DIMENSIONAL MODEL DESIGN

Banking Transaction Warehouse

Star Schema, SCD Logic, Surrogate Keys, Performance Strategy

Reporting Period	Design Window: Q1 2026
Document Version	1.0
Date Prepared	April 2026
Classification	Internal - Data Engineering
Prepared By	Data & Analytics Team

DISTRIBUTION

Data Engineering - Data Quality - Internal Audit - Analytics Leadership

PREPARED BY	REVIEWED BY	APPROVED BY
_____ *Signature / Date*	_____ *Signature / Date*	_____ *Signature / Date*

1. Purpose and Scope

This document specifies the dimensional warehouse for the Bank's retail banking transaction data. The source is a flat transaction file with 15 columns; the output is a star schema implemented in PostgreSQL 18, with full SCD support, an embedded data quality framework, and a monthly aggregate table for reporting performance.

The model answers five business questions:

- Which customer segments generate the most fee income?
- How does transaction behavior vary by branch and channel?
- Are high-value customers reducing their activity?
- Which products drive deposits versus withdrawals?
- How do metrics compare month-over-month, quarter-over-quarter, and year-over-year?

Scope summary

Element	Detail
Source data	Flat transaction file: Txn_ID, Txn_Date, Customer_ID, Customer_Name, Tier, Branch_ID, Branch_Name, State, Product_ID, Product_Name, Product_Type, Txn_Type, Channel, Amount, Balance_After.
Target schema	Star schema with one fact table, six dimensions, one staging table, one monthly aggregate.
Database	PostgreSQL 18 (pgAdmin 4 for development).
Refresh cadence	Daily incremental load from the operational system. Historical reload supported when required.
Out of scope	Fee calculation rules (source file lacks fee data; column reserved). Customer demographic data. Loan and credit portfolio data.

Table 1. Project scope.

2. The Grain Statement

> **FOUNDATIONAL DECISION**
> The grain of fact_transactions is one row per banking transaction. Every unique Txn_ID appears once in the fact table. Every measure is anchored to this grain.

The grain statement[object Object] determines what each measure means:

Aggregation	Interpretation at this grain
COUNT(*)	Number of transactions.
SUM(amount_ngn)	Total transaction value (sum of all individual amounts).
SUM(deposit_amount_ngn)	Total deposit value across selected rows.
SUM(withdrawal_amount_ngn)	Total withdrawal value across selected rows.
AVG(amount_ngn)	Average transaction size.
MAX(balance_after_ngn)	Latest balance in the selected window. SUM is invalid here.[object Object]

Table 2. Aggregation semantics at the transaction grain.

Anything other than a transaction-level row would violate the grain. A row per customer would force averaging or summing across transactions before storage, losing the ability to answer transaction-level questions. A row per day would lose the per-transaction detail needed for behavior analysis.

3. Fact and Dimension Classification

Every column in the source file must be classified before the model can be built. The classification answers: is this a measure, a descriptive attribute, or an identifier?

Source column	Classification	Final location	Why
Txn_ID	Degenerate dimension	fact_transactions	Identifies the transaction; no descriptive attributes.[object Object]
Txn_Date	Date dimension / fact timestamp	dim_date, fact_transactions	Time analysis requires day-of-week, month, quarter; dim_date provides these. Timestamp stays on fact for sub-day analysis.
Customer_ID	Natural key	dim_customer (as attribute)	Source-system identifier; preserved for joins back to source.

Source column	Classification	Final location	Why
Customer_Name	Dimension attribute	dim_customer	Describes the customer. Type 1 (corrections only).
Tier	Dimension attribute, SCD Type 2	dim_customer	Changes over time and the change matters for analysis.
Branch_ID	Natural key	dim_branch (as attribute)	Source identifier preserved.
Branch_Name	Dimension attribute, SCD Type 2	dim_branch	Branch rebrands must preserve historical names.
State	Dimension attribute, Type 1	dim_branch	Geography rarely changes; corrections handled in place.
Product_ID, Product_Name, Product_Type	Dimension attributes	dim_product	Type 1; product changes are typically corrections.
Txn_Type	Standalone dimension	dim_transaction_type	Independent of channel; needs separate reporting axis.
Channel	Standalone dimension	dim_channel	Independent of transaction type; needs separate reporting axis.
Amount	Fully additive fact	fact_transactions	Can be summed across all dimensions and time.
Balance_After	Semi-additive fact	fact_transactions	Sums correctly across customers at a point in time; not across time.[object Object]

Table 3. Column-by-column classification.

