End-to-End PostgreSQL Data Architecture: Partitioning, Query Optimization, and ETL Automation at Scale

-- Keywords: data engineering, query optimization, window functions, CTE, partitioning, indexing,

-- materialized view, JSONB, full-text search, concurrency, CDC, incremental ETL, schema evolution, upsert

-- DB: PostgreSQL (12+), adjust dialect-specific parts for other systems

```
-------------------------------------------------------------------------------
-- 0) Safety: Work in a sandbox schema
-------------------------------------------------------------------------------
DROP SCHEMA IF EXISTS prod_analytics CASCADE;
CREATE SCHEMA prod_analytics AUTHORIZATION CURRENT_USER;
SET search_path = prod_analytics, public;

-------------------------------------------------------------------------------
-- 1) Core OLTP table: users & organizations (demonstrates FK, constraints, generated cols)
-------------------------------------------------------------------------------
CREATE TABLE prod_analytics.organizations (
    org_id          BIGSERIAL PRIMARY KEY,
    org_name        TEXT NOT NULL,
    industry        TEXT,
```

```
    created_at       TIMESTAMP WITH TIME ZONE DEFAULT now(),
    metadata         JSONB DEFAULT '{}'::JSONB
);

COMMENT ON TABLE prod_analytics.organizations IS 'Organization master (data
engineering, lineage metadata)';


CREATE TABLE prod_analytics.users (
    user_id          BIGSERIAL PRIMARY KEY,
    org_id           BIGINT NOT NULL REFERENCES prod_analytics.organizations(org_id) ON
DELETE CASCADE,
    email            CITEXT NOT NULL UNIQUE,
    username         TEXT,
    signup_ts        TIMESTAMP WITH TIME ZONE DEFAULT now(),
    attrs            JSONB DEFAULT '{}'::JSONB,
    -- generated column: domain extracted from email for analytics segmentation
    email_domain     TEXT GENERATED ALWAYS AS (split_part(email, '@', 2)) STORED,
    is_active        BOOLEAN DEFAULT TRUE,
    last_login       TIMESTAMP WITH TIME ZONE,
    created_at       TIMESTAMP WITH TIME ZONE DEFAULT now()
);

CREATE INDEX idx_users_org_created ON prod_analytics.users (org_id, created_at DESC);

COMMENT ON TABLE prod_analytics.users IS 'User master table with generated column for
email_domain';


-------------------------------------------------------------------------------

-- 2) Event table for analytics (append-only, partitioned by day, optimized for high ingest)
```

```
--    Uses RANGE partitioning for time-series scale; BRIN index for fast scans on large data.
-------------------------------------------------------------------------------
CREATE TABLE prod_analytics.events (
    event_id         BIGSERIAL NOT NULL,
    user_id          BIGINT REFERENCES prod_analytics.users(user_id),
    org_id           BIGINT,
    event_type       TEXT NOT NULL,
    event_props      JSONB,
    event_time       TIMESTAMP WITH TIME ZONE NOT NULL DEFAULT now(),
    received_at      TIMESTAMP WITH TIME ZONE DEFAULT now(),
    -- useful derived column for micro-bucketing (example)
    event_date       DATE GENERATED ALWAYS AS (event_time::date) STORED
) PARTITION BY RANGE (event_date);


-- create monthly partitions for 12 months (example)
DO $$
BEGIN
  FOR i IN 0..11 LOOP
    EXECUTE format('
      CREATE TABLE IF NOT EXISTS prod_analytics.events_p%s PARTITION OF
prod_analytics.events
      FOR VALUES FROM (date_trunc("month", current_date) + INTERVAL "%s month")
                  TO (date_trunc("month", current_date) + INTERVAL "%s month")
    ', i, i, i+1);
  END LOOP;
```

```
END $$;


-- Indexing: composite btree for common filters; BRIN for very large time scans;

CREATE INDEX IF NOT EXISTS idx_events_user_event_time ON prod_analytics.events (user_id, event_time DESC);

CREATE INDEX IF NOT EXISTS idx_events_org_time ON prod_analytics.events (org_id, event_time DESC);

-- BRIN index on partitioned table (benefit for huge append-only data)

CREATE INDEX IF NOT EXISTS brin_events_event_date ON prod_analytics.events USING BRIN (event_date);


COMMENT ON TABLE prod_analytics.events IS 'Append-only event table partitioned by date. Keywords: partitioning, BRIN, time-series.';


-------------------------------------------------------------------------------

-- 3) Full-text search & semantic tagging using tsvector and GIN (search across articles/support logs)

-------------------------------------------------------------------------------

CREATE TABLE prod_analytics.knowledge_base (

