGIN
    -- Validate before delete
    v_validation := validate_purchase_delete(p_invoice_id);

    IF (v_validation->>'is_valid')::BOOLEAN = FALSE THEN
        RAISE EXCEPTION '%', v_validation->>'message';
    END IF;

    -- Remove stock movements
    DELETE FROM StockMovements
    WHERE reference_type = 'PurchaseInvoice'
      AND reference_id = p_invoice_id;

    -- Remove journal entry
    SELECT journal_id INTO v_journal_id
    FROM PurchaseInvoices
    WHERE purchase_invoice_id = p_invoice_id;

    IF v_journal_id IS NOT NULL THEN
        DELETE FROM JournalLines WHERE journal_id = v_journal_id;
        DELETE FROM JournalEntries WHERE journal_id = v_journal_id;
    END IF;

    -- Delete invoice (cascades to items and units)
    DELETE FROM PurchaseInvoices
    WHERE purchase_invoice_id = p_invoice_id;
END;
$$;
```

Parameters:

- `p_invoice_id` BIGINT - Purchase invoice ID to delete

Returns:

`VOID`

Function Behavior:

1. **Validates Deletion**: Ensures no serials have been sold or returned
2. **Removes Stock Movements**: Deletes audit trail entries
3. **Removes Journal Entries**: Deletes accounting records
4. **Cascading Delete**: Removes PurchaseInvoice (auto-deletes PurchaseItems, PurchaseUnits)

Example SQL Call:

```
-- First validate
SELECT validate_purchase_delete(101);
-- Returns: {"is_valid": true, "message": "Safe to delete"}

-- Then delete
SELECT delete_purchase(101);
```

Function 4: `validate_purchase_update2()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.validate_purchase_update2(
    p_invoice_id BIGINT,
    p_items JSONB
) RETURNS JSONB
LANGUAGE plpgsql
AS $$
DECLARE
    v_existing_serials TEXT[];
    v_new_serials TEXT[];
    v_removed_serials TEXT[];
    v_sold_serials TEXT[];
    v_returned_serials TEXT[];
    v_message TEXT;
BEGIN
    -- 1️⃣ Existing serials in invoice
    SELECT ARRAY_AGG(pu.serial_number)
    INTO v_existing_serials
    FROM PurchaseUnits pu
    JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
    WHERE pi.purchase_invoice_id = p_invoice_id;

    IF v_existing_serials IS NULL THEN
        v_existing_serials := ARRAY[]::TEXT[];
    END IF;

    -- 2️⃣ Extract serials from NEW JSON (object format)
    SELECT ARRAY_AGG(serial_obj->>'serial')
    INTO v_new_serials
    FROM jsonb_array_elements(p_items) AS item,
         jsonb_array_elements(item->'serials') AS serial_obj;

    IF v_new_serials IS NULL THEN
        v_new_serials := ARRAY[]::TEXT[];
    END IF;

    -- 3️⃣ Identify removed serials
    SELECT ARRAY_AGG(s)
    INTO v_removed_serials
```

```
    FROM unnest(v_existing_serials) AS s
    WHERE s <> ALL(v_new_serials);

    IF v_removed_serials IS NULL THEN
        v_removed_serials := ARRAY[]::TEXT[];
    END IF;

    -- 4️⃣ Check SOLD serials
    SELECT ARRAY_AGG(pu.serial_number)
    INTO v_sold_serials
    FROM SoldUnits su
    JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
    WHERE pu.serial_number = ANY(v_removed_serials);

    IF v_sold_serials IS NULL THEN
        v_sold_serials := ARRAY[]::TEXT[];
    END IF;

    -- 5️⃣ Check RETURNED serials
    SELECT ARRAY_AGG(pri.serial_number)
    INTO v_returned_serials
    FROM PurchaseReturnItems pri
    WHERE pri.serial_number = ANY(v_removed_serials);

    IF v_returned_serials IS NULL THEN
        v_returned_serials := ARRAY[]::TEXT[];
    END IF;

    -- 6️⃣ Conflict check
    IF array_length(v_sold_serials, 1) IS NOT NULL
       OR array_length(v_returned_serials, 1) IS NOT NULL THEN

        v_message := '❌ Cannot update Purchase Invoice ' || p_invoice_id || '.';

        IF array_length(v_sold_serials, 1) IS NOT NULL THEN
            v_message := v_message || ' ' || array_length(v_sold_serials, 1) ||
                         ' serial(s) already sold cannot be removed.';
        END IF;

        IF array_length(v_returned_serials, 1) IS NOT NULL THEN
            v_message := v_message || ' ' || array_length(v_returned_serials, 1)
||
                         ' serial(s) already returned cannot be removed.';
        END IF;

        RETURN jsonb_build_object(
            'is_valid', FALSE,
            'message', v_message,
            'sold_serials', v_sold_serials,
            'returned_serials', v_returned_serials,
            'removed_serials', v_removed_serials
        );
    END IF;
```

```
    -- 7️⃣ Safe
    RETURN jsonb_build_object(
        'is_valid', TRUE,
        'message', '☑️ Safe to update — no sold or returned serials will be
removed.',
        'sold_serials', v_sold_serials,
        'returned_serials', v_returned_serials,
        'removed_serials', v_removed_serials
    );
END;
$$;
```

Parameters:

- `p_invoice_id BIGINT`
- `p_items JSONB`

Returns:

`JSONB`

Purpose:

Validate Purchase Update2 - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT validate_purchase_update2('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `validate_purchase_delete()`

Complete SQL Code:

```
CREATE FUNCTION public.validate_purchase_delete(p_invoice_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_invoice_serials TEXT[];
    v_sold_serials TEXT[];
    v_returned_serials TEXT[];
```

```
    v_message TEXT;
BEGIN
    -- 1️⃣ Get all serial numbers from this purchase invoice
    SELECT ARRAY_AGG(pu.serial_number)
    INTO v_invoice_serials
    FROM PurchaseUnits pu
    JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
    WHERE pi.purchase_invoice_id = p_invoice_id;

    IF v_invoice_serials IS NULL THEN
        v_invoice_serials := ARRAY[]::TEXT[];
    END IF;

    -- 2️⃣ Check if any of these serials are sold
    SELECT ARRAY_AGG(pu.serial_number)
    INTO v_sold_serials
    FROM SoldUnits su
    JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
    WHERE pu.serial_number = ANY(v_invoice_serials);

    IF v_sold_serials IS NULL THEN
        v_sold_serials := ARRAY[]::TEXT[];
    END IF;

    -- 3️⃣ Check if any of these serials are already returned to vendor
    SELECT ARRAY_AGG(pri.serial_number)
    INTO v_returned_serials
    FROM PurchaseReturnItems pri
    WHERE pri.serial_number = ANY(v_invoice_serials);

    IF v_returned_serials IS NULL THEN
        v_returned_serials := ARRAY[]::TEXT[];
    END IF;

    -- 4️⃣ If any sold or returned serials exist, prevent deletion
    IF array_length(v_sold_serials, 1) IS NOT NULL
       OR array_length(v_returned_serials, 1) IS NOT NULL THEN

        v_message := '❌ Purchase Invoice ' || p_invoice_id || ' cannot be
deleted.';

        IF array_length(v_sold_serials, 1) IS NOT NULL THEN
            v_message := v_message || ' ' || array_length(v_sold_serials, 1) || '
sold serial(s) found.';
        END IF;

        IF array_length(v_returned_serials, 1) IS NOT NULL THEN
            v_message := v_message || ' ' || array_length(v_returned_serials, 1)
|| ' returned serial(s) found.';
        END IF;

        RETURN jsonb_build_object(
            'is_valid', FALSE,
            'message', v_message,
```

```
            'sold_serials', v_sold_serials,
            'returned_serials', v_returned_serials
        );
    END IF;

    -- 5️⃣ Otherwise, safe to delete
    RETURN jsonb_build_object(
        'is_valid', TRUE,
        'message', '✅ Safe to delete — no sold or returned serials found in this
invoice.',
        'sold_serials', v_sold_serials,
        'returned_serials', v_returned_serials
    );
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`jsonb`

Purpose:

Validate Purchase Delete - Deletes a record after validation, cleaning up all related entries.

Example SQL Call:

```
SELECT validate_purchase_delete(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 6: `rebuild_purchase_journal()`

Complete SQL Code:

```
CREATE FUNCTION public.rebuild_purchase_journal(p_invoice_id bigint) RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    j_id BIGINT;
    inv_acc BIGINT;
```

```
    party_acc BIGINT;
    v_total NUMERIC(14,2);
BEGIN
    -- 1. Remove old journal if exists
    SELECT journal_id INTO j_id
    FROM PurchaseInvoices
    WHERE purchase_invoice_id = p_invoice_id;

    IF j_id IS NOT NULL THEN
        DELETE FROM JournalEntries WHERE journal_id = j_id;
    END IF;

    -- 2. Get accounts
    SELECT account_id INTO inv_acc FROM ChartOfAccounts WHERE
account_name='Inventory';
    SELECT ap_account_id INTO party_acc FROM Parties p
    JOIN PurchaseInvoices pi ON pi.vendor_id = p.party_id
    WHERE pi.purchase_invoice_id = p_invoice_id;

    -- 3. Get invoice total
    SELECT total_amount INTO v_total
    FROM PurchaseInvoices WHERE purchase_invoice_id = p_invoice_id;

    -- 4. Insert new journal entry
    INSERT INTO JournalEntries(entry_date, description)
    SELECT invoice_date, 'Purchase Invoice ' || purchase_invoice_id
    FROM PurchaseInvoices
    WHERE purchase_invoice_id = p_invoice_id
    RETURNING journal_id INTO j_id;

    -- 5. Update invoice with new journal_id
    UPDATE PurchaseInvoices
    SET journal_id = j_id
    WHERE purchase_invoice_id = p_invoice_id;

    -- 6. Debit Inventory
    INSERT INTO JournalLines(journal_id, account_id, debit)
    VALUES (j_id, inv_acc, v_total);

    -- 7. Credit Vendor (AP)
    INSERT INTO JournalLines(journal_id, account_id, party_id, credit)
    VALUES (j_id, party_acc, (
        SELECT vendor_id FROM PurchaseInvoices WHERE purchase_invoice_id =
p_invoice_id
    ), v_total);
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`void`

Purpose:

Rebuild Purchase Journal - Rebuilds accounting journal entries for a transaction.

Example SQL Call:

```
SELECT rebuild_purchase_journal(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 7: `get_last_purchase_id()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_purchase_id() RETURNS bigint
    LANGUAGE plpgsql
    AS $$
DECLARE
    last_id BIGINT;
BEGIN
    SELECT purchase_invoice_id
    INTO last_id
    FROM PurchaseInvoices
    ORDER BY purchase_invoice_id DESC
    LIMIT 1;

    RETURN last_id;
END;
$$;
```

Parameters:

- None

Returns:

`bigint`

Purpose:

Get Last Purchase Id - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_purchase_id();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 8: `get_purchase_summary()`

Complete SQL Code:

```
CREATE FUNCTION public.get_purchase_summary(p_start_date date DEFAULT NULL::date,
p_end_date date DEFAULT NULL::date) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSON;
BEGIN
    IF p_start_date IS NOT NULL AND p_end_date IS NOT NULL THEN
        -- 📄 Case 1: Purchases between given dates (latest first)
        SELECT json_agg(p ORDER BY p.invoice_date DESC)
        INTO result
        FROM (
            SELECT
                pi.purchase_invoice_id,
                pi.invoice_date,
                pa.party_name AS vendor,
                pi.total_amount
            FROM PurchaseInvoices pi
            JOIN Parties pa ON pi.vendor_id = pa.party_id
            WHERE pi.invoice_date BETWEEN p_start_date AND p_end_date
            ORDER BY pi.invoice_date DESC
        ) AS p;

    ELSE
        -- 📄 Case 2: Last 20 purchases (latest first)
        SELECT json_agg(p ORDER BY p.invoice_date DESC)
        INTO result
        FROM (
            SELECT
                pi.purchase_invoice_id,
                pi.invoice_date,
                pa.party_name AS vendor,
                pi.total_amount
            FROM PurchaseInvoices pi
```

```
            JOIN Parties pa ON pi.vendor_id = pa.party_id
            ORDER BY pi.invoice_date DESC
            LIMIT 20
        ) AS p;
    END IF;

    RETURN COALESCE(result, '[]'::json);
END;
$$;
```

Parameters:

- `p_start_date date DEFAULT NULL::date`
- `p_end_date date DEFAULT NULL::date`

Returns:

`json`

Purpose:

Get Purchase Summary - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_purchase_summary(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 9: `get_current_purchase()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.get_current_purchase(p_invoice_id bigint)
RETURNS json
LANGUAGE plpgsql
AS $$
DECLARE
    result JSON;
BEGIN
    SELECT json_build_object(
        'purchase_invoice_id', pi.purchase_invoice_id,
        'Party', p.party_name,
        'invoice_date', pi.invoice_date,
```

```
        'total_amount', pi.total_amount,
        'description', je.description,
        'items', (
            SELECT json_agg(
                json_build_object(
                    'item_name', i.item_name,
                    'qty', pi2.quantity,
                    'unit_price', pi2.unit_price,
                    'serials', (
                        SELECT json_agg(
                            json_build_object(
                                'serial', pu.serial_number,
                                'comment', pu.serial_comment
                            )
                        )
                        FROM PurchaseUnits pu
                        WHERE pu.purchase_item_id = pi2.purchase_item_id
                    )
                )
            )
            FROM PurchaseItems pi2
            JOIN Items i ON i.item_id = pi2.item_id
            WHERE pi2.purchase_invoice_id = pi.purchase_invoice_id
        )
    )
    INTO result
    FROM PurchaseInvoices pi
    JOIN Parties p ON p.party_id = pi.vendor_id
    LEFT JOIN JournalEntries je ON je.journal_id = pi.journal_id
    WHERE pi.purchase_invoice_id = p_invoice_id;

    RETURN result;
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`json`

Purpose:

Get Current Purchase - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_current_purchase(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 10: `get_next_purchase()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.get_next_purchase(p_invoice_id bigint)
RETURNS json
LANGUAGE plpgsql
AS $$
DECLARE
    next_id BIGINT;
BEGIN
    SELECT purchase_invoice_id INTO next_id
    FROM PurchaseInvoices
    WHERE purchase_invoice_id > p_invoice_id
    ORDER BY purchase_invoice_id ASC
    LIMIT 1;

    IF next_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_purchase(next_id);
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`json`

Purpose:

Get Next Purchase - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_next_purchase(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 11: `get_previous_purchase()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.get_previous_purchase(p_invoice_id bigint)
RETURNS json
LANGUAGE plpgsql
AS $$
DECLARE
    prev_id BIGINT;
BEGIN
    SELECT purchase_invoice_id INTO prev_id
    FROM PurchaseInvoices
    WHERE purchase_invoice_id < p_invoice_id
    ORDER BY purchase_invoice_id DESC
    LIMIT 1;

    IF prev_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_purchase(prev_id);
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`json`

Purpose:

Get Previous Purchase - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_previous_purchase(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 12: `get_last_purchase()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.get_last_purchase()
RETURNS json
LANGUAGE plpgsql
AS $$
DECLARE
    last_id BIGINT;
BEGIN
    SELECT purchase_invoice_id INTO last_id
    FROM PurchaseInvoices
    ORDER BY purchase_invoice_id DESC
    LIMIT 1;

    RETURN get_current_purchase(last_id);
END;
$$;
```

Parameters:

- None

Returns:

`json`

Purpose:

Get Last Purchase - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_purchase();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

💰 SALES FUNCTIONS

Total Functions: 12

Function 1: `create_sale()`

Complete SQL Code:

```
CREATE FUNCTION public.create_sale(p_party_id bigint, p_invoice_date date, p_items
jsonb) RETURNS bigint
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_invoice_id BIGINT;
    v_sales_item_id BIGINT;
    v_total NUMERIC(14,2) := 0;
    v_unit_id BIGINT;
    v_serial TEXT;
    v_item_id BIGINT;
    v_item JSONB;
BEGIN
    -- 1. Create Invoice (header)
    INSERT INTO SalesInvoices(customer_id, invoice_date, total_amount)
    VALUES (p_party_id, p_invoice_date, 0)
    RETURNING sales_invoice_id INTO v_invoice_id;

    -- 2. Loop through items
    FOR v_item IN SELECT * FROM jsonb_array_elements(p_items)
    LOOP
        -- Resolve item_id from item_name
        SELECT item_id INTO v_item_id
        FROM Items
        WHERE item_name = (v_item->>'item_name')
        LIMIT 1;

        IF v_item_id IS NULL THEN
            RAISE EXCEPTION 'Item "%" not found in Items table', (v_item-
>>'item_name');
        END IF;

        -- Insert sales item
        INSERT INTO SalesItems(sales_invoice_id, item_id, quantity, unit_price)
        VALUES (
            v_invoice_id,
            v_item_id,
            (v_item->>'qty')::INT,
            (v_item->>'unit_price')::NUMERIC
        )
        RETURNING sales_item_id INTO v_sales_item_id;

        -- Accumulate total
        v_total := v_total + ((v_item->>'qty')::INT * (v_item-
>>'unit_price')::NUMERIC);

        -- Insert sold units from serials
```

```
        FOR v_serial IN SELECT jsonb_array_elements_text(v_item->'serials')
        LOOP
            -- find unit_id for this serial
            SELECT unit_id INTO v_unit_id
            FROM PurchaseUnits
            WHERE serial_number = v_serial
              AND in_stock = TRUE
            LIMIT 1;

            IF v_unit_id IS NULL THEN
                RAISE EXCEPTION 'Serial % not found or already sold', v_serial;
            END IF;

            -- insert sold unit
            INSERT INTO SoldUnits(sales_item_id, unit_id, sold_price, status)
            VALUES (v_sales_item_id, v_unit_id, (v_item->>'unit_price')::NUMERIC,
'Sold');

            -- mark purchase unit as not in stock
            UPDATE PurchaseUnits
            SET in_stock = FALSE
            WHERE unit_id = v_unit_id;

            -- log stock movement (OUT)
            INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
            VALUES (v_item_id, v_serial, 'OUT', 'SalesInvoice', v_invoice_id, 1);
        END LOOP;
    END LOOP;

    -- 3. Update invoice total
    UPDATE SalesInvoices
    SET total_amount = v_total
    WHERE sales_invoice_id = v_invoice_id;

    -- 4. Build Journal Entry (explicit, no trigger needed)
    PERFORM rebuild_sales_journal(v_invoice_id);

    RETURN v_invoice_id;
END;
$$;
```

Parameters:

- `p_party_id bigint`
- `p_invoice_date date`
- `p_items jsonb`

Returns:

`bigint`

Purpose:

Create Sale - Creates a new record in the database with all related entries and accounting journal.

Example SQL Call:

```
SELECT create_sale('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `delete_sale()`

Complete SQL Code:

```
CREATE FUNCTION public.delete_sale(p_invoice_id bigint) RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    rec RECORD;
    v_journal_id BIGINT;
BEGIN
    -- 1. Restore stock for all sold units of this sale
    FOR rec IN
        SELECT su.unit_id, pu.serial_number, si.item_id
        FROM SoldUnits su
        JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
        JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
        WHERE si.sales_invoice_id = p_invoice_id
    LOOP
        -- restore stock
        UPDATE PurchaseUnits
        SET in_stock = TRUE
        WHERE unit_id = rec.unit_id;

        -- log stock movement (IN)
        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (rec.item_id, rec.serial_number, 'IN', 'SalesInvoice-Delete',
p_invoice_id, 1);
    END LOOP;

    -- 2. Delete associated journal entries (accounting)
    SELECT journal_id INTO v_journal_id
    FROM SalesInvoices
```

```
    WHERE sales_invoice_id = p_invoice_id;

    IF v_journal_id IS NOT NULL THEN
        DELETE FROM JournalLines WHERE journal_id = v_journal_id;
        DELETE FROM JournalEntries WHERE journal_id = v_journal_id;
    END IF;

    -- 3. Delete the invoice (cascade removes SalesItems + SoldUnits)
    DELETE FROM SalesInvoices
    WHERE sales_invoice_id = p_invoice_id;

END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`void`

Purpose:

Delete Sale - Deletes a record after validation, cleaning up all related entries.

Example SQL Call:

```
SELECT delete_sale(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `validate_sales_delete()`

Complete SQL Code:

```
CREATE FUNCTION public.validate_sales_delete(p_invoice_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_invoice_serials TEXT[];
    v_returned_serials TEXT[];
    v_message TEXT;
BEGIN
```

```
    -- 1️⃣ Get all serials belonging to this Sales Invoice
    SELECT ARRAY_AGG(pu.serial_number)
    INTO v_invoice_serials
    FROM SoldUnits su
    JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
    JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
    WHERE si.sales_invoice_id = p_invoice_id;

    IF v_invoice_serials IS NULL THEN
        v_invoice_serials := ARRAY[]::TEXT[];
    END IF;

    -- 2️⃣ Check which of these serials are already returned
    SELECT ARRAY_AGG(sri.serial_number)
    INTO v_returned_serials
    FROM SalesReturnItems sri
    WHERE sri.serial_number = ANY(v_invoice_serials);

    IF v_returned_serials IS NULL THEN
        v_returned_serials := ARRAY[]::TEXT[];
    END IF;

    -- 3️⃣ If any serials are returned, block deletion
    IF array_length(v_returned_serials, 1) IS NOT NULL THEN
        v_message := '❌ Cannot delete Sales Invoice ' || p_invoice_id ||
                     ' — ' || array_length(v_returned_serials, 1) ||
                     ' serial(s) already returned.';

        RETURN jsonb_build_object(
            'is_valid', FALSE,
            'message', v_message,
            'returned_serials', v_returned_serials
        );
    END IF;

    -- 4️⃣ Otherwise, safe to delete
    RETURN jsonb_build_object(
        'is_valid', TRUE,
        'message', '☑️ Safe to delete — no returned serials found.',
        'returned_serials', v_returned_serials
    );
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`jsonb`

Purpose:

Validate Sales Delete - Deletes a record after validation, cleaning up all related entries.