4. The Star Schema

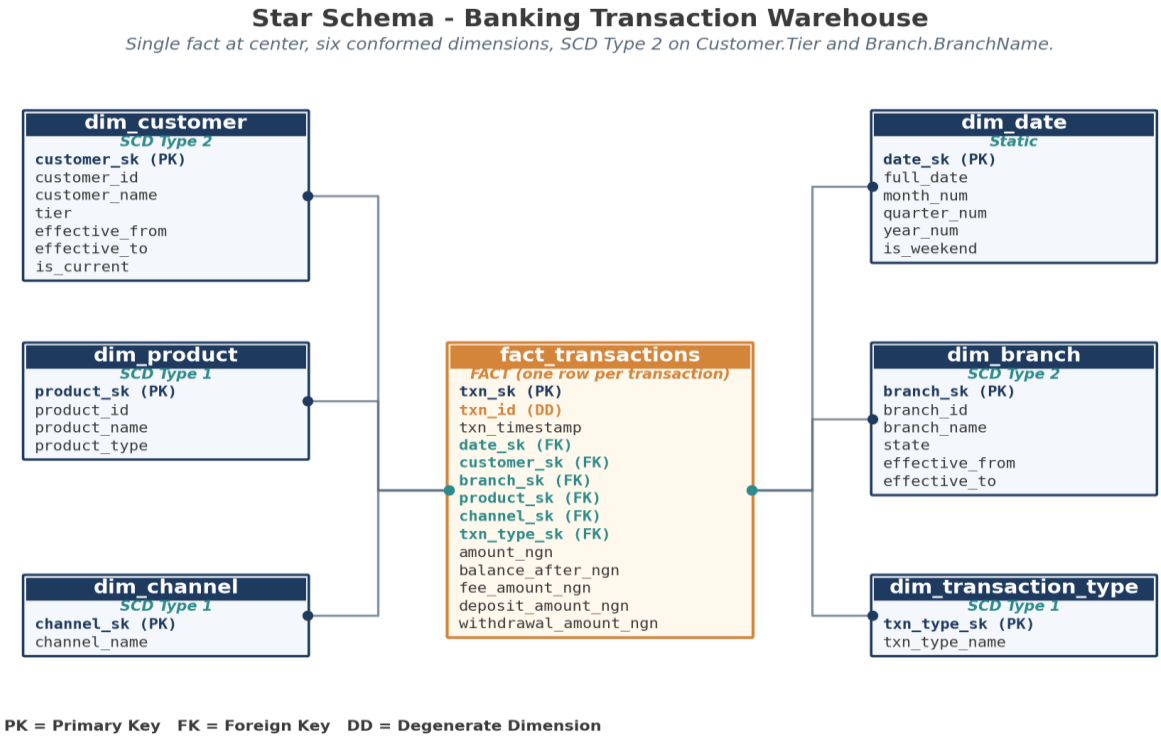

Figure 1. The star schema. One fact at center, six dimensions, surrogate-key foreign relationships.

Why a star, not a snowflake

A snowflake schema would normalize the dimensions further (for example, splitting dim_product into product, category, and subcategory tables). The star design keeps each dimension flat: dim_product contains product_id, product_name, and product_type in one row.

Star (chosen)	Snowflake (rejected)
Fewer joins required for queries.	More joins required; every query touches more tables.
Faster on typical analytical workloads.	Slightly less storage but rarely matters at this scale.
Easier for analysts to understand and write SQL against.	Stricter normalization but harder to use.
Industry standard for OLAP workloads.	Used in special cases where storage is critical or hierarchies are deeply variable.

