STUDENT 26: INSURANCE CLAIM & POLICY

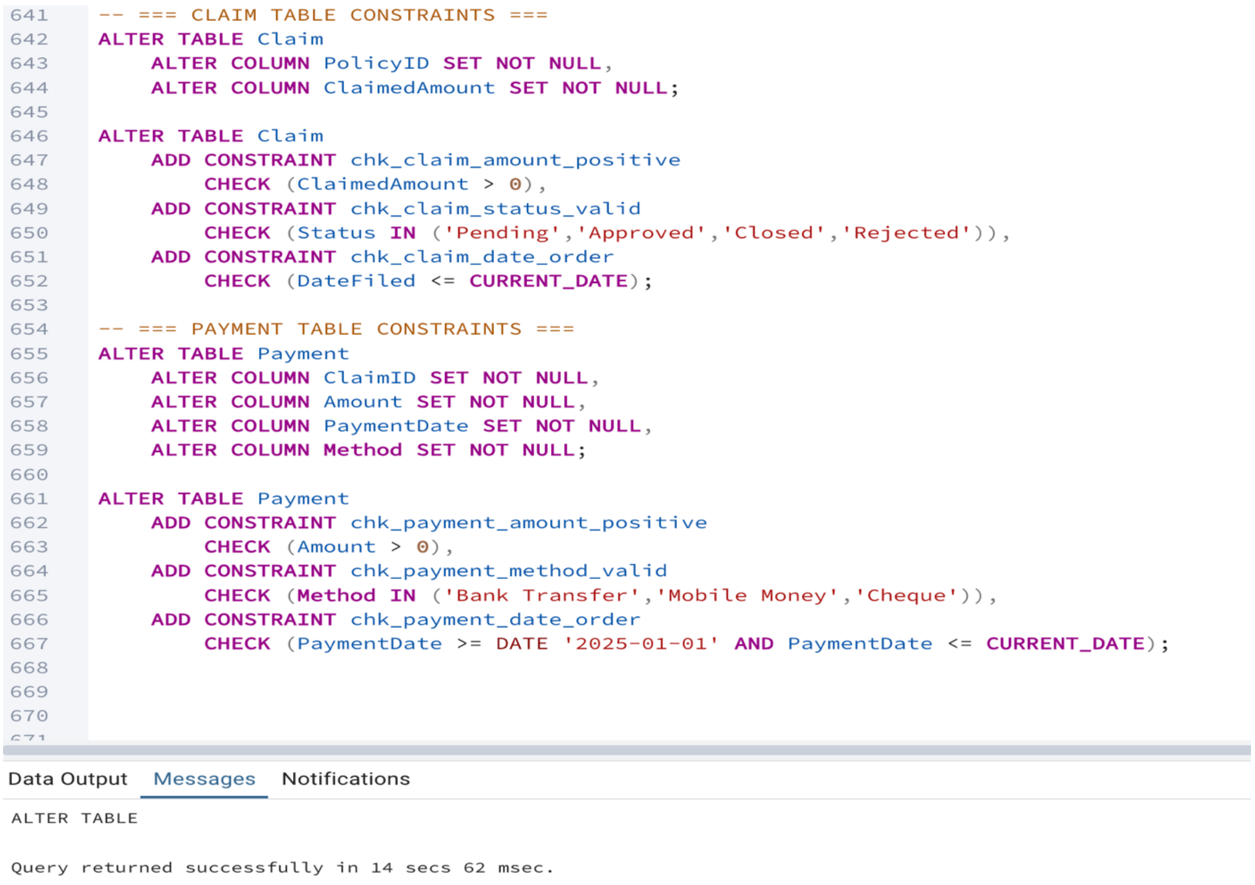
B6: Declarative Rules Hardening (≤10 committed rows)

WHAT TO DO

1. On tables Claim and Payment, add/verify NOT NULL and domain CHECK constraints suitable for claims and approvals (e.g., positive amounts, valid statuses, date order).
2. Prepare 2 failing and 2 passing INSERTs per table to validate rules, but wrap failing ones in a block and ROLLBACK so committed rows stay within ≤10 total.
3. Show clean error handling for failing cases.