Example SQL Call:

```
SELECT validate_sales_delete(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: update_sale_invoice()

Complete SQL Code:

```
CREATE FUNCTION public.update_sale_invoice(p_invoice_id bigint, p_items jsonb,
p_party_name text DEFAULT NULL::text, p_invoice_date date DEFAULT NULL::date)
RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_item JSONB;
    v_item_id BIGINT;
    v_total NUMERIC(14,2) := 0;
    v_sales_item_id BIGINT;
    v_serial TEXT;
    v_unit_id BIGINT;
    v_new_party_id BIGINT;
BEGIN
    -- ============================================================
    -- 1️⃣ Update Party (Customer) if given
    -- ============================================================
    IF p_party_name IS NOT NULL THEN
        SELECT party_id INTO v_new_party_id
        FROM Parties
        WHERE party_name = p_party_name
        LIMIT 1;

        IF v_new_party_id IS NULL THEN
            RAISE EXCEPTION 'Customer "%" not found in Parties table.',
p_party_name;
        END IF;

        UPDATE SalesInvoices
        SET customer_id = v_new_party_id
        WHERE sales_invoice_id = p_invoice_id;
    END IF;
```

```
    -- ============================================================
    -- 2️⃣ Update Invoice Date (if provided)
    -- ============================================================
    IF p_invoice_date IS NOT NULL THEN
        UPDATE SalesInvoices
        SET invoice_date = p_invoice_date
        WHERE sales_invoice_id = p_invoice_id;
    END IF;

    -- ============================================================
    -- 3️⃣ Delete old items + sold units + stock movements
    -- ============================================================
    DELETE FROM StockMovements
    WHERE reference_type = 'SalesInvoice' AND reference_id = p_invoice_id;

    DELETE FROM SoldUnits
    WHERE sales_item_id IN (
        SELECT sales_item_id FROM SalesItems WHERE sales_invoice_id = p_invoice_id
    );

    DELETE FROM SalesItems
    WHERE sales_invoice_id = p_invoice_id;

    -- ============================================================
    -- 4️⃣ Insert new/updated items and serials
    -- ============================================================
    FOR v_item IN SELECT * FROM jsonb_array_elements(p_items)
    LOOP
        -- Find item_id
        SELECT item_id INTO v_item_id
        FROM Items
        WHERE item_name = (v_item->>'item_name')
        LIMIT 1;

        IF v_item_id IS NULL THEN
            RAISE EXCEPTION 'Item "%" not found in Items table for
update_sale_invoice', (v_item->>'item_name');
        END IF;

        -- Insert sales item
        INSERT INTO SalesItems(sales_invoice_id, item_id, quantity, unit_price)
        VALUES (
            p_invoice_id,
            v_item_id,
            (v_item->>'qty')::INT,
            (v_item->>'unit_price')::NUMERIC
        )
        RETURNING sales_item_id INTO v_sales_item_id;

        -- Add to total
        v_total := v_total + ((v_item->>'qty')::INT * (v_item-
>>'unit_price')::NUMERIC);
```

```
        -- Insert sold units + stock movements
        FOR v_serial IN SELECT jsonb_array_elements_text(v_item->'serials')
        LOOP
            -- get matching purchase unit
            SELECT unit_id INTO v_unit_id
            FROM PurchaseUnits
            WHERE serial_number = v_serial
            LIMIT 1;

            IF v_unit_id IS NULL THEN
                RAISE EXCEPTION 'Serial % not found in PurchaseUnits', v_serial;
            END IF;

            -- mark unit as sold (in_stock = FALSE)
            UPDATE PurchaseUnits
            SET in_stock = FALSE
            WHERE unit_id = v_unit_id;

            -- insert into SoldUnits
            INSERT INTO SoldUnits(sales_item_id, unit_id, sold_price, status)
            VALUES (v_sales_item_id, v_unit_id, (v_item->>'unit_price')::NUMERIC,
'Sold');

            -- log stock OUT
            INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
            VALUES (v_item_id, v_serial, 'OUT', 'SalesInvoice', p_invoice_id, 1);
        END LOOP;
    END LOOP;

    -- ============================================================
    -- 5️⃣ Update total amount
    -- ============================================================
    UPDATE SalesInvoices
    SET total_amount = v_total
    WHERE sales_invoice_id = p_invoice_id;

    -- ============================================================
    -- 6️⃣ Rebuild journal (refreshes AR, Revenue, COGS, Inventory)
    -- ============================================================
    PERFORM rebuild_sales_journal(p_invoice_id);

END;
$$;
```

Parameters:

- `p_invoice_id bigint`
- `p_items jsonb`
- `p_party_name text DEFAULT NULL::text`
- `p_invoice_date date DEFAULT NULL::date`

Returns:

`void`

Purpose:

Update Sale Invoice - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT update_sale_invoice('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `validate_sales_update()`

Complete SQL Code:

```
CREATE FUNCTION public.validate_sales_update(p_invoice_id bigint, p_items jsonb)
RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_existing_serials TEXT[];
    v_new_serials TEXT[];
    v_removed_serials TEXT[];
    v_returned_serials TEXT[];
    v_message TEXT;
BEGIN
    -- 1 Get all serials currently in this sales invoice
    SELECT ARRAY_AGG(pu.serial_number)
    INTO v_existing_serials
    FROM SoldUnits su
    JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
    JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
    WHERE si.sales_invoice_id = p_invoice_id;

    IF v_existing_serials IS NULL THEN
        v_existing_serials := ARRAY[]::TEXT[];
    END IF;

    -- 2 Extract all serials from the new JSON data (flatten correctly)
    SELECT ARRAY_AGG(serial::TEXT)
    INTO v_new_serials
```

```
    FROM jsonb_array_elements(p_items) AS item,
         jsonb_array_elements_text(item->'serials') AS serial;

    IF v_new_serials IS NULL THEN
        v_new_serials := ARRAY[]::TEXT[];
    END IF;

    -- 3️⃣ Find removed serials (those that existed before but not now)
    SELECT ARRAY_AGG(s)
    INTO v_removed_serials
    FROM unnest(v_existing_serials) AS s
    WHERE s <> ALL(v_new_serials);

    IF v_removed_serials IS NULL THEN
        v_removed_serials := ARRAY[]::TEXT[];
    END IF;

    -- 4️⃣ Check if removed serials are already in Sales Return
    SELECT ARRAY_AGG(sri.serial_number)
    INTO v_returned_serials
    FROM SalesReturnItems sri
    WHERE sri.serial_number = ANY(v_removed_serials);

    IF v_returned_serials IS NULL THEN
        v_returned_serials := ARRAY[]::TEXT[];
    END IF;

    -- 5️⃣ If any conflicts found, return descriptive message
    IF array_length(v_returned_serials, 1) IS NOT NULL THEN
        v_message := '❌ Some serials cannot be removed. ' ||
                     array_length(v_returned_serials, 1) || ' serial(s) already
returned.';

        RETURN jsonb_build_object(
            'is_valid', FALSE,
            'message', v_message,
            'returned_serials', v_returned_serials
        );
    END IF;

    -- 6️⃣ Otherwise, all safe
    RETURN jsonb_build_object(
        'is_valid', TRUE,
        'message', '☑️ Safe to update — no returned serials will be removed.',
        'returned_serials', v_returned_serials
    );
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

- `p_items jsonb`

Returns:

`jsonb`

Purpose:

Validate Sales Update - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT validate_sales_update('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 6: `rebuild_sales_journal()`

Complete SQL Code:

```
CREATE FUNCTION public.rebuild_sales_journal(p_invoice_id bigint) RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    j_id BIGINT;
    rev_acc BIGINT;
    party_acc BIGINT;
    cogs_acc BIGINT;
    inv_acc BIGINT;
    total_cost NUMERIC(14,2);
    total_revenue NUMERIC(14,2);
    v_customer_id BIGINT;
    v_invoice_date DATE;
BEGIN
    -- 1. Get existing journal_id (if any)
    SELECT journal_id INTO j_id
    FROM SalesInvoices
    WHERE sales_invoice_id = p_invoice_id;

    -- 2. If exists, clear old lines + entry
    IF j_id IS NOT NULL THEN
        DELETE FROM JournalLines WHERE journal_id = j_id;
        DELETE FROM JournalEntries WHERE journal_id = j_id;
    END IF;
```

```
    -- 3. Get invoice details
    SELECT s.customer_id, s.total_amount, s.invoice_date
    INTO v_customer_id, total_revenue, v_invoice_date
    FROM SalesInvoices s
    WHERE s.sales_invoice_id = p_invoice_id;

    -- 4. Get accounts
    SELECT account_id INTO rev_acc FROM ChartOfAccounts WHERE account_name='Sales
Revenue';
    SELECT account_id INTO cogs_acc FROM ChartOfAccounts WHERE account_name='Cost
of Goods Sold';
    SELECT account_id INTO inv_acc FROM ChartOfAccounts WHERE
account_name='Inventory';
    SELECT ar_account_id INTO party_acc FROM Parties WHERE party_id =
v_customer_id;

    -- 5. Insert new journal entry
    INSERT INTO JournalEntries(entry_date, description)
    VALUES (v_invoice_date, 'Sale Invoice ' || p_invoice_id)
    RETURNING journal_id INTO j_id;

    -- 6. Update invoice with new journal_id
    UPDATE SalesInvoices
    SET journal_id = j_id
    WHERE sales_invoice_id = p_invoice_id;

    -- (1) Debit Customer (AR)
    INSERT INTO JournalLines(journal_id, account_id, party_id, debit)
    VALUES (j_id, party_acc, v_customer_id, total_revenue);

    -- (2) Credit Revenue
    INSERT INTO JournalLines(journal_id, account_id, credit)
    VALUES (j_id, rev_acc, total_revenue);

    -- (3) Debit COGS / Credit Inventory
    SELECT COALESCE(SUM(pi.unit_price),0) INTO total_cost
    FROM SoldUnits su
    JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
    JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
    JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
    WHERE si.sales_invoice_id = p_invoice_id;

    IF total_cost > 0 THEN
        INSERT INTO JournalLines(journal_id, account_id, debit)
        VALUES (j_id, cogs_acc, total_cost);

        INSERT INTO JournalLines(journal_id, account_id, credit)
        VALUES (j_id, inv_acc, total_cost);
    END IF;
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`void`

Purpose:

Rebuild Sales Journal - Rebuilds accounting journal entries for a transaction.

Example SQL Call:

```
SELECT rebuild_sales_journal(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 7: `get_current_sale()`

Complete SQL Code:

```
CREATE FUNCTION public.get_current_sale(p_invoice_id bigint) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSON;
BEGIN
    SELECT json_build_object(
        'sales_invoice_id', si.sales_invoice_id,
        'Party', p.party_name,
        'invoice_date', si.invoice_date,
        'total_amount', si.total_amount,
        'description', je.description,
        'items', (
            SELECT json_agg(
                json_build_object(
                    'item_name', i.item_name,
                    'qty', s_items.quantity,
                    'unit_price', s_items.unit_price,
                    'serials', (
                        SELECT json_agg(pu.serial_number)
                        FROM SoldUnits su
                        JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
                        WHERE su.sales_item_id = s_items.sales_item_id
                    )
```

```
                )
            )
            FROM SalesItems s_items
            JOIN Items i ON i.item_id = s_items.item_id
            WHERE s_items.sales_invoice_id = si.sales_invoice_id
        )
    )
    INTO result
    FROM SalesInvoices si
    JOIN Parties p ON p.party_id = si.customer_id
    LEFT JOIN JournalEntries je ON je.journal_id = si.journal_id
    WHERE si.sales_invoice_id = p_invoice_id;

    RETURN result;
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`json`

Purpose:

Get Current Sale - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_current_sale(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 8: `get_last_sale()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_sale() RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    last_id BIGINT;
```

```
BEGIN
    SELECT sales_invoice_id INTO last_id
    FROM SalesInvoices
    ORDER BY sales_invoice_id DESC
    LIMIT 1;

    RETURN get_current_sale(last_id);
END;
$$;
```

Parameters:

- None

Returns:

`json`

Purpose:

Get Last Sale - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_sale();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 9: `get_last_sale_id()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_sale_id() RETURNS bigint
    LANGUAGE plpgsql
    AS $$
DECLARE
    last_id BIGINT;
BEGIN
    SELECT sales_invoice_id
    INTO last_id
    FROM SalesInvoices
    ORDER BY sales_invoice_id DESC
    LIMIT 1;
```

```
    RETURN last_id;
END;
$$;
```

Parameters:

- None

Returns:

`bigint`

Purpose:

Get Last Sale Id - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_sale_id();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 10: `get_next_sale()`

Complete SQL Code:

```
CREATE FUNCTION public.get_next_sale(p_invoice_id bigint) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    next_id BIGINT;
BEGIN
    SELECT sales_invoice_id INTO next_id
    FROM SalesInvoices
    WHERE sales_invoice_id > p_invoice_id
    ORDER BY sales_invoice_id ASC
    LIMIT 1;

    IF next_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_sale(next_id);
```

```
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`json`

Purpose:

Get Next Sale - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_next_sale(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 11: `get_previous_sale()`

Complete SQL Code:

```
CREATE FUNCTION public.get_previous_sale(p_invoice_id bigint) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    prev_id BIGINT;
BEGIN
    SELECT sales_invoice_id INTO prev_id
    FROM SalesInvoices
    WHERE sales_invoice_id < p_invoice_id
    ORDER BY sales_invoice_id DESC
    LIMIT 1;

    IF prev_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_sale(prev_id);
END;
$$;
```

Parameters:

- `p_invoice_id bigint`

Returns:

`json`

Purpose:

Get Previous Sale - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_previous_sale(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 12: `get_sales_summary()`

Complete SQL Code:

```
CREATE FUNCTION public.get_sales_summary(p_start_date date DEFAULT NULL::date,
p_end_date date DEFAULT NULL::date) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSON;
BEGIN
    IF p_start_date IS NOT NULL AND p_end_date IS NOT NULL THEN
        -- 📄 Case 1: Sales between given dates (latest first)
        SELECT json_agg(p ORDER BY p.invoice_date DESC)
        INTO result
        FROM (
            SELECT
                si.sales_invoice_id,
                si.invoice_date,
                pa.party_name AS customer,
                si.total_amount
            FROM SalesInvoices si
            JOIN Parties pa ON si.customer_id = pa.party_id
            WHERE si.invoice_date BETWEEN p_start_date AND p_end_date
            ORDER BY si.invoice_date DESC
        ) AS p;
```

```
    ELSE
        -- 📄 Case 2: Last 20 sales (latest first)
        SELECT json_agg(p ORDER BY p.invoice_date DESC)
        INTO result
        FROM (
            SELECT
                si.sales_invoice_id,
                si.invoice_date,
                pa.party_name AS customer,
                si.total_amount
            FROM SalesInvoices si
            JOIN Parties pa ON si.customer_id = pa.party_id
            ORDER BY si.invoice_date DESC
            LIMIT 20
        ) AS p;
    END IF;

    RETURN COALESCE(result, '[]'::json);
END;
$$;
```

Parameters:

- `p_start_date date DEFAULT NULL::date`
- `p_end_date date DEFAULT NULL::date`

Returns:

`json`

Purpose:

Get Sales Summary - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_sales_summary(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

↩ PURCHASE RETURN FUNCTIONS

Total Functions: 11

Function 1: `create_purchase_return()`

Complete SQL Code:

```
CREATE FUNCTION public.create_purchase_return(p_party_name text, p_serials jsonb)
RETURNS bigint
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_party_id BIGINT;
    v_return_id BIGINT;
    v_serial TEXT;
    v_unit RECORD;
    v_total NUMERIC(14,2) := 0;
BEGIN
    -- 1. Find Vendor
    SELECT party_id INTO v_party_id
    FROM Parties
    WHERE party_name = p_party_name;

    IF v_party_id IS NULL THEN
        RAISE EXCEPTION 'Vendor % not found', p_party_name;
    END IF;

    -- 2. Create Return Header
    INSERT INTO PurchaseReturns(vendor_id, return_date, total_amount)
    VALUES (v_party_id, CURRENT_DATE, 0)
    RETURNING purchase_return_id INTO v_return_id;

    -- 3. Process each serial
    FOR v_serial IN SELECT jsonb_array_elements_text(p_serials)
    LOOP
        SELECT pu.unit_id, pu.serial_number, pi.item_id, pi.unit_price,
p.vendor_id, p.purchase_invoice_id
        INTO v_unit
        FROM PurchaseUnits pu
        JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
        JOIN PurchaseInvoices p ON pi.purchase_invoice_id = p.purchase_invoice_id
        WHERE pu.serial_number = v_serial;

        IF NOT FOUND THEN
            RAISE EXCEPTION 'Serial % not found in PurchaseUnits', v_serial;
        END IF;

        -- check if in stock
        IF NOT EXISTS (
            SELECT 1 FROM PurchaseUnits WHERE unit_id = v_unit.unit_id AND
in_stock = TRUE
        ) THEN
            RAISE EXCEPTION 'Serial % is not currently in stock', v_serial;
        END IF;
```

```
        -- check vendor match
        IF v_unit.vendor_id <> v_party_id THEN
            RAISE EXCEPTION 'Serial % was purchased from a different vendor',
v_serial;
        END IF;

        -- mark as returned (remove from stock)
        UPDATE PurchaseUnits
        SET in_stock = FALSE
        WHERE unit_id = v_unit.unit_id;

        -- log stock OUT
        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (v_unit.item_id, v_serial, 'OUT', 'PurchaseReturn', v_return_id,
1);

        -- insert return line (☑ unit_price instead of cost_price)
        INSERT INTO PurchaseReturnItems(purchase_return_id, item_id, unit_price,
serial_number)
        VALUES (v_return_id, v_unit.item_id, v_unit.unit_price, v_serial);

        -- accumulate total
        v_total := v_total + v_unit.unit_price;
    END LOOP;

    -- 4. Update header total
    UPDATE PurchaseReturns
    SET total_amount = v_total
    WHERE purchase_return_id = v_return_id;

    -- 5. Build Journal
    PERFORM rebuild_purchase_return_journal(v_return_id);

    RETURN v_return_id;
END;
$$;
```

Parameters:

- `p_party_name text`
- `p_serials jsonb`

Returns:

`bigint`

Purpose:

Create Purchase Return - Creates a new record in the database with all related entries and accounting journal.

Example SQL Call:

```
SELECT create_purchase_return('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `delete_purchase_return()`

Complete SQL Code:

```
CREATE FUNCTION public.delete_purchase_return(p_return_id bigint) RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    rec RECORD;
    v_vendor_id BIGINT;
    v_unit_vendor_id BIGINT;
    v_journal_id BIGINT;
BEGIN
    -- 1. Get vendor id from return header
    SELECT vendor_id, journal_id
    INTO v_vendor_id, v_journal_id
    FROM PurchaseReturns
    WHERE purchase_return_id = p_return_id;

    IF v_vendor_id IS NULL THEN
        RAISE EXCEPTION 'Purchase Return % not found', p_return_id;
    END IF;

    -- 2. Restore stock for returned items
    FOR rec IN
        SELECT serial_number, item_id
        FROM PurchaseReturnItems
        WHERE purchase_return_id = p_return_id
    LOOP
        -- fetch the vendor of the original purchase for safety
        SELECT p.vendor_id
        INTO v_unit_vendor_id
        FROM PurchaseUnits pu
        JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
        JOIN PurchaseInvoices p ON pi.purchase_invoice_id = p.purchase_invoice_id
        WHERE pu.serial_number = rec.serial_number;

        IF v_unit_vendor_id IS DISTINCT FROM v_vendor_id THEN
```

```
            RAISE EXCEPTION 'Serial % does not belong to vendor % (return %)',
                rec.serial_number, v_vendor_id, p_return_id;
        END IF;

        -- restore in stock
        UPDATE PurchaseUnits
        SET in_stock = TRUE
        WHERE serial_number = rec.serial_number;

        -- log stock IN
        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (rec.item_id, rec.serial_number, 'IN', 'PurchaseReturn-Delete',
p_return_id, 1);
    END LOOP;

    -- 3. Remove journal (if exists)
    IF v_journal_id IS NOT NULL THEN
        DELETE FROM JournalEntries WHERE journal_id = v_journal_id;
    END IF;

    -- 4. Delete return items
    DELETE FROM PurchaseReturnItems WHERE purchase_return_id = p_return_id;

    -- 5. Delete return header
    DELETE FROM PurchaseReturns WHERE purchase_return_id = p_return_id;
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`void`

Purpose:

Delete Purchase Return - Deletes a record after validation, cleaning up all related entries.

