STUDENT 13: RESTAURANT ORDER & BILLING

This README explains all SQL scripts from A1 through B10 with actual code and explanations.

A1: Fragment & Recombine Main Fact (≤10 rows)

It split order details across two database locations and bring them back together.

Code & Explanations:

sql

```
CREATE SCHEMA IF NOT EXISTS node_a;
CREATE SCHEMA IF NOT EXISTS node_b;
```

Creates two separate storage areas (schemas) to hold fragmented data.

sql

```
CREATE TABLE IF NOT EXISTS node_a.orderdetail_a (
  detailid BIGINT PRIMARY KEY,
  orderid  BIGINT NOT NULL,
  menuid   BIGINT NOT NULL,
  quantity INT    NOT NULL CHECK (quantity > 0),
  subtotal NUMERIC(12,2) NOT NULL CHECK (subtotal >= 0)
);
```

Creates first fragment table with columns for order details and validation rules ensuring positive quantities and non-negative subtotals.

sql

INSERT INTO node_a.orderdetail_a VALUES (1,101,11,2,5000), (2,101,15,1,1500), (3,102,12,1,3500);

INSERT INTO node_b.orderdetail_b VALUES (4,103,13,3,8400), (5,103,18,2,1600), (6,104,14,1,3000);

Inserts 3 rows into each fragment, splitting 6 total rows across two locations.

sql

CREATE EXTENSION IF NOT EXISTS postgres_fdw;

CREATE SERVER proj_link FOREIGN DATA WRAPPER postgres_fdw OPTIONS (host 'localhost', dbname 'restaurantdb', port '5432');

Sets up foreign data wrapper extension and creates database link named 'proj_link' to access remote data.

sql

CREATE FOREIGN TABLE node_a.orderdetail_b_ft (...) SERVER proj_link OPTIONS (schema_name 'node_b', table_name 'orderdetail_b');

Creates foreign table that acts as a window to view remote node_b data from node_a.

sql

CREATE OR REPLACE VIEW node_a.orderdetail_all AS

SELECT * FROM node_a.orderdetail_a UNION ALL SELECT * FROM node_a.orderdetail_b_ft;

Creates unified view combining both fragments using UNION ALL to stack all rows together.

sql

SELECT 'Fragment A' AS source, (SELECT COUNT(*) FROM node_a.orderdetail_a) AS row_count

UNION ALL SELECT 'Fragment B (via FT)', (SELECT COUNT(*) FROM node_a.orderdetail_b_ft);

Validates row counts from both fragments to ensure no data loss during fragmentation.

A2: Database Link & Cross-Node Join (3–10 rows result)

It connect to remote data and combine information from different locations.

sql

SELECT srvname AS server_name, fdwname AS wrapper_name FROM pg_foreign_server fs

JOIN pg_foreign_data_wrapper fdw ON fs.srvfdw = fdw.oid WHERE srvname = 'proj_link';

It verifies the database link 'proj_link' exists and shows its configuration details.

sql

SELECT * FROM node_a.orderdetail_b_ft ORDER BY detailid FETCH FIRST 5 ROWS ONLY;

It fetches up to 5 sample rows from the remote node_b table through the foreign table.

sql

SELECT COALESCE(a.orderid, b.orderid) AS orderid, a.detailid AS detailid_a, b.detailid AS detailid_b

FROM node_a.orderdetail_a a FULL OUTER JOIN node_a.orderdetail_b_ft b ON a.orderid = b.orderid

WHERE COALESCE(a.orderid, b.orderid) IN (101,102,103,104);

It performs distributed join combining local and remote data, showing side-by-side comparison of matching orders using FULL OUTER JOIN.

