ChatGPT

Forecast vs Actual (Accuracy) Report – Final Implementation Design

This document describes the final implementation for the **Forecast vs Actual (Accuracy)** report. It defines the storage schema, provides a deterministic SQL query to calculate forecast and actual revenue for a given period, and includes example Python code for executing the report. This implementation assumes forecasts are generated using current data (i.e., `forecast_as_of` is ignored or equal to now). More advanced snapshot handling would require historical state reconstruction or snapshot tables.

A) SQL DDL – Report Storage

1. `report_run` (shared header)

Ensure the `report_run` table from previous final designs exists. It records each report execution's metadata.

2. `report_forecast_accuracy_fact`

Create the fact table for storing forecast vs actual metrics. Each row corresponds to one combination of pipeline and principal for a particular run.

```
CREATE TABLE IF NOT EXISTS dyno_crm.report_forecast_accuracy_fact (
    run_id UUID NOT NULL,
    tenant_id UUID NOT NULL,
    pipeline_id UUID,
    principal_type VARCHAR(5),
    principal_id UUID,
    period_start DATE NOT NULL,
    period_end DATE NOT NULL,
    forecast_as_of TIMESTAMPTZ,
    forecast_amount NUMERIC(20,2) NOT NULL,
    actual_amount NUMERIC(20,2) NOT NULL,
    forecast_count BIGINT NOT NULL,
    actual_count BIGINT NOT NULL,
    forecast_accuracy NUMERIC(10,4),
    variance_amount NUMERIC(20,2),
    variance_percent NUMERIC(10,4),
    PRIMARY KEY (run_id, pipeline_id, principal_type, principal_id),
    CONSTRAINT fk_rfaf_run FOREIGN KEY (run_id) REFERENCES dyno_crm.report_run
(run_id) ON DELETE CASCADE
);
```

```
CREATE INDEX IF NOT EXISTS ix_rfaf_tenant_pipeline
    ON dyno_crm.report_forecast_accuracy_fact (tenant_id, pipeline_id);

CREATE INDEX IF NOT EXISTS ix_rfaf_principal
    ON dyno_crm.report_forecast_accuracy_fact (tenant_id, principal_type,
principal_id);
```