Example SQL Call:

```
SELECT delete_purchase_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: update_purchase_return()

Complete SQL Code:

```
CREATE FUNCTION public.update_purchase_return(p_return_id bigint, p_serials jsonb)
RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    rec RECORD;
    v_serial TEXT;
    v_unit RECORD;
    v_total NUMERIC(14,2) := 0;
    v_vendor_id BIGINT;
BEGIN
    -- 1. Get vendor id from return header
    SELECT vendor_id INTO v_vendor_id
    FROM PurchaseReturns
    WHERE purchase_return_id = p_return_id;

    IF v_vendor_id IS NULL THEN
        RAISE EXCEPTION 'Purchase Return % not found', p_return_id;
    END IF;

    -- 2. Reverse old items (restore stock)
    FOR rec IN
        SELECT serial_number, item_id
        FROM PurchaseReturnItems
        WHERE purchase_return_id = p_return_id
    LOOP
        UPDATE PurchaseUnits
        SET in_stock = TRUE
        WHERE serial_number = rec.serial_number;

        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (rec.item_id, rec.serial_number, 'IN', 'PurchaseReturn-Update-
Reverse', p_return_id, 1);
    END LOOP;

    -- 3. Remove old items
    DELETE FROM PurchaseReturnItems WHERE purchase_return_id = p_return_id;

    -- 4. Insert new items
    FOR v_serial IN SELECT jsonb_array_elements_text(p_serials)
    LOOP
        SELECT pu.unit_id, pu.serial_number, pi.item_id, pi.unit_price,
p.vendor_id
        INTO v_unit
        FROM PurchaseUnits pu
        JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
        JOIN PurchaseInvoices p ON pi.purchase_invoice_id = p.purchase_invoice_id
```

```
        WHERE pu.serial_number = v_serial;

        IF NOT FOUND THEN
            RAISE EXCEPTION 'Serial % not found in PurchaseUnits', v_serial;
        END IF;

        -- check if in stock
        IF NOT EXISTS (
            SELECT 1 FROM PurchaseUnits WHERE unit_id = v_unit.unit_id AND
in_stock = TRUE
        ) THEN
            RAISE EXCEPTION 'Serial % is not currently in stock', v_serial;
        END IF;

        -- check vendor match
        IF v_unit.vendor_id <> v_vendor_id THEN
            RAISE EXCEPTION 'Serial % was purchased from a different vendor',
v_serial;
        END IF;

        -- mark as returned (out of stock)
        UPDATE PurchaseUnits
        SET in_stock = FALSE
        WHERE unit_id = v_unit.unit_id;

        -- log stock OUT
        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (v_unit.item_id, v_serial, 'OUT', 'PurchaseReturn-Update',
p_return_id, 1);

        -- insert return line (☑ unit_price instead of cost_price)
        INSERT INTO PurchaseReturnItems(purchase_return_id, item_id, unit_price,
serial_number)
        VALUES (p_return_id, v_unit.item_id, v_unit.unit_price, v_serial);

        v_total := v_total + v_unit.unit_price;
    END LOOP;

    -- 5. Update header total
    UPDATE PurchaseReturns
    SET total_amount = v_total
    WHERE purchase_return_id = p_return_id;

    -- 6. Rebuild journal
    PERFORM rebuild_purchase_return_journal(p_return_id);
END;
$$;
```

Parameters:

- `p_return_id bigint`

- `p_serials jsonb`

Returns:

`void`

Purpose:

Update Purchase Return - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT update_purchase_return('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: `get_current_purchase_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_current_purchase_return(p_return_id bigint) RETURNS
json
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSON;
BEGIN
    SELECT json_build_object(
        'purchase_return_id', pr.purchase_return_id,
        'Vendor', pa.party_name,
        'return_date', pr.return_date,
        'total_amount', pr.total_amount,
        'description', je.description,
        'items', (
            SELECT json_agg(
                json_build_object(
                    'item_name', i.item_name,
                    'unit_price', pri.unit_price,
                    'serial_number', pri.serial_number
                )
            )
            FROM PurchaseReturnItems pri
            JOIN Items i ON i.item_id = pri.item_id
            WHERE pri.purchase_return_id = pr.purchase_return_id
```

```
        )
    )
    INTO result
    FROM PurchaseReturns pr
    JOIN Parties pa ON pa.party_id = pr.vendor_id
    LEFT JOIN JournalEntries je ON je.journal_id = pr.journal_id
    WHERE pr.purchase_return_id = p_return_id;

    RETURN result;
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`json`

Purpose:

Get Current Purchase Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_current_purchase_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `get_last_purchase_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_purchase_return() RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    last_id BIGINT;
BEGIN
    SELECT purchase_return_id INTO last_id
    FROM PurchaseReturns
    ORDER BY purchase_return_id DESC
    LIMIT 1;
```

```
    RETURN get_current_purchase_return(last_id);
END;
$$;
```

Parameters:

- None

Returns:

`json`

Purpose:

Get Last Purchase Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_purchase_return();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 6: `get_last_purchase_return_id()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_purchase_return_id() RETURNS bigint
    LANGUAGE plpgsql
    AS $$
DECLARE
    last_id BIGINT;
BEGIN
    SELECT purchase_return_id
    INTO last_id
    FROM PurchaseReturns
    ORDER BY purchase_return_id DESC
    LIMIT 1;

    RETURN last_id;
END;
$$;
```

Parameters:

- None

Returns:

`bigint`

Purpose:

Get Last Purchase Return Id - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_purchase_return_id();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 7: `get_next_purchase_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_next_purchase_return(p_return_id bigint) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    next_id BIGINT;
BEGIN
    SELECT purchase_return_id INTO next_id
    FROM PurchaseReturns
    WHERE purchase_return_id > p_return_id
    ORDER BY purchase_return_id ASC
    LIMIT 1;

    IF next_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_purchase_return(next_id);
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`json`

Purpose:

Get Next Purchase Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_next_purchase_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 8: `get_previous_purchase_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_previous_purchase_return(p_return_id bigint) RETURNS
json
    LANGUAGE plpgsql
    AS $$
DECLARE
    prev_id BIGINT;
BEGIN
    SELECT purchase_return_id INTO prev_id
    FROM PurchaseReturns
    WHERE purchase_return_id < p_return_id
    ORDER BY purchase_return_id DESC
    LIMIT 1;

    IF prev_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_purchase_return(prev_id);
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`json`

Purpose:

Get Previous Purchase Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_previous_purchase_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 9: `get_purchase_return_summary()`

Complete SQL Code:

```
CREATE FUNCTION public.get_purchase_return_summary(p_start_date date DEFAULT
NULL::date, p_end_date date DEFAULT NULL::date) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSON;
BEGIN
    IF p_start_date IS NOT NULL AND p_end_date IS NOT NULL THEN
        -- 📄 Case 1: Returns between given dates (latest first)
        SELECT json_agg(p ORDER BY p.return_date DESC)
        INTO result
        FROM (
            SELECT
                pr.purchase_return_id,
                pr.return_date,
                pa.party_name AS vendor,
                pr.total_amount
            FROM PurchaseReturns pr
            JOIN Parties pa ON pr.vendor_id = pa.party_id
            WHERE pr.return_date BETWEEN p_start_date AND p_end_date
            ORDER BY pr.return_date DESC
        ) AS p;

    ELSE
        -- 📄 Case 2: Last 20 purchase returns (latest first)
        SELECT json_agg(p ORDER BY p.return_date DESC)
        INTO result
        FROM (
```

```
            SELECT
                pr.purchase_return_id,
                pr.return_date,
                pa.party_name AS vendor,
                pr.total_amount
            FROM PurchaseReturns pr
            JOIN Parties pa ON pr.vendor_id = pa.party_id
            ORDER BY pr.return_date DESC
            LIMIT 20
        ) AS p;
    END IF;

    RETURN COALESCE(result, '[]'::json);
END;
$$;
```

Parameters:

- `p_start_date date DEFAULT NULL::date`
- `p_end_date date DEFAULT NULL::date`

Returns:

`json`

Purpose:

Get Purchase Return Summary - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_purchase_return_summary(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 10: `serial_exists_in_purchase_return()`

Complete SQL Code:

```
CREATE FUNCTION public.serial_exists_in_purchase_return(p_purchase_return_id
bigint, p_serial_number text) RETURNS boolean
    LANGUAGE plpgsql
    AS $$
DECLARE
```

```
    v_exists BOOLEAN;
BEGIN
    SELECT TRUE
    INTO v_exists
    FROM PurchaseReturnItems
    WHERE purchase_return_id = p_purchase_return_id
      AND serial_number = p_serial_number
    LIMIT 1;

    RETURN COALESCE(v_exists, FALSE);
END;
$$;
```

Parameters:

- `p_purchase_return_id bigint`
- `p_serial_number text`

Returns:

`boolean`

Purpose:

Serial Exists In Purchase Return - Performs specialized database operation.

Example SQL Call:

```
SELECT serial_exists_in_purchase_return(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 11: `rebuild_purchase_return_journal()`

Complete SQL Code:

```
CREATE FUNCTION public.rebuild_purchase_return_journal(p_return_id bigint) RETURNS
void
    LANGUAGE plpgsql
    AS $$
DECLARE
    j_id BIGINT;
    inv_acc BIGINT;
    party_acc BIGINT;
```

```
    v_total NUMERIC(14,2);
    v_vendor_id BIGINT;
    v_date DATE;
BEGIN
    -- 1. Remove old journal if exists
    SELECT journal_id INTO j_id
    FROM PurchaseReturns
    WHERE purchase_return_id = p_return_id;

    IF j_id IS NOT NULL THEN
        DELETE FROM JournalEntries WHERE journal_id = j_id;
    END IF;

    -- 2. Get totals
    SELECT vendor_id, total_amount, return_date
    INTO v_vendor_id, v_total, v_date
    FROM PurchaseReturns
    WHERE purchase_return_id = p_return_id;

    -- 3. Accounts
    SELECT account_id INTO inv_acc
    FROM ChartOfAccounts
    WHERE account_name='Inventory';

    SELECT ap_account_id INTO party_acc
    FROM Parties
    WHERE party_id = v_vendor_id;

    -- 4. New journal
    INSERT INTO JournalEntries(entry_date, description)
    VALUES (v_date, 'Purchase Return ' || p_return_id)
    RETURNING journal_id INTO j_id;

    UPDATE PurchaseReturns
    SET journal_id = j_id
    WHERE purchase_return_id = p_return_id;

    -- 5. Journal lines (with conditions)
    -- (1) Debit Vendor (reduce AP balance)
    IF v_total > 0 THEN
        INSERT INTO JournalLines(journal_id, account_id, party_id, debit)
        VALUES (j_id, party_acc, v_vendor_id, v_total);
    END IF;

    -- (2) Credit Inventory (stock reduced)
    IF v_total > 0 THEN
        INSERT INTO JournalLines(journal_id, account_id, credit)
        VALUES (j_id, inv_acc, v_total);
    END IF;
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`void`

Purpose:

Rebuild Purchase Return Journal - Rebuilds accounting journal entries for a transaction.

Example SQL Call:

```
SELECT rebuild_purchase_return_journal(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

⬅ BACK SALES RETURN FUNCTIONS

Total Functions: 11

Function 1: `create_sale_return()`

Complete SQL Code:

```
CREATE FUNCTION public.create_sale_return(p_party_name text, p_serials jsonb)
RETURNS bigint
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_party_id BIGINT;
    v_return_id BIGINT;
    v_serial TEXT;
    v_unit RECORD;
    v_total NUMERIC(14,2) := 0;
    v_cost NUMERIC(14,2) := 0;
BEGIN
    -- 1. Find Customer
    SELECT party_id INTO v_party_id
    FROM Parties
    WHERE party_name = p_party_name;

    IF v_party_id IS NULL THEN
```

```
        RAISE EXCEPTION 'Customer % not found', p_party_name;
    END IF;

    -- 2. Create Return Header
    INSERT INTO SalesReturns(customer_id, return_date, total_amount)
    VALUES (v_party_id, CURRENT_DATE, 0)
    RETURNING sales_return_id INTO v_return_id;

    -- 3. Process each serial
    FOR v_serial IN SELECT jsonb_array_elements_text(p_serials)
    LOOP
        SELECT su.sold_unit_id, su.unit_id, su.sold_price, si.item_id,
               si.sales_invoice_id, pu.serial_number, pi.unit_price, s.customer_id
        INTO v_unit
        FROM SoldUnits su
        JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
        JOIN SalesInvoices s ON si.sales_invoice_id = s.sales_invoice_id
        JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
        JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
        WHERE pu.serial_number = v_serial;

        IF NOT FOUND THEN
            RAISE EXCEPTION 'Serial % not found in SoldUnits', v_serial;
        END IF;

        -- check customer match
        IF v_unit.customer_id <> v_party_id THEN
            RAISE EXCEPTION 'Serial % was not sold to %', v_serial, p_party_name;
        END IF;

        -- mark sold unit as returned
        UPDATE SoldUnits SET status = 'Returned' WHERE sold_unit_id =
v_unit.sold_unit_id;

        -- restore stock
        UPDATE PurchaseUnits SET in_stock = TRUE WHERE unit_id = v_unit.unit_id;

        -- log stock IN
        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (v_unit.item_id, v_serial, 'IN', 'SalesReturn', v_return_id, 1);

        -- insert return line item
        INSERT INTO SalesReturnItems(sales_return_id, item_id, sold_price,
cost_price, serial_number)
        VALUES (v_return_id, v_unit.item_id, v_unit.sold_price, v_unit.unit_price,
v_serial);

        -- accumulate totals
        v_total := v_total + v_unit.sold_price;
        v_cost := v_cost + v_unit.unit_price;
    END LOOP;

    -- 4. Update return total
```

```
    UPDATE SalesReturns
    SET total_amount = v_total
    WHERE sales_return_id = v_return_id;

    -- Build Journal
    PERFORM rebuild_sales_return_journal(v_return_id);

    RETURN v_return_id;
END;
$$;
```

Parameters:

- `p_party_name text`
- `p_serials jsonb`

Returns:

`bigint`

Purpose:

Create Sale Return - Creates a new record in the database with all related entries and accounting journal.

Example SQL Call:

```
SELECT create_sale_return('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `delete_sale_return()`

Complete SQL Code:

```
CREATE FUNCTION public.delete_sale_return(p_return_id bigint) RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    rec RECORD;
    v_journal_id BIGINT;
BEGIN
    -- 1. Revert each returned unit
```

```
    FOR rec IN
        SELECT sri.serial_number, sri.item_id
        FROM SalesReturnItems sri
        WHERE sri.sales_return_id = p_return_id
    LOOP
        -- mark sold unit back as Sold
        UPDATE SoldUnits
        SET status = 'Sold'
        WHERE unit_id = (
            SELECT unit_id
            FROM PurchaseUnits
            WHERE serial_number = rec.serial_number
            LIMIT 1
        );

        -- remove from stock again
        UPDATE PurchaseUnits
        SET in_stock = FALSE
        WHERE serial_number = rec.serial_number;

        -- log stock OUT
        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (rec.item_id, rec.serial_number, 'OUT', 'SalesReturn-Delete',
p_return_id, 1);
    END LOOP;

    -- 2. Remove journal (if exists)
    SELECT journal_id INTO v_journal_id
    FROM SalesReturns
    WHERE sales_return_id = p_return_id;

    IF v_journal_id IS NOT NULL THEN
        DELETE FROM JournalEntries WHERE journal_id = v_journal_id;
    END IF;

    -- 2. Delete return items
    DELETE FROM SalesReturnItems WHERE sales_return_id = p_return_id;

    -- 4. Delete return header (triggers remove journal)
    DELETE FROM SalesReturns WHERE sales_return_id = p_return_id;
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`void`

Purpose:

Delete Sale Return - Deletes a record after validation, cleaning up all related entries.

Example SQL Call:

```
SELECT delete_sale_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `update_sale_return()`

Complete SQL Code:

```
CREATE FUNCTION public.update_sale_return(p_return_id bigint, p_serials jsonb)
RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    rec RECORD;
    v_serial TEXT;
    v_unit RECORD;
    v_total NUMERIC(14,2) := 0;
    v_cost NUMERIC(14,2) := 0;
    v_customer_id BIGINT;
BEGIN
    -- 1. Reverse old items
    FOR rec IN
        SELECT serial_number, item_id
        FROM SalesReturnItems
        WHERE sales_return_id = p_return_id
    LOOP
        -- revert SoldUnits status
        UPDATE SoldUnits
        SET status = 'Sold'
        WHERE unit_id = (
            SELECT unit_id FROM PurchaseUnits WHERE serial_number =
rec.serial_number LIMIT 1
        );

        -- remove stock
        UPDATE PurchaseUnits
        SET in_stock = FALSE
        WHERE serial_number = rec.serial_number;

        -- log OUT
```

```
        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (rec.item_id, rec.serial_number, 'OUT', 'SalesReturn-Update-
Reverse', p_return_id, 1);
    END LOOP;

    -- 2. Remove old items
    DELETE FROM SalesReturnItems WHERE sales_return_id = p_return_id;

    -- 3. Get customer id
    SELECT customer_id INTO v_customer_id
    FROM SalesReturns
    WHERE sales_return_id = p_return_id;

    -- 4. Insert new items
    FOR v_serial IN SELECT jsonb_array_elements_text(p_serials)
    LOOP
        SELECT su.sold_unit_id, su.unit_id, su.sold_price, si.item_id,
               si.sales_invoice_id, pu.serial_number, pi.unit_price, s.customer_id
        INTO v_unit
        FROM SoldUnits su
        JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
        JOIN SalesInvoices s ON si.sales_invoice_id = s.sales_invoice_id
        JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
        JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
        WHERE pu.serial_number = v_serial;

        IF NOT FOUND THEN
            RAISE EXCEPTION 'Serial % not found in SoldUnits', v_serial;
        END IF;

        IF v_unit.customer_id <> v_customer_id THEN
            RAISE EXCEPTION 'Serial % was not sold to this customer', v_serial;
        END IF;

        UPDATE SoldUnits SET status = 'Returned' WHERE sold_unit_id =
v_unit.sold_unit_id;
        UPDATE PurchaseUnits SET in_stock = TRUE WHERE unit_id = v_unit.unit_id;

        INSERT INTO StockMovements(item_id, serial_number, movement_type,
reference_type, reference_id, quantity)
        VALUES (v_unit.item_id, v_serial, 'IN', 'SalesReturn-Update', p_return_id,
1);

        INSERT INTO SalesReturnItems(sales_return_id, item_id, sold_price,
cost_price, serial_number)
        VALUES (p_return_id, v_unit.item_id, v_unit.sold_price, v_unit.unit_price,
v_serial);

        v_total := v_total + v_unit.sold_price;
        v_cost := v_cost + v_unit.unit_price;
    END LOOP;

    -- 5. Update header total
```

```
    UPDATE SalesReturns
    SET total_amount = v_total
    WHERE sales_return_id = p_return_id;

    -- 6. Rebuild journal
    PERFORM rebuild_sales_return_journal(p_return_id);
END;
$$;
```

Parameters:

- `p_return_id bigint`
- `p_serials jsonb`

Returns:

`void`

Purpose:

Update Sale Return - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT update_sale_return('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: `get_current_sales_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_current_sales_return(p_return_id bigint) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSON;
BEGIN
    SELECT json_build_object(
        'sales_return_id', sr.sales_return_id,
        'Customer', pa.party_name,
        'return_date', sr.return_date,
```

```
            'total_amount', sr.total_amount,
            'description', je.description,
            'items', (
                SELECT json_agg(
                    json_build_object(
                        'item_name', i.item_name,
                        'sold_price', sri.sold_price,
                        'cost_price', sri.cost_price,
                        'serial_number', sri.serial_number
                    )
                )
                FROM SalesReturnItems sri
                JOIN Items i ON i.item_id = sri.item_id
                WHERE sri.sales_return_id = sr.sales_return_id
            )
        )
        INTO result
        FROM SalesReturns sr
        JOIN Parties pa ON pa.party_id = sr.customer_id
        LEFT JOIN JournalEntries je ON je.journal_id = sr.journal_id
        WHERE sr.sales_return_id = p_return_id;

        RETURN result;
    END;
    $$;
```

Parameters:

- `p_return_id bigint`

Returns:

`json`

Purpose:

Get Current Sales Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_current_sales_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `get_last_sales_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_sales_return() RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    last_id BIGINT;
BEGIN
    SELECT sales_return_id INTO last_id
    FROM SalesReturns
    ORDER BY sales_return_id DESC
    LIMIT 1;

    RETURN get_current_sales_return(last_id);
END;
$$;
```

Parameters:

- None

Returns:

`json`

Purpose:

Get Last Sales Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_sales_return();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 6: `get_last_sales_return_id()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_sales_return_id() RETURNS bigint
    LANGUAGE plpgsql
    AS $$
DECLARE
```

```
    last_id BIGINT;
BEGIN
    SELECT sales_return_id
    INTO last_id
    FROM SalesReturns
    ORDER BY sales_return_id DESC
    LIMIT 1;

    RETURN last_id;
END;
$$;
```

Parameters:

- None

Returns:

`bigint`

Purpose:

Get Last Sales Return Id - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_sales_return_id();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 7: `get_next_sales_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_next_sales_return(p_return_id bigint) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    next_id BIGINT;
BEGIN
    SELECT sales_return_id INTO next_id
    FROM SalesReturns
    WHERE sales_return_id > p_return_id
    ORDER BY sales_return_id ASC
```

```
    LIMIT 1;

    IF next_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_sales_return(next_id);
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`json`

Purpose:

Get Next Sales Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_next_sales_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 8: `get_previous_sales_return()`

Complete SQL Code:

```
CREATE FUNCTION public.get_previous_sales_return(p_return_id bigint) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    prev_id BIGINT;
BEGIN
    SELECT sales_return_id INTO prev_id
    FROM SalesReturns
    WHERE sales_return_id < p_return_id
    ORDER BY sales_return_id DESC
    LIMIT 1;
```

```
    IF prev_id IS NULL THEN
        RETURN NULL;
    END IF;

    RETURN get_current_sales_return(prev_id);
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`json`

Purpose:

Get Previous Sales Return - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_previous_sales_return(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 9: `get_sales_return_summary()`

Complete SQL Code:

```
CREATE FUNCTION public.get_sales_return_summary(p_start_date date DEFAULT
NULL::date, p_end_date date DEFAULT NULL::date) RETURNS json
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSON;
BEGIN
    IF p_start_date IS NOT NULL AND p_end_date IS NOT NULL THEN
        -- 🧮 Filter by date range
        SELECT json_agg(p ORDER BY p.return_date DESC)
        INTO result
        FROM (
            SELECT
                sr.sales_return_id,
```

```
                sr.return_date,
                pa.party_name AS customer,
                sr.total_amount
            FROM SalesReturns sr
            JOIN Parties pa ON sr.customer_id = pa.party_id
            WHERE sr.return_date BETWEEN p_start_date AND p_end_date
            ORDER BY sr.return_date DESC
        ) AS p;
    ELSE
        -- 🧮 Last 20 returns
        SELECT json_agg(p ORDER BY p.return_date DESC)
        INTO result
        FROM (
            SELECT
                sr.sales_return_id,
                sr.return_date,
                pa.party_name AS customer,
                sr.total_amount
            FROM SalesReturns sr
            JOIN Parties pa ON sr.customer_id = pa.party_id
            ORDER BY sr.return_date DESC
            LIMIT 20
        ) AS p;
    END IF;