Table 4. Star versus snowflake trade-off.

5. Surrogate Keys

Every dimension uses a surrogate key as its primary key. The natural key from the source remains as an attribute.

Dimension	Surrogate key	Natural key (kept as attribute)
dim_customer	customer_sk	customer_id
dim_branch	branch_sk	branch_id
dim_product	product_sk	product_id
dim_channel	channel_sk	(none - channel_name is the natural identifier)
dim_transaction_type	txn_type_sk	(none)
dim_date	date_sk	(date_sk is YYYYMMDD - acts as both)

Table 5. Surrogate key naming across all dimensions.

Why surrogate keys

- Source keys can be reused (rare but real). A reused customer ID would create ambiguity in joins; the surrogate key isolates the warehouse from this.
- Source keys can change format (the bank may migrate from 8-character to 12-character IDs). The surrogate key is stable across such changes.
- Source keys can have gaps. Surrogate keys are dense and sequential.
- Surrogate keys enable SCD Type 2. The same customer_id can appear in multiple rows (one per version) but each must have a unique primary key. The surrogate key fills that role.
- Joins on smaller integer keys are faster than joins on varchar natural keys.

6. Slowly Changing Dimensions

Three of the six dimensions need SCD logic[object Object]. The choice of SCD type per attribute reflects the business question the model must answer.

6.1 dim_customer - SCD Type 2 on Tier

Customer tier (Silver, Gold, Platinum) changes over time. One of the five business questions is *are high-value customers reducing activity*. To answer that correctly, every transaction must report the tier the customer **had at transaction time**, not their current tier.

If only the current tier is stored, a customer who was Silver in 2023 and is now Gold would have all their 2023 transactions report as Gold. The historical churn analysis would be wrong.

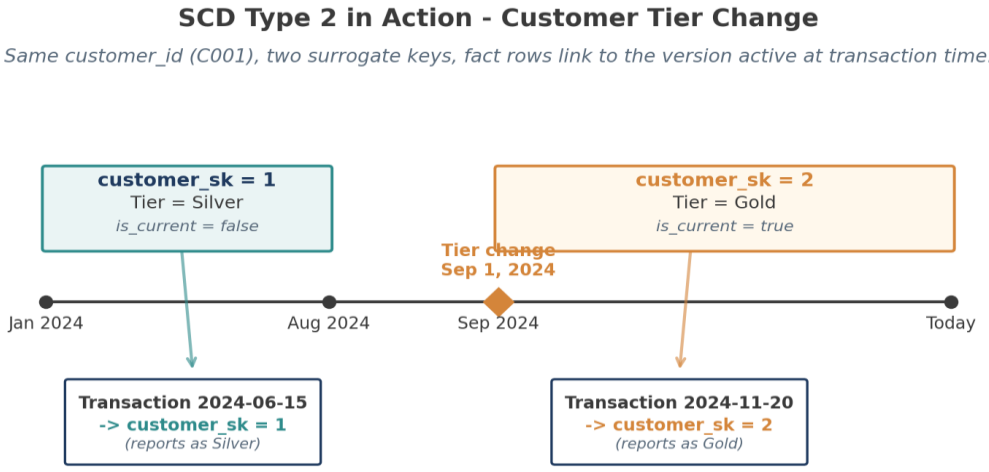

Figure 2. SCD Type 2 example. A customer changes from Silver to Gold; the warehouse keeps both versions; each transaction links to the version active when it occurred.

6.2 dim_customer - SCD Type 1 on Customer_Name

Customer name changes are usually corrections (a typo, a married name, a legal name change). The business doesn't need to see the old spelling on old transactions. Type 1 overwrites the previous value; the row_hash is recalculated.

6.3 dim_branch - SCD Type 2 on Branch_Name

Branches occasionally get renamed (rebrand, merger, address change). Old reports should still show the name the branch had at the time of the transaction. Type 2 preserves this.

6.4 dim_branch - SCD Type 1 on State

Branch state changes are extremely rare. When they do occur, they are usually corrections. Type 1 unless the business specifically requires historical state tracking (in which case promote to Type 2).

