ChatGPT

Representative Performance Report – Final Implementation

This document finalizes the Representative Performance report, defining the storage schema, the SQL needed to compute performance metrics, and Python code to execute and store the results. It implements the detailed design based on the extracted CRM schema.

A) Report Storage DDL

```
-- Fact table for Representative Performance metrics
CREATE TABLE IF NOT EXISTS dyno_crm.report_rep_performance_fact (
    run_id UUID NOT NULL,
    tenant_id UUID NOT NULL,
    principal_type VARCHAR(20) NOT NULL,
    principal_id UUID NOT NULL,
    period_start TIMESTAMPTZ NOT NULL,
    period_end TIMESTAMPTZ NOT NULL,
    deals_created_count BIGINT NOT NULL,
    deals_closed_won_count BIGINT NOT NULL,
    deals_closed_lost_count BIGINT NOT NULL,
    total_amount_won NUMERIC(14,2) NOT NULL,
    avg_deal_size_won NUMERIC(14,2),
    avg_time_to_close_days NUMERIC(10,4),
    win_rate NUMERIC(5,4),
    CONSTRAINT fk_rrp_run FOREIGN KEY (run_id)
        REFERENCES dyno_crm.report_run (id) ON DELETE CASCADE,
    CONSTRAINT fk_rrp_tenant FOREIGN KEY (tenant_id)
        REFERENCES dyno_crm.report_run (tenant_id) ON DELETE CASCADE,
    CONSTRAINT ux_rrp_unique_row UNIQUE (run_id)
);

-- Indexes to support queries by tenant/principal/period
CREATE INDEX IF NOT EXISTS ix_rrp_tenant_principal
    ON dyno_crm.report_rep_performance_fact (tenant_id, principal_type,
principal_id);

CREATE INDEX IF NOT EXISTS ix_rrp_period
    ON dyno_crm.report_rep_performance_fact (period_start, period_end);
```

B) Report Generation Query

The following SQL query computes performance metrics for a single principal and period. It uses subqueries to calculate each component (created deals, won deals, lost deals, amounts, durations) and combines them in a single SELECT for insertion into the fact table.

```
--
Parameters: :run_id, :tenant_id, :principal_type, :principal_id, :period_start, :period_end

WITH attributed_deals AS (
    -- Deals created in the period and attributed to the principal
    SELECT d.id AS deal_id, d.amount, d.created_at
    FROM dyno_crm.deal d
    WHERE d.tenant_id = :tenant_id
      AND d.created_at >= :period_start
      AND d.created_at < :period_end
      AND (
        (
            :principal_type = 'USER'
            AND (d.owned_by_user_id = :principal_id OR d.assigned_user_id
= :principal_id)
        )
        OR (
            :principal_type = 'GROUP'
            AND (d.owned_by_group_id = :principal_id OR d.assigned_group_id
= :principal_id)
        )
      )
),
terminal_stage_entries AS (
    -- Find the earliest terminal stage entry for each deal within the period
    SELECT DISTINCT ON (dpss.deal_id)
        dpss.deal_id,
        dpss.entered_at AS close_ts,
        ps.stage_state
    FROM dyno_crm.deal_pipeline_stage_state dpss
    JOIN dyno_crm.pipeline_stage ps ON ps.id = dpss.pipeline_stage_id
    WHERE dpss.tenant_id = :tenant_id
      AND dpss.is_current = TRUE
      AND ps.stage_state IN ('DONE_SUCCESS', 'DONE_FAILED')
      AND dpss.entered_at >= :period_start
      AND dpss.entered_at < :period_end
    ORDER BY dpss.deal_id, dpss.entered_at
),
closed_deals AS (
    SELECT
```

```
        ad.deal_id,
        ad.amount,
        ad.created_at,
        ts.close_ts,
        ts.stage_state
    FROM attributed_deals ad
    JOIN terminal_stage_entries ts ON ts.deal_id = ad.deal_id
),
created_ct AS (
    SELECT COUNT(*) AS deals_created_count
    FROM attributed_deals
),
closed_won AS (
    SELECT COUNT(*) AS won_count,
           COALESCE(SUM(COALESCE(cd.amount, 0)), 0) AS total_amount_won,
           CASE WHEN COUNT(*) > 0 THEN AVG(COALESCE(cd.amount, 0)) ELSE NULL
END AS avg_deal_size_won
    FROM closed_deals cd
    WHERE cd.stage_state = 'DONE_SUCCESS'
),
closed_lost AS (
    SELECT COUNT(*) AS lost_count
    FROM closed_deals cd
    WHERE cd.stage_state = 'DONE_FAILED'
),
close_durations AS (
    SELECT
        COUNT(*) AS closed_count,
        CASE WHEN COUNT(*) > 0 THEN AVG(EXTRACT(EPOCH FROM
(COALESCE(cd.close_ts, cd.created_at) - cd.created_at)) / 86400.0) ELSE NULL
END AS avg_time_days
    FROM closed_deals cd
)
SELECT
    :run_id AS run_id,
    :tenant_id AS tenant_id,
    :principal_type AS principal_type,
    :principal_id AS principal_id,
    :period_start AS period_start,
    :period_end AS period_end,
    ct.deals_created_count,
    cw.won_count AS deals_closed_won_count,
    cl.lost_count AS deals_closed_lost_count,
    cw.total_amount_won,
    cw.avg_deal_size_won,
    cdur.avg_time_days AS avg_time_to_close_days,
    CASE WHEN (cw.won_count + cl.lost_count) > 0 THEN
         ROUND(cw.won_count::NUMERIC / (cw.won_count + cl.lost_count), 4)
```

```
        ELSE NULL
    END AS win_rate
FROM created_ct ct, closed_won cw, closed_lost cl, close_durations cdur;
```