    kb_id           BIGSERIAL PRIMARY KEY,

    org_id          BIGINT REFERENCES prod_analytics.organizations(org_id),

    title           TEXT NOT NULL,

    content         TEXT NOT NULL,

    content_json    JSONB DEFAULT '{}'::JSONB,

    created_at      TIMESTAMP WITH TIME ZONE DEFAULT now(),

    updated_at      TIMESTAMP WITH TIME ZONE DEFAULT now(),

    -- materialized searchable vector (for full-text search)
```

```
    search_vector    TSVECTOR GENERATED ALWAYS AS (
        setweight(to_tsvector('english', coalesce(title,'')), 'A') ||
        setweight(to_tsvector('english', coalesce(content,'')), 'B')
    ) STORED
);

CREATE INDEX idx_kb_search_vector ON prod_analytics.knowledge_base USING GIN (search_vector);

COMMENT ON TABLE prod_analytics.knowledge_base IS 'Knowledge base with full-text search support (GIN).';


-------------------------------------------------------------------------------

-- 4) Upsert pattern (ON CONFLICT) for idempotent ETL / CDC ingestion

-------------------------------------------------------------------------------

-- Example staging table for CDC

CREATE TABLE prod_analytics.staging_users (
    src_id          TEXT PRIMARY KEY,
    email           CITEXT,
    username        TEXT,
    org_src         TEXT,
    payload         JSONB,
    ingested_at     TIMESTAMP WITH TIME ZONE DEFAULT now()
);


-- Upsert into users from staging: idempotent CDC pattern

-- demonstrates upsert strategy, conflict_target, and returning clause

WITH upsert AS (
```

```
    INSERT INTO prod_analytics.users (org_id, email, username, attrs, created_at)
    SELECT o.org_id, s.email, s.username, s.payload, now()
    FROM prod_analytics.staging_users s
    LEFT JOIN LATERAL (
        SELECT org_id FROM prod_analytics.organizations WHERE org_name = s.org_src
LIMIT 1
    ) o ON TRUE
    ON CONFLICT (email) DO UPDATE
      SET username = EXCLUDED.username,
          attrs = users.attrs || EXCLUDED.attrs,
          last_login = CASE WHEN EXCLUDED.attrs->>'last_login' IS NOT NULL THEN
(EXCLUDED.attrs->>'last_login')::timestamptz ELSE users.last_login END
    RETURNING users.user_id, users.email, users.username
)
SELECT 'UPSERT_COMPLETE' as result, count(*) FROM upsert;


-------------------------------------------------------------------------------
-- 5) Recursive CTE: org hierarchy / reporting chain analytics
--    Demonstrates recursive CTEs and graph traversal
-------------------------------------------------------------------------------
CREATE TABLE prod_analytics.org_hierarchy (
    node_id     BIGSERIAL PRIMARY KEY,
    org_id      BIGINT REFERENCES prod_analytics.organizations(org_id),
    parent_id   BIGINT REFERENCES prod_analytics.org_hierarchy(node_id),
    role        TEXT,
    metadata    JSONB
```

```
);


-- Recursive traversal to get full path and depth
WITH RECURSIVE hierarchy AS (
    SELECT node_id, org_id, parent_id, role, metadata, 1 AS depth, array[node_id] AS path
    FROM prod_analytics.org_hierarchy WHERE parent_id IS NULL
  UNION ALL
    SELECT h.node_id, h.org_id, h.parent_id, h.role, h.metadata, r.depth + 1, r.path || h.node_id
    FROM prod_analytics.org_hierarchy h
    JOIN hierarchy r ON h.parent_id = r.node_id
)
SELECT * FROM hierarchy ORDER BY depth DESC LIMIT 100;


-------------------------------------------------------------------------------
-- 6) Analytical query: rolling metrics and retention using window functions
--    Common MAANG interview prompt: DAU/MAU, retention, event funnel
-------------------------------------------------------------------------------
-- DAU (daily active users) and 7-day rolling active users:
WITH dau AS (
  SELECT event_date,
         count(DISTINCT user_id) AS dau_count
  FROM prod_analytics.events
  WHERE event_time >= now() - INTERVAL '90 days'
  GROUP BY event_date
)
```

```
SELECT
  event_date,
  dau_count,
  SUM(dau_count) OVER (ORDER BY event_date ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS rolling_7_day_active_users
FROM dau
ORDER BY event_date DESC
LIMIT 90;


-- 7-day retention cohort analysis: cohort by signup_date, retention by event_date
WITH cohorts AS (
  SELECT u.user_id, u.created_at::date AS signup_date, e.event_time::date AS event_date
  FROM prod_analytics.users u
  LEFT JOIN prod_analytics.events e ON e.user_id = u.user_id
  WHERE u.created_at >= current_date - INTERVAL '90 days'
)
SELECT
  signup_date,
  event_date,