A3: Parallel vs Serial Aggregation (≤10 rows data)

Compare processing speed using one worker versus multiple workers.

sql

```
SET max_parallel_workers_per_gather = 0;
SELECT menuid, SUM(quantity) AS total_qty, SUM(subtotal) AS total_amount
FROM node_a.orderdetail_all GROUP BY menuid ORDER BY menuid;
```

It disables parallel processing and runs aggregation with single worker, grouping order details by menu item.

sql

```
SET max_parallel_workers_per_gather = 2;
SELECT menuid, SUM(quantity) AS total_qty, SUM(subtotal) AS total_amount
FROM node_a.orderdetail_all GROUP BY menuid ORDER BY menuid;
```

Enables parallel processing with 2 workers to split work and potentially speed up the same aggregation query.

sql

```
EXPLAIN (ANALYZE, BUFFERS, VERBOSE) SELECT menuid, SUM(quantity) AS total_qty
FROM node_a.orderdetail_all GROUP BY menuid;
```

Shows detailed execution plan with timing information to compare serial versus parallel performance.

sql

```
SELECT 'Serial' AS execution_mode, (SELECT COUNT(*) FROM node_a.orderdetail_all) AS total_rows_processed
UNION ALL SELECT 'Parallel', (SELECT COUNT(*) FROM node_a.orderdetail_all);
```

It creates 2-row comparison table showing rows processed in serial versus parallel modes.

A4: Two-Phase Commit & Recovery (2 rows)

It safely save data to multiple locations with ability to undo if something fails.

sql

```
BEGIN;
INSERT INTO node_a.orderdetail_a VALUES (7,105,11,1,2500);
INSERT INTO node_a.orderdetail_b_ft VALUES (8,105,15,1,1500);
PREPARE TRANSACTION 'tx_restaurant_1';
```

It starts transaction, inserts one row locally and one remotely, then prepares transaction for two-phase commit without finalizing yet.

sql

```
SELECT gid AS transaction_id, prepared AS prepared_at, owner AS owner_user, database AS db_name
FROM pg_prepared_xacts WHERE gid = 'tx_restaurant_1';
```

Queries prepare transactions table to show pending transaction waiting to be committed or rolled back.

sql

COMMIT PREPARED 'tx_restaurant_1';

It finalizes the prepared transaction, permanently saving both local and remote changes.

sql

BEGIN; INSERT INTO node_a.orderdetail_a VALUES (9,106,12,1,3500);

PREPARE TRANSACTION 'tx_restaurant_2'; ROLLBACK PREPARED 'tx_restaurant_2';

It demonstrates rollback scenario: prepares a transaction then undoes it, discarding all changes.

A5: Distributed Lock Conflict & Diagnosis (no extra rows)

It shows what happens when two users try to modify the same data simultaneously.

sql

BEGIN; UPDATE node_a.orderdetail_a SET quantity = quantity + 1 WHERE detailid = 1;

The session 1 starts transaction and locks a row by updating it, keeping transaction open without committing.

sql

UPDATE node_a.orderdetail_b_ft SET quantity = quantity + 1 WHERE orderid = 101;

The session 2 tries to update related data but will wait because Session 1 holds a lock.

sql

SELECT pid, state, wait_event_type, wait_event, NOW() - query_start AS waiting_duration

FROM pg_stat_activity WHERE state = 'active' AND wait_event IS NOT NULL;

It shows which database processes are waiting for locks and how long they've been waiting.

sql

```
SELECT pg_blocking_pids(pid) AS blocking_pids, pid AS waiting_pid, state, query FROM pg_stat_activity WHERE state = 'active';
```

Identifies which process is blocking others, helping diagnose lock conflicts.

sql

```
SELECT locktype, mode, granted, relation::regclass AS table_name, pid FROM pg_locks WHERE NOT granted;
```

Shows detailed lock information including which locks are waiting (not granted) and which table they're trying to access.