    RETURN COALESCE(result, '[]'::json);
END;
$$;
```

Parameters:

- `p_start_date date DEFAULT NULL::date`
- `p_end_date date DEFAULT NULL::date`

Returns:

`json`

Purpose:

Get Sales Return Summary - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_sales_return_summary(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 10: `serial_exists_in_sales_return()`

Complete SQL Code:

```
CREATE FUNCTION public.serial_exists_in_sales_return(p_sales_return_id bigint,
p_serial_number text) RETURNS boolean
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_exists BOOLEAN;
BEGIN
    SELECT TRUE
    INTO v_exists
    FROM SalesReturnItems
    WHERE sales_return_id = p_sales_return_id
      AND serial_number = p_serial_number
    LIMIT 1;

    RETURN COALESCE(v_exists, FALSE);
END;
$$;
```

Parameters:

- `p_sales_return_id bigint`
- `p_serial_number text`

Returns:

`boolean`

Purpose:

Serial Exists In Sales Return - Performs specialized database operation.

Example SQL Call:

```
SELECT serial_exists_in_sales_return(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 11: `rebuild_sales_return_journal()`

Complete SQL Code:

```
CREATE FUNCTION public.rebuild_sales_return_journal(p_return_id bigint) RETURNS
void
    LANGUAGE plpgsql
    AS $$
DECLARE
    j_id BIGINT;
    rev_acc BIGINT;
    cogs_acc BIGINT;
    inv_acc BIGINT;
    party_acc BIGINT;
    v_total NUMERIC(14,2);
    v_cost NUMERIC(14,2);
    v_customer_id BIGINT;
    v_date DATE;
BEGIN
    -- remove old journal
    SELECT journal_id INTO j_id FROM SalesReturns WHERE sales_return_id =
p_return_id;
    IF j_id IS NOT NULL THEN
        DELETE FROM JournalEntries WHERE journal_id = j_id;
    END IF;

    -- totals
    SELECT customer_id, total_amount, return_date
    INTO v_customer_id, v_total, v_date
    FROM SalesReturns WHERE sales_return_id = p_return_id;

    SELECT COALESCE(SUM(cost_price),0) INTO v_cost
    FROM SalesReturnItems WHERE sales_return_id = p_return_id;

    -- accounts
    SELECT account_id INTO rev_acc FROM ChartOfAccounts WHERE account_name='Sales
Revenue';
    SELECT account_id INTO cogs_acc FROM ChartOfAccounts WHERE account_name='Cost
of Goods Sold';
    SELECT account_id INTO inv_acc FROM ChartOfAccounts WHERE
account_name='Inventory';
    SELECT ar_account_id INTO party_acc FROM Parties WHERE party_id =
v_customer_id;

    -- new journal
    INSERT INTO JournalEntries(entry_date, description)
    VALUES (v_date, 'Sales Return ' || p_return_id)
    RETURNING journal_id INTO j_id;

    UPDATE SalesReturns SET journal_id = j_id WHERE sales_return_id = p_return_id;

    -- (1) Debit Sales Revenue
```

```
    IF v_total > 0 THEN
        INSERT INTO JournalLines(journal_id, account_id, debit)
        VALUES (j_id, rev_acc, v_total);
    END IF;

    -- (2) Credit Customer AR
    IF v_total > 0 THEN
        INSERT INTO JournalLines(journal_id, account_id, party_id, credit)
        VALUES (j_id, party_acc, v_customer_id, v_total);
    END IF;

    -- (3) Debit Inventory
    IF v_cost > 0 THEN
        INSERT INTO JournalLines(journal_id, account_id, debit)
        VALUES (j_id, inv_acc, v_cost);
    END IF;

    -- (4) Credit COGS
    IF v_cost > 0 THEN
        INSERT INTO JournalLines(journal_id, account_id, credit)
        VALUES (j_id, cogs_acc, v_cost);
    END IF;
END;
$$;
```

Parameters:

- `p_return_id bigint`

Returns:

`void`

Purpose:

Rebuild Sales Return Journal - Rebuilds accounting journal entries for a transaction.

Example SQL Call:

```
SELECT rebuild_sales_return_journal(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

💵 PAYMENT FUNCTIONS

Total Functions: 9

Function 1: `make_payment()`

Complete SQL Code:

```
CREATE FUNCTION public.make_payment(p_data jsonb) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_party_id   BIGINT;
    v_account_id BIGINT;
    v_amount     NUMERIC(14,4);
    v_method     TEXT;
    v_reference  TEXT;
    v_desc       TEXT;
    v_date       DATE;
    v_id         BIGINT;
BEGIN
    -- Extract
    v_amount    := (p_data->>'amount')::NUMERIC;
    v_method    := p_data->>'method';
    v_reference := p_data->>'reference_no';
    v_desc      := p_data->>'description';
    v_date      := NULLIF(p_data->>'payment_date','')::DATE;

    IF v_amount IS NULL OR v_amount <= 0 THEN
        RAISE EXCEPTION 'Invalid amount: must be > 0';
    END IF;

    -- Get Vendor
    SELECT party_id INTO v_party_id
    FROM Parties
    WHERE party_name = p_data->>'party_name'
    LIMIT 1;

    IF v_party_id IS NULL THEN
        RAISE EXCEPTION 'Vendor % not found', p_data->>'party_name';
    END IF;

    -- Always Cash for now
    SELECT account_id INTO v_account_id
    FROM ChartOfAccounts
    WHERE account_name = 'Cash';

    IF v_account_id IS NULL THEN
        RAISE EXCEPTION 'Cash account not found';
    END IF;

    -- Auto ref
    IF v_reference IS NULL OR v_reference = '' THEN
        v_reference := 'PMT-' || nextval('payments_ref_seq');
    END IF;
```

```
    -- Insert (use given date or default CURRENT_DATE)
    INSERT INTO Payments(party_id, account_id, amount, method, reference_no,
description, payment_date)
    VALUES (v_party_id, v_account_id, v_amount, v_method, v_reference, v_desc,
COALESCE(v_date, CURRENT_DATE))
    RETURNING payment_id INTO v_id;

    RETURN jsonb_build_object(
        'status','success',
        'message','Payment created successfully',
        'payment_id',v_id
    );
END;
$$;
```

Parameters:

- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Make Payment - Performs specialized database operation.

Example SQL Call:

```
SELECT make_payment('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `delete_payment()`

Complete SQL Code:

```
CREATE FUNCTION public.delete_payment(p_payment_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
BEGIN
```

```
    DELETE FROM Payments WHERE payment_id = p_payment_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Payment ID % not found', p_payment_id;
    END IF;

    RETURN jsonb_build_object(
        'status','success',
        'message','Payment deleted successfully',
        'payment_id',p_payment_id
    );
END;
$$;
```

Parameters:

- `p_payment_id bigint`

Returns:

`jsonb`

Purpose:

Delete Payment - Deletes a record after validation, cleaning up all related entries.

Example SQL Call:

```
SELECT delete_payment(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `update_payment()`

Complete SQL Code:

```
CREATE FUNCTION public.update_payment(p_payment_id bigint, p_data jsonb) RETURNS
jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_amount    NUMERIC(14,4);
    v_method    TEXT;
    v_reference TEXT;
```

```
    v_desc      TEXT;
    v_date      DATE;
    v_party_id  BIGINT;
    v_updated   RECORD;
BEGIN
    v_amount    := NULLIF(p_data->>'amount','')::NUMERIC;
    v_method    := NULLIF(p_data->>'method','');
    v_reference := NULLIF(p_data->>'reference_no','');
    v_desc      := NULLIF(p_data->>'description','');
    v_date      := NULLIF(p_data->>'payment_date','')::DATE;

    IF p_data ? 'party_name' THEN
        SELECT party_id INTO v_party_id
        FROM Parties
        WHERE party_name = p_data->>'party_name'
        LIMIT 1;
        IF v_party_id IS NULL THEN
            RAISE EXCEPTION 'Vendor % not found', p_data->>'party_name';
        END IF;
    END IF;

    IF v_amount IS NOT NULL AND v_amount <= 0 THEN
        RAISE EXCEPTION 'Invalid amount';
    END IF;

    UPDATE Payments
    SET amount       = COALESCE(v_amount, amount),
        method       = COALESCE(v_method, method),
        reference_no = COALESCE(v_reference, reference_no),
        party_id     = COALESCE(v_party_id, party_id),
        description  = COALESCE(v_desc, description),
        payment_date = COALESCE(v_date, payment_date)
    WHERE payment_id = p_payment_id
    RETURNING * INTO v_updated;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Payment ID % not found', p_payment_id;
    END IF;

    RETURN jsonb_build_object(
        'status','success',
        'message','Payment updated successfully',
        'payment', to_jsonb(v_updated)
    );
END;
$$;
```

Parameters:

- `p_payment_id bigint`
- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Update Payment - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT update_payment('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: `get_last_20_payments_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_20_payments_json(p_data jsonb) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_party TEXT;
    result  JSONB;
BEGIN
    -- Extract optional party filter
    v_party := p_data->>'party_name';

    SELECT jsonb_agg(row_data)
    INTO result
    FROM (
        SELECT to_jsonb(p) || jsonb_build_object('party_name', pt.party_name) AS
row_data
        FROM Payments p
        JOIN Parties pt ON pt.party_id = p.party_id
        WHERE (v_party IS NULL OR pt.party_name ILIKE v_party)
        ORDER BY p.payment_date DESC, p.payment_id DESC
        LIMIT 20
    ) sub;

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Get Last 20 Payments Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_20_payments_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `get_last_payment()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_payment() RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(p) || jsonb_build_object('party_name', pt.party_name)
    INTO result
    FROM Payments p
    LEFT JOIN Parties pt ON pt.party_id = p.party_id
    ORDER BY p.payment_id DESC
    LIMIT 1;

    RETURN result;
END;
$$;
```

Parameters:

- None

Returns:

`jsonb`

Purpose:

Get Last Payment - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_payment();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 6: `get_next_payment()`

Complete SQL Code:

```
CREATE FUNCTION public.get_next_payment(p_payment_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(p) || jsonb_build_object('party_name', pt.party_name)
    INTO result
    FROM Payments p
    LEFT JOIN Parties pt ON pt.party_id = p.party_id
    WHERE p.payment_id > p_payment_id
    ORDER BY p.payment_id ASC
    LIMIT 1;

    RETURN result;
END;
$$;
```

Parameters:

- `p_payment_id bigint`

Returns:

`jsonb`

Purpose:

Get Next Payment - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_next_payment(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 7: `get_payment_details()`

Complete SQL Code:

```
CREATE FUNCTION public.get_payment_details(p_payment_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(p) || jsonb_build_object('party_name', pt.party_name)
    INTO result
    FROM Payments p
    LEFT JOIN Parties pt ON pt.party_id = p.party_id
    WHERE p.payment_id = p_payment_id;

    RETURN result;
END;
$$;
```

Parameters:

- `p_payment_id bigint`

Returns:

`jsonb`

Purpose:

Get Payment Details - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_payment_details(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 8: `get_payments_by_date_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_payments_by_date_json(p_data jsonb) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_start DATE;
    v_end   DATE;
    v_party TEXT;
    result  JSONB;
BEGIN
    -- Extract from JSON
    v_start := (p_data->>'start_date')::DATE;
    v_end   := (p_data->>'end_date')::DATE;
    v_party := p_data->>'party_name';

    IF v_start IS NULL OR v_end IS NULL THEN
        RAISE EXCEPTION 'Both start_date and end_date must be provided in JSON';
    END IF;

    SELECT jsonb_agg(to_jsonb(p) || jsonb_build_object('party_name',
pt.party_name)
                     ORDER BY p.payment_date DESC, p.payment_id DESC)
    INTO result
    FROM Payments p
    JOIN Parties pt ON pt.party_id = p.party_id
    WHERE p.payment_date BETWEEN v_start AND v_end
      AND (v_party IS NULL OR pt.party_name ILIKE v_party);

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Get Payments By Date Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_payments_by_date_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 9: `get_previous_payment()`

Complete SQL Code:

```
CREATE FUNCTION public.get_previous_payment(p_payment_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(p) || jsonb_build_object('party_name', pt.party_name)
    INTO result
    FROM Payments p
    LEFT JOIN Parties pt ON pt.party_id = p.party_id
    WHERE p.payment_id < p_payment_id
    ORDER BY p.payment_id DESC
    LIMIT 1;

    RETURN result;
END;
$$;
```

Parameters:

- `p_payment_id bigint`

Returns:

`jsonb`

Purpose:

Get Previous Payment - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_previous_payment(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

RECEIPT FUNCTIONS

Total Functions: 9

Function 1: `make_receipt()`

Complete SQL Code:

```
CREATE FUNCTION public.make_receipt(p_data jsonb) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_party_id   BIGINT;
    v_account_id BIGINT;
    v_amount     NUMERIC(14,4);
    v_method     TEXT;
    v_reference  TEXT;
    v_desc       TEXT;
    v_date       DATE;
    v_id         BIGINT;
BEGIN
    -- Extract
    v_amount    := (p_data->>'amount')::NUMERIC;
    v_method    := p_data->>'method';
    v_reference := p_data->>'reference_no';
    v_desc      := p_data->>'description';
    v_date      := NULLIF(p_data->>'receipt_date','')::DATE;

    IF v_amount IS NULL OR v_amount <= 0 THEN
        RAISE EXCEPTION 'Invalid amount: must be > 0';
    END IF;

    -- Get Customer
    SELECT party_id INTO v_party_id
    FROM Parties
```

```
    WHERE party_name = p_data->>'party_name'
    LIMIT 1;

    IF v_party_id IS NULL THEN
        RAISE EXCEPTION 'Customer % not found', p_data->>'party_name';
    END IF;

    -- Always Cash for now
    SELECT account_id INTO v_account_id
    FROM ChartOfAccounts
    WHERE account_name = 'Cash';

    IF v_account_id IS NULL THEN
        RAISE EXCEPTION 'Cash account not found';
    END IF;

    -- Auto ref
    IF v_reference IS NULL OR v_reference = '' THEN
        v_reference := 'RCT-' || nextval('receipts_ref_seq');
    END IF;

    -- Insert
    INSERT INTO Receipts(party_id, account_id, amount, method, reference_no,
description, receipt_date)
    VALUES (v_party_id, v_account_id, v_amount, v_method, v_reference, v_desc,
COALESCE(v_date, CURRENT_DATE))
    RETURNING receipt_id INTO v_id;

    RETURN jsonb_build_object(
        'status','success',
        'message','Receipt created successfully',
        'receipt_id',v_id
    );
END;
$$;
```

Parameters:

- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Make Receipt - Performs specialized database operation.

Example SQL Call:

```
SELECT make_receipt('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `delete_receipt()`

Complete SQL Code:

```
CREATE FUNCTION public.delete_receipt(p_receipt_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
BEGIN
    DELETE FROM Receipts WHERE receipt_id = p_receipt_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Receipt ID % not found', p_receipt_id;
    END IF;

    RETURN jsonb_build_object(
        'status','success',
        'message','Receipt deleted successfully',
        'receipt_id',p_receipt_id
    );
END;
$$;
```

Parameters:

- `p_receipt_id bigint`

Returns:

`jsonb`

Purpose:

Delete Receipt - Deletes a record after validation, cleaning up all related entries.

Example SQL Call:

```
SELECT delete_receipt(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `update_receipt()`

Complete SQL Code:

```
CREATE FUNCTION public.update_receipt(p_receipt_id bigint, p_data jsonb) RETURNS
jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_amount    NUMERIC(14,4);
    v_method    TEXT;
    v_reference TEXT;
    v_desc      TEXT;
    v_date      DATE;
    v_party_id  BIGINT;
    v_updated   RECORD;
BEGIN
    v_amount    := NULLIF(p_data->>'amount','')::NUMERIC;
    v_method    := NULLIF(p_data->>'method','');
    v_reference := NULLIF(p_data->>'reference_no','');
    v_desc      := NULLIF(p_data->>'description','');
    v_date      := NULLIF(p_data->>'receipt_date','')::DATE;

    IF p_data ? 'party_name' THEN
        SELECT party_id INTO v_party_id
        FROM Parties
        WHERE party_name = p_data->>'party_name'
        LIMIT 1;
        IF v_party_id IS NULL THEN
            RAISE EXCEPTION 'Customer % not found', p_data->>'party_name';
        END IF;
    END IF;

    IF v_amount IS NOT NULL AND v_amount <= 0 THEN
        RAISE EXCEPTION 'Invalid amount';
    END IF;

    UPDATE Receipts
    SET amount       = COALESCE(v_amount, amount),
        method       = COALESCE(v_method, method),
        reference_no = COALESCE(v_reference, reference_no),
        party_id     = COALESCE(v_party_id, party_id),
        description  = COALESCE(v_desc, description),
        receipt_date = COALESCE(v_date, receipt_date)
    WHERE receipt_id = p_receipt_id
    RETURNING * INTO v_updated;
```

```
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Receipt ID % not found', p_receipt_id;
    END IF;

    RETURN jsonb_build_object(
        'status','success',
        'message','Receipt updated successfully',
        'receipt', to_jsonb(v_updated)
    );
END;
$$;
```

Parameters:

- `p_receipt_id bigint`
- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Update Receipt - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT update_receipt('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: `get_last_20_receipts_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_20_receipts_json(p_data jsonb) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_party TEXT;
    result  JSONB;
```

```
BEGIN
    v_party := p_data->>'party_name';

    SELECT jsonb_agg(row_data)
    INTO result
    FROM (
        SELECT to_jsonb(r) || jsonb_build_object('party_name', pt.party_name) AS
row_data
        FROM Receipts r
        JOIN Parties pt ON pt.party_id = r.party_id
        WHERE (v_party IS NULL OR pt.party_name ILIKE v_party)
        ORDER BY r.receipt_date DESC, r.receipt_id DESC
        LIMIT 20
    ) sub;

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Get Last 20 Receipts Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_20_receipts_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `get_last_receipt()`

Complete SQL Code:

```
CREATE FUNCTION public.get_last_receipt() RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(r) || jsonb_build_object('party_name', pt.party_name)
    INTO result
    FROM Receipts r
    LEFT JOIN Parties pt ON pt.party_id = r.party_id
    ORDER BY r.receipt_id DESC
    LIMIT 1;

    RETURN result;
END;
$$;
```

Parameters:

- None

Returns:

`jsonb`

Purpose:

Get Last Receipt - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_last_receipt();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 6: `get_next_receipt()`

Complete SQL Code:

```
CREATE FUNCTION public.get_next_receipt(p_receipt_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
```

```
    result JSONB;
BEGIN
    SELECT to_jsonb(r) || jsonb_build_object('party_name', pt.party_name)
    INTO result
    FROM Receipts r
    LEFT JOIN Parties pt ON pt.party_id = r.party_id
    WHERE r.receipt_id > p_receipt_id
    ORDER BY r.receipt_id ASC
    LIMIT 1;

    RETURN result;
END;
$$;
```

Parameters:

- `p_receipt_id bigint`

Returns:

`jsonb`

Purpose:

Get Next Receipt - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_next_receipt(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 7: `get_previous_receipt()`

Complete SQL Code:

```
CREATE FUNCTION public.get_previous_receipt(p_receipt_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(r) || jsonb_build_object('party_name', pt.party_name)
    INTO result
```

```
    FROM Receipts r
    LEFT JOIN Parties pt ON pt.party_id = r.party_id
    WHERE r.receipt_id < p_receipt_id
    ORDER BY r.receipt_id DESC
    LIMIT 1;

    RETURN result;
END;
$$;
```

Parameters:

- `p_receipt_id bigint`

Returns:

`jsonb`

Purpose:

Get Previous Receipt - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_previous_receipt(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 8: `get_receipt_details()`

Complete SQL Code:

```
CREATE FUNCTION public.get_receipt_details(p_receipt_id bigint) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(r) || jsonb_build_object('party_name', pt.party_name)
    INTO result
    FROM Receipts r
    LEFT JOIN Parties pt ON pt.party_id = r.party_id
    WHERE r.receipt_id = p_receipt_id;
```

```
    RETURN result;
END;
$$;
```

Parameters:

- `p_receipt_id bigint`

Returns:

`jsonb`

Purpose:

Get Receipt Details - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_receipt_details(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 9: `get_receipts_by_date_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_receipts_by_date_json(p_data jsonb) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    v_start DATE;
    v_end   DATE;
    v_party TEXT;
    result  JSONB;
BEGIN
    v_start := (p_data->>'start_date')::DATE;
    v_end   := (p_data->>'end_date')::DATE;
    v_party := p_data->>'party_name';

    IF v_start IS NULL OR v_end IS NULL THEN
        RAISE EXCEPTION 'Both start_date and end_date must be provided in JSON';
    END IF;

    SELECT jsonb_agg(to_jsonb(r) || jsonb_build_object('party_name',
```

```
    pt.party_name)
                        ORDER BY r.receipt_date DESC, r.receipt_id DESC)
    INTO result
    FROM Receipts r
    JOIN Parties pt ON pt.party_id = r.party_id
    WHERE r.receipt_date BETWEEN v_start AND v_end
      AND (v_party IS NULL OR pt.party_name ILIKE v_party);

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- `p_data jsonb`

Returns:

`jsonb`

Purpose:

Get Receipts By Date Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_receipts_by_date_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

👥 PARTY FUNCTIONS

Total Functions: 6

Function 1: `add_party_from_json()`

Complete SQL Code:

```
CREATE FUNCTION public.add_party_from_json(party_data jsonb) RETURNS void
    LANGUAGE plpgsql
    AS $$
```

```
DECLARE
    v_party_type TEXT := TRIM(BOTH '"' FROM party_data->>'party_type');
    v_party_name TEXT := TRIM(BOTH '"' FROM party_data->>'party_name');
    v_opening_balance NUMERIC := COALESCE((party_data-
>>'opening_balance')::NUMERIC, 0);
    v_balance_type TEXT := COALESCE(party_data->>'balance_type', 'Debit');
    v_expense_account_id BIGINT;
BEGIN
    -- Handle Expense-type Party (auto-create its expense COA account)
    IF v_party_type = 'Expense' THEN
        -- Check if Expense account already exists in COA
        SELECT account_id INTO v_expense_account_id
        FROM ChartOfAccounts
        WHERE account_name ILIKE v_party_name
          AND account_type = 'Expense'
        LIMIT 1;

        -- Create a new Expense account if not found
        IF v_expense_account_id IS NULL THEN
            INSERT INTO ChartOfAccounts (
                account_code, account_name, account_type, parent_account,
date_created
            )
            VALUES (
                CONCAT('EXP-', LPAD((SELECT COUNT(*) + 1 FROM ChartOfAccounts
WHERE account_type='Expense')::TEXT, 4, '0')),
                v_party_name,
                'Expense',
                (SELECT account_id FROM ChartOfAccounts WHERE account_name ILIKE
'Expenses' LIMIT 1),
                CURRENT_TIMESTAMP
            )
            RETURNING account_id INTO v_expense_account_id;
        END IF;
    END IF;

    -- Insert into Parties table
    INSERT INTO Parties (
        party_name, party_type, contact_info, address,
        opening_balance, balance_type,
        ar_account_id, ap_account_id
    )
    VALUES (
        v_party_name,
        v_party_type,
        party_data->>'contact_info',
        party_data->>'address',
        v_opening_balance,
        v_balance_type,
        CASE
            WHEN v_party_type IN ('Customer','Both','Expense') THEN
                (SELECT account_id FROM ChartOfAccounts WHERE account_name ILIKE
'Accounts Receivable' LIMIT 1)
            ELSE NULL
```

```
        END,
        CASE
            WHEN v_party_type IN ('Vendor','Both') THEN
                (SELECT account_id FROM ChartOfAccounts WHERE account_name ILIKE
'Accounts Payable' LIMIT 1)
            WHEN v_party_type = 'Expense' THEN
                v_expense_account_id
            ELSE NULL
        END
    );
END;
$$;
```

Parameters:

- `party_data jsonb`

Returns:

`void`

Purpose:

Add Party From Json - Performs specialized database operation.