Notes:

- `attributed_deals` picks deals created in the period with owner/assignee fields matching the principal at creation time.
- `terminal_stage_entries` finds the current stage entry for each deal (the latest row with `is_current = true`) that is terminal and falls within the period. If a deal has not yet closed, it will not appear here.
- `closed_deals` joins attributed deals with terminal entries to compute amounts and durations.
- Aggregation subqueries compute counts, sums, and averages. The final SELECT assembles all metrics into a single row.
- `avg_time_to_close_days` uses either `close_ts` (the `entered_at` timestamp of the terminal stage) or falls back to `created_at` if no terminal entry exists. In practice, deals that haven't closed should not contribute to closed metrics.

C) Python Execution Code

```
import json
import psycopg2
from datetime import datetime

def run_rep_performance_report(conn, tenant_id, principal_type, principal_id,
period_start, period_end):
    """
    Execute the Representative Performance report and insert the result.
    Parameters:
        conn: psycopg2 connection
        tenant_id: UUID
        principal_type: 'USER' or 'GROUP'
        principal_id: UUID
        period_start: datetime (inclusive)
        period_end: datetime (exclusive)
    """
    with conn.cursor() as cur:
        # Insert run header
        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,
                'REP_PERFORMANCE',
                period_start,
                period_end,
                json.dumps({'principal_type': principal_type, 'principal_id':
str(principal_id)})
            )
        )
        run_id = cur.fetchone()[0]

        # Execute performance query
        sql = open('rep_performance_query.sql').read()
        cur.execute(
            sql,
            {
                'run_id': run_id,
                'tenant_id': tenant_id,
                'principal_type': principal_type,
                'principal_id': principal_id,
                'period_start': period_start,
                'period_end': period_end
            }
        )
        result = cur.fetchone()

        # Insert into fact table
        insert_sql = """
            INSERT INTO dyno_crm.report_rep_performance_fact (
                run_id, tenant_id, principal_type, principal_id,
                period_start, period_end, deals_created_count,
                deals_closed_won_count, deals_closed_lost_count,
                total_amount_won, avg_deal_size_won,
                avg_time_to_close_days, win_rate
            ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s)
        """
        cur.execute(insert_sql, result)

        conn.commit()

    return run_id
```

Explanation:

1. The function records a new run with type `'REP_PERFORMANCE'` and stores the input window and principal. The parameters are serialized to JSON for auditing.

2. It executes the complex SQL query (stored in `rep_performance_query.sql`) with the run and filter parameters. The query returns exactly one row containing all metrics.
3. The resulting row is inserted into the `report_rep_performance_fact` table. A unique constraint on `run_id` ensures one row per execution.
4. The transaction is committed, finalising the snapshot.
