OPERATIONS RUNBOOK

ETL Operations Runbook

Initial Load, Incremental Load, Data Quality, Troubleshooting

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

DISTRIBUTION

Data Engineering - DataOps - On-Call - Operations Leadership

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

1. Purpose

This runbook documents the day-to-day operation of the Bank warehouse ETL pipeline. It is the document the on-call engineer opens at 2 AM when a load fails. Every procedure includes the SQL, the expected result, the failure modes, and the remediation steps.

Two scenarios are covered:

- Initial load - full historical reload from source.
- Incremental load - daily delta load from the operational source.

Both share the same staging-to-dim-to-fact-to-aggregate sequence; they differ in volume and dimension-version detection logic.

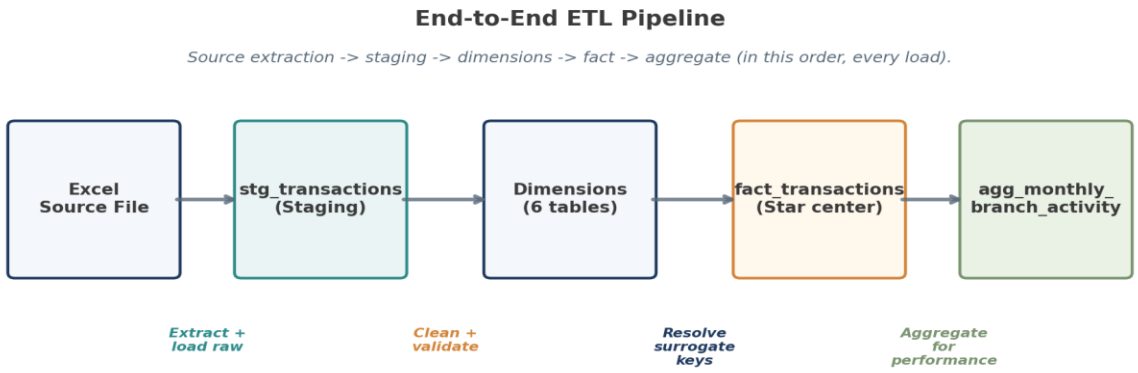

Figure 1. ETL pipeline at a glance. Same flow for both load types; the difference is volume and how dimension changes are detected.

2. Initial Load Procedure

The initial load runs once when the warehouse is first built, against the full historical transaction file. It is also used for disaster recovery (restoring the warehouse from source).

2.1 Pre-load checklist

Check	Action if failed
dim_date populated for the full historical range	Run the date-dimension generator script with the earliest and latest expected transaction dates.
Source file landed at expected path	Verify with file system / S3 listing. Reload from source if missing.
Source file row count matches manifest	If mismatch, halt and contact source team.
No open transactions in fact_transactions	Initial load assumes empty target. TRUNCATE if needed.

Table 1. Pre-flight checks for initial load.

2.2 Initial load sequence

Run these steps in order[object Object]:

Step	Action	Why this position
1	Truncate stg_transactions	Clean slate.
2	Load source file into stg_transactions	Single COPY command. Fast for bulk load.
3	Run data quality checks (see Section 4)	Catch issues before they propagate into dimensions.
4	Load dim_date (if not pre-populated)	No dependencies.
5	Load dim_channel from distinct staging values	No dependencies.
6	Load dim_transaction_type from distinct staging values	No dependencies.
7	Load dim_product from distinct staging values	No dependencies.
8	Load dim_branch (SCD Type 2 - initial version per branch)	All initial rows get effective_from = earliest transaction date.

Step	Action	Why this position
9	Load dim_customer (SCD Type 2 - initial version per customer)	Same pattern as dim_branch.
10	Load fact_transactions	All FKs now exist.
11	Validate row counts (staging vs fact)	Catch missing rows.
12	Build agg_monthly_branch_activity	Final step. Depends on fact table being complete.

Table 2. Initial load step sequence.