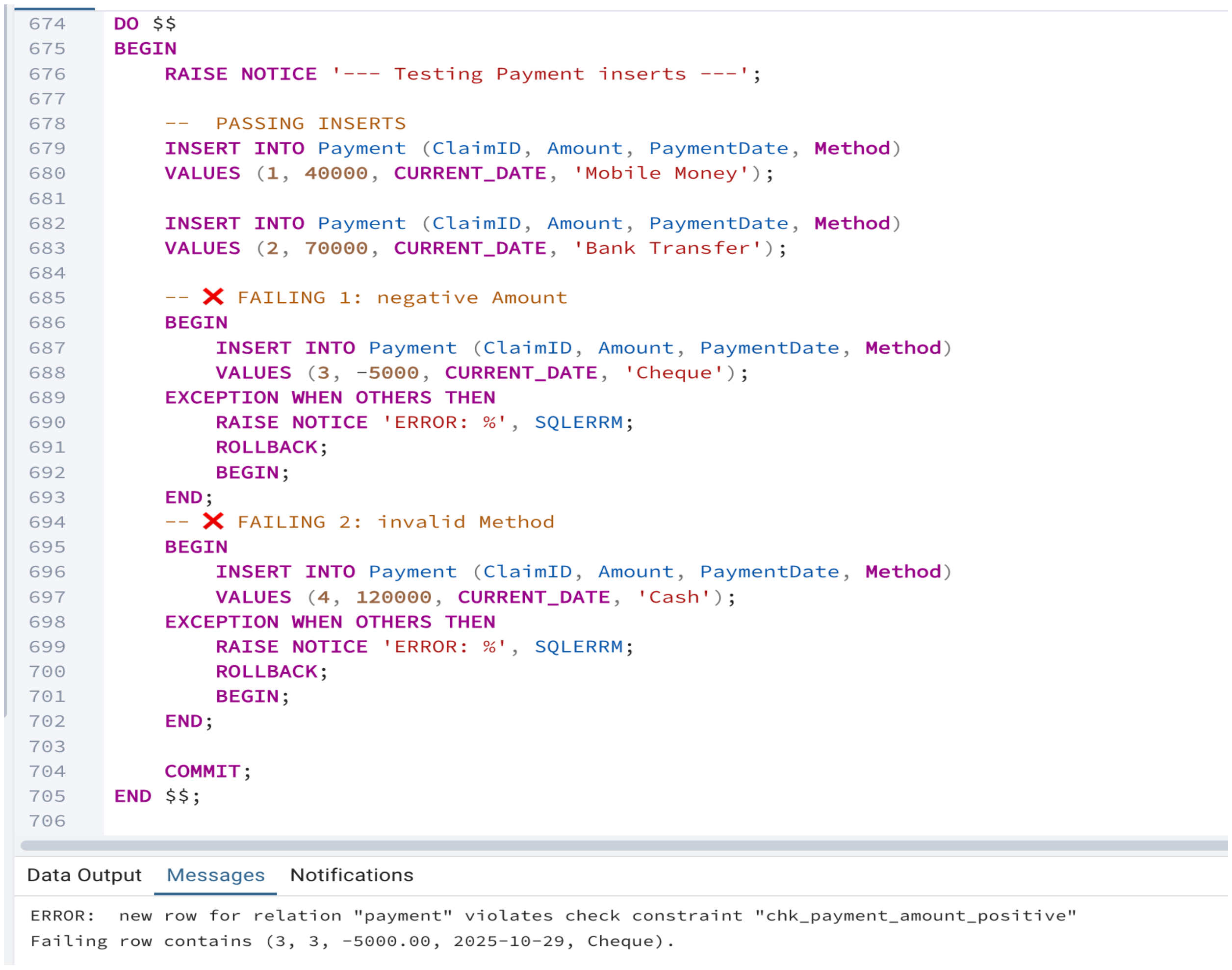
EXPECTED OUTPUT

- ✓ - ALTER TABLE statements for added constraints (named consistently).
- ✓ - Script with test INSERTs and captured ORA- errors for failing cases.
- ✓ - SELECT proof that only the passing rows were committed; total committed rows ≤10.