6.5 Other dimensions - SCD Type 1

Dimension	SCD treatment and rationale
dim_product	Type 1 across all attributes. Product changes are usually corrections; if a product is genuinely new, it gets a new product_id.
dim_channel	Type 1. Channel names rarely change; when they do, it's a rename of an existing channel and old data should reflect the new name.
dim_transaction_type	Type 1. Transaction type names are stable taxonomy.
dim_date	No SCD. Dates do not change.

Table 6. SCD treatment for the remaining four dimensions.

7. Table Definitions

7.1 Staging - stg_transactions

The staging table receives raw data from the source. It is rebuilt on every load. No constraints (apart from a default timestamp); the goal is to land the data before any validation.

```
CREATE TABLE stg_transactions (
    txn_id              VARCHAR(30),
    txn_datetime        TIMESTAMP,
    customer_id         VARCHAR(30),
    customer_name       VARCHAR(150),
    tier                VARCHAR(30),
    branch_id           VARCHAR(30),
    branch_name         VARCHAR(150),
    state               VARCHAR(80),
    product_id          VARCHAR(30),
    product_name        VARCHAR(150),
    product_type        VARCHAR(80),
    txn_type            VARCHAR(50),
    channel             VARCHAR(50),
    amount_ngn          NUMERIC(18,2),
    balance_after_ngn   NUMERIC(18,2),
    load_batch_id       VARCHAR(50),
    loaded_at           TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Listing 1. Staging table definition.

7.2 dim_customer (representative dimension with SCD Type 2)

```
CREATE TABLE dim_customer (
    customer_sk     BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    customer_id     VARCHAR(30) NOT NULL,        -- natural key from source
    customer_name   VARCHAR(150) NOT NULL,       -- Type 1: overwrite on change
    tier            VARCHAR(30) NOT NULL,        -- Type 2: track history
    effective_from  DATE NOT NULL,
    effective_to    DATE NOT NULL DEFAULT DATE '9999-12-31',
    is_current      BOOLEAN NOT NULL DEFAULT TRUE,
    row_hash        TEXT,                        -- for change detection
    CONSTRAINT uq_customer_version UNIQUE (customer_id, effective_from)
);

CREATE INDEX idx_dim_customer_lookup
    ON dim_customer (customer_id, effective_from, effective_to);

CREATE INDEX idx_dim_customer_current
    ON dim_customer (customer_id, is_current);
```

Listing 2. Customer dimension with SCD Type 2 columns and indexes.

Key features of this definition:

- customer_sk uses GENERATED ALWAYS AS IDENTITY - PostgreSQL's preferred replacement for SERIAL.
- effective_to defaults to '9999-12-31' for current rows. This sentinel value avoids NULL handling in queries that compare against transaction dates.
- is_current is a denormalized boolean derived from effective_to. It exists for query speed: filtering on a boolean is faster than comparing dates.
- The unique constraint on (customer_id, effective_from) prevents duplicate version inserts during ETL errors.
- row_hash is used during incremental loads to detect changes without comparing column-by-column.

7.3 fact_transactions

```
CREATE TABLE fact_transactions (
    txn_sk                BIGINT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    txn_id                VARCHAR(30) NOT NULL,
    txn_timestamp         TIMESTAMP NOT NULL,

    -- Foreign keys to dimensions
    date_sk               INT NOT NULL,
    customer_sk           BIGINT NOT NULL,
    branch_sk             BIGINT NOT NULL,
    product_sk            BIGINT NOT NULL,
    channel_sk            BIGINT NOT NULL,
    txn_type_sk           BIGINT NOT NULL,

    -- Measures
    amount_ngn            NUMERIC(18,2) NOT NULL,
    balance_after_ngn     NUMERIC(18,2),
    fee_amount_ngn        NUMERIC(18,2),
    deposit_amount_ngn    NUMERIC(18,2) NOT NULL DEFAULT 0,
    withdrawal_amount_ngn NUMERIC(18,2) NOT NULL DEFAULT 0,
    transaction_count     INT NOT NULL DEFAULT 1,