2.3 Initial load SQL

```
-- Reset staging between batches
TRUNCATE TABLE stg_transactions RESTART IDENTITY;
```

Listing 1. Reset staging.

```
-- PostgreSQL COPY (production form uses \copy from psql)
COPY stg_transactions (
    txn_id, txn_datetime, customer_id, customer_name, tier,
    branch_id, branch_name, state, product_id, product_name,
    product_type, txn_type, channel, amount_ngn, balance_after_ngn,
    load_batch_id
)
FROM '/data/incoming/transactions_2026_03_15.csv'
WITH (FORMAT CSV, HEADER TRUE, ENCODING 'UTF8');
```

Listing 2. Load source CSV into staging via COPY.

2.4 Initial load validation

Three validations must pass before the initial load is considered complete:

- Row count: COUNT(*) in fact_transactions equals COUNT(*) in stg_transactions, minus any rows rejected by data quality.
- Referential integrity: zero orphans on each dimension FK (see DQ Check 5 in Section 4).
- Sum reconciliation: SUM(amount_ngn) in fact_transactions equals SUM(amount_ngn) in stg_transactions (after rejections).

3. Incremental Load Procedure

The incremental load runs daily against the previous day's transactions from the operational source. Volumes are typically 1-5% of the historical load; the procedure must be fast and idempotent[object Object].

3.1 Incremental load sequence

Step	Action	Notes
1	Truncate stg_transactions	Same starting state as initial.
2	Load daily delta into stg_transactions	Only new transactions; source filters by timestamp.
3	Run DQ checks	Same checks; smaller dataset; same blocking rules.
4	Detect already-loaded txn_ids	Compare staging vs fact. Log duplicates (idempotency).
5	Insert new dim_product, dim_channel, dim_transaction_type values	Type 1 dimensions: simple INSERT-if-not-exists.
6	Detect customer SCD Type 2 changes	Compare row hashes; close and insert versions.
7	Detect branch SCD Type 2 changes	Same pattern as customer.
8	Resolve surrogate keys and load facts	Single INSERT with all FK joins.
9	Refresh agg_monthly_branch_activity	Current and prior month only.
10	Log batch completion	Record load_batch_id, row counts, duration.

Table 3. Incremental load step sequence.

3.2 Duplicate detection

```
-- Identify duplicate txn_ids already in fact_transactions
SELECT s.txn_id, COUNT(*) AS already_exists
FROM stg_transactions s
WHERE EXISTS (
    SELECT 1 FROM fact_transactions f WHERE f.txn_id = s.txn_id
)
GROUP BY s.txn_id;
-- Rows returned here are skipped on insert (idempotency).
```

Listing 3. Check for transactions already in fact_transactions.

Rows returned by this query are not failures - they indicate the staging file contains transactions already loaded (re-runs, retries). The downstream fact load skips them via NOT EXISTS.

3.3 SCD Type 2 change detection

```
-- Detect customers needing SCD Type 2 update
WITH source_customers AS (
    SELECT DISTINCT customer_id, customer_name, tier,
           MD5(customer_name || '|' || tier) AS new_hash
    FROM stg_transactions
    WHERE customer_id IS NOT NULL
)
SELECT
    s.customer_id, s.customer_name, s.tier, s.new_hash,
    d.tier AS current_tier, d.row_hash AS current_hash
FROM source_customers s
JOIN dim_customer d
    ON d.customer_id = s.customer_id
   AND d.is_current = TRUE
WHERE d.row_hash <> s.new_hash;
```

Listing 4. Detect customers with tier or name changes since last load.

Customers returned by this query are candidates for SCD Type 2 update. For each: close out the current row and insert a new version. Wrap both operations in a single transaction (see Document 4, Section 11 for the SQL).