B) SQL Query – Report Generation

The following SQL computes forecast and actual revenue for a given tenant, period and optional filters. It uses two subqueries—one for the forecast (weighted pipeline) and one for actual closed deals—and then joins them. Note that `forecast_as_of` is accepted as a parameter but not used; current stage data is employed for forecasts.

```
WITH params AS (
    SELECT
        :tenant_id      AS tenant_id,
        :period_start   AS period_start,
        :period_end     AS period_end,
        :forecast_as_of AS forecast_as_of,
        :pipeline_id    AS pipeline_id,
        :principal_type AS principal_type,
        :principal_id   AS principal_id
),
-- Forecast data: open deals expected to close within the period
forecast_candidates AS (
    SELECT
        d.id AS deal_id,
        d.pipeline_id,
        d.amount,
        d.expected_close_date,
        d.forecast_probability,
        cs.pipeline_stage_id,
        d.owned_by_user_id,
        d.owned_by_group_id,
        d.assigned_user_id,
        d.assigned_group_id
    FROM dyno_crm.deal d
    JOIN dyno_crm.deal_pipeline_stage_state cs
      ON cs.deal_id = d.id
     AND cs.tenant_id = d.tenant_id
     AND cs.is_current = TRUE
    JOIN dyno_crm.pipeline_stage ps
      ON ps.id = cs.pipeline_stage_id
     AND ps.tenant_id = d.tenant_id
```

```sql
    JOIN params p ON true
    WHERE d.tenant_id = p.tenant_id
      AND ps.stage_state NOT IN ('DONE_SUCCESS','DONE_FAILED')
      AND d.expected_close_date >= p.period_start
      AND d.expected_close_date <  p.period_end
      AND (p.pipeline_id IS NULL OR d.pipeline_id = p.pipeline_id)
      AND (
            p.principal_type IS NULL
            OR (
                 p.principal_type = 'USER'  AND (d.owned_by_user_id =
p.principal_id OR d.assigned_user_id = p.principal_id)
               )
            OR (
                 p.principal_type = 'GROUP' AND (d.owned_by_group_id =
p.principal_id OR d.assigned_group_id = p.principal_id)
               )
          )
),
forecast_data AS (
    SELECT
        COALESCE(fc.pipeline_id, params.pipeline_id) AS pipeline_id,
        params.principal_type                        AS principal_type,
        params.principal_id                          AS principal_id,
        COUNT(*)                                     AS forecast_count,
        SUM(fc.amount * COALESCE(fc.forecast_probability, ps.probability, 0))
AS forecast_amount
    FROM forecast_candidates fc
    JOIN dyno_crm.pipeline_stage ps
      ON ps.id = fc.pipeline_stage_id
     AND ps.tenant_id = :tenant_id
    JOIN params ON true
    GROUP BY COALESCE(fc.pipeline_id, params.pipeline_id),
params.principal_type, params.principal_id
),
-- Actual data: deals closed won during the period
actual_candidates AS (
    SELECT
        d.id AS deal_id,
        d.pipeline_id,
        d.amount,
        dpss.entered_at AS closed_at,
        d.owned_by_user_id,
        d.owned_by_group_id,
        d.assigned_user_id,
        d.assigned_group_id
    FROM dyno_crm.deal d
    JOIN dyno_crm.deal_pipeline_stage_state dpss
      ON dpss.deal_id = d.id
     AND dpss.tenant_id = d.tenant_id
     AND dpss.is_current = TRUE
    JOIN dyno_crm.pipeline_stage ps
      ON ps.id = dpss.pipeline_stage_id
     AND ps.tenant_id = d.tenant_id
    JOIN params p ON true
    WHERE d.tenant_id = p.tenant_id
      AND ps.stage_state = 'DONE_SUCCESS'
      AND dpss.entered_at >= p.period_start
      AND dpss.entered_at <  p.period_end
      AND (p.pipeline_id IS NULL OR d.pipeline_id = p.pipeline_id)
      AND (
            p.principal_type IS NULL
            OR (
                 p.principal_type = 'USER'  AND (d.owned_by_user_id =
p.principal_id OR d.assigned_user_id = p.principal_id)
               )
            OR (
                 p.principal_type = 'GROUP' AND (d.owned_by_group_id =
p.principal_id OR d.assigned_group_id = p.principal_id)
               )
          )
),
actual_data AS (
    SELECT
        COALESCE(ac.pipeline_id, params.pipeline_id) AS pipeline_id,
        params.principal_type                        AS principal_type,
        params.principal_id                          AS principal_id,
        COUNT(*)                                     AS actual_count,
        SUM(COALESCE(ac.amount,0))                   AS actual_amount
    FROM actual_candidates ac
    JOIN params ON true
    GROUP BY COALESCE(ac.pipeline_id, params.pipeline_id),
params.principal_type, params.principal_id
)
SELECT
    COALESCE(fd.pipeline_id, ad.pipeline_id) AS pipeline_id,
    COALESCE(fd.principal_type, ad.principal_type) AS principal_type,
    COALESCE(fd.principal_id, ad.principal_id) AS principal_id,
    params.period_start,
    params.period_end,
    params.forecast_as_of,
    COALESCE(fd.forecast_amount, 0) AS forecast_amount,
    COALESCE(ad.actual_amount, 0)   AS actual_amount,
    COALESCE(fd.forecast_count, 0)  AS forecast_count,
    COALESCE(ad.actual_count, 0)    AS actual_count,
    CASE WHEN COALESCE(fd.forecast_amount, 0) > 0
         THEN COALESCE(ad.actual_amount, 0) / COALESCE(fd.forecast_amount, 0)
    END AS forecast_accuracy,
    COALESCE(ad.actual_amount, 0) - COALESCE(fd.forecast_amount, 0) AS
variance_amount,
    CASE WHEN COALESCE(fd.forecast_amount, 0) > 0
         THEN (COALESCE(ad.actual_amount, 0) - COALESCE(fd.forecast_amount,
0)) / COALESCE(fd.forecast_amount, 0)
    END AS variance_percent
FROM params
LEFT JOIN forecast_data fd
  ON true
LEFT JOIN actual_data ad
  ON (COALESCE(fd.pipeline_id, params.pipeline_id) = COALESCE(ad.pipeline_id,
params.pipeline_id))
 AND (COALESCE(fd.principal_type, params.principal_type) =
COALESCE(ad.principal_type, params.principal_type))
 AND (COALESCE(fd.principal_id, params.principal_id) =
COALESCE(ad.principal_id, params.principal_id));
```