    CONSTRAINT uq_fact_txn_id UNIQUE (txn_id),
    CONSTRAINT fk_fact_date     FOREIGN KEY (date_sk)
        REFERENCES dim_date(date_sk),
    CONSTRAINT fk_fact_customer FOREIGN KEY (customer_sk)
        REFERENCES dim_customer(customer_sk),
    CONSTRAINT fk_fact_branch   FOREIGN KEY (branch_sk)
        REFERENCES dim_branch(branch_sk),
    CONSTRAINT fk_fact_product  FOREIGN KEY (product_sk)
        REFERENCES dim_product(product_sk),
    CONSTRAINT fk_fact_channel  FOREIGN KEY (channel_sk)
        REFERENCES dim_channel(channel_sk),
```

```
    CONSTRAINT fk_fact_txn_type FOREIGN KEY (txn_type_sk)
        REFERENCES dim_transaction_type(txn_type_sk),
    CONSTRAINT chk_transaction_count CHECK (transaction_count = 1)
);
```

Listing 3. Fact table definition with all foreign keys and constraints.

Key features of the fact table definition:

- txn_id is preserved as a degenerate dimension - directly on the fact row, with a unique constraint to prevent duplicate transaction insertion.
- Six surrogate-key foreign keys: one per dimension.
- Pre-calculated deposit_amount_ngn and withdrawal_amount_ngn are stored alongside amount_ngn. This is denormalization for query speed; it avoids CASE WHEN repeated in every reporting query.
- transaction_count = 1 is a constant on every row; the CHECK constraint enforces the grain (one row per transaction). Reports SUM this column to get transaction count without needing COUNT(*).
- fee_amount_ngn is nullable - reserved for future fee data.

8. Hierarchies

The model supports four navigation hierarchies. Each hierarchy enables drill-down from summary to detail.

Hierarchy	Levels
Date	Year → Quarter → Month → Day. Implemented in dim_date.
Geography	State → Branch. Implemented in dim_branch.
Product	Product Type → Product. Implemented in dim_product.
Customer	Tier → Customer. Implemented in dim_customer (via SCD Type 2 history).

Table 7. Drill-down hierarchies supported by the model.

Example query path: an executive starts with yearly transaction value (year_num), drills into Q3 (quarter_num), then into September (month_num), then to a specific day (full_date). Each drill operates on dim_date and joins back to fact_transactions through date_sk.

9. Performance Strategy

Three performance techniques are built into the model.

9.1 Partitioning

fact_transactions is partitioned by month on txn_timestamp. Most banking reports are time-scoped (this quarter, last 30 days, year-to-date), and monthly partitions let PostgreSQL skip irrelevant data.

Query type	Partition pruning benefit
Last 30 days	Reads 1 partition. ~1/24 of total data scanned.
Current quarter	Reads 3 partitions. ~1/8 of total data scanned.
Year-over-year same month	Reads 2 partitions across years.
Full history (audit)	Reads all partitions; no pruning benefit but partitioning doesn't slow it down.

Table 8. Partitioning benefits by query type.

9.2 Indexes

Foreign key columns on fact_transactions are individually indexed:

- date_sk - used in every time-bounded query.
- customer_sk - used in customer behavior queries.
- branch_sk - used in branch performance queries.
- product_sk - used in product performance queries.
- channel_sk and txn_type_sk - used in channel and transaction-type breakdowns.
- txn_timestamp - used for fine-grained time-range scans inside a partition.

dim_customer has two indexes beyond the primary key: (customer_id, effective_from, effective_to) for SCD lookups during fact loading, and (customer_id, is_current) for queries against the current version.

9.3 Aggregate table

agg_monthly_branch_activity pre-calculates monthly performance at branch and product type grain. It supports the most common recurring reports (monthly branch dashboards) and reduces repeated full-table scans of fact_transactions.

Stored measure	Calculation
total_transaction_value	SUM(amount_ngn)
deposit_value_ngn	SUM(deposit_amount_ngn)
withdrawal_value_ngn	SUM(withdrawal_amount_ngn)
fee_income_ngn	SUM(fee_amount_ngn) - nullable until fee rules supplied
transaction_count	COUNT(*)
active_customer_count	COUNT(DISTINCT customer_sk)

Table 9. Pre-aggregated measures.

10. Sample Query

To illustrate how the model answers a business question, here is the SQL for: which customer tier drives the most transaction volume in Q3 2024.