3.4 Fact load with surrogate-key resolution

```
-- Load facts with resolved surrogate keys
INSERT INTO fact_transactions (
    txn_id, txn_timestamp,
    date_sk, customer_sk, branch_sk, product_sk,
    channel_sk, txn_type_sk,
    amount_ngn, balance_after_ngn,
    deposit_amount_ngn, withdrawal_amount_ngn
)
SELECT
    s.txn_id, s.txn_datetime,
    TO_CHAR(s.txn_datetime, 'YYYYMMDD')::INT AS date_sk,
    c.customer_sk, b.branch_sk,
    p.product_sk, ch.channel_sk, tt.txn_type_sk,
    s.amount_ngn, s.balance_after_ngn,
    CASE WHEN s.txn_type = 'Deposit'    THEN s.amount_ngn ELSE 0 END,
    CASE WHEN s.txn_type = 'Withdrawal' THEN s.amount_ngn ELSE 0 END
FROM stg_transactions s
JOIN dim_customer c
    ON c.customer_id = s.customer_id
```

```
   AND s.txn_datetime >= c.effective_from
   AND s.txn_datetime <  c.effective_to + INTERVAL '1 day'
JOIN dim_branch b
    ON b.branch_id = s.branch_id
   AND s.txn_datetime >= b.effective_from
   AND s.txn_datetime <  b.effective_to + INTERVAL '1 day'
JOIN dim_product p           ON p.product_id = s.product_id
JOIN dim_channel ch          ON ch.channel_name = s.channel
JOIN dim_transaction_type tt ON tt.txn_type_name = s.txn_type
WHERE NOT EXISTS (
    SELECT 1 FROM fact_transactions f WHERE f.txn_id = s.txn_id
);
```

Listing 5. Fact load with SCD-aware joins and NOT EXISTS idempotency check.

Two things to notice in this SQL:

- The joins to dim_customer and dim_branch include effective-date range predicates. A transaction on 2024-06-15 joins to the customer version where effective_from <= '2024-06-15' < effective_to + 1 day.
- The NOT EXISTS clause makes the entire INSERT idempotent. Re-running the same staging data twice produces the same final state.

3.5 Aggregate refresh

```
-- Refresh agg_monthly_branch_activity for current + prior month
BEGIN;

DELETE FROM agg_monthly_branch_activity
WHERE month_key IN (
    TO_CHAR(CURRENT_DATE, 'YYYYMM')::INT,
    TO_CHAR(CURRENT_DATE - INTERVAL '1 month', 'YYYYMM')::INT
);

INSERT INTO agg_monthly_branch_activity (
    month_key, branch_sk, product_type,
    total_transaction_value,
    deposit_value_ngn, withdrawal_value_ngn,
    fee_income_ngn, transaction_count, active_customer_count
)
SELECT
    (d.year_num * 100 + d.month_num) AS month_key,
    f.branch_sk, p.product_type,
    SUM(f.amount_ngn),
    SUM(f.deposit_amount_ngn),
    SUM(f.withdrawal_amount_ngn),
    SUM(f.fee_amount_ngn),
    COUNT(*),
    COUNT(DISTINCT f.customer_sk)
FROM fact_transactions f
```

```
JOIN dim_date d    ON f.date_sk = d.date_sk
JOIN dim_product p ON f.product_sk = p.product_sk
WHERE (d.year_num * 100 + d.month_num) IN (
    TO_CHAR(CURRENT_DATE, 'YYYYMM')::INT,
    TO_CHAR(CURRENT_DATE - INTERVAL '1 month', 'YYYYMM')::INT
)
GROUP BY (d.year_num * 100 + d.month_num), f.branch_sk, p.product_type;

COMMIT;
```

Listing 6. Aggregate table refresh for current and prior month.

Only the current and prior month are recomputed each load. Older months never change (assuming no late-arriving facts beyond a month). If late-arriving facts are common in this warehouse, extend the WHERE clause to include the affected months.

4. Data Quality Framework

Data quality checks run after staging load, before any dimension or fact load. Failures are categorized by severity[object Object]:

Severity	Examples	Behavior
Blocking	Duplicate txn_id in staging. Null txn_id. Future-dated transaction. Invalid amount on non-reversal. Unmatched dimension FK after dim load.	Halt load. Page on-call.
Warning	Unexpected channel name (not in dim_channel reference). Customer tier outside known values. Unusual amount magnitude (>10x typical).	Log to error table. Continue load. Review next business day.
Info	First-time customer. First-time branch (legitimate new branch opening). Spike in transaction volume.	Log only. No action needed.

Table 4. Data quality severity tiers and behaviors.