  COUNT(DISTINCT user_id) AS active_users,
  (event_date - signup_date) AS days_after_signup
FROM cohorts
GROUP BY signup_date, event_date
ORDER BY signup_date DESC, days_after_signup
LIMIT 200;
```

```
-------------------------------------------------------------------------------
-- 7) Lateral join and JSONB-heavy query: extract nested properties & flatten arrays
-------------------------------------------------------------------------------
-- Example event_props has schema {"product": {"id": "...", "attributes": [{"k":"", "v":""}]}}
SELECT e.event_id, e.user_id, e.event_time,
    (e.event_props->'product'->>'id')::text AS product_id,
    k.attr->>'k' AS attr_key,
    k.attr->>'v' AS attr_value
FROM prod_analytics.events e
CROSS JOIN LATERAL jsonb_array_elements(coalesce(e.event_props->'product'->'attributes', '[]'::jsonb)) AS k(attr)
WHERE e.event_type = 'product_view'
LIMIT 500;


-------------------------------------------------------------------------------
-- 8) Materialized View for expensive aggregated reports + incremental refresh strategy
-------------------------------------------------------------------------------
CREATE MATERIALIZED VIEW IF NOT EXISTS prod_analytics.mv_org_daily_metrics
AS
SELECT
  org_id,
  event_date,
  count(DISTINCT user_id) AS dau,
  count(*) FILTER (WHERE event_type = 'purchase') AS purchases,
```

```
  sum((event_props->>'price')::numeric) FILTER (WHERE (event_props->>'price') IS NOT
NULL) AS revenue,

  now() AS computed_at

FROM prod_analytics.events

GROUP BY org_id, event_date;


-- For incremental refresh: using CONCURRENTLY (note: requires unique index on mat view)

CREATE UNIQUE INDEX IF NOT EXISTS mv_org_daily_metrics_idx ON
prod_analytics.mv_org_daily_metrics (org_id, event_date);

-- Refresh concurrently in maintenance job (example):

-- REFRESH MATERIALIZED VIEW CONCURRENTLY
prod_analytics.mv_org_daily_metrics;


-------------------------------------------------------------------------------

-- 9) PL/pgSQL: idempotent incremental ETL function + observability

--    Demonstrates transactional control, exception handling, logging, and metrics table

-------------------------------------------------------------------------------

CREATE TABLE IF NOT EXISTS prod_analytics.etl_jobs (

    job_id      BIGSERIAL PRIMARY KEY,

    job_name    TEXT NOT NULL,

    started_at  TIMESTAMPTZ,

    finished_at TIMESTAMPTZ,

    status      TEXT,

    rows_processed BIGINT,

    details     JSONB

);
```

```
-- simple incremental ETL: move from events staging into partitioned events table
CREATE OR REPLACE FUNCTION prod_analytics.etl_ingest_events(batch_size INT DEFAULT 10000)
RETURNS JSONB
LANGUAGE plpgsql
AS $$
DECLARE
  v_start TIMESTAMPTZ := now();
  v_rows BIGINT := 0;
  v_job_id BIGINT;
BEGIN
  INSERT INTO prod_analytics.etl_jobs (job_name, started_at, status)
  VALUES ('ingest_events', v_start, 'RUNNING') RETURNING job_id INTO v_job_id;

  -- Simulated idempotent move: assume staging_events exists with unique event_id
  WITH moved AS (
    DELETE FROM prod_analytics.staging_events se
    WHERE se.event_time < now() + INTERVAL '1 second'
    RETURNING se.*
  )
  INSERT INTO prod_analytics.events (user_id, org_id, event_type, event_props, event_time, received_at)
  SELECT user_id, org_id, event_type, payload, event_time, now()
  FROM moved
  ON CONFLICT (event_id) DO NOTHING
```

```
  RETURNING 1 INTO v_rows;

  -- update job record
  UPDATE prod_analytics.etl_jobs
     SET finished_at = now(), status = 'SUCCESS', rows_processed = COALESCE(v_rows,0)
   WHERE job_id = v_job_id;

  RETURN jsonb_build_object('job_id', v_job_id, 'rows', COALESCE(v_rows,0),
'duration_seconds', EXTRACT(EPOCH FROM now() - v_start));
EXCEPTION WHEN OTHERS THEN
  UPDATE prod_analytics.etl_jobs
     SET finished_at = now(), status = 'FAILED', details = jsonb_build_object('error', SQLERRM)
   WHERE job_id = v_job_id;
  RAISE;
END;
$$;

-------------------------------------------------------------------------------
-- 10) Concurrency & advisory locks: ensure single-run for ETL, lock-based coordination
-------------------------------------------------------------------------------
-- Use advisory locks for cross-process coordination:
-- SELECT pg_try_advisory_lock(hashtext('etl_ingest_events'));
-- run function
-- SELECT prod_analytics.etl_ingest_events();
-------------------------------------------------------------------------------
```