Explanation:

1. The `params` CTE binds input parameters for readability.
2. `forecast_candidates` selects open deals expected to close within the period, applying pipeline and principal filters. It joins the current stage to ensure we only consider open opportunities.
3. `forecast_data` groups forecast candidates by pipeline and principal, computing the weighted forecast using either the deal's `forecast_probability` override or the stage's default probability. The sum of weighted amounts and the count of deals form the forecast metrics.
4. `actual_candidates` selects deals that closed won during the period by looking at current stage instances with state `DONE_SUCCESS`. Pipeline and principal filters ensure symmetry with the forecast.
5. `actual_data` groups actual candidates to produce sums and counts.
6. The final SELECT joins forecast and actual data on matching dimensions, computing accuracy ratios and variance values. When a group is present in one set but not the other, missing values default to zero.

C) Python Execution Code

The Python function below demonstrates how to run the forecast vs actual report. It follows the common pattern of inserting a run header, executing the query with bound parameters, inserting fact rows, and marking the run as succeeded. For clarity, the SQL is assumed to be stored in a separate file (`forecast_vs_actual_query.sql`).

```
import uuid
import json
import psycopg2
from datetime import datetime
```

```
def run_forecast_vs_actual_report(conn, tenant_id, period_start, period_end,
                                  forecast_as_of=None,
                                  pipeline_id=None, principal_type=None,
principal_id=None):
    """
    Executes the Forecast vs Actual (Accuracy) report for a given tenant, period
and optional filters.
    Inserts a row into report_run and fact rows into
report_forecast_accuracy_fact.
    Returns the run_id.
    """
    run_id = uuid.uuid4()
    now_ts = datetime.utcnow()
    input_params = {
        "period_start": period_start.isoformat(),
        "period_end": period_end.isoformat(),
        "forecast_as_of": forecast_as_of.isoformat() if forecast_as_of else
None,
        "pipeline_id": str(pipeline_id) if pipeline_id else None,
        "principal_type": principal_type,
        "principal_id": str(principal_id) if principal_id else None,
    }
    with conn.cursor() as cur:
        # Create report run header
        cur.execute(
            "INSERT INTO dyno_crm.report_run (run_id, tenant_id, report_type,
generated_at, period_start, period_end, input_params, status) "
            "VALUES (%s, %s, %s, %s, %s, %s, %s, 'IN_PROGRESS')",
            (run_id, tenant_id, 'forecast_vs_actual', now_ts, period_start,
period_end, json.dumps(input_params))
        )
        # Execute the report query
        cur.execute(
            open('forecast_vs_actual_query.sql').read(),  # Save SQL above into
this file
            {
                'tenant_id': tenant_id,
                'period_start': period_start,
                'period_end': period_end,
                'forecast_as_of': forecast_as_of,
                'pipeline_id': pipeline_id,
                'principal_type': principal_type,
                'principal_id': principal_id,
            }
        )
        rows = cur.fetchall()
        for (pipeline_id_val, principal_type_val, principal_id_val,
             period_start_val, period_end_val, forecast_as_of_val,
               forecast_amount, actual_amount, forecast_count, actual_count,
               forecast_accuracy, variance_amount, variance_percent) in rows:
              cur.execute(
                  "INSERT INTO dyno_crm.report_forecast_accuracy_fact (
                      run_id, tenant_id, pipeline_id, principal_type,
principal_id,
                      period_start, period_end, forecast_as_of,
                      forecast_amount, actual_amount,
                      forecast_count, actual_count,
                      forecast_accuracy, variance_amount, variance_percent
                  ) VALUES (%s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s, %s,
%s, %s)",
                  (
                      run_id, tenant_id, pipeline_id_val, principal_type_val,
principal_id_val,
                      period_start_val, period_end_val, forecast_as_of_val,
                      forecast_amount or 0, actual_amount or 0,
                      forecast_count or 0, actual_count or 0,
                      forecast_accuracy, variance_amount, variance_percent
                  )
              )
          # Mark run as succeeded
          cur.execute(
              "UPDATE dyno_crm.report_run SET status = 'SUCCEEDED' WHERE run_id =
%s",
              (run_id,)
          )
      conn.commit()
      return run_id
```

Note: The SQL should be stored in `forecast_vs_actual_query.sql` or embedded directly in the Python string. In a full implementation, add error handling to capture and record exceptions.
