ChatGPT

Stage Conversion Report – Final Implementation

This document translates the Stage Conversion report design into concrete SQL DDL, query logic, and executable Python code. It adheres to the common reporting methodology defined in `high-level-reporting-methodology.md` and relies solely on the canonical schema in `pipeline_schema.sql`.

A) Report Storage DDL

The report results are stored in a tenant-scoped fact table. Each run inserts one or more rows depending on how many stage pairs are evaluated.

```
-- Report fact table for Stage Conversion metrics
CREATE TABLE IF NOT EXISTS dyno_crm.report_stage_conversion_fact (
    run_id UUID NOT NULL,
    tenant_id UUID NOT NULL,
    pipeline_id UUID NOT NULL,
    from_stage_id UUID NOT NULL,
    to_stage_id UUID NOT NULL,
    entered_count BIGINT NOT NULL,
    converted_count BIGINT NOT NULL,
    conversion_rate NUMERIC(5,4) NOT NULL,
    -- Foreign key back to report_run header for auditing
    CONSTRAINT fk_rsc_run FOREIGN KEY (run_id)
        REFERENCES dyno_crm.report_run (id) ON DELETE CASCADE,
    CONSTRAINT fk_rsc_tenant FOREIGN KEY (tenant_id)
        REFERENCES dyno_crm.report_run (tenant_id) ON DELETE CASCADE,
    -- Prevent duplicate rows per run/stage pair
    CONSTRAINT ux_rsc_run_stage UNIQUE (run_id, pipeline_id, from_stage_id,
to_stage_id)
);

-- Indexes for efficient lookups
CREATE INDEX IF NOT EXISTS ix_rsc_tenant_pipeline
    ON dyno_crm.report_stage_conversion_fact (tenant_id, pipeline_id);

CREATE INDEX IF NOT EXISTS ix_rsc_stage_pair
    ON dyno_crm.report_stage_conversion_fact (tenant_id, from_stage_id,
to_stage_id);
```

B) Report Generation Query

The following SQL query produces stage conversion metrics for a given tenant, pipeline, time window, and stage pairs. It relies on two CTEs:

1. `stage_entries` – returns one row per deal entry into a stage within the period, deduplicated by deal and stage.
2. `conversions` – for each stage entry, determines whether a later entry exists into the target stage.

When `stage_pairs` is not specified, the query derives consecutive stage pairs from `pipeline_stage` using `display_order`.

```
-- :tenant_id, :pipeline_id, :period_start, :period_end are parameters
-- :stage_pairs is a JSON array of objects {from_stage_id UUID, to_stage_id
UUID} or NULL

WITH stage_ordering AS (
    -- Derive consecutive stage pairs when stage_pairs is null
    SELECT ps.pipeline_id, ps.id AS from_stage_id, ps_next.id AS to_stage_id
    FROM dyno_crm.pipeline_stage ps
    JOIN dyno_crm.pipeline_stage ps_next
      ON ps_next.pipeline_id = ps.pipeline_id
     AND ps_next.display_order = ps.display_order + 1
    WHERE ps.pipeline_id = :pipeline_id
),
input_pairs AS (
    SELECT p.pipeline_id, p.from_stage_id, p.to_stage_id
    FROM jsonb_to_recordset(:stage_pairs::jsonb) AS p(pipeline_id UUID,
from_stage_id UUID, to_stage_id UUID)
    WHERE p.pipeline_id = :pipeline_id
),
pairs AS (
    SELECT pipeline_id, from_stage_id, to_stage_id
    FROM (
        SELECT * FROM input_pairs
        UNION ALL
        SELECT * FROM stage_ordering
    ) AS sp
),
stage_entries AS (
    -- For each deal and stage, pick the earliest entry in the period
(deduplicate re-entries)
    SELECT DISTINCT ON (dpss.deal_id, dpss.pipeline_stage_id)
           dpss.deal_id,
           dpss.pipeline_stage_id AS stage_id,
           dpss.entered_at
```

```
    FROM dyno_crm.deal_pipeline_stage_state dpss
    WHERE dpss.tenant_id = :tenant_id
      AND dpss.pipeline_id = :pipeline_id
      AND dpss.entered_at >= :period_start
      AND dpss.entered_at < :period_end
    ORDER BY dpss.deal_id, dpss.pipeline_stage_id, dpss.entered_at
),
conversions AS (
    SELECT
        p.pipeline_id,
        p.from_stage_id,
        p.to_stage_id,
        COUNT(DISTINCT se_from.deal_id) AS entered_count,
        COUNT(DISTINCT se_to.deal_id) AS converted_count
    FROM pairs p
    LEFT JOIN stage_entries se_from
      ON se_from.stage_id = p.from_stage_id
    LEFT JOIN stage_entries se_to
      ON se_to.stage_id = p.to_stage_id
     AND se_to.deal_id = se_from.deal_id
     AND se_to.entered_at > se_from.entered_at
    GROUP BY p.pipeline_id, p.from_stage_id, p.to_stage_id
)
SELECT
    :run_id AS run_id,
    :tenant_id AS tenant_id,
    c.pipeline_id,
    c.from_stage_id,
    c.to_stage_id,
    c.entered_count,
    c.converted_count,
    CASE WHEN c.entered_count > 0 THEN
         ROUND((c.converted_count::NUMERIC / c.entered_count), 4)
         ELSE 0
    END AS conversion_rate
FROM conversions c;
```