-- 11) Observability & introspection queries: query plan + statistics

---

-- Show top 10 slowest queries (requires pg_stat_statements extension). This is an example query; extension must be enabled.

-- (In a real environment, enable extension and run periodically)

CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

-- Top queries by total_time

SELECT queryid, query, calls, total_time, mean_time, rows

FROM pg_stat_statements

ORDER BY total_time DESC

LIMIT 10;

-- Explain analyze template: show how to capture query plan metadata (demonstration only)

-- EXPLAIN (ANALYZE, BUFFERS, VERBOSE) SELECT * FROM prod_analytics.events WHERE event_time > now() - INTERVAL '1 day';

-- 12) Advanced indexing: partial, expression, bloom, GIN for JSONB & full-text

---

-- partial index for active users with recent activity

CREATE INDEX IF NOT EXISTS idx_users_active_recent ON prod_analytics.users (last_login) WHERE is_active = TRUE;

-- expression index for lower(username) lookups

CREATE INDEX IF NOT EXISTS idx_users_username_lower ON prod_analytics.users ((lower(username)));

-- GIN for JSONB path existence/containment

CREATE INDEX IF NOT EXISTS idx_events_props_gin ON prod_analytics.events USING GIN (event_props jsonb_path_ops);

-- trigram index for fuzzy match (pg_trgm extension)

CREATE EXTENSION IF NOT EXISTS pg_trgm;

CREATE INDEX IF NOT EXISTS idx_kb_title_trgm ON prod_analytics.knowledge_base USING gin (title gin_trgm_ops);

-- BRIN index already created for event_date earlier - cheap for large time series tables.