Example SQL Call:

```
SELECT add_party_from_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `update_party_from_json()`

Complete SQL Code:

```
CREATE FUNCTION public.update_party_from_json(p_id bigint, party_data jsonb)
RETURNS void
    LANGUAGE plpgsql
    AS $$
DECLARE
    -- Old party data
    old_opening NUMERIC(14,2);
```

```
    old_balance_type VARCHAR(10);
    old_party_type VARCHAR(20);
    old_party_name VARCHAR(150);

    -- New values
    new_opening NUMERIC(14,2);
    new_balance_type VARCHAR(10);
    new_party_type VARCHAR(20);
    new_party_name VARCHAR(150);

    -- Accounting
    cap_acc BIGINT;
    j_id BIGINT;
    debit_acc BIGINT;
    credit_acc BIGINT;
    v_expense_account_id BIGINT;
BEGIN
    -- ============= FETCH EXISTING DATA =============
    SELECT opening_balance, balance_type, party_type, party_name
    INTO old_opening, old_balance_type, old_party_type, old_party_name
    FROM Parties
    WHERE party_id = p_id;

    -- ============= PARSE NEW VALUES =============
    new_opening := COALESCE((party_data->>'opening_balance')::NUMERIC,
old_opening);
    new_balance_type := COALESCE(party_data->>'balance_type', old_balance_type);
    new_party_type := COALESCE(party_data->>'party_type', old_party_type);
    new_party_name := COALESCE(party_data->>'party_name', old_party_name);

    -- ============= EXPENSE PARTY LOGIC =============
    IF new_party_type = 'Expense' THEN
        -- Try to fetch the linked expense account (stored in ap_account_id)
        SELECT ap_account_id INTO v_expense_account_id
        FROM Parties WHERE party_id = p_id;

        -- If found, rename COA to match new name
        IF v_expense_account_id IS NOT NULL THEN
            UPDATE ChartOfAccounts
            SET account_name = new_party_name
            WHERE account_id = v_expense_account_id;
        ELSE
            -- Otherwise create a new Expense COA account
            INSERT INTO ChartOfAccounts (
                account_code, account_name, account_type, parent_account,
date_created
            )
            VALUES (
                CONCAT('EXP-', LPAD((SELECT COUNT(*) + 1 FROM ChartOfAccounts
WHERE account_type='Expense')::TEXT, 4, '0')),
                new_party_name,
                'Expense',
                (SELECT account_id FROM ChartOfAccounts WHERE account_name ILIKE
'Expenses' LIMIT 1),
```

```
                CURRENT_TIMESTAMP
            )
            RETURNING account_id INTO v_expense_account_id;
        END IF;
    END IF;

    -- ============= UPDATE PARTY DETAILS =============
    UPDATE Parties
    SET
        party_name     = new_party_name,
        party_type     = new_party_type,
        contact_info   = COALESCE(party_data->>'contact_info', contact_info),
        address        = COALESCE(party_data->>'address', address),
        opening_balance = new_opening,
        balance_type   = new_balance_type,
        ar_account_id  = CASE
                            WHEN new_party_type IN ('Customer','Both')
                            THEN (SELECT account_id FROM ChartOfAccounts WHERE
account_name ILIKE 'Accounts Receivable' LIMIT 1)
                            ELSE NULL END,
        ap_account_id  = CASE
                            WHEN new_party_type IN ('Vendor','Both')
                                THEN (SELECT account_id FROM ChartOfAccounts WHERE
account_name ILIKE 'Accounts Payable' LIMIT 1)
                            WHEN new_party_type = 'Expense'
                                THEN v_expense_account_id
                            ELSE NULL
                        END
    WHERE party_id = p_id;

    -- ============= SYNC JOURNAL DESCRIPTION IF PARTY NAME CHANGED =============
    IF new_party_name IS DISTINCT FROM old_party_name THEN
        UPDATE JournalEntries
        SET description = 'Opening Balance for ' || new_party_name
        WHERE journal_id IN (
            SELECT DISTINCT jl.journal_id
            FROM JournalLines jl
            WHERE jl.party_id = p_id
        )
        AND description ILIKE 'Opening Balance for%';
    END IF;

    -- ============= HANDLE OPENING BALANCE CHANGES =============
    IF new_opening IS DISTINCT FROM old_opening
       OR new_balance_type IS DISTINCT FROM old_balance_type
       OR new_party_type IS DISTINCT FROM old_party_type THEN

        -- Delete old Opening Balance journals
        DELETE FROM JournalEntries je
        WHERE je.journal_id IN (
            SELECT jl.journal_id
            FROM JournalLines jl
            WHERE jl.party_id = p_id
        )
```

```
        AND je.description ILIKE 'Opening Balance for%';

        -- Get Owner's Capital account
        SELECT account_id INTO cap_acc
        FROM ChartOfAccounts WHERE account_name = 'Owner''s Capital';

        IF cap_acc IS NULL THEN
            RAISE EXCEPTION 'Owner''s Capital account not found in COA';
        END IF;

        -- Recreate new Opening Balance entry
        INSERT INTO JournalEntries(entry_date, description)
        VALUES (CURRENT_DATE, 'Opening Balance for ' || new_party_name)
        RETURNING journal_id INTO j_id;

        -- Customer / Both (Debit balance)
        IF new_party_type IN ('Customer','Both') AND new_balance_type = 'Debit'
AND new_opening > 0 THEN
            debit_acc := (SELECT ar_account_id FROM Parties WHERE party_id =
p_id);
            credit_acc := cap_acc;

            INSERT INTO JournalLines(journal_id, account_id, party_id, debit)
            VALUES (j_id, debit_acc, p_id, new_opening);

            INSERT INTO JournalLines(journal_id, account_id, credit)
            VALUES (j_id, credit_acc, new_opening);
        END IF;

        -- Vendor / Both (Credit balance)
        IF new_party_type IN ('Vendor','Both') AND new_balance_type = 'Credit' AND
new_opening > 0 THEN
            debit_acc := cap_acc;
            credit_acc := (SELECT ap_account_id FROM Parties WHERE party_id =
p_id);

            INSERT INTO JournalLines(journal_id, account_id, debit)
            VALUES (j_id, debit_acc, new_opening);

            INSERT INTO JournalLines(journal_id, account_id, party_id, credit)
            VALUES (j_id, credit_acc, p_id, new_opening);
        END IF;
    END IF;
END;
$$;
```

Parameters:

- `p_id bigint`
- `party_data jsonb`

Returns:

`void`

Purpose:

Update Party From Json - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT update_party_from_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `get_parties_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_parties_json() RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT jsonb_agg(
               jsonb_build_object(
                   'party_name', party_name,
                   'party_type', party_type
               )
           )
    INTO result
    FROM Parties;

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- None

Returns:

`jsonb`

Purpose:

Get Parties Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_parties_json();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: `get_party_balances_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_party_balances_json() RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT jsonb_agg(
               jsonb_build_object(
                   'name', name,
                   'balance', balance
               )
           )
    INTO result
    FROM vw_trial_balance
    WHERE code IS NULL  -- only parties (not chart of accounts)
      AND type NOT ILIKE '%Expense%'  -- exclude expense parties if any
      AND balance <> 0;  -- optional: skip zero balances

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- None

Returns:

`jsonb`

Purpose:

Get Party Balances Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_party_balances_json();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `get_party_by_name()`

Complete SQL Code:

```
CREATE FUNCTION public.get_party_by_name(p_name text) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT to_jsonb(p)
    INTO result
    FROM Parties p
    WHERE LOWER(p.party_name) = LOWER(p_name)
    LIMIT 1;

    IF result IS NULL THEN
        RETURN '[]'::jsonb;
    END IF;

    RETURN result;
END;
$$;
```

Parameters:

- `p_name text`

Returns:

`jsonb`

Purpose:

Get Party By Name - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_party_by_name(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 6: `get_expense_party_balances_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_expense_party_balances_json() RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT jsonb_agg(
               jsonb_build_object(
                   'name', name,
                   'balance', balance
               )
           )
    INTO result
    FROM vw_trial_balance
    WHERE code IS NULL  -- only parties (not chart of accounts)
      AND type = 'Expense Party'  -- specifically Expense Party
      AND balance <> 0;  -- optional: skip zero balances

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- None

Returns:

`jsonb`

Purpose:

Get Expense Party Balances Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_expense_party_balances_json();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

ITEM FUNCTIONS

Total Functions: 4

Function 1: add_item_from_json()

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.add_item_from_json(item_data jsonb) RETURNS void
    LANGUAGE plpgsql
    AS $$
BEGIN
    INSERT INTO Items (
        item_name,
        storage,
        sale_price,
        item_code,
        category,
        brand,
        created_at,
        updated_at
    )
    VALUES (
        item_data->>'item_name',
        COALESCE(item_data->>'storage', 'Main Warehouse'),          -- default
storage
        COALESCE((item_data->>'sale_price')::NUMERIC, 0.00),        -- default
0.00
        NULLIF(item_data->>'item_code', ''),                        -- NULL if
empty
        NULLIF(item_data->>'category', ''),                         -- optional
        NULLIF(item_data->>'brand', ''),                            -- optional
        COALESCE((item_data->>'created_at')::TIMESTAMP, NOW()),     -- default
current time
        COALESCE((item_data->>'updated_at')::TIMESTAMP, NOW())      -- default
current time
    );
```

```
END;
$$;
```

Parameters:

- `item_data jsonb`

Returns:

`void`

Purpose:

Add Item From Json - Performs specialized database operation.

Example SQL Call:

```
SELECT add_item_from_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `update_item_from_json()`

Complete SQL Code:

```
CREATE FUNCTION public.update_item_from_json(item_data jsonb) RETURNS void
    LANGUAGE plpgsql
    AS $$
BEGIN
    UPDATE Items
    SET
        item_name   = COALESCE(item_data->>'item_name', item_name),
        storage     = COALESCE(item_data->>'storage', storage),
        sale_price  = COALESCE(NULLIF(item_data->>'sale_price','')::NUMERIC,
sale_price),
        item_code   = COALESCE(NULLIF(item_data->>'item_code',''), item_code),
        category    = COALESCE(NULLIF(item_data->>'category',''), category),
        brand       = COALESCE(NULLIF(item_data->>'brand',''), brand),
        updated_at  = NOW()
    WHERE item_id = (item_data->>'item_id')::BIGINT;
END;
$$;
```

Parameters:

- `item_data jsonb`

Returns:

`void`

Purpose:

Update Item From Json - Updates an existing record with validation to maintain data integrity.

Example SQL Call:

```
SELECT update_item_from_json('{
    -- JSON parameters here
}'::jsonb);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `get_items_json()`

Complete SQL Code:

```
CREATE FUNCTION public.get_items_json() RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT jsonb_agg(
               jsonb_build_object(
                   'item_name', item_name,
                   'brand', brand
               )
           )
    INTO result
    FROM Items;

    RETURN COALESCE(result, '[]'::jsonb);
END;
$$;
```

Parameters:

- None

Returns:

`jsonb`

Purpose:

Get Items Json - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_items_json();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: `get_item_by_name()`

Complete SQL Code:

```
CREATE FUNCTION public.get_item_by_name(p_item_name text) RETURNS jsonb
    LANGUAGE plpgsql
    AS $$
DECLARE
    result JSONB;
BEGIN
    SELECT COALESCE(
        jsonb_agg(to_jsonb(i) - 'updated_at' - 'created_at'),  -- convert rows to
JSON array
        '[]'::jsonb
    )
    INTO result
    FROM Items i
    WHERE i.item_name ILIKE p_item_name;   -- case-insensitive match

    RETURN result;
END;
$$;
```

Parameters:

- `p_item_name text`

Returns:

`jsonb`

Purpose:

Get Item By Name - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_item_by_name(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

ACCOUNT REPORT FUNCTIONS

Total Functions: 2

Function 1: `detailed_ledger()`

Complete SQL Code:

```
CREATE FUNCTION public.detailed_ledger(p_party_name text, p_start_date date,
p_end_date date) RETURNS TABLE(entry_date date, journal_id bigint, description
text, party_name text, account_type text, debit numeric, credit numeric,
running_balance numeric)
    LANGUAGE plpgsql
    AS $$
BEGIN
    RETURN QUERY
    WITH party_ledger AS (
        SELECT
            je.entry_date AS entry_date,
            je.journal_id AS journal_id,
            je.description::TEXT AS description,
            p.party_name::TEXT AS party_name,
            a.account_name::TEXT AS account_name,
            jl.debit AS debit,
            jl.credit AS credit,
            (jl.debit - jl.credit) AS amount
        FROM JournalLines jl
        JOIN JournalEntries je ON jl.journal_id = je.journal_id
        JOIN ChartOfAccounts a ON jl.account_id = a.account_id
        LEFT JOIN Parties p ON jl.party_id = p.party_id
        WHERE
```

```
            p.party_name = p_party_name
            AND je.entry_date BETWEEN p_start_date AND p_end_date
    )
    SELECT
        pl.entry_date,
        pl.journal_id,
        pl.description,
        pl.party_name,
        pl.account_name AS account_type,
        pl.debit,
        pl.credit,
        SUM(pl.amount) OVER (ORDER BY pl.entry_date, pl.journal_id ROWS UNBOUNDED
PRECEDING) AS running_balance
    FROM party_ledger pl
    ORDER BY pl.entry_date, pl.journal_id;
END;
$$;
```

Parameters:

- `p_party_name text`
- `p_start_date date`
- `p_end_date date`

Returns:

`TABLE(entry_date date, journal_id bigint, description text, party_name text, account_type text, debit numeric, credit numeric, running_balance numeric)`

Purpose:

Detailed Ledger - Performs specialized database operation.

Example SQL Call:

```
SELECT detailed_ledger(..., ..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `get_cash_ledger_with_party()`

Complete SQL Code:

```

```

```
CREATE OR REPLACE FUNCTION public.get_cash_ledger_with_party(
    p_start_date DATE DEFAULT NULL,
    p_end_date DATE DEFAULT NULL
) RETURNS TABLE (
    entry_date DATE,
    journal_id BIGINT,
    party_name VARCHAR(150),
    description TEXT,
    debit NUMERIC(14,4),
    credit NUMERIC(14,4),
    balance NUMERIC(14,4)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_cash_account_id BIGINT;
    v_opening_balance NUMERIC(14,4) := 0;
BEGIN
    -- Get Cash account ID
    SELECT account_id INTO v_cash_account_id
    FROM ChartOfAccounts
    WHERE account_name = 'Cash'
    LIMIT 1;

    IF v_cash_account_id IS NULL THEN
        RAISE EXCEPTION 'Cash account not found in Chart of Accounts';
    END IF;

    -- Set default dates if not provided
    p_start_date := COALESCE(p_start_date, '1900-01-01'::DATE);
    p_end_date := COALESCE(p_end_date, CURRENT_DATE);

    -- Calculate opening balance
    SELECT COALESCE(SUM(jl.debit) - SUM(jl.credit), 0)
    INTO v_opening_balance
    FROM JournalLines jl
    JOIN JournalEntries je ON jl.journal_id = je.journal_id
    WHERE jl.account_id = v_cash_account_id
      AND je.entry_date < p_start_date;

    -- Return opening balance if there are prior transactions
    IF v_opening_balance <> 0 THEN
        RETURN QUERY
        SELECT
            p_start_date AS entry_date,
            NULL::BIGINT AS journal_id,
            NULL::VARCHAR(150) AS party_name,
            'Opening Balance'::TEXT AS description,
            CASE WHEN v_opening_balance > 0 THEN v_opening_balance ELSE 0 END AS
debit,
            CASE WHEN v_opening_balance < 0 THEN ABS(v_opening_balance) ELSE 0 END
AS credit,
            v_opening_balance AS balance;
```

```
    END IF;

    -- Return all cash transactions with party information
    RETURN QUERY
    WITH cash_transactions AS (
        SELECT
            je.entry_date,
            je.journal_id,
            -- Get party name from the OTHER journal line in the same journal
entry
            (SELECT p.party_name
             FROM JournalLines jl2
             LEFT JOIN Parties p ON jl2.party_id = p.party_id
             WHERE jl2.journal_id = je.journal_id
               AND jl2.account_id != v_cash_account_id
               AND jl2.party_id IS NOT NULL
             LIMIT 1) AS party_name,
            je.description,
            jl.debit,
            jl.credit,
            (jl.debit - jl.credit) AS net_amount
        FROM JournalLines jl
        JOIN JournalEntries je ON jl.journal_id = je.journal_id
        WHERE jl.account_id = v_cash_account_id
          AND je.entry_date >= p_start_date
          AND je.entry_date <= p_end_date
        ORDER BY je.entry_date, je.journal_id
    )
    SELECT
        ct.entry_date,
        ct.journal_id,
        ct.party_name,
        ct.description,
        ct.debit,
        ct.credit,
        v_opening_balance + SUM(ct.net_amount) OVER (ORDER BY ct.entry_date,
ct.journal_id) AS balance
    FROM cash_transactions ct;

END;
$$;
```

Parameters:

- `p_start_date DATE DEFAULT NULL`
- `p_end_date DATE DEFAULT NULL`

Returns:

`TABLE ( entry_date DATE, journal_id BIGINT, party_name VARCHAR(150), description TEXT, debit NUMERIC(14,4), credit NUMERIC(14,4), balance NUMERIC(14,4) )`

Purpose:

Get Cash Ledger With Party - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_cash_ledger_with_party(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

PROFIT REPORT FUNCTIONS

Total Functions: 1

Function 1: `sale_wise_profit()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.sale_wise_profit(
    p_from_date DATE,
    p_to_date DATE
) RETURNS TABLE (
    sale_date DATE,
    item_name TEXT,
    serial_number TEXT,
    serial_comment TEXT,
    sale_price NUMERIC,
    purchase_price NUMERIC,
    profit_loss NUMERIC,
    profit_loss_percent NUMERIC,
    vendor_name TEXT
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    WITH sold_serials AS (
        SELECT
            su.sold_unit_id,
            su.sold_price,
            pu.serial_number::TEXT AS serial_number,
            pu.serial_comment::TEXT AS serial_comment,
            si.sales_item_id,
            s.sales_invoice_id,
            s.invoice_date AS sale_date,
```

```
            i.item_name::TEXT AS item_name,
            i.item_code,
            i.brand,
            i.category,
            si.item_id,
            pu.unit_id
        FROM SoldUnits su
        JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
        JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
        JOIN SalesInvoices s ON si.sales_invoice_id = s.sales_invoice_id
        JOIN Items i ON si.item_id = i.item_id
        WHERE s.invoice_date BETWEEN p_from_date AND p_to_date
    ),
    purchased_serials AS (
        SELECT
            pu.unit_id,
            pu.serial_number::TEXT AS serial_number,
            pi.purchase_item_id,
            p.purchase_invoice_id,
            p.vendor_id,
            i.item_id,
            i.item_name::TEXT AS item_name,
            pi.unit_price AS purchase_price
        FROM PurchaseUnits pu
        JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
        JOIN PurchaseInvoices p ON pi.purchase_invoice_id = p.purchase_invoice_id
        JOIN Items i ON pi.item_id = i.item_id
    )
    SELECT
        ss.sale_date,
        ss.item_name,
        ss.serial_number,
        ss.serial_comment,
        ss.sold_price AS sale_price,
        ps.purchase_price,
        ROUND(ss.sold_price - ps.purchase_price, 2) AS profit_loss,
        CASE
            WHEN ps.purchase_price > 0 THEN
                ROUND(((ss.sold_price - ps.purchase_price) / ps.purchase_price) *
100, 2)
            ELSE
                NULL
        END AS profit_loss_percent,
        v.party_name::TEXT AS vendor_name
    FROM sold_serials ss
    LEFT JOIN purchased_serials ps
        ON ss.unit_id = ps.unit_id
    LEFT JOIN Parties v
        ON ps.vendor_id = v.party_id
    ORDER BY ss.sale_date, ss.item_name, ss.serial_number;
END;
$$;
```

Parameters:

- `p_from_date DATE`
- `p_to_date DATE`

Returns:

`TABLE ( sale_date DATE, item_name TEXT, serial_number TEXT, serial_comment TEXT, sale_price NUMERIC, purchase_price NUMERIC, profit_loss NUMERIC, profit_loss_percent NUMERIC, vendor_name TEXT )`

Purpose:

Sale Wise Profit - Performs specialized database operation.