4.1 Standard DQ check suite

```
-- Data quality checks (run after staging load, before fact load)

-- 1. Duplicate txn_id
SELECT txn_id, COUNT(*) AS dup_count
FROM stg_transactions
WHERE txn_id IS NOT NULL
GROUP BY txn_id
HAVING COUNT(*) > 1;
-- Expected: 0 rows

-- 2. Missing transaction IDs
SELECT COUNT(*) AS null_txn_ids
FROM stg_transactions
WHERE txn_id IS NULL;
-- Expected: 0

-- 3. Invalid amounts (null or zero on non-reversal types)
SELECT txn_id, txn_type, amount_ngn
FROM stg_transactions
WHERE (amount_ngn IS NULL OR amount_ngn <= 0)
  AND txn_type NOT IN ('Reversal', 'Adjustment');
-- Expected: 0 rows

-- 4. Future-dated transactions
SELECT txn_id, txn_datetime
```

```
FROM stg_transactions
WHERE txn_datetime > CURRENT_TIMESTAMP;
-- Expected: 0 rows

-- 5. Unmatched dimension lookups (after dim load, before fact load)
SELECT s.txn_id, s.customer_id
FROM stg_transactions s
LEFT JOIN dim_customer c
    ON c.customer_id = s.customer_id AND c.is_current = TRUE
WHERE c.customer_sk IS NULL;
-- Expected: 0 rows
```

Listing 7. The five standard DQ checks. All must return zero rows for the load to proceed.

4.2 Custom checks

In addition to the standard suite, project-specific checks should be added:

- Amount sign vs transaction type. Withdrawals should be positive in the amount column (the type determines direction). A negative withdrawal is suspicious.
- Balance continuity. balance_after_ngn for the same customer on consecutive transactions should differ by (signed) amount_ngn. Discontinuities flag potential data loss or duplicate transactions.
- Channel-type compatibility. ATM channel rarely sees Transfer transaction types; Branch rarely sees POS. Build a compatibility matrix; flag exceptions.

5. Troubleshooting Playbook

Common failures and their resolutions.

5.1 Symptom - load fails with 'unique constraint violation' on fact_transactions

Element	Detail
Cause	The same txn_id is being inserted twice. The NOT EXISTS clause should prevent this; if it triggers, either two staging rows have the same txn_id, or the staging table was loaded twice in the same batch.
Diagnosis	Query stg_transactions: SELECT txn_id, COUNT(*) FROM stg_transactions GROUP BY txn_id HAVING COUNT(*) > 1.
Resolution	Truncate stg_transactions. Re-run from staging-load step. Confirm source file does not contain duplicates.
Prevention	Add the duplicate-txn_id check to the pre-load DQ suite (Section 4) as a blocking check.

Table 5. Resolution for duplicate txn_id failures.

5.2 Symptom - fact load completes but row count is lower than staging

Element	Detail
Cause	Most common: unmatched dimension FK. A staging transaction references a customer_id or branch_id that does not exist in the dimension at the transaction's effective date.
Diagnosis	Run DQ Check 5 (Section 4) which surfaces unmatched dimension lookups. If it returns rows, the dimension load missed those records.
Resolution	Investigate whether the customer or branch was new in this batch. If yes, the dimension load step before fact load should have caught it - check the dimension load logs.
Prevention	Ensure DQ Check 5 is run AFTER dimension loads and BEFORE fact load.

Table 6. Resolution for missing fact rows.

5.3 Symptom - SCD Type 2 produces duplicate current rows

Element	Detail
Cause	The 'close old row' step in the SCD update failed or was skipped. Multiple rows for the same customer_id have is_current = TRUE.

Element	Detail
Diagnosis	Query: SELECT customer_id, COUNT(*) FROM dim_customer WHERE is_current = TRUE GROUP BY customer_id HAVING COUNT(*) > 1.
Resolution	Identify the duplicate rows. Set is_current = FALSE on all but the most recent (highest customer_sk) for each customer_id. Set effective_to on the closed rows to (effective_from of next version) minus one day.
Prevention	Wrap the SCD update in a transaction (BEGIN/COMMIT). Test for is_current = TRUE count after each batch as a post-load validation.