-- 13) Data quality checks (DQ) & assertions: example SQL tests

---

-- 13.1 Referential integrity check: events.user_id that have no user in users table

SELECT e.user_id, count(*) AS orphan_event_count

FROM prod_analytics.events e

LEFT JOIN prod_analytics.users u ON u.user_id = e.user_id

WHERE u.user_id IS NULL

GROUP BY e.user_id

ORDER BY orphan_event_count DESC

LIMIT 50;

-- 13.2 Schema validation: expected keys present in event_props for 'purchase' events

```
SELECT
  count(*) FILTER (WHERE (event_props ? 'order_id') IS NOT TRUE) AS missing_order_id,
  count(*) FILTER (WHERE (event_props ? 'price') IS NOT TRUE) AS missing_price,
  count(*) AS total_purchase_events
FROM prod_analytics.events
WHERE event_type = 'purchase';


-------------------------------------------------------------------------------
-- 14) Example complex Business Intelligence query: funnel conversion & attribution
-------------------------------------------------------------------------------
-- Attribution: count unique users who viewed product -> added_to_cart -> purchased within 7 days
WITH views AS (
  SELECT user_id, event_time, (event_props->>'product_id') AS product_id
  FROM prod_analytics.events WHERE event_type = 'product_view'
),
adds AS (
  SELECT user_id, event_time, (event_props->>'product_id') AS product_id
  FROM prod_analytics.events WHERE event_type = 'add_to_cart'
),
purchases AS (
  SELECT user_id, event_time, (event_props->>'product_id') AS product_id
  FROM prod_analytics.events WHERE event_type = 'purchase'
)
SELECT
```

```
  v.product_id,
  COUNT(DISTINCT v.user_id) AS viewers,
  COUNT(DISTINCT a.user_id) AS adders,
  COUNT(DISTINCT p.user_id) AS purchasers,
  ROUND(100.0 * COUNT(DISTINCT p.user_id) / NULLIF(COUNT(DISTINCT v.user_id),0), 2) AS conv_rate_from_view
FROM views v
LEFT JOIN adds a ON a.user_id = v.user_id AND a.product_id = v.product_id AND a.event_time BETWEEN v.event_time AND v.event_time + INTERVAL '7 days'
LEFT JOIN purchases p ON p.user_id = v.user_id AND p.product_id = v.product_id AND p.event_time BETWEEN v.event_time AND v.event_time + INTERVAL '7 days'
GROUP BY v.product_id
ORDER BY conv_rate_from_view DESC NULLS LAST
LIMIT 100;


-------------------------------------------------------------------------------
-- 15) Security-conscious examples: RBAC hints (showcase understanding of permissions)
-------------------------------------------------------------------------------
-- create a read-only role for analytics consumers
DO $$
BEGIN
  CREATE ROLE analytics_reader NOLOGIN;
EXCEPTION WHEN duplicate_object THEN
  -- ignore
END $$;
```

```
GRANT USAGE ON SCHEMA prod_analytics TO analytics_reader;

GRANT SELECT ON ALL TABLES IN SCHEMA prod_analytics TO analytics_reader;

ALTER DEFAULT PRIVILEGES IN SCHEMA prod_analytics GRANT SELECT ON
TABLES TO analytics_reader;


-------------------------------------------------------------------------------

-- 16) Example show-casing schema evolution & migrations (safe ALTER patterns)

-------------------------------------------------------------------------------

-- Add a new column with minimal locking (add column with NULL and then backfill in
batches)

ALTER TABLE prod_analytics.events ADD COLUMN IF NOT EXISTS device_type TEXT;

-- Backfill in small batches (example pattern) to avoid long locks:

-- UPDATE prod_analytics.events SET device_type = (event_props->>'device') WHERE
device_type IS NULL LIMIT 10000;


-------------------------------------------------------------------------------

-- 17) Demonstration of advanced joins: anti-join, semi-join, lateral correlated subquery

-------------------------------------------------------------------------------

-- Find users who never purchased (anti-join)

SELECT u.user_id, u.email, u.signup_ts

FROM prod_analytics.users u