Example SQL Call:

```
SELECT sale_wise_profit(..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

STOCK REPORT FUNCTIONS

Total Functions: 5

Function 1: `item_transaction_history()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.item_transaction_history(
    p_item_name text,
    p_from_date date DEFAULT NULL::date,
    p_to_date date DEFAULT NULL::date
)
RETURNS TABLE(
    item_name text,
    serial_number text,
    serial_comment text,
    transaction_date date,
    transaction_type text,
    counterparty text,
    price numeric
)
LANGUAGE plpgsql
```

```
AS $$
BEGIN
    RETURN QUERY
    WITH purchase_history AS (
        SELECT
            i.item_id,
            i.item_name::TEXT AS item_name,
            pu.serial_number::TEXT AS serial_number,
            pu.serial_comment::TEXT AS serial_comment,
            p.invoice_date AS transaction_date,
            'PURCHASE'::TEXT AS transaction_type,
            v.party_name::TEXT AS counterparty,
            pi.unit_price AS price
        FROM PurchaseUnits pu
        JOIN PurchaseItems pi ON pu.purchase_item_id = pi.purchase_item_id
        JOIN PurchaseInvoices p ON pi.purchase_invoice_id = p.purchase_invoice_id
        JOIN Items i ON pi.item_id = i.item_id
        JOIN Parties v ON p.vendor_id = v.party_id
        WHERE i.item_name ILIKE ('%' || p_item_name || '%')
          AND (p_from_date IS NULL OR p.invoice_date >= p_from_date)
          AND (p_to_date IS NULL OR p.invoice_date <= p_to_date)
    ),
    sale_history AS (
        SELECT
            i.item_id,
            i.item_name::TEXT AS item_name,
            pu.serial_number::TEXT AS serial_number,
            pu.serial_comment::TEXT AS serial_comment,
            s.invoice_date AS transaction_date,
            'SALE'::TEXT AS transaction_type,
            c.party_name::TEXT AS counterparty,
            su.sold_price AS price
        FROM SoldUnits su
        JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
        JOIN SalesItems si ON su.sales_item_id = si.sales_item_id
        JOIN SalesInvoices s ON si.sales_invoice_id = s.sales_invoice_id
        JOIN Items i ON si.item_id = i.item_id
        JOIN Parties c ON s.customer_id = c.party_id
        WHERE i.item_name ILIKE ('%' || p_item_name || '%')
          AND (p_from_date IS NULL OR s.invoice_date >= p_from_date)
          AND (p_to_date IS NULL OR s.invoice_date <= p_to_date)
    )
    SELECT
        ph.item_name,
        ph.serial_number,
        ph.serial_comment,
        ph.transaction_date,
        ph.transaction_type,
        ph.counterparty,
        ph.price
    FROM (
        SELECT * FROM purchase_history
        UNION ALL
        SELECT * FROM sale_history
```

```
    ) AS ph
    ORDER BY ph.transaction_date,
             ph.transaction_type DESC,    -- ensures PURCHASE before SALE if same
date
             ph.serial_number;
END;
$$;
```

Parameters:

- `p_item_name text`
- `p_from_date date DEFAULT NULL::date`
- `p_to_date date DEFAULT NULL::date`

Returns:

`TABLE( item_name text, serial_number text, serial_comment text, transaction_date date, transaction_type text, counterparty text, price numeric )`

Purpose:

Item Transaction History - Performs specialized database operation.

Example SQL Call:

```
SELECT item_transaction_history(..., ..., ...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `get_item_stock_by_name()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION get_item_stock_by_name(p_item_name VARCHAR)
RETURNS TABLE (
    item_id_out TEXT,
    item_name_out VARCHAR(150),
    serial_number_out VARCHAR(100),
    serial_comment_out TEXT,
    quantity_out TEXT
)
LANGUAGE plpgsql
AS $$
BEGIN
```

```
    RETURN QUERY
    WITH stock AS (
        SELECT
            i.item_id,
            i.item_name,
            pu.serial_number,
            pu.serial_comment,
            COUNT(*) OVER () AS total_quantity,
            ROW_NUMBER() OVER (ORDER BY pu.serial_number) AS rn
        FROM purchaseunits pu
        JOIN purchaseitems pit ON pu.purchase_item_id = pit.purchase_item_id
        JOIN items i ON pit.item_id = i.item_id
        WHERE i.item_name = p_item_name
          AND pu.in_stock = true
    )
    SELECT
        CASE WHEN rn = 1 THEN item_id::TEXT ELSE '' END,
        CASE WHEN rn = 1 THEN item_name ELSE ''::VARCHAR END,
        serial_number,
        serial_comment,
        CASE WHEN rn = 1 THEN total_quantity::TEXT ELSE '' END
    FROM stock
    ORDER BY rn;
END;
$$;
```

Parameters:

- `p_item_name VARCHAR`

Returns:

`TABLE ( item_id_out TEXT, item_name_out VARCHAR(150), serial_number_out VARCHAR(100), serial_comment_out TEXT, quantity_out TEXT )`

Purpose:

Get Item Stock By Name - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_item_stock_by_name(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `get_serial_ledger()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.get_serial_ledger(p_serial text)
RETURNS TABLE(
    serial_number text,
    serial_comment text,
    item_name text,
    txn_date date,
    particulars text,
    reference text,
    qty_in integer,
    qty_out integer,
    balance integer,
    party_name text,
    purchase_price numeric,
    sale_price numeric,
    profit numeric
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY

    WITH item_info AS (
        SELECT
            pu.serial_number::text AS serial_number,
            pu.serial_comment::text AS serial_comment,
            i.item_name::text AS item_name
        FROM PurchaseUnits pu
        JOIN PurchaseItems pit ON pu.purchase_item_id = pit.purchase_item_id
        JOIN Items i ON pit.item_id = i.item_id
        WHERE pu.serial_number = p_serial
        LIMIT 1
    ),

    purchase AS (
        SELECT
            pi.invoice_date AS dt,
            'Purchase'::text AS particulars,
            pi.purchase_invoice_id::text AS reference,
            1 AS qty_in,
            0 AS qty_out,
            p.party_name::text AS party_name,
            pit.unit_price AS purchase_price,
            NULL::numeric AS sale_price
        FROM PurchaseUnits pu
        JOIN PurchaseItems pit ON pu.purchase_item_id = pit.purchase_item_id
        JOIN PurchaseInvoices pi ON pit.purchase_invoice_id =
pi.purchase_invoice_id
        JOIN Parties p ON pi.vendor_id = p.party_id
        WHERE pu.serial_number = p_serial
    ),
```

```
    purchase_return AS (
        SELECT
            pr.return_date AS dt,
            'Purchase Return'::text AS particulars,
            pr.purchase_return_id::text AS reference,
            0 AS qty_in,
            1 AS qty_out,
            p.party_name::text AS party_name,
            pri.unit_price AS purchase_price,
            NULL::numeric AS sale_price
        FROM PurchaseReturnItems pri
        JOIN PurchaseReturns pr ON pri.purchase_return_id = pr.purchase_return_id
        JOIN Parties p ON pr.vendor_id = p.party_id
        WHERE pri.serial_number = p_serial
    ),

    sale AS (
        SELECT
            si.invoice_date AS dt,
            'Sale'::text AS particulars,
            si.sales_invoice_id::text AS reference,
            0 AS qty_in,
            1 AS qty_out,
            c.party_name::text AS party_name,
            pit.unit_price AS purchase_price,
            su.sold_price AS sale_price
        FROM SoldUnits su
        JOIN SalesItems sitm ON su.sales_item_id = sitm.sales_item_id
        JOIN SalesInvoices si ON sitm.sales_invoice_id = si.sales_invoice_id
        JOIN Parties c ON si.customer_id = c.party_id
        JOIN PurchaseUnits pu ON su.unit_id = pu.unit_id
        JOIN PurchaseItems pit ON pu.purchase_item_id = pit.purchase_item_id
        WHERE pu.serial_number = p_serial
    ),

    sales_return AS (
        SELECT
            sr.return_date AS dt,
            'Sales Return'::text AS particulars,
            sr.sales_return_id::text AS reference,
            1 AS qty_in,
            0 AS qty_out,
            c.party_name::text AS party_name,
            sri.cost_price AS purchase_price,
            sri.sold_price AS sale_price
        FROM SalesReturnItems sri
        JOIN SalesReturns sr ON sri.sales_return_id = sr.sales_return_id
        JOIN Parties c ON sr.customer_id = c.party_id
        WHERE sri.serial_number = p_serial
    )

    SELECT
        ii.serial_number,
```

```
        ii.serial_comment,
        ii.item_name,
        l.dt AS txn_date,
        l.particulars,
        l.reference,
        l.qty_in,
        l.qty_out,
        CAST(SUM(l.qty_in - l.qty_out) OVER (ORDER BY l.dt, l.reference) AS INT)
AS balance,
        l.party_name,
        l.purchase_price,
        l.sale_price,
        CASE
            WHEN l.sale_price IS NOT NULL AND l.purchase_price IS NOT NULL
            THEN l.sale_price - l.purchase_price
        END AS profit
    FROM (
        SELECT * FROM purchase
        UNION ALL SELECT * FROM purchase_return
        UNION ALL SELECT * FROM sale
        UNION ALL SELECT * FROM sales_return
    ) l
    CROSS JOIN item_info ii
    ORDER BY l.dt, l.reference;

END;
$$;
```

Parameters:

- `p_serial text`

Returns:

`TABLE( serial_number text, serial_comment text, item_name text, txn_date date, particulars text, reference text, qty_in integer, qty_out integer, balance integer, party_name text, purchase_price numeric, sale_price numeric, profit numeric )`

Purpose:

Get Serial Ledger - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_serial_ledger(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 4: `stock_summary()`

Complete SQL Code:

```
CREATE OR REPLACE FUNCTION public.stock_summary()
RETURNS TABLE (
    item_id BIGINT,
    item_name VARCHAR,
    category VARCHAR,
    brand VARCHAR,
    quantity_in_stock BIGINT
)
LANGUAGE plpgsql
AS $$
BEGIN
    RETURN QUERY
    SELECT
        i.item_id,
        i.item_name,
        i.category,
        i.brand,
        COUNT(pu.unit_id) FILTER (WHERE pu.in_stock = TRUE) AS quantity_in_stock
    FROM Items i
    LEFT JOIN PurchaseItems pi ON i.item_id = pi.item_id
    LEFT JOIN PurchaseUnits pu ON pi.purchase_item_id = pu.purchase_item_id
    GROUP BY
        i.item_id,
        i.item_name,
        i.category,
        i.brand
    ORDER BY
        i.item_name ASC;
END;
$$;
```

Parameters:

- None

Returns:

`TABLE ( item_id BIGINT, item_name VARCHAR, category VARCHAR, brand VARCHAR, quantity_in_stock BIGINT )`

Purpose:

Stock Summary - Performs specialized database operation.

Example SQL Call:

```
SELECT stock_summary();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 5: `get_serial_number_details()`

Complete SQL Code:

```
CREATE FUNCTION public.get_serial_number_details(serial text) RETURNS
TABLE(serial_number character varying, item_name character varying, brand
character varying, category character varying, purchase_invoice_id bigint,
vendor_name character varying, purchase_date date, purchase_price numeric,
in_stock boolean, sales_invoice_id bigint, customer_name character varying,
sale_date date, sold_price numeric, current_status character varying)
    LANGUAGE plpgsql
    AS $$
BEGIN
    RETURN QUERY
    SELECT
        pu.serial_number,
        i.item_name,
        i.brand,
        i.category,
        pi.purchase_invoice_id,
        p.party_name AS vendor_name,
        pi.invoice_date AS purchase_date,
        pit.unit_price AS purchase_price,
        pu.in_stock,
        si.sales_invoice_id,
        c.party_name AS customer_name,
        si.invoice_date AS sale_date,
        su.sold_price,
        COALESCE(su.status, CASE WHEN pu.in_stock THEN 'In Stock' ELSE
'Sold/Unknown' END) AS current_status
    FROM PurchaseUnits pu
    JOIN PurchaseItems pit ON pu.purchase_item_id = pit.purchase_item_id
    JOIN Items i ON pit.item_id = i.item_id
    JOIN PurchaseInvoices pi ON pit.purchase_invoice_id = pi.purchase_invoice_id
    JOIN Parties p ON pi.vendor_id = p.party_id
    LEFT JOIN SoldUnits su ON su.unit_id = pu.unit_id
    LEFT JOIN SalesItems si_itm ON su.sales_item_id = si_itm.sales_item_id
    LEFT JOIN SalesInvoices si ON si_itm.sales_invoice_id = si.sales_invoice_id
    LEFT JOIN Parties c ON si.customer_id = c.party_id
    WHERE pu.serial_number = serial;
```

```
END;
$$;
```

Parameters:

- `serial text`

Returns:

`TABLE(serial_number character varying, item_name character varying, brand character varying, category character varying, purchase_invoice_id bigint, vendor_name character varying, purchase_date date, purchase_price numeric, in_stock boolean, sales_invoice_id bigint, customer_name character varying, sale_date date, sold_price numeric, current_status character varying)`

Purpose:

Get Serial Number Details - Retrieves data from the database in JSON format.

Example SQL Call:

```
SELECT get_serial_number_details(...);
```

Function Behavior:

See complete SQL implementation above for detailed logic.

TRIGGER FUNCTIONS

Total Functions: 3

Function 1: `trg_party_opening_balance()`

Complete SQL Code:

```
CREATE FUNCTION public.trg_party_opening_balance() RETURNS trigger
    LANGUAGE plpgsql
    AS $$
DECLARE
    j_id BIGINT;
    debit_acc BIGINT;
    credit_acc BIGINT;
    cap_acc BIGINT;
BEGIN
    IF NEW.opening_balance > 0 THEN
```

```
        -- Owner's Capital account
        SELECT account_id INTO cap_acc
        FROM ChartOfAccounts
        WHERE account_name = 'Owner''s Capital';

        IF cap_acc IS NULL THEN
            RAISE EXCEPTION 'Owner''s Capital account not found in COA';
        END IF;

        -- Create a new Journal Entry
        INSERT INTO JournalEntries(entry_date, description)
        VALUES (CURRENT_DATE, 'Opening Balance for ' || NEW.party_name)
        RETURNING journal_id INTO j_id;

        -- ---------------------------
        -- CUSTOMER or BOTH
        -- ---------------------------
        IF NEW.party_type IN ('Customer','Both') AND NEW.balance_type = 'Debit'
THEN
            debit_acc := NEW.ar_account_id;
            credit_acc := cap_acc;

            INSERT INTO JournalLines(journal_id, account_id, party_id, debit)
            VALUES (j_id, debit_acc, NEW.party_id, NEW.opening_balance);

            INSERT INTO JournalLines(journal_id, account_id, credit)
            VALUES (j_id, credit_acc, NEW.opening_balance);
        END IF;

        -- ---------------------------
        -- VENDOR or BOTH
        -- ---------------------------
        IF NEW.party_type IN ('Vendor','Both') AND NEW.balance_type = 'Credit'
THEN
            debit_acc := cap_acc;
            credit_acc := NEW.ap_account_id;

            INSERT INTO JournalLines(journal_id, account_id, debit)
            VALUES (j_id, debit_acc, NEW.opening_balance);

            INSERT INTO JournalLines(journal_id, account_id, party_id, credit)
            VALUES (j_id, credit_acc, NEW.party_id, NEW.opening_balance);
        END IF;

        -- ---------------------------
        -- EXPENSE PARTY
        -- ---------------------------
        IF NEW.party_type = 'Expense' THEN
            debit_acc := NEW.ap_account_id;  -- Expense account
            credit_acc := cap_acc;           -- Funded by Owner's Capital

            INSERT INTO JournalLines(journal_id, account_id, party_id, debit)
            VALUES (j_id, debit_acc, NEW.party_id, NEW.opening_balance);
```

```
            INSERT INTO JournalLines(journal_id, account_id, credit)
            VALUES (j_id, credit_acc, NEW.opening_balance);
        END IF;
    END IF;

    RETURN NEW;
END;
$$;
```

Parameters:

- None

Returns:

`trigger`

Purpose:

Trg Party Opening Balance - Performs specialized database operation.

Example SQL Call:

```
SELECT trg_party_opening_balance();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 2: `trg_payment_journal()`

Complete SQL Code:

```
CREATE FUNCTION public.trg_payment_journal() RETURNS trigger
    LANGUAGE plpgsql
    AS $$
DECLARE
    j_id BIGINT;
    party_acc BIGINT;
    v_party_name  TEXT;
    journal_desc TEXT;
BEGIN
    -- Handle DELETE: remove related journal
    IF TG_OP = 'DELETE' THEN
        DELETE FROM JournalEntries WHERE journal_id = OLD.journal_id;
        RETURN OLD;
```

```
    END IF;

    -- Handle UPDATE: only regenerate if relevant fields changed
    IF TG_OP = 'UPDATE' THEN
        IF OLD.amount = NEW.amount
           AND OLD.account_id = NEW.account_id
           AND OLD.party_id = NEW.party_id
           AND OLD.description IS NOT DISTINCT FROM NEW.description
           AND OLD.payment_date = NEW.payment_date THEN
            RETURN NEW;
        END IF;

        DELETE FROM JournalEntries WHERE journal_id = OLD.journal_id;
    END IF;

    -- Handle INSERT or UPDATE
    IF TG_OP IN ('INSERT','UPDATE') THEN
        -- Find AP account for vendor
        SELECT ap_account_id, p.party_name
        INTO party_acc, v_party_name
        FROM Parties AS p
        WHERE party_id = NEW.party_id;

        IF party_acc IS NULL THEN
            RAISE EXCEPTION 'No AP account found for vendor %', NEW.party_id;
        END IF;

        -- Description: custom if provided, else fallback with ref no
        journal_desc := COALESCE(
            NEW.description,
            'Payment to ' || v_party_name ||
            CASE WHEN NEW.reference_no IS NOT NULL AND NEW.reference_no <> ''
                 THEN ' (Ref: ' || NEW.reference_no || ')'
                 ELSE '' END
        );

        -- Insert Journal Entry
        INSERT INTO JournalEntries(entry_date, description)
        VALUES (NEW.payment_date, journal_desc)
        RETURNING journal_id INTO j_id;

        -- Prevent recursion when linking back
        PERFORM pg_catalog.set_config('session_replication_role', 'replica',
true);
        UPDATE Payments
        SET journal_id = j_id
        WHERE payment_id = NEW.payment_id;
        PERFORM pg_catalog.set_config('session_replication_role', 'origin', true);

        -- Debit Vendor (reduce liability)
        INSERT INTO JournalLines(journal_id, account_id, party_id, debit)
        VALUES (j_id, party_acc, NEW.party_id, NEW.amount);

        -- Credit Cash/Bank
```

```
        INSERT INTO JournalLines(journal_id, account_id, credit)
        VALUES (j_id, NEW.account_id, NEW.amount);
    END IF;

    RETURN NEW;
END;
$$;
```

Parameters:

- None

Returns:

`trigger`

Purpose:

Trg Payment Journal - Performs specialized database operation.

Example SQL Call:

```
SELECT trg_payment_journal();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

Function 3: `trg_receipt_journal()`

Complete SQL Code:

```
CREATE FUNCTION public.trg_receipt_journal() RETURNS trigger
    LANGUAGE plpgsql
    AS $$
DECLARE
    j_id BIGINT;
    party_acc BIGINT;
    v_party_name  TEXT;
    journal_desc TEXT;
BEGIN
    -- Handle DELETE: remove related journal
    IF TG_OP = 'DELETE' THEN
        DELETE FROM JournalEntries WHERE journal_id = OLD.journal_id;
        RETURN OLD;
    END IF;
```

```
    -- Handle UPDATE: only regenerate if relevant fields changed
    IF TG_OP = 'UPDATE' THEN
        IF OLD.amount = NEW.amount
           AND OLD.account_id = NEW.account_id
           AND OLD.party_id = NEW.party_id
           AND OLD.description IS NOT DISTINCT FROM NEW.description
           AND OLD.receipt_date = NEW.receipt_date THEN
            RETURN NEW;
        END IF;

        DELETE FROM JournalEntries WHERE journal_id = OLD.journal_id;
    END IF;

    -- Handle INSERT or UPDATE
    IF TG_OP IN ('INSERT','UPDATE') THEN
        -- Find AR account for customer
        SELECT ar_account_id, p.party_name
        INTO party_acc, v_party_name
        FROM Parties AS p
        WHERE party_id = NEW.party_id;

        IF party_acc IS NULL THEN
            RAISE EXCEPTION 'No AR account found for customer %', NEW.party_id;
        END IF;

        -- Description: custom if provided, else fallback with ref no
        journal_desc := COALESCE(
            NEW.description,
            'Receipt from ' || v_party_name ||
            CASE WHEN NEW.reference_no IS NOT NULL AND NEW.reference_no <> ''
                 THEN ' (Ref: ' || NEW.reference_no || ')'
                 ELSE '' END
        );

        -- Insert Journal Entry
        INSERT INTO JournalEntries(entry_date, description)
        VALUES (NEW.receipt_date, journal_desc)
        RETURNING journal_id INTO j_id;

        -- Prevent recursion when linking back
        PERFORM pg_catalog.set_config('session_replication_role', 'replica',
true);
        UPDATE Receipts
        SET journal_id = j_id
        WHERE receipt_id = NEW.receipt_id;
        PERFORM pg_catalog.set_config('session_replication_role', 'origin', true);

        -- Debit Cash/Bank (increase asset)
        INSERT INTO JournalLines(journal_id, account_id, debit)
        VALUES (j_id, NEW.account_id, NEW.amount);

        -- Credit Customer (reduce receivable)
        INSERT INTO JournalLines(journal_id, account_id, party_id, credit)
```

```
            VALUES (j_id, party_acc, NEW.party_id, NEW.amount);
        END IF;

        RETURN NEW;
    END;
    $$;
```

Parameters:

- None

Returns:

`trigger`

Purpose:

Trg Receipt Journal - Performs specialized database operation.