```
-- Q: Which customer tier drives the most transaction volume in Q3 2024?
SELECT
    c.tier,
    COUNT(*)              AS txn_count,
    SUM(f.amount_ngn)     AS total_value,
    SUM(f.deposit_amount_ngn)    AS deposit_value,
    SUM(f.withdrawal_amount_ngn) AS withdrawal_value
FROM fact_transactions f
JOIN dim_customer c ON f.customer_sk = c.customer_sk
JOIN dim_date d     ON f.date_sk = d.date_sk
WHERE d.year_num = 2024
  AND d.quarter_num = 3
GROUP BY c.tier
ORDER BY total_value DESC;

-- Note: uses customer_sk, not customer_id.
-- This means the tier reported is the tier at the time of the transaction,
-- not the customer's current tier. That distinction is the entire point of SCD Type 2.
```

Listing 4. Sample query using surrogate-key joins. Note how SCD Type 2 affects the result.

> **WHY THE SURROGATE-KEY JOIN MATTERS HERE**
>
> The query joins on customer_sk, not customer_id. This means a customer who upgraded from Silver to Gold mid-quarter would have their pre-upgrade transactions counted in Silver and post-upgrade transactions counted in Gold. That's the correct historical reading. A join on customer_id would either count everything as the current tier (wrong) or require additional date-range logic to find the right version (slower).

11. SCD Type 2 Update Logic

When a customer's tier changes, the ETL must close the old row and insert a new one in a single transaction:

```
-- SCD Type 2 update on dim_customer when tier changes
BEGIN;

-- Step 1: Close out the old row
UPDATE dim_customer
SET effective_to = CURRENT_DATE - INTERVAL '1 day',
    is_current   = FALSE
WHERE customer_id = 'C001'
```

```
  AND is_current  = TRUE;

-- Step 2: Insert the new current row
INSERT INTO dim_customer (
    customer_id, customer_name, tier,
    effective_from, effective_to, is_current, row_hash
) VALUES (
    'C001', 'Sample Customer', 'Gold',
    CURRENT_DATE, DATE '9999-12-31', TRUE,
    MD5('Sample Customer|Gold')
);

COMMIT;
```

Listing 5. SCD Type 2 update logic for a tier change.

Three things to note:

- The BEGIN/COMMIT wrap ensures atomicity. A failure between the UPDATE and INSERT would otherwise leave the customer with no current row.
- effective_to on the old row is set to yesterday (CURRENT_DATE - INTERVAL '1 day'). The new row's effective_from is today. There is no gap.
- row_hash is recalculated on the new row. The next change-detection pass uses this hash to determine whether the customer needs another SCD update.

12. Fee Income Limitation

> **OPEN ISSUE - DOCUMENTED HONESTLY**
> The source transaction file does not include a fee column. The business question 'which customer segments generate the most fee income' cannot be answered with current data. The model includes fee_amount_ngn as a nullable column so the question becomes answerable as soon as fee data or fee rules are supplied. Fee income is not fabricated.

If the Bank later supplies a fee rules table (for example: percentage on withdrawal, flat fee on transfer, tier-based fee waiver), the ETL can compute fee_amount_ngn during fact loading and the existing aggregate table will start including fee income automatically. No schema changes required.

Document Control

Version	Date	Change Summary	Author
0.1	2026-02-08	Initial schema draft - grain and classification	Data Engineering
0.5	2026-02-22	SCD logic, surrogate keys, performance section	Data Engineering
0.9	2026-03-05	Sample queries and SCD update logic added	Data Engineering
1.0	2026-03-15	Approved as reference design	Data Engineering

Table 10. Revision history.