Flux POS & Inventory System – Core Database Design (Updated)

This document presents the **updated** core entity-relationship design for **Flux**, an offline-first POS & inventory system built on **Next.js**, **TypeScript** and **Supabase**. The original design delivered a walking skeleton (Phase 0) and a trimmed MVP (Phase 1) while adhering to the global rules defined in the project charter: all primary keys use UUIDs for offline generation; every transactional or master table (except system-wide configurations) contains a `location_id` column to enforce branch-level row-level security (RLS); audit fields (`created_at`, `updated_at`, `created_by`) are included; and monetary amounts use `DECIMAL(15,2)`. The MVP definition emphasised that core masters (branches/locations, categories, units, items, suppliers and customers) and inventory control with batch/expiry tracking are essential.

Since the design commits to **offline inserts with later sync**, **branch isolation using `location_id` and RLS**, **multi-location transfers** and **batch/expiry (lots) with stock balances**, several small but high-value improvements are necessary to make the schema robust under real-world offline conditions. In particular, offline retries can insert duplicate rows, so we now **enforce uniqueness in SQL** rather than relying solely on documentation. Transfers now carry their own `location_id` for clean RLS policies, sales invoices link directly to customers, and the offline sync flag is kept only on the client – once a record is in Supabase it is considered synced. These changes are incorporated below.

Phase 0 – Walking Skeleton (Core ERD v0)

Phase 0 demonstrates that the stack works end-to-end: authenticate a user, create minimal master data (category, unit, item), sell a single item while offline and sync the sale to Supabase. A simple POS prototype sells one dummy item; sales are stored offline, synced later and update stock on hand. The schema therefore includes tables for sales invoices and lines, payments, stock balances and the stock ledger. In this updated version we also enforce uniqueness on stock balances and barcodes to prevent duplicate rows during offline sync, remove the `sync_status` flag from the database (it remains in the client's IndexedDB), and add a business `status` on invoices to track draft/posted/cancelled/void states.

DBML for Phase 0

```
// Locations/branches.  Each record represents a shop or warehouse.
Table locations {
  id uuid [pk, note: 'Primary key generated client-side']
  name text [note: 'unique']
  address text
  created_at timestamp
  updated_at timestamp
  created_by uuid [note: 'References auth.users.id']
}

// Product categories.  Category data is isolated by location for RLS.
Table categories {
  id uuid [pk]
  name text
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Units of measure (e.g. piece, kg).  Also scoped by location.
Table units {
  id uuid [pk]
  name text
  symbol text
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Inventory items.  Each item belongs to a category and unit.  Price and
cost use DECIMAL(15,2).
Table items {
  id uuid [pk]
  name text
  barcode text [note: 'unique per location when not null']
  price decimal(15,2)
  cost decimal(15,2)
  category_id uuid [ref: > categories.id]
  unit_id uuid [ref: > units.id]
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Sales invoice header.  Tracks one sale transaction; payment_status
indicates paid/unpaid/partial and status indicates the business state (draft,
posted, cancelled, void).  Offline sync flags are stored only on the client.
Table sales_invoices {
  id uuid [pk]
  invoice_number text
  total_amount decimal(15,2)
  payment_status varchar(20)
  status varchar(20) [note: 'draft | posted | cancelled | void']
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}
```

```
// Sales invoice lines.  Each line references an item and records quantity,
unit price and line total.
Table sales_invoice_lines {
  id uuid [pk]
  invoice_id uuid [ref: > sales_invoices.id]
  item_id uuid [ref: > items.id]
  qty decimal(15,3) [note: 'Supports fractional quantities']
  unit_price decimal(15,2)
  total decimal(15,2)
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Payment records for a sales invoice.  Multiple payments allow split
tender.
Table payments {
  id uuid [pk]
  invoice_id uuid [ref: > sales_invoices.id]
  method varchar(20) [note: 'Cash, Card, Credit']
  amount decimal(15,2)
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Snapshot table holding current stock on hand per item per location.
Derived from stock_moves.  location_id + item_id must be unique (enforced via
a unique index).
Table stock_balances {
  id uuid [pk]
  item_id uuid [ref: > items.id]
  location_id uuid [ref: > locations.id]
  quantity_on_hand decimal(20,3)
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Stock ledger capturing every change in inventory.  A sale generates a
negative quantity_change; purchases and adjustments generate positive
changes.
Table stock_moves {
  id uuid [pk]
  item_id uuid [ref: > items.id]
  location_id uuid [ref: > locations.id]
  quantity_change decimal(20,3)
  move_type varchar(20) [note: 'Sale, Return, Adjustment, TransferOut,
TransferIn, GRN']
  reference_id uuid [note: 'References the document causing the move, e.g.
sales_invoices.id or grns.id']
  created_at timestamp
  created_by uuid
}

// Relationships
Ref: categories.location_id > locations.id
Ref: units.location_id > locations.id
Ref: items.category_id > categories.id
Ref: items.unit_id > units.id
Ref: items.location_id > locations.id
Ref: sales_invoices.location_id > locations.id
Ref: sales_invoice_lines.invoice_id > sales_invoices.id
Ref: sales_invoice_lines.item_id > items.id
Ref: sales_invoice_lines.location_id > locations.id
Ref: payments.invoice_id > sales_invoices.id
Ref: payments.location_id > locations.id
Ref: stock_balances.item_id > items.id
Ref: stock_balances.location_id > locations.id
Ref: stock_moves.item_id > items.id
Ref: stock_moves.location_id > locations.id
```