B6: Declarative Rules Hardening (≤10 committed rows)

It adds safety rules to prevent invalid data from being saved.

sql

```
ALTER TABLE node_a.orderdetail_a ALTER COLUMN orderid SET NOT NULL,
ADD CONSTRAINT chk_a_amount CHECK (quantity > 0 AND subtotal >= 0);
```

It makes orderid required and adds check constraint ensuring quantity is positive and subtotal is non-negative.

sql

```
BEGIN; INSERT INTO node_a.orderdetail_a VALUES (10, NULL, 11, 1, 2500);
```

EXCEPTION WHEN not_null_violation THEN RAISE NOTICE 'Expected error: NOT NULL constraint violation'; ROLLBACK;

Tests constraint by attempting to insert NULL orderid, catches the error, and rolls back to prove bad data was rejected.

sql

UPDATE node_a.orderdetail_a SET subtotal = 5200 WHERE detailid = 1;

It demonstrates successful update that satisfies all constraints (positive quantity, non-negative subtotal, non-null values).

sql

SELECT detailid, orderid, quantity, subtotal,

CASE WHEN orderid IS NULL OR quantity <= 0 OR subtotal < 0 THEN 'INVALID' ELSE 'VALID' END AS validation_status

FROM node_a.orderdetail_a;

It validates all rows against constraints and marks each as VALID or INVALID.

B7: E–C–A Trigger for Denormalized Totals (small DML set)

It automatically track changes to order totals and log them for auditing.

sql

CREATE TABLE IF NOT EXISTS node_a.orderinfo_audit (

  bef_total NUMERIC(12,2), aft_total NUMERIC(12,2), changed_at TIMESTAMP DEFAULT now(), key_col TEXT

);

Creates audit table to store before/after totals with timestamp for tracking changes over time.

sql

```
CREATE OR REPLACE FUNCTION node_a.fn_retotal_order() RETURNS TRIGGER AS $$
BEGIN
  SELECT COALESCE(SUM(subtotal),0) INTO v_bef FROM node_a.orderdetail_all WHERE orderid = v_orderids[1];
  INSERT INTO node_a.orderinfo_audit(bef_total, aft_total, key_col) VALUES (v_bef, v_aft, 'orderid='||v_orderids[1]);
  RETURN NULL;
END; $$ LANGUAGE plpgsql;
```

It creates trigger function that calculates order totals before and after changes, then logs them to audit table.

sql

```
CREATE TRIGGER trg_retotal_order AFTER INSERT OR UPDATE OR DELETE ON node_a.orderdetail_a
DEFERRABLE INITIALLY DEFERRED FOR EACH STATEMENT EXECUTE FUNCTION node_a.fn_retotal_order();
```

It creates statement-level trigger that runs after any data modification, deferred until end of transaction.

sql

```
UPDATE node_a.orderdetail_a SET quantity = quantity + 1, subtotal = subtotal + 2500 WHERE detailid IN (1,2);
```

Executes mixed DML updating 2 rows, which automatically triggers audit logging.

sql

SELECT bef_total, aft_total, changed_at, key_col, aft_total - bef_total AS difference

FROM node_a.orderinfo_audit ORDER BY changed_at DESC LIMIT 5;

It shows audit log entries with before/after totals and calculated difference to track changes.

B8: Recursive Hierarchy Roll-Up

It navigates parent-child relationships to find all descendants in a tree structure.

sql

CREATE TEMP TABLE hier(parent_id INT, child_id INT);

INSERT INTO hier VALUES (1,2), (1,3), (2,4), (2,5), (3,6), (3,7);

It creates temporary hierarchy table and inserts 6 rows forming 3-level tree: root (1) has children (2,3), which have their own children.

sql

WITH RECURSIVE tree AS (

  SELECT child_id, parent_id, parent_id AS root_id, 1 AS depth FROM hier WHERE parent_id = 1

  UNION ALL

  SELECT h.child_id, h.parent_id, t.root_id, t.depth + 1 FROM hier h JOIN tree t ON h.parent_id = t.child_id

)

SELECT child_id, root_id, depth FROM tree ORDER BY root_id, depth, child_id;

It uses recursive CTE to traverse entire tree: starts with direct children of root, then recursively finds children of children, tracking depth level.

sql

```
SELECT depth, COUNT(*) AS nodes_at_depth, STRING_AGG(child_id::TEXT, ', ' ORDER BY child_id) AS children_list
FROM tree GROUP BY depth ORDER BY depth;
```

It groups results by depth level showing how many nodes exist at each level and listing all children.