Table 7. Resolution for SCD Type 2 duplicate currents.

5.4 Symptom - aggregate table out of sync with fact_transactions

Element	Detail
Cause	Aggregate refresh either did not run or covered the wrong months. Late-arriving facts are the most common cause.
Diagnosis	Compare SUM from fact_transactions against agg_monthly_branch_activity for the suspect month. If they differ by more than rounding error, the aggregate is stale.
Resolution	Run the aggregate refresh manually for the affected months (extend the date range in Listing 6).
Prevention	Add a post-load validation step that compares fact-level SUM against aggregate SUM for the current month. Page on-call if the difference exceeds 0.1%.

Table 8. Resolution for stale aggregates.

6. Operational Monitoring

Three things should be monitored on every load:

6.1 Load duration

Phase	Expected duration (incremental)	Page if exceeds
Staging load	2 minutes	10 minutes
Dimension updates	1 minute	5 minutes
Fact load	3 minutes	15 minutes
Aggregate refresh	2 minutes	10 minutes
Total batch	10 minutes	30 minutes

Table 9. Phase-level duration expectations. Initial load durations scale with historical volume (typically 60-90 minutes total).

6.2 Row count drift

Compare daily row counts against the rolling 7-day average. Spikes (more than 3x typical) or drops (less than 30% of typical) are warning signs:

- Spike: likely batch double-load, source-system migration, or genuine business event. Investigate before next run.
- Drop: likely source extraction failed mid-run. Confirm source file is complete before declaring load successful.

6.3 DQ failure rate

Track the rate at which DQ checks reject rows. A sudden rise in any single check is a signal that something upstream has changed. Examples:

- Rise in null txn_id rate: source system may have a new code path that omits the ID.
- Rise in unmatched dimension FK: dimensional changes (new branch, new product type) not flowing through the dimension load step.
- Rise in future-dated transactions: clock skew on the source system, or a timezone misconfiguration in the staging load.

7. Recovery Procedures

7.1 Failed staging load

Symptoms: COPY command returns error mid-load; stg_transactions partially populated.

- Truncate stg_transactions.
- Investigate the source file - encoding errors, malformed rows, schema drift.
- Once source issue resolved, re-run from Step 1.
- Idempotency: re-running affects only staging; no downstream impact yet.

7.2 Failed dimension load

Symptoms: dimension load step throws an error after staging load succeeded.

- Identify the failing dimension from logs.
- Manually inspect the offending rows in staging.
- Resolve and re-run from the failing step onward.
- Idempotency: dimension loads use INSERT-IF-NOT-EXISTS patterns; safe to re-run.

7.3 Failed fact load

Symptoms: fact load completes partially or throws constraint violation.

- Check fact_transactions for the latest txn_id loaded. The NOT EXISTS clause should have skipped duplicates.
- If constraint violation is on a foreign key, run DQ Check 5 to identify orphan facts.
- Re-run the fact load step. NOT EXISTS handles the partial-load case.

7.4 Complete batch rollback

If a batch is unsalvageable and the changes must be reversed:

- Identify the load_batch_id.
- DELETE FROM fact_transactions WHERE txn_id IN (SELECT txn_id FROM stg_transactions). Skip if facts have been queried by downstream consumers - prefer to investigate forward.
- Roll back SCD Type 2 changes: identify dim_customer and dim_branch rows inserted with effective_from = batch date; either delete (if no facts depend on them) or extend effective_to of the previous version back to today.
- Rebuild aggregate for affected months.

> **ROLLBACK IS A LAST RESORT**
> In most cases the right answer is forward correction, not rollback. Downstream consumers may have already queried the new data; reversing the load creates audit complications. Use rollback only when the data is genuinely corrupted, not when it is merely inconvenient.

Document Control

Version	Date	Change Summary	Author
0.1	2026-02-20	Initial load procedure documented	Data Engineering
0.5	2026-03-02	Incremental load and DQ framework added	Data Engineering
0.9	2026-03-11	Troubleshooting playbook and recovery procedures added	Data Engineering
1.0	2026-03-15	Approved as operational reference	Data Engineering

Table 10. Revision history.