SQL to Create Phase 0 Schema in Supabase

The following SQL uses PostgreSQL syntax compatible with Supabase. It relies on the `pgcrypto` extension for UUID generation. All tables include `location_id`, `created_at`, `updated_at` and `created_by`. Monetary fields use `NUMERIC(15,2)` in accordance with the currency rule. Foreign keys reference the corresponding parent tables; the `created_by` columns should reference `auth.users(id)` but cannot be declared here because `auth.users` lives in a separate schema in Supabase. After creating the tables you must define RLS policies to restrict access by `location_id`.

```
-- Enable required extension for UUID generation
create extension if not exists "pgcrypto";

-- Locations
create table if not exists public.locations (
  id uuid primary key default gen_random_uuid(),
  name text not null,
  address text,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Enforce unique location names to avoid duplicate branches
create unique index if not exists ux_locations_name
  on public.locations (name);

-- Categories
create table if not exists public.categories (
  id uuid primary key default gen_random_uuid(),
  name text not null,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Units
create table if not exists public.units (
  id uuid primary key default gen_random_uuid(),
  name text not null,
  symbol text,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Items
create table if not exists public.items (
  id uuid primary key default gen_random_uuid(),
  name text not null,
  barcode text,
  price numeric(15,2),
  cost numeric(15,2),
  category_id uuid not null references public.categories(id) on delete
restrict,
  unit_id uuid not null references public.units(id) on delete restrict,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Ensure barcode is unique per location when not null
create unique index if not exists ux_items_barcode_per_location
  on public.items (location_id, barcode)
  where barcode is not null;

-- Sales invoices
create table if not exists public.sales_invoices (
  id uuid primary key default gen_random_uuid(),
  invoice_number text not null,
  total_amount numeric(15,2) not null,
  payment_status varchar(20) not null,
  status varchar(20) not null default 'draft', -- draft | posted | cancelled
| void
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);
```

```
-- Ensure invoice numbers are unique per location
create unique index if not exists
ux_sales_invoices_invoice_number_per_location
  on public.sales_invoices (location_id, invoice_number);

-- Sales invoice lines
create table if not exists public.sales_invoice_lines (
  id uuid primary key default gen_random_uuid(),
  invoice_id uuid not null references public.sales_invoices(id) on delete
cascade,
  item_id uuid not null references public.items(id) on delete restrict,
  qty numeric(15,3) not null,
  unit_price numeric(15,2) not null,
  total numeric(15,2) not null,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Payments
create table if not exists public.payments (
  id uuid primary key default gen_random_uuid(),
  invoice_id uuid not null references public.sales_invoices(id) on delete
cascade,
  method varchar(20) not null,
  amount numeric(15,2) not null,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Stock balances
create table if not exists public.stock_balances (
  id uuid primary key default gen_random_uuid(),
  item_id uuid not null references public.items(id) on delete cascade,
  location_id uuid not null references public.locations(id) on delete
cascade,
  quantity_on_hand numeric(20,3) not null default 0,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Enforce one balance row per location and item
create unique index if not exists ux_stock_balances_location_item
  on public.stock_balances (location_id, item_id);

-- Stock moves (ledger)
create table if not exists public.stock_moves (
  id uuid primary key default gen_random_uuid(),
  item_id uuid not null references public.items(id) on delete cascade,
  location_id uuid not null references public.locations(id) on delete
cascade,
  quantity_change numeric(20,3) not null,
  move_type varchar(20) not null,
  reference_id uuid,
  created_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Indexes for performance (optional but recommended)
create index if not exists idx_stock_moves_item_location on
  public.stock_moves(item_id, location_id);
create index if not exists idx_stock_balances_item_location on
  public.stock_balances(item_id, location_id);
```