B9: Mini-Knowledge Base with Transitive Inference (≤10 facts)

Used to store facts about food items and automatically infer indirect relationships.

sql

```
CREATE TEMP TABLE triple(s TEXT, p TEXT, o TEXT);
INSERT INTO triple VALUES ('Isombe','isA','Traditional'), ('Traditional','isA','Dish'), ('Dish','isA','Food');
```

Creates knowledge base table storing facts as subject-predicate-object triples, forming chains like "Isombe is a Traditional is a Dish is a Food".

sql

```
WITH RECURSIVE isa(s,o) AS (
  SELECT s, o FROM triple WHERE p='isA'
  UNION
  SELECT i.s, t.o FROM isa i JOIN triple t ON i.o = t.s AND t.p='isA'
)
SELECT DISTINCT s AS entity, o AS label FROM isa WHERE o IN ('Dish','Food','Item');
```

Uses recursive CTE to infer transitive relationships: if "Isombe isA Traditional" and "Traditional isA Dish", then infers "Isombe isA Dish".

sql

```
SELECT o AS label, COUNT(DISTINCT s) AS num_entities, STRING_AGG(DISTINCT s, ', ') AS entities_list
FROM isa WHERE o IN ('Dish','Food','Item') GROUP BY o;
```

It groups inferred relationships by label showing how many entities have each label and listing them.

B10: Business Limit Alert (Function + Trigger) (row-budget safe)

Used to enforce business rules by preventing orders that exceed item limits.

sql

```
CREATE TABLE IF NOT EXISTS node_a.business_limits(rule_key TEXT PRIMARY KEY, threshold INT NOT NULL, active CHAR(1) CHECK (active IN ('Y','N')));
INSERT INTO node_a.business_limits VALUES ('max_items_per_order', 5, 'Y');
```

It creates business rules table and inserts one active rule: maximum 5 items per order.

sql

```
CREATE OR REPLACE FUNCTION node_a.fn_should_alert(p_orderid BIGINT) RETURNS INT AS $$
BEGIN
  SELECT threshold INTO lim FROM node_a.business_limits WHERE rule_key='max_items_per_order' AND active='Y';
  SELECT COALESCE(SUM(quantity),0) INTO tot FROM node_a.orderdetail_a WHERE orderid=p_orderid;
```

```
  IF lim IS NOT NULL AND tot > lim THEN RETURN 1; END IF; RETURN 0;
END; $$ LANGUAGE plpgsql;
```

Creates function that reads active threshold from business_limits table, counts total items for an order, returns 1 if limit exceeded.

sql

```
CREATE OR REPLACE FUNCTION node_a.fn_enforce_limit() RETURNS TRIGGER AS $$
BEGIN
  IF node_a.fn_should_alert(COALESCE(NEW.orderid, OLD.orderid)) = 1 THEN
    RAISE EXCEPTION 'Business limit exceeded for order %', COALESCE(NEW.orderid, OLD.orderid);
  END IF; RETURN COALESCE(NEW, OLD);
END; $$ LANGUAGE plpgsql;
```

It creates trigger function that calls fn_should_alert and raises error if limit would be exceeded, preventing the change.

sql

```
CREATE TRIGGER trg_enforce_limit BEFORE INSERT OR UPDATE ON node_a.orderdetail_a
FOR EACH ROW EXECUTE FUNCTION node_a.fn_enforce_limit();
```

It creates BEFORE trigger that runs for each row being inserted or updated, enforcing business limit before data is saved.

sql

```
BEGIN; UPDATE node_a.orderdetail_a SET quantity = 10 WHERE detailid = 2;
EXCEPTION WHEN others THEN RAISE NOTICE 'Expected error caught'; ROLLBACK;
```

It tests enforcement by attempting to set quantity to 10 (exceeds limit of 5), catches error, and rolls back.

sql

```
UPDATE node_a.orderdetail_a SET quantity = 3 WHERE detailid = 1;
```

Demonstrates successful update with quantity = 3 (within limit of 5), which passes validation and commits.