Notes:

- The `jsonb_to_recordset` usage allows passing an array of stage pairs. If `:stage_pairs` is null or empty, only the derived consecutive pairs from `stage_ordering` are used.
- `DISTINCT ON` in `stage_entries` selects the earliest occurrence of each stage entry per deal in the period. Without deduplication, re-entries would inflate the conversion counts.
- The query returns one row per stage pair with counts and conversion rate. Stage pairs that have no entries produce `entered_count = 0` and `conversion_rate = 0`.

C) Python Execution Code

Below is an example Python function (using `psycopg2`) to generate and persist a Stage Conversion report. It assumes the existence of a `report_run` table defined in the shared methodology and that a database connection is available. Error handling and connection management should be adapted to your environment.

```
import json
import psycopg2
from datetime import datetime

def run_stage_conversion_report(conn, tenant_id, pipeline_id, period_start,
period_end, stage_pairs=None):
    """
    Execute the Stage Conversion report for a tenant and pipeline over a given
period.
    Parameters:
        conn:        psycopg2 connection
        tenant_id:   UUID of the tenant
        pipeline_id: UUID of the pipeline
        period_start: datetime (inclusive)
        period_end:   datetime (exclusive)
        stage_pairs: Optional list of dicts {from_stage_id: UUID, to_stage_id:
UUID}
    """
    with conn.cursor() as cur:
        # 1. Insert a row into report_run to capture metadata
        cur.execute(
            """
            INSERT INTO dyno_crm.report_run (tenant_id, report_type,
generated_at, period_start, period_end, parameters)
            VALUES (%s, %s, NOW(), %s, %s, %s)
            RETURNING id
            """,
            (
                tenant_id,
                'STAGE_CONVERSION',
                period_start,
                period_end,
                json.dumps({'pipeline_id': str(pipeline_id), 'stage_pairs':
stage_pairs})
            )
        )
        run_id = cur.fetchone()[0]

        # 2. Execute the conversion query
```

```
        sql = open('stage_conversion_query.sql').read()
        cur.execute(
            sql,
            {
                'run_id': run_id,
                'tenant_id': tenant_id,
                'pipeline_id': pipeline_id,
                'period_start': period_start,
                'period_end': period_end,
                'stage_pairs': json.dumps(stage_pairs) if stage_pairs else None
            }
        )
        results = cur.fetchall()

        # 3. Insert fact rows
        insert_sql = """
            INSERT INTO dyno_crm.report_stage_conversion_fact
                (run_id, tenant_id, pipeline_id, from_stage_id, to_stage_id,
                 entered_count, converted_count, conversion_rate)
            VALUES (%s, %s, %s, %s, %s, %s, %s, %s)
        """
        for row in results:
            cur.execute(insert_sql, row)

        # 4. Commit transaction
        conn.commit()

    return run_id
```

Explanation:

1. A new row is inserted into `report_run` to record the report execution, including input parameters. The `report_type` is set to `'STAGE_CONVERSION'`.
2. The conversion SQL (stored in a separate `.sql` file for clarity) is executed with the run parameters. The parameter `:stage_pairs` is passed as JSON or null.
3. Each result row from the query is inserted into `report_stage_conversion_fact` with the generated `run_id` to tie facts to the run.
4. The transaction is committed. In production, error handling should roll back on exceptions and update the `report_run.status` accordingly.