Phase 1 – Trimmed MVP (Core ERD v1)

Phase 1 expands the schema to support real retail operations: purchasing, goods received notes (GRNs), stock transfers, simple stock adjustments and batch/expiry tracking. The MVP definition emphasises that core masters must include suppliers and customers, that inventory needs batch/expiry fields and that stock on hand must be maintained per location. To implement these requirements we add a `stock_lots` table and extend existing tables to include a `lot_id`. We also introduce headers and line tables for GRNs, stock transfers and stock adjustments. In this updated version we enforce uniqueness for balances per lot and per non-lot, ensure barcodes remain unique, add a `customer_id` column to sales invoices to support credit sales, add a `location_id` to stock transfer headers to simplify RLS, and add unique constraints on document numbers.

DBML for Phase 1

```
// Lot/batch tracking.  Each lot belongs to an item and location, with an
optional expiry date.
Table stock_lots {
  id uuid [pk]
  item_id uuid [ref: > items.id]
  batch_number text
  expiry_date date
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Extend stock balances to track per-lot quantities.  location_id + item_id
+ lot_id must be unique (enforced via partial unique indexes).
Table stock_balances {
  lot_id uuid [ref: > stock_lots.id, note: 'nullable for items without
batches']
```

```
}

// Extend stock moves to record the lot affected by each movement.
Table stock_moves {
  lot_id uuid [ref: > stock_lots.id, note: 'nullable']
}

// Customers master.  Includes basic credit fields.
Table customers {
  id uuid [pk]
  name text
  mobile text
  email text
  credit_limit decimal(15,2)
  credit_days integer
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Suppliers master.
Table suppliers {
  id uuid [pk]
  name text
  contact_info text
  credit_days integer
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Goods Received Note (GRN) header.  Records receipts of goods from
suppliers.
Table grns {
  id uuid [pk]
  grn_number text
  supplier_id uuid [ref: > suppliers.id]
  total_amount decimal(15,2)
  status varchar(20) [note: 'draft or posted']
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// GRN lines.  Each line references an item, lot and quantity.  When a GRN is
posted, stock_moves and balances update accordingly.
Table grn_lines {
  id uuid [pk]
  grn_id uuid [ref: > grns.id]
  item_id uuid [ref: > items.id]
  lot_id uuid [ref: > stock_lots.id]
  qty decimal(20,3)
  unit_cost decimal(15,2)
  total_cost decimal(15,2)
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Stock transfer header.  Transfers goods from one location to another.
Includes its own location_id equal to the source branch for RLS.
Table stock_transfers {
  id uuid [pk]
  transfer_number text
  from_location_id uuid [ref: > locations.id]
  to_location_id uuid [ref: > locations.id]
  location_id uuid [ref: > locations.id, note: 'same as from_location_id']
  status varchar(20) [note: 'Pending, InTransit, Received']
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Stock transfer lines.  Each line moves a quantity of an item (and lot).
For RLS, location_id references the source location.
Table stock_transfer_lines {
  id uuid [pk]
  transfer_id uuid [ref: > stock_transfers.id]
  item_id uuid [ref: > items.id]
  lot_id uuid [ref: > stock_lots.id, note: 'nullable']
  qty decimal(20,3)
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Stock adjustment header (simple increase/decrease with reason).
Table stock_adjustments {
  id uuid [pk]
  reason text
  status varchar(20) [note: 'draft or posted']
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

// Stock adjustment lines.  Each line specifies the quantity change (positive
or negative) for a given item and lot.
Table stock_adjustment_lines {
  id uuid [pk]
  adjustment_id uuid [ref: > stock_adjustments.id]
  item_id uuid [ref: > items.id]
  lot_id uuid [ref: > stock_lots.id, note: 'nullable']
  qty_change decimal(20,3)
  location_id uuid [ref: > locations.id]
  created_at timestamp
  updated_at timestamp
  created_by uuid
}

  // Extend sales invoices to link a customer.  Cash sales can leave this
field null but credit sales must reference a customer.
  Table sales_invoices {
    customer_id uuid [ref: > customers.id, note: 'nullable for cash sales']
  }

// Relationships added in Phase 1
Ref: stock_lots.item_id > items.id
Ref: stock_lots.location_id > locations.id
Ref: stock_balances.lot_id > stock_lots.id
Ref: stock_moves.lot_id > stock_lots.id
Ref: customers.location_id > locations.id
Ref: suppliers.location_id > locations.id
Ref: grns.supplier_id > suppliers.id
Ref: grns.location_id > locations.id
Ref: grn_lines.grn_id > grns.id
Ref: grn_lines.item_id > items.id
Ref: grn_lines.lot_id > stock_lots.id
Ref: stock_transfers.from_location_id > locations.id
Ref: stock_transfers.to_location_id > locations.id
Ref: stock_transfers.location_id > locations.id
Ref: stock_transfer_lines.transfer_id > stock_transfers.id
Ref: stock_transfer_lines.item_id > items.id
Ref: stock_transfer_lines.lot_id > stock_lots.id
Ref: stock_adjustments.location_id > locations.id
Ref: stock_adjustment_lines.adjustment_id > stock_adjustments.id
Ref: stock_adjustment_lines.item_id > items.id
Ref: stock_adjustment_lines.lot_id > stock_lots.id
```