LEFT JOIN prod_analytics.events e ON e.user_id = u.user_id AND e.event_type = 'purchase'

WHERE e.event_id IS NULL

LIMIT 100;


-- Semi-join: users who have at least one 'product_view'
```

```
SELECT u.user_id, u.email
FROM prod_analytics.users u
WHERE EXISTS (
  SELECT 1 FROM prod_analytics.events e WHERE e.user_id = u.user_id AND e.event_type = 'product_view'
)
LIMIT 100;

-------------------------------------------------------------------------------
-- 18) Example: approximate distinct counts for cardinality estimation (hyperloglog via extension)
-------------------------------------------------------------------------------
CREATE EXTENSION IF NOT EXISTS postgresql_hll; -- if installed in environment
-- pseudo usage (may not be available in all envs): hll_add_agg & hll_cardinality
-- SELECT org_id, hll_cardinality(hll_add_agg(hll_hash_integer(user_id))) FROM prod_analytics.events GROUP BY org_id;

-------------------------------------------------------------------------------
-- 19) Sample test data insertion (kept minimal here; in a real test, swap with generated dataset)
-------------------------------------------------------------------------------
INSERT INTO prod_analytics.organizations (org_name, industry)
VALUES ('Acme Corp', 'ecommerce'), ('Globex', 'cloud');

INSERT INTO prod_analytics.users (org_id, email, username, attrs)
SELECT o.org_id, format('user%s@example.com', gs), format('user%s', gs), jsonb_build_object('signup_source','email')
FROM generate_series(1,100) gs
```

```
CROSS JOIN LATERAL (SELECT org_id FROM prod_analytics.organizations ORDER BY random() LIMIT 1) o;


-- insert synthetic events (small sample)

INSERT INTO prod_analytics.events (user_id, org_id, event_type, event_props, event_time)

SELECT u.user_id, u.org_id,

    CASE WHEN random() < 0.02 THEN 'purchase' WHEN random() < 0.2 THEN 'add_to_cart' ELSE 'product_view' END,

    jsonb_build_object('price', round(random()*100,2), 'product_id', (1000 + (random()*100)::int)::text, 'device', CASE WHEN random() < 0.5 THEN 'mobile' ELSE 'desktop' END),

    now() - (random() * interval '30 days')

FROM prod_analytics.users u

ORDER BY random()

LIMIT 5000;


-------------------------------------------------------------------------------

-- 20) Example README-style queries to demonstrate you understand trade-offs

-------------------------------------------------------------------------------

-- EXPLAIN (ANALYZE, BUFFERS, VERBOSE) <complex_query_here>;

-- For very large tables: prefer BRIN indexes, partition-pruning, and clustered order for HOT updates.

-- Use VACUUM and ANALYZE regularly; use autovacuum tuning for high-throughput ingestion.

-- For JSONB heavy workloads, GIN jsonb_path_ops is superior for key-existence queries; use expression indexes for frequent lookups.


-------------------------------------------------------------------------------
```

-- 21) Cleanup helper (safe drops for exercise / demos)

-- DROP SCHEMA prod_analytics CASCADE; -- uncomment to teardown demo

---

-- End of script: highlights & keywords for HR copy/paste

-- HIGHLIGHTS:

-- * Partitioning (RANGE by date) for high throughput time-series data

-- * Indexing strategies: BTREE, BRIN, GIN (JSONB & full-text), trigram for fuzzy search

-- * Advanced SQL constructs: window functions, recursive CTEs, lateral joins

-- * Idempotent ETL patterns (staging + upsert ON CONFLICT)

-- * Materialized views + concurrent refresh for expensive aggregates

-- * PL/pgSQL ETL job examples with advisory locks for single-run concurrency

-- * Observability: pg_stat_statements example and explain analyze pattern

-- Keywords: data engineering, partitioning, query optimization, window functions, JSONB, full-text search, upsert, CDC, ETL, materialized view, indexing, BRIN, GIN, PL/pgSQL, concurrency.