Example SQL Call:

```
SELECT trg_receipt_journal();
```

Function Behavior:

See complete SQL implementation above for detailed logic.

DATABASE VIEWS

Overview

This document provides comprehensive documentation for all database views in the ERP system. Views are virtual tables that provide simplified access to complex queries and reporting data.

Total Views: 4

Table of Contents

---

1. vw_trial_balance

Purpose

Generates a complete trial balance report showing all account balances and party balances, properly classified by account type.

Complete SQL Code

```
CREATE VIEW public.vw_trial_balance AS
 WITH journal_summary AS (
         SELECT jl.account_id,
            jl.party_id,
            COALESCE(sum(jl.debit), (0)::numeric) AS debit,
            COALESCE(sum(jl.credit), (0)::numeric) AS credit
           FROM public.journallines jl
          GROUP BY jl.account_id, jl.party_id
        ), account_totals AS (
         SELECT coa.account_id,
            coa.account_code,
            coa.account_name,
            coa.account_type,
            sum(js.debit) AS total_debit,
            sum(js.credit) AS total_credit
           FROM (public.chartofaccounts coa
             LEFT JOIN journal_summary js ON (((coa.account_id = js.account_id)
AND (((coa.account_name)::text = ANY (ARRAY[('Accounts Receivable'::character
varying)::text, ('Accounts Payable'::character varying)::text])) OR (js.party_id
IS NULL)))))
          WHERE (NOT (coa.account_id IN ( SELECT DISTINCT p.ap_account_id
                   FROM public.parties p
                  WHERE (((p.party_type)::text = 'Expense'::text) AND
(p.ap_account_id IS NOT NULL)))))
          GROUP BY coa.account_id, coa.account_code, coa.account_name,
coa.account_type
        ), party_totals AS (
         SELECT p.party_id,
            p.party_name,
            p.party_type,
            COALESCE(sum(js.debit), (0)::numeric) AS total_debit,
            COALESCE(sum(js.credit), (0)::numeric) AS total_credit,
            (COALESCE(sum(js.debit), (0)::numeric) - COALESCE(sum(js.credit),
(0)::numeric)) AS balance
           FROM (public.parties p
             LEFT JOIN journal_summary js ON ((js.party_id = p.party_id)))
          GROUP BY p.party_id, p.party_name, p.party_type
        ), classified_parties AS (
         SELECT pt.party_id,
            pt.party_name,
            pt.party_type,
            pt.total_debit,
            pt.total_credit,
            pt.balance,
                CASE
```

```
                    WHEN (((pt.party_type)::text = 'Customer'::text) AND
(pt.balance < (0)::numeric)) THEN 'Accounts Payable'::text
                    WHEN (((pt.party_type)::text = 'Vendor'::text) AND (pt.balance
> (0)::numeric)) THEN 'Accounts Receivable'::text
                    WHEN ((pt.party_type)::text = 'Both'::text) THEN
                    CASE
                        WHEN (pt.balance >= (0)::numeric) THEN 'Accounts
Receivable'::text
                        ELSE 'Accounts Payable'::text
                    END
                    WHEN ((pt.party_type)::text = 'Customer'::text) THEN 'Accounts
Receivable'::text
                    WHEN ((pt.party_type)::text = 'Vendor'::text) THEN 'Accounts
Payable'::text
                    ELSE 'Expense Party'::text
                END AS effective_type
           FROM party_totals pt
        ), control_adjustment AS (
         SELECT classified_parties.effective_type AS account_name,
            sum(GREATEST(classified_parties.balance, (0)::numeric)) AS debit_side,
            sum(abs(LEAST(classified_parties.balance, (0)::numeric))) AS
credit_side
           FROM classified_parties
          WHERE (classified_parties.effective_type = ANY (ARRAY['Accounts
Receivable'::text, 'Accounts Payable'::text]))
          GROUP BY classified_parties.effective_type
        )
 SELECT at.account_code AS code,
    at.account_name AS name,
    at.account_type AS type,
    COALESCE(ca.debit_side, at.total_debit, (0)::numeric) AS total_debit,
    COALESCE(ca.credit_side, at.total_credit, (0)::numeric) AS total_credit,
    (COALESCE(ca.debit_side, at.total_debit, (0)::numeric) -
COALESCE(ca.credit_side, at.total_credit, (0)::numeric)) AS balance
   FROM (account_totals at
     LEFT JOIN control_adjustment ca ON ((ca.account_name =
(at.account_name)::text)))
UNION ALL
 SELECT NULL::character varying AS code,
    pt.party_name AS name,
    pt.effective_type AS type,
    pt.total_debit,
    pt.total_credit,
    pt.balance
   FROM classified_parties pt
  WHERE ((pt.total_debit <> (0)::numeric) OR (pt.total_credit <> (0)::numeric))
  ORDER BY 1 NULLS FIRST, 2;
```

Output Columns

Column	Type	Description

Column	Type	Description
code	VARCHAR	Account code (NULL for party entries)
name	VARCHAR	Account name or party name
type	VARCHAR	Account type or effective classification
total_debit	NUMERIC	Total debit amount
total_credit	NUMERIC	Total credit amount
balance	NUMERIC	Net balance (debit - credit)

How It Works

The view uses multiple Common Table Expressions (CTEs) to build the trial balance:

1. journal_summary CTE

- Aggregates all journal lines by account_id and party_id
- Calculates total debits and credits for each combination

2. account_totals CTE

- Summarizes balances for all Chart of Accounts entries
- Excludes expense-type parties (they're handled separately)
- Handles AR/AP accounts specially to exclude party-level detail at account level

3. party_totals CTE

- Calculates balances for all parties
- Computes running balance (debit - credit)

4. classified_parties CTE

- Classifies each party into its effective type
- **Logic:**
  - Customer with credit balance → "Accounts Payable" (they owe us but balance is negative)
  - Vendor with debit balance → "Accounts Receivable" (we owe them but balance is positive)
  - "Both" type → Classified based on balance sign
  - Expense parties → "Expense Party"

5. control_adjustment CTE

- Aggregates party balances by effective type (AR/AP)
- Separates debit-side and credit-side amounts
- Used to adjust control account (AR/AP) totals

6. Final SELECT

- Combines account-level totals with party-level details
- Shows account totals first (with adjusted AR/AP)
- Then shows individual party details
- Orders by code (NULLs first for parties), then by name

Usage Example

```
-- Get complete trial balance
SELECT * FROM vw_trial_balance;

-- Get only account-level totals (no party detail)
SELECT * FROM vw_trial_balance WHERE code IS NOT NULL;

-- Get only party details
SELECT * FROM vw_trial_balance WHERE code IS NULL;

-- Get trial balance for specific account type
SELECT * FROM vw_trial_balance WHERE type = 'Asset';

-- Verify trial balance (debits should equal credits)
SELECT
    SUM(total_debit) as total_debits,
    SUM(total_credit) as total_credits,
    SUM(total_debit) - SUM(total_credit) as difference
FROM vw_trial_balance
WHERE code IS NOT NULL;
```

Business Logic

Key Features:

1. **Control Account Reconciliation**: AR and AP control accounts are adjusted to match subsidiary ledger (party balances)
2. **Party Classification**: Parties are dynamically classified based on their balance
3. **Expense Party Handling**: Expense-type parties shown separately from regular expense accounts
4. **Complete Audit Trail**: Shows both summary and detail views

Expected Output:

```
code  | name                    | type              | total_debit | total_credit |
balance
------|-------------------------|-------------------|-------------|--------------
|--------
1000  | Cash                    | Asset             | 500000.00   | 300000.00    |
200000.00
1100  | Accounts Receivable     | Asset             | 250000.00   | 100000.00    |
150000.00
NULL  | ABC Corp                | Accounts Receivable| 50000.00    | 20000.00
| 30000.00
```

```
NULL  | XYZ Ltd                | Accounts Receivable| 30000.00    | 10000.00
| 20000.00
...
```

---

2. standing_company_worth_view

Purpose

Provides a comprehensive financial standing report showing the company's financial position (Balance Sheet) and profitability (P&L) in JSON format.

Complete SQL Code

```
CREATE VIEW public.standing_company_worth_view AS
 WITH journal_summary AS (
         SELECT jl.account_id,
            jl.party_id,
            COALESCE(sum(jl.debit), (0)::numeric) AS debit,
            COALESCE(sum(jl.credit), (0)::numeric) AS credit
           FROM public.journallines jl
          GROUP BY jl.account_id, jl.party_id
        ), account_totals AS (
         SELECT coa.account_id,
            coa.account_code,
            coa.account_name,
            coa.account_type,
            COALESCE(sum(js.debit), (0)::numeric) AS total_debit,
            COALESCE(sum(js.credit), (0)::numeric) AS total_credit
           FROM (public.chartofaccounts coa
             LEFT JOIN journal_summary js ON (((coa.account_id = js.account_id)
AND (((coa.account_name)::text = ANY (ARRAY[('Accounts Receivable'::character
varying)::text, ('Accounts Payable'::character varying)::text])) OR (js.party_id
IS NULL)))))
          WHERE (NOT (coa.account_id IN ( SELECT DISTINCT p.ap_account_id
                   FROM public.parties p
                  WHERE (((p.party_type)::text = 'Expense'::text) AND
(p.ap_account_id IS NOT NULL)))))
          GROUP BY coa.account_id, coa.account_code, coa.account_name,
coa.account_type
        ), party_totals AS (
         SELECT p.party_id,
            p.party_name,
            p.party_type,
            COALESCE(sum(js.debit), (0)::numeric) AS total_debit,
            COALESCE(sum(js.credit), (0)::numeric) AS total_credit,
            (COALESCE(sum(js.debit), (0)::numeric) - COALESCE(sum(js.credit),
(0)::numeric)) AS balance
           FROM (public.parties p
             LEFT JOIN journal_summary js ON ((js.party_id = p.party_id)))
          GROUP BY p.party_id, p.party_name, p.party_type
```

```
        ), classified_parties AS (
         SELECT pt.party_id,
            pt.party_name,
            pt.party_type,
            pt.total_debit,
            pt.total_credit,
            pt.balance,
                CASE
                    WHEN (((pt.party_type)::text = 'Customer'::text) AND
(pt.balance < (0)::numeric)) THEN 'Accounts Payable'::text
                    WHEN (((pt.party_type)::text = 'Vendor'::text) AND (pt.balance
> (0)::numeric)) THEN 'Accounts Receivable'::text
                    WHEN ((pt.party_type)::text = 'Both'::text) THEN
                    CASE
                        WHEN (pt.balance >= (0)::numeric) THEN 'Accounts
Receivable'::text
                        ELSE 'Accounts Payable'::text
                    END
                    WHEN ((pt.party_type)::text = 'Customer'::text) THEN 'Accounts
Receivable'::text
                    WHEN ((pt.party_type)::text = 'Vendor'::text) THEN 'Accounts
Payable'::text
                    ELSE 'Expense Party'::text
                END AS effective_type
           FROM party_totals pt
        ), control_adjustment AS (
         SELECT classified_parties.effective_type AS account_name,
            sum(GREATEST(classified_parties.balance, (0)::numeric)) AS debit_side,
            sum(abs(LEAST(classified_parties.balance, (0)::numeric))) AS
credit_side
           FROM classified_parties
          WHERE (classified_parties.effective_type = ANY (ARRAY['Accounts
Receivable'::text, 'Accounts Payable'::text]))
          GROUP BY classified_parties.effective_type
        ), merged_totals AS (
         SELECT coa.account_type,
                CASE
                    WHEN ((coa.account_type)::text = ANY
(ARRAY[('Asset'::character varying)::text, ('Expense'::character varying)::text]))
THEN (COALESCE(ca.debit_side, at.total_debit, (0)::numeric) -
COALESCE(ca.credit_side, at.total_credit, (0)::numeric))
                    WHEN ((coa.account_type)::text = ANY
(ARRAY[('Liability'::character varying)::text, ('Equity'::character
varying)::text, ('Revenue'::character varying)::text])) THEN
(COALESCE(ca.credit_side, at.total_credit, (0)::numeric) - COALESCE(ca.debit_side,
at.total_debit, (0)::numeric))
                    ELSE (0)::numeric
                END AS net_balance
           FROM ((account_totals at
             JOIN public.chartofaccounts coa ON ((at.account_id =
coa.account_id)))
             LEFT JOIN control_adjustment ca ON ((ca.account_name =
(coa.account_name)::text)))
        ), summary AS (
```

```
          SELECT merged_totals.account_type,
             sum(merged_totals.net_balance) AS total
            FROM merged_totals
           GROUP BY merged_totals.account_type
         ), party_expenses AS (
          SELECT sum(classified_parties.balance) AS total_party_expenses
            FROM classified_parties
           WHERE (classified_parties.effective_type = 'Expense Party'::text)
         ), totals AS (
          SELECT COALESCE(sum(
                CASE
                    WHEN ((summary.account_type)::text = 'Asset'::text) THEN
summary.total
                    ELSE NULL::numeric
                END), (0)::numeric) AS assets,
             COALESCE(sum(
                CASE
                    WHEN ((summary.account_type)::text = 'Liability'::text) THEN
summary.total
                    ELSE NULL::numeric
                END), (0)::numeric) AS liabilities,
             COALESCE(sum(
                CASE
                    WHEN ((summary.account_type)::text = 'Equity'::text) THEN
summary.total
                    ELSE NULL::numeric
                END), (0)::numeric) AS equity,
             COALESCE(sum(
                CASE
                    WHEN ((summary.account_type)::text = 'Revenue'::text) THEN
summary.total
                    ELSE NULL::numeric
                END), (0)::numeric) AS revenue,
             (COALESCE(sum(
                CASE
                    WHEN ((summary.account_type)::text = 'Expense'::text) THEN
summary.total
                    ELSE NULL::numeric
                END), (0)::numeric) + COALESCE(( SELECT
party_expenses.total_party_expenses
                   FROM party_expenses), (0)::numeric)) AS expenses
            FROM summary
         )
  SELECT json_build_object('financial_position', json_build_object('total_assets',
round(assets, 2), 'total_liabilities', round(liabilities, 2), 'total_equity',
round(equity, 2), 'net_worth', round((assets - liabilities), 2)),
'profit_and_loss', json_build_object('total_revenue', round(revenue, 2),
'total_expenses', round(expenses, 2), 'net_profit_loss', round((revenue -
expenses), 2))) AS company_standing
    FROM totals;
```

Output Structure

Returns: Single JSON object

JSON Output Format

```
{
  "financial_position": {
    "total_assets": 1500000.00,
    "total_liabilities": 500000.00,
    "total_equity": 800000.00,
    "net_worth": 1000000.00
  },
  "profit_and_loss": {
    "total_revenue": 2000000.00,
    "total_expenses": 1500000.00,
    "net_profit_loss": 500000.00
  }
}
```

How It Works

The view uses a series of CTEs similar to `vw_trial_balance`, then adds additional calculations:

Additional CTEs Beyond Trial Balance Base:

6. merged_totals CTE

- Calculates net balance for each account type
- **For Assets & Expenses**: Net = Debit - Credit (normal debit balance)
- **For Liabilities, Equity, Revenue**: Net = Credit - Debit (normal credit balance)

7. summary CTE

- Aggregates totals by account type (Asset, Liability, Equity, Revenue, Expense)

8. party_expenses CTE

- Separately calculates total for expense-type parties
- These are added to regular expenses

9. totals CTE

- Pivots summary data into individual columns
- Adds party expenses to regular expenses

10. Final SELECT

- Builds nested JSON structure
- Calculates derived values:
  - **Net Worth** = Total Assets - Total Liabilities
  - **Net Profit/Loss** = Total Revenue - Total Expenses

- Rounds all values to 2 decimal places

Usage Example

```
-- Get complete company standing
SELECT * FROM standing_company_worth_view;

-- Extract specific values from JSON
SELECT
    company_standing->'financial_position'->>'total_assets' as total_assets,
    company_standing->'financial_position'->>'net_worth' as net_worth,
    company_standing->'profit_and_loss'->>'net_profit_loss' as net_profit
FROM standing_company_worth_view;

-- Pretty print the JSON
SELECT jsonb_pretty(company_standing::jsonb)
FROM standing_company_worth_view;
```

Business Logic

Financial Position (Balance Sheet):

- **Total Assets** = Sum of all asset account balances (debit - credit)
- **Total Liabilities** = Sum of all liability account balances (credit - debit)
- **Total Equity** = Sum of all equity account balances (credit - debit)
- **Net Worth** = Total Assets - Total Liabilities (should equal Total Equity)

Profit and Loss (Income Statement):

- **Total Revenue** = Sum of all revenue account balances (credit - debit)
- **Total Expenses** = Sum of all expense accounts + expense-type parties (debit - credit)
- **Net Profit/Loss** = Total Revenue - Total Expenses

Accounting Equation Verification:

```
Assets = Liabilities + Equity
Net Worth = Assets - Liabilities
Net Worth ≈ Equity (should be equal)
```

---

3. stock_worth_report

Purpose

Provides detailed inventory valuation report showing purchase price vs market price for all items in stock, with running totals.

Complete SQL Code

```
CREATE VIEW public.stock_worth_report AS
WITH stock AS (
    SELECT
        i.item_id,
        i.item_name,
        COUNT(pu.unit_id) OVER (PARTITION BY i.item_id) AS quantity,
        pu.serial_number,
        pu.serial_comment,
        pit.unit_price AS purchase_price,
        i.sale_price AS market_price,
        ROW_NUMBER() OVER (PARTITION BY i.item_id ORDER BY pu.serial_number) AS rn
    FROM public.purchaseunits pu
    JOIN public.purchaseitems pit ON pu.purchase_item_id = pit.purchase_item_id
    JOIN public.items i ON pit.item_id = i.item_id
    WHERE pu.in_stock = true
),
running AS (
    SELECT
        stock.item_id,
        stock.item_name,
        stock.quantity,
        stock.serial_number,
        stock.serial_comment,
        stock.purchase_price,
        stock.market_price,
        SUM(stock.purchase_price) OVER (ORDER BY stock.item_id, stock.rn) AS
running_total_purchase,
        SUM(stock.market_price) OVER (ORDER BY stock.item_id, stock.rn) AS
running_total_market,
        stock.rn
    FROM stock
)
SELECT
    CASE WHEN rn = 1 THEN item_id::TEXT ELSE ''::TEXT END AS item_id,
    CASE WHEN rn = 1 THEN item_name ELSE ''::VARCHAR END AS item_name,
    CASE WHEN rn = 1 THEN quantity::TEXT ELSE ''::TEXT END AS quantity,
    serial_number,
    serial_comment,
    purchase_price,
    market_price,
    running_total_purchase,
    running_total_market
FROM running
ORDER BY item_id::INTEGER, rn;
```

Output Columns

Column	Type	Description
item_id	TEXT	Item ID (shown only on first row per item)

Column	Type	Description
item_name	VARCHAR	Item name (shown only on first row per item)
quantity	TEXT	Total quantity in stock (shown only on first row per item)
serial_number	VARCHAR	Serial/IMEI number
serial_comment	TEXT	Optional comment for this serial
purchase_price	NUMERIC	Original purchase price for this unit
market_price	NUMERIC	Current market/sale price
running_total_purchase	NUMERIC	Running total of purchase prices
running_total_market	NUMERIC	Running total of market prices

How It Works

1. stock CTE

- Joins PurchaseUnits → PurchaseItems → Items
- Filters only items where `in_stock = true`
- Uses window function `COUNT() OVER (PARTITION BY item_id)` to get quantity per item
- Assigns row number within each item group

2. running CTE

- Calculates running totals using window functions
- `SUM() OVER (ORDER BY item_id, rn)` creates cumulative sums
- Running totals show cumulative inventory value

3. Final SELECT

- Display logic: Shows item_id, item_name, quantity only on first row (rn=1)
- Subsequent rows for same item show empty strings for grouping clarity
- Each serial number gets its own row

Usage Example

```
-- Get complete stock worth report
SELECT * FROM stock_worth_report;

-- Get total inventory value
SELECT
    MAX(running_total_purchase) as total_cost,
    MAX(running_total_market) as total_market_value,
    MAX(running_total_market) - MAX(running_total_purchase) as potential_profit
FROM stock_worth_report;

-- Get stock worth for specific item
```

```
SELECT * FROM stock_worth_report
WHERE item_id::INTEGER = 5 OR item_name LIKE '%iPhone%';

-- Count total items in stock
SELECT COUNT(DISTINCT item_id::INTEGER) as total_items,
       SUM(CASE WHEN quantity <> '' THEN quantity::INTEGER ELSE 0 END) as
total_units
FROM stock_worth_report;
```

Sample Output

```
item_id | item_name      | quantity | serial_number  | serial_comment |
purchase_price | market_price | running_total_purchase | running_total_market
--------|----------------|----------|----------------|----------------|-----------
-----|--------------|----------------------|-------------------
1       | iPhone 15 Pro  | 3        | IMEI001        | Black 256GB    | 45000.00
| 52000.00     | 45000.00             | 52000.00
        |                |          | IMEI002        | White 512GB    | 46000.00
| 52000.00     | 91000.00             | 104000.00
        |                |          | IMEI003        | NULL           | 45500.00
| 52000.00     | 136500.00            | 156000.00
2       | MacBook Pro    | 1        | MBP001         | M3 Max         | 120000.00
| 135000.00    | 256500.00            | 291000.00
```

Business Logic

Key Features:

1. **Inventory Valuation**: Shows both cost basis (purchase_price) and current value (market_price)
2. **Serial-Level Detail**: Every unit tracked individually
3. **Running Totals**: Cumulative values help track total inventory worth
4. **Profit Potential**: Difference between market and purchase price shows potential profit
5. **Visual Grouping**: Empty strings for item details after first row make report easier to read

Calculations:

- **Total Cost** = Sum of all purchase_price (last running_total_purchase)
- **Total Market Value** = Sum of all market_price (last running_total_market)
- **Unrealized Profit** = Total Market Value - Total Cost

---

4. stock_report

Purpose

Simplified stock report showing current inventory with serial numbers and comments, without pricing information.