SQL to Create Phase 1 Schema in Supabase

Below is SQL to create the new tables and to alter existing tables from Phase 0. The statements preserve UUID primary keys and audit fields. You should also set up triggers or Supabase function hooks to update stock balances whenever GRNs are posted, stock transfers occur or adjustments are made. This updated version adds the missing `customer_id` on invoices, includes `location_id` on transfers, enforces uniqueness on barcodes, stock balances (both with and without lots) and document numbers, and keeps the offline sync flag out of the database.

```
-- New table for stock lots (batches/expiry)
create table if not exists public.stock_lots (
  id uuid primary key default gen_random_uuid(),
  item_id uuid not null references public.items(id) on delete cascade,
  batch_number text,
  expiry_date date,
  location_id uuid not null references public.locations(id) on delete
```

```
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Update stock_balances to include lot_id
alter table public.stock_balances
  add column if not exists lot_id uuid references public.stock_lots(id);

-- Enforce unique stock balances for items with and without lots
create unique index if not exists ux_stock_balances_with_lot
  on public.stock_balances (location_id, item_id, lot_id)
  where lot_id is not null;
create unique index if not exists ux_stock_balances_no_lot
  on public.stock_balances (location_id, item_id)
  where lot_id is null;

-- Update stock_moves to include lot_id
alter table public.stock_moves
  add column if not exists lot_id uuid references public.stock_lots(id);

-- Customers table
create table if not exists public.customers (
  id uuid primary key default gen_random_uuid(),
  name text not null,
  mobile text,
  email text,
  credit_limit numeric(15,2) default 0,
  credit_days integer default 0,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Suppliers table
create table if not exists public.suppliers (
  id uuid primary key default gen_random_uuid(),
  name text not null,
  contact_info text,
  credit_days integer default 0,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- GRN header
create table if not exists public.grns (
  id uuid primary key default gen_random_uuid(),
  grn_number text not null,
  supplier_id uuid not null references public.suppliers(id) on delete
restrict,
  total_amount numeric(15,2) not null,
  status varchar(20) not null default 'draft',
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Ensure GRN numbers are unique per location
create unique index if not exists ux_grns_grn_number_per_location
  on public.grns (location_id, grn_number);

-- GRN lines
create table if not exists public.grn_lines (
  id uuid primary key default gen_random_uuid(),
  grn_id uuid not null references public.grns(id) on delete cascade,
  item_id uuid not null references public.items(id) on delete restrict,
  lot_id uuid references public.stock_lots(id),
  qty numeric(20,3) not null,
  unit_cost numeric(15,2) not null,
  total_cost numeric(15,2) not null,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Stock transfer header
create table if not exists public.stock_transfers (
  id uuid primary key default gen_random_uuid(),
  transfer_number text not null,
  from_location_id uuid not null references public.locations(id),
  to_location_id uuid not null references public.locations(id),
  location_id uuid, -- will be set equal to from_location_id
  status varchar(20) not null default 'Pending',
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Populate location_id = from_location_id for existing rows then make
non-null
update public.stock_transfers
  set location_id = from_location_id
  where location_id is null;

alter table public.stock_transfers
  alter column location_id set not null;

alter table public.stock_transfers
```

```
  add constraint fk_stock_transfers_location
  foreign key (location_id) references public.locations(id);

-- Ensure transfer numbers are unique per location
create unique index if not exists
ux_stock_transfers_transfer_number_per_location
  on public.stock_transfers (location_id, transfer_number);

-- Stock transfer lines
create table if not exists public.stock_transfer_lines (
  id uuid primary key default gen_random_uuid(),
  transfer_id uuid not null references public.stock_transfers(id) on delete
cascade,
  item_id uuid not null references public.items(id) on delete restrict,
  lot_id uuid references public.stock_lots(id),
  qty numeric(20,3) not null,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Stock adjustment header
create table if not exists public.stock_adjustments (
  id uuid primary key default gen_random_uuid(),
  reason text not null,
  status varchar(20) not null default 'draft',
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Stock adjustment lines
create table if not exists public.stock_adjustment_lines (
  id uuid primary key default gen_random_uuid(),
  adjustment_id uuid not null references public.stock_adjustments(id) on
delete cascade,
  item_id uuid not null references public.items(id) on delete restrict,
  lot_id uuid references public.stock_lots(id),
  qty_change numeric(20,3) not null,
  location_id uuid not null references public.locations(id) on delete
cascade,
  created_at timestamp with time zone not null default now(),
  updated_at timestamp with time zone not null default now(),
  created_by uuid
);

-- Add customer_id to sales_invoices (credit sales)
alter table public.sales_invoices
  add column if not exists customer_id uuid references public.customers(id);
```

```
-- Indexes to accelerate queries on lot-based stock balances and moves
create index if not exists idx_stock_balances_lot on
  public.stock_balances(lot_id);
create index if not exists idx_stock_moves_lot on
  public.stock_moves(lot_id);

-- Ensure barcodes remain unique per location when not null
create unique index if not exists ux_items_barcode_per_location_phase1
  on public.items (location_id, barcode)
  where barcode is not null;

-- Ensure invoice numbers remain unique per location in Phase 1
create unique index if not exists
ux_sales_invoices_invoice_number_per_location_phase1
  on public.sales_invoices (location_id, invoice_number);
```

How a Sale Creates a Stock Movement

When a cashier completes a sale in Flux, the POS creates a record in `sales_invoices` with one or more lines in `sales_invoice_lines`. Each line contains the `item_id`, quantity and unit price. In order to maintain accurate inventory, a trigger or Supabase function executes after the invoice is inserted (or when the invoice is marked as posted). For every line:

1. **Insert a ledger entry into** `stock_moves` with the same `item_id`, `location_id` and, if applicable, `lot_id`. The `quantity_change` is set to the negative quantity sold (because stock decreases), and `move_type` is `Sale`. The `reference_id` field stores the `sales_invoices.id` to trace the source document.
2. **Update** `stock_balances` by subtracting the quantity sold from `quantity_on_hand` for the given `item_id`, `location_id` and `lot_id`. If there is no existing balance row, create one with an initial negative quantity. Unique indexes on `stock_balances` ensure that duplicate rows are not created during offline retries.

This mechanism ensures that each sale reduces stock on hand while maintaining a historical ledger. The roadmap emphasises that sales must work offline and update stock during sync; therefore, the client stores the invoice locally with a sync flag **in IndexedDB only**. When the device reconnects, a background sync service posts the invoice to Supabase, which runs the trigger to insert `stock_moves` and adjust balances. Because `stock_moves` is the single source of truth and `stock_balances` is derived from it, other processes such as GRNs, transfers and adjustments follow the same pattern with positive quantity changes for increases.