Complete SQL Code

```sql
CREATE VIEW public.stock_report AS
WITH stock AS (
    SELECT
        i.item_id,
        i.item_name,
        COUNT(pu.unit_id) OVER (PARTITION BY i.item_id) AS quantity,
        pu.serial_number,
        pu.serial_comment,
        ROW_NUMBER() OVER (PARTITION BY i.item_id ORDER BY pu.serial_number) AS rn
    FROM public.purchaseunits pu
    JOIN public.purchaseitems pit ON pu.purchase_item_id = pit.purchase_item_id
    JOIN public.items i ON pit.item_id = i.item_id
    WHERE pu.in_stock = true
)
SELECT
    CASE WHEN rn = 1 THEN item_id::TEXT ELSE ''::TEXT END AS item_id,
    CASE WHEN rn = 1 THEN item_name ELSE ''::VARCHAR END AS item_name,
    CASE WHEN rn = 1 THEN quantity::TEXT ELSE ''::TEXT END AS quantity,
    serial_number,
    serial_comment
FROM stock
ORDER BY item_id::INTEGER, rn;
```

Output Columns

Column	Type	Description
item_id	TEXT	Item ID (shown only on first row per item)
item_name	VARCHAR	Item name (shown only on first row per item)
quantity	TEXT	Total quantity in stock (shown only on first row per item)
serial_number	VARCHAR	Serial/IMEI number
serial_comment	TEXT	Optional comment for this serial

How It Works

1. stock CTE

- Similar to `stock_worth_report` but without pricing columns
- Joins PurchaseUnits → PurchaseItems → Items
- Filters `in_stock = true`
- Counts quantity per item using window function
- Assigns row numbers for ordering

2. Final SELECT

- Same display logic as `stock_worth_report`
- Shows item details only on first row (rn=1)

- Each serial gets its own row

Usage Example

```
-- Get complete stock report
SELECT * FROM stock_report;

-- Count items by category
SELECT
    item_name,
    MAX(quantity::INTEGER) as qty
FROM stock_report
WHERE quantity <> ''
GROUP BY item_name
ORDER BY qty DESC;

-- Search for specific serial
SELECT * FROM stock_report
WHERE serial_number LIKE '%IMEI001%';

-- Get all serials for an item
SELECT serial_number, serial_comment
FROM stock_report
WHERE item_name = 'iPhone 15 Pro';

-- Total units in stock
SELECT SUM(quantity::INTEGER) as total_units
FROM stock_report
WHERE quantity <> '';
```

Sample Output

```
item_id | item_name      | quantity | serial_number  | serial_comment
--------|----------------|----------|----------------|-----------------
1       | iPhone 15 Pro  | 3        | IMEI001        | Black 256GB
        |                |          | IMEI002        | White 512GB
        |                |          | IMEI003        | NULL
2       | MacBook Pro    | 1        | MBP001         | M3 Max 16-inch
3       | iPad Air       | 5        | IPAD001        | Blue
        |                |          | IPAD002        | Pink
        |                |          | IPAD003        | NULL
        |                |          | IPAD004        | Space Gray
        |                |          | IPAD005        | NULL
```

Business Logic

Key Features:

1. **Simplified View**: No pricing information - pure inventory count

2. **Serial Tracking**: Complete list of all serials in stock
3. **Comments Visible**: Shows any notes/comments added during purchase
4. **Clean Display**: Grouped by item for easy reading

Use Cases:

- Quick stock check
- Serial number lookup
- Inventory count verification
- Non-financial stock reporting

Difference from stock_worth_report:

- No pricing information (simpler, faster)
- No running totals
- Suitable for warehouse/operations staff (not accounting)

📊 View Comparison Summary

Feature	vw_trial_balance	standing_company_worth_view	stock_worth_report	stock_report
Purpose	Trial balance	Financial summary	Inventory valuation	Simple stock list
Output Format	Rows	JSON	Rows	Rows
Includes Pricing	No	Yes (aggregated)	Yes (detailed)	No
Party Detail	Yes	No	No	No
Serial Detail	No	No	Yes	Yes
Running Totals	No	No	Yes	No
Primary Users	Accountants	Management	Accounting/Inventory	Warehouse

---

🔧 Common View Patterns

Pattern 1: Window Functions for Grouping

All stock views use:

```
COUNT(*) OVER (PARTITION BY item_id) -- Total per group
ROW_NUMBER() OVER (PARTITION BY item_id ORDER BY serial) -- Row number within
```

```
group
```

Pattern 2: Conditional Display

All stock views use:

```
CASE WHEN rn = 1 THEN value ELSE '' END
```

This shows grouped values only once for readability.

Pattern 3: CTEs for Complex Logic

All views use Common Table Expressions (WITH clauses) to:

- Break complex queries into logical steps
- Improve readability and maintenance
- Enable reuse of intermediate results

Usage Best Practices

Performance Considerations

1. **Views are Recomputed**: Views execute the query each time accessed
2. **Index Recommendations**:
    - Index on `PurchaseUnits.in_stock`
    - Index on `JournalLines.account_id, party_id`
    - Index on `Parties.party_type`
3. **Materialized Views** (if needed for performance):

```
-- Convert to materialized view if data doesn't need to be real-time
CREATE MATERIALIZED VIEW mv_stock_report AS
SELECT * FROM stock_report;

-- Refresh when needed
REFRESH MATERIALIZED VIEW mv_stock_report;
```

Query Optimization

```
-- Instead of SELECT * on large reports, specify columns
SELECT item_name, quantity, serial_number
FROM stock_report;
```

```
-- Use WHERE clauses to filter early
SELECT * FROM stock_report
WHERE item_id::INTEGER = 5;  -- Filter specific item
```

Integration with Functions

These views work alongside stored functions:

```
-- Get stock report
SELECT * FROM stock_report;

-- Then create a sale using the serial numbers
SELECT create_sale(
    customer_id,
    CURRENT_DATE,
    '[{"item_name":"iPhone 15 Pro","qty":1,"unit_price":52000,"serials":
["IMEI001"]}]'::jsonb
);

-- Verify stock updated
SELECT * FROM stock_report WHERE serial_number = 'IMEI001';
-- Should not appear (in_stock=FALSE after sale)
```

Data Flow

Overview

The system follows these primary business flows:

1. **Party Creation Flow**
2. **Item Creation Flow**
3. **Purchase Flow**
4. **Sales Flow**
5. **Payment Flow**
6. **Receipt Flow**
7. **Purchase Return Flow**
8. **Sales Return Flow**

Each flow integrates inventory tracking with double-entry accounting.

---

1. Party Creation Flow

Steps:

1. User creates party via `add_party_from_json()`

2. System validates party_type (Customer/Vendor/Both/Expense)
3. For Customer/Both: Links to AR account
4. For Vendor/Both: Links to AP account
5. For Expense: Auto-creates expense GL account
6. If opening_balance > 0: Trigger `trg_party_opening_balance()` creates journal entry
    - Customer Debit: Dr AR, Cr Owner's Capital
    - Vendor Credit: Dr Owner's Capital, Cr AP
    - Expense: Dr Expense Account, Cr Owner's Capital

Database Impact:

- 1 row in Parties table
- 0-1 row in ChartOfAccounts (if Expense type)
- 0-1 journal entry with 2 lines (if opening balance)

Example:

```
SELECT add_party_from_json('{
    "party_name": "ABC Corp",
    "party_type": "Customer",
    "opening_balance": 10000,
    "balance_type": "Debit",
    "contact_info": "555-1234",
    "address": "123 Main St"
}'::jsonb);
```

---

2. Item Creation Flow

Steps:

1. User creates item via `add_item_from_json()`
2. System validates item_name uniqueness
3. Sets default values (sale_price=0, storage='Main Warehouse')
4. Auto-generates timestamps

Database Impact:

- 1 row in Items table

Example:

```
SELECT add_item_from_json('{
    "item_name": "iPhone 15 Pro",
    "sale_price": 52000,
    "category": "Mobile Phones",
    "brand": "Apple",
    "storage": "Warehouse A"
}'::jsonb);
```

---

3. Purchase Flow

Steps:

1. User calls `create_purchase(vendor_id, date, items_jsonb)`
2. System creates PurchaseInvoices header
3. For each item:
    - Resolves or creates item_id
    - Creates PurchaseItems record
    - For each serial:
        - Creates PurchaseUnits record (in_stock=TRUE)
        - Creates StockMovements record (IN)
4. Calculates total_amount
5. Calls `rebuild_purchase_journal()`:
    - Creates JournalEntries record
    - Creates JournalLines: Dr Inventory, Cr AP

Database Impact:

- 1 row in PurchaseInvoices
- N rows in PurchaseItems (N = number of item types)
- M rows in PurchaseUnits (M = total quantity/serials)
- M rows in StockMovements
- 1 row in JournalEntries
- 2 rows in JournalLines

Accounting Entry:

```
Dr Inventory                  $45,000
  Cr Accounts Payable - Vendor   $45,000
(To record purchase of inventory)
```

---

4. Sales Flow

Steps:

1. User calls `create_sale(customer_id, date, items_jsonb)`
2. System creates SalesInvoices header
3. For each item:
    - Validates item exists
    - Creates SalesItems record
    - For each serial:
        - Validates serial is in_stock=TRUE
        - Creates SoldUnits record

        - Updates PurchaseUnits.in_stock=FALSE
        - Creates StockMovements record (OUT)
4. Calculates total_amount
5. Calls `rebuild_sales_journal()`:
    - Calculates revenue and COGS
    - Creates journal with 4 lines

Database Impact:

- 1 row in SalesInvoices
- N rows in SalesItems
- M rows in SoldUnits
- M rows updated in PurchaseUnits
- M rows in StockMovements
- 1 row in JournalEntries
- 4 rows in JournalLines

Accounting Entry:

```
Dr Accounts Receivable - Customer    $52,000
  Cr Sales Revenue                       $52,000
(To record sale)

Dr Cost of Goods Sold                 $45,000
  Cr Inventory                            $45,000
(To record cost of goods sold)
```

---

5. Payment Flow

Steps:

1. User calls `make_payment(data_jsonb)`
2. System validates amount > 0
3. Resolves vendor party_id
4. Uses Cash account (hardcoded for now)
5. Auto-generates reference if not provided (PMT-XXXX)
6. Inserts into Payments table
7. Trigger `trg_payment_journal()` fires:
    - Creates journal entry
    - Links back to payment

Database Impact:

- 1 row in Payments
- 1 row in JournalEntries
- 2 rows in JournalLines

Accounting Entry:

```
Dr Accounts Payable - Vendor    $10,000
  Cr Cash                           $10,000
(To record payment to vendor)
```

---

6. Receipt Flow

Similar to Payment Flow but for incoming money from customers.

Accounting Entry:

```
Dr Cash                               $15,000
  Cr Accounts Receivable - Customer     $15,000
(To record receipt from customer)
```

---

7. Purchase Return Flow

Steps:

1. User calls `create_purchase_return(vendor_id, date, items_jsonb)`
2. System validates each serial was previously purchased
3. Creates PurchaseReturns header
4. For each serial:
   - Creates PurchaseReturnItems record
   - Removes serial from PurchaseUnits (DELETE or mark)
   - Creates StockMovements record (OUT)
5. Calls `rebuild_purchase_return_journal()`:
   - Creates reversing entry

Accounting Entry:

```
Dr Accounts Payable - Vendor    $5,000
  Cr Inventory                      $5,000
(To record return of goods to vendor)
```

---

8. Sales Return Flow

Steps:

1. User calls `create_sales_return(customer_id, date, items_jsonb)`
2. System validates each serial was previously sold
3. Creates SalesReturns header
4. For each serial:

    - Creates SalesReturnItems record (with cost_price)
    - Updates PurchaseUnits.in_stock=TRUE
    - Creates StockMovements record (IN)
5. Calls `rebuild_sales_return_journal()`:
    - Reverses revenue and COGS

Accounting Entry:

```
Dr Sales Revenue                    $7,000
  Cr Accounts Receivable - Customer      $7,000
(To reverse revenue)

Dr Inventory                        $6,000
  Cr Cost of Goods Sold                  $6,000
(To restore inventory at cost)
```

Key Design Principles

1. **Double-Entry Accounting**

Every financial transaction creates balanced journal entries (Debits = Credits).

2. **Serial Number Tracking**

Every inventory unit tracked individually from purchase through sale to return.

3. **FIFO Costing**

Cost of Goods Sold calculated using First-In-First-Out via serial number purchase sequence.

4. **Automatic Journal Entries**

All business transactions automatically create appropriate accounting entries.

5. **Referential Integrity**

Comprehensive foreign key constraints ensure data consistency.

6. **Audit Trail**

Complete StockMovements table tracks every inventory change.

7. **Party-Centric Design**

Single Parties table handles customers, vendors, and expense categories.

8. **Validation Before Modification**

Prevent updates/deletes that would break business logic (e.g., can't delete sold serials).

9. **Flexible Reporting**

JSON-returning functions enable easy front-end integration.

10. **Django Integration**

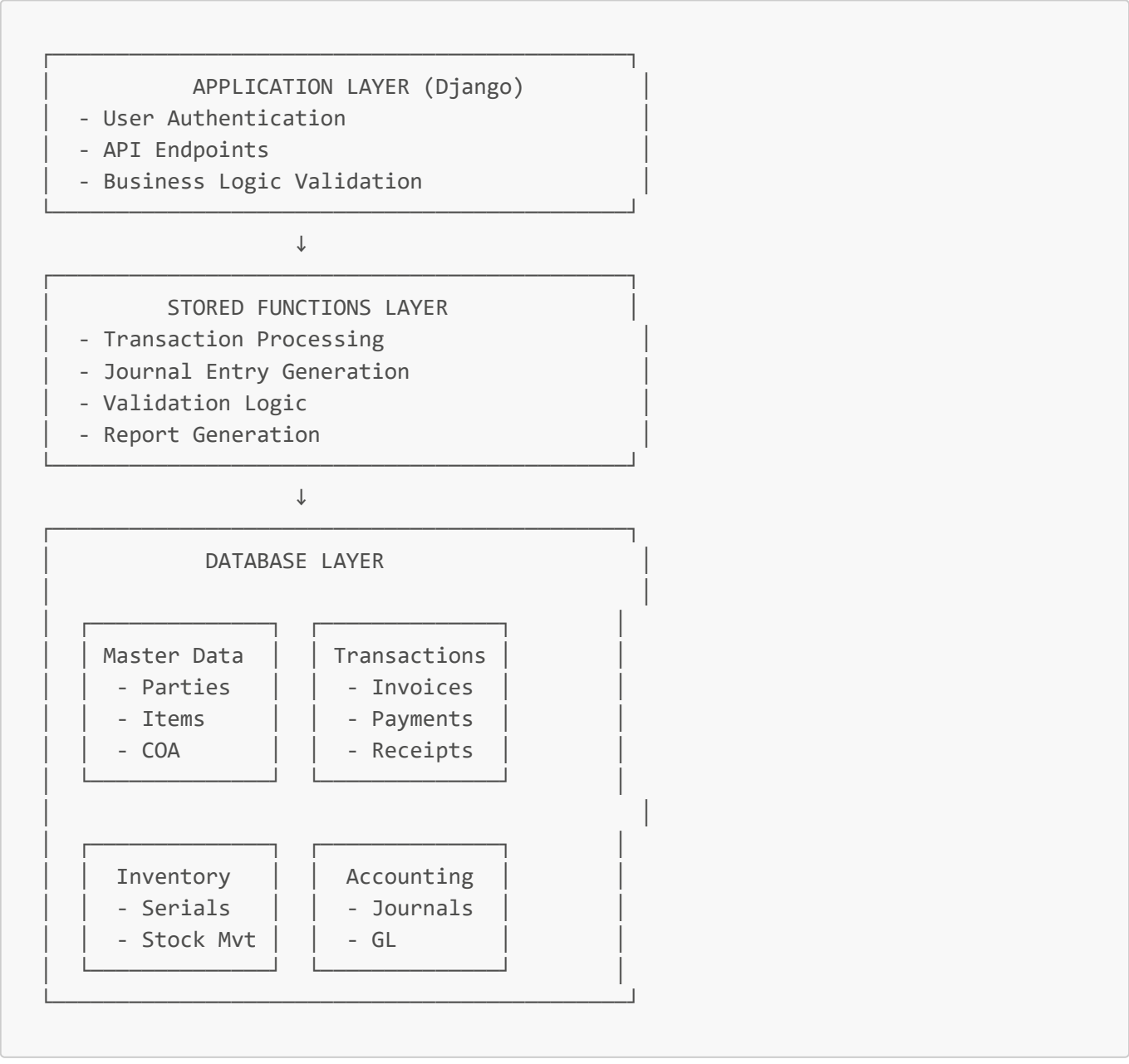
Standard Django authentication and admin tables for user management.

🏗 System Architecture

Database Layer

- **PostgreSQL 14+** with PL/pgSQL stored procedures
- **Django ORM** for application-level queries
- **Stored Functions** for complex business logic

Data Organization



Transaction Flow Architecture

```
User Action (Frontend)
    ↓
Django View/API
    ↓
Stored Function Call
    ↓
┌─────────────────────────┐
│ 1. Validate Input       │
│ 2. Create Header Record │
│ 3. Create Detail Records│
│ 4. Update Inventory     │
│ 5. Create Journal Entry │
│ 6. Return Result        │
└─────────────────────────┘
    ↓
Response to Frontend
```

Accounting Integration

```
Business Transaction
    ↓
Stored Function
    ↓
rebuild_*_journal() function
    ↓
┌─────────────────────────┐
│ 1. Delete old journal   │
│ 2. Create new JournalEntry│
│ 3. Insert Debit lines   │
│ 4. Insert Credit lines  │
│ 5. Link to source doc   │
└─────────────────────────┘
    ↓
General Ledger Updated
```

📊 Summary Statistics

Database Objects

Object Type	Count	Description
Business Tables	18	Core ERP tables
Django System Tables	10+	Authentication, sessions, migrations
Stored Functions	85	Business logic procedures

Object Type	Count	Description
Trigger Functions	3	Automatic journal creation
Foreign Keys	30+	Referential integrity
Check Constraints	20+	Data validation
Unique Constraints	15+	Data uniqueness

Functional Coverage

Module	Functions	Purpose
Purchase	12	Purchase invoice management
Sales	12	Sales invoice management
Purchase Returns	11	Vendor return processing
Sales Returns	11	Customer return processing
Payments	9	Vendor payment tracking
Receipts	9	Customer receipt tracking
Parties	6	Customer/vendor/expense management
Items	4	Inventory item master
Account Reports	2	Financial reporting
Profit Reports	1	P&L analysis
Stock Reports	5	Inventory reporting
Triggers	3	Automatic journal creation

Data Volumes (Typical)

Table	Expected Volume	Notes
Parties	100-1,000	Customers, vendors, expenses
Items	500-5,000	Product catalog
PurchaseInvoices	1,000-10,000/year	Purchase transactions
SalesInvoices	2,000-20,000/year	Sales transactions
PurchaseUnits	10,000-100,000	Individual serial numbers
JournalEntries	5,000-50,000/year	All accounting entries
JournalLines	10,000-200,000/year	Debit/credit lines

🔧 Technical Requirements

Database

- PostgreSQL 14 or higher
- PL/pgSQL language support
- JSONB support
- Trigger support

Application

- Django 4.x or higher
- Python 3.9+
- psycopg2 database adapter

Features Used

- Serial/BigSerial auto-increment
- Foreign key constraints with CASCADE/SET NULL
- Check constraints for data validation
- Trigger functions for automation
- JSONB for flexible data structures
- Stored procedures for business logic

Usage Guidelines

Best Practices

1. **Always use stored functions** for transactions (don't INSERT directly)
2. **Validate before delete** using validate_* functions
3. **Use serial numbers** for all inventory tracking
4. **Maintain GL accounts** required by system (Cash, Inventory, AR, AP, Revenue, COGS)
5. **Regular backups** of database
6. **Monitor journal balance** (debits should equal credits)

Common Patterns

Creating a Purchase:

```
SELECT create_purchase(
    vendor_id,
    CURRENT_DATE,
    '[{"item_name":"Product A","qty":2,"unit_price":100,"serials":
[{"serial":"S1"},{"serial":"S2"}]}]'::jsonb
);
```

Creating a Sale:

```
SELECT create_sale(
    customer_id,
    CURRENT_DATE,
    '[{"item_name":"Product A","qty":1,"unit_price":150,"serials":["S1"]}]'::jsonb
);
```

Recording a Payment:

```
SELECT make_payment('{
    "party_name":"Vendor ABC",
    "amount":5000,
    "method":"Bank",
    "description":"Invoice payment"
}'::jsonb);
```

📚 Additional Resources

Function Quick Reference

- **Purchase**: create_purchase, update_purchase_invoice, delete_purchase, get_purchase_summary
- **Sales**: create_sale, update_sale_invoice, delete_sale, get_sales_summary
- **Returns**: create_purchase_return, create_sales_return
- **Payments/Receipts**: make_payment, make_receipt, update_payment, update_receipt
- **Parties**: add_party_from_json, update_party_from_json
- **Items**: add_item_from_json, update_item_from_json, get_items_json
- **Reports**: get_trial_balance, get_profit_and_loss, get_stock_summary

Report Functions

All report functions return JSONB for easy front-end integration:

- Trial Balance: Shows all account balances
- P&L: Revenue vs expenses with profit calculation
- Stock Reports: Current inventory by item, serial status, movements

---

🎓 Conclusion

This ERP system provides a comprehensive, integrated solution for accounting and inventory management. The database schema ensures:

☑ **Data Integrity** through foreign keys and constraints
☑ **Accurate Accounting** via automatic journal entries
☑ **Complete Traceability** with serial number tracking
☑ **Flexible Reporting** through JSON-returning functions

☑ **Business Logic Enforcement** via stored procedures
☑ **Audit Trail** for all transactions

The system is production-ready and designed for scalability, maintainability, and integration with modern web applications.

Document Version: 1.4
Last Updated: February 13, 2026
Maintained By: Maaz Rehan