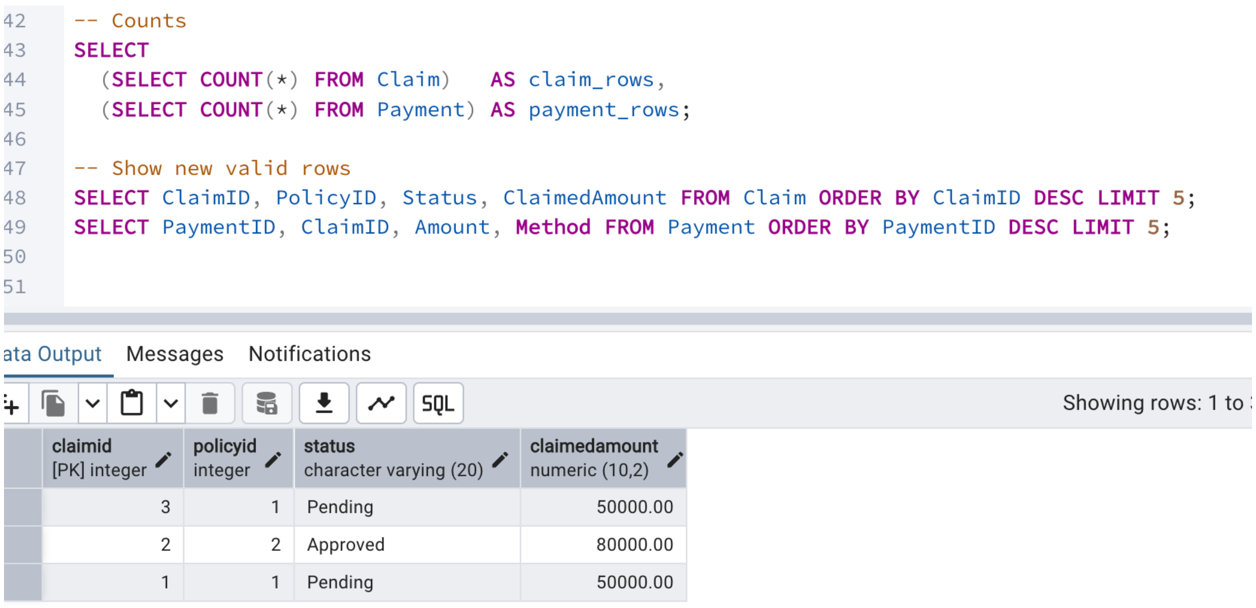
```sql
-- === CLAIM TABLE CONSTRAINTS ===
ALTER TABLE Claim
    ALTER COLUMN PolicyID SET NOT NULL,
    ALTER COLUMN ClaimedAmount SET NOT NULL;

ALTER TABLE Claim
    ADD CONSTRAINT chk_claim_amount_positive
        CHECK (ClaimedAmount > 0),
    ADD CONSTRAINT chk_claim_status_valid
        CHECK (Status IN ('Pending','Approved','Closed','Rejected')),
    ADD CONSTRAINT chk_claim_date_order
        CHECK (DateFiled <= CURRENT_DATE);

-- === PAYMENT TABLE CONSTRAINTS ===
ALTER TABLE Payment
    ALTER COLUMN ClaimID SET NOT NULL,
    ALTER COLUMN Amount SET NOT NULL,
    ALTER COLUMN PaymentDate SET NOT NULL,
    ALTER COLUMN Method SET NOT NULL;

ALTER TABLE Payment
    ADD CONSTRAINT chk_payment_amount_positive
        CHECK (Amount > 0),
    ADD CONSTRAINT chk_payment_method_valid
        CHECK (Method IN ('Bank Transfer','Mobile Money','Cheque')),
    ADD CONSTRAINT chk_payment_date_order
        CHECK (PaymentDate >= DATE '2025-01-01' AND PaymentDate <= CURRENT_DATE);
```

```
ALTER TABLE

Query returned successfully in 14 secs 62 msec.
```



```sql
DO $$
BEGIN
    RAISE NOTICE '--- Testing Payment inserts ---';

    --  PASSING INSERTS
    INSERT INTO Payment (ClaimID, Amount, PaymentDate, Method)
    VALUES (1, 40000, CURRENT_DATE, 'Mobile Money');

    INSERT INTO Payment (ClaimID, Amount, PaymentDate, Method)
    VALUES (2, 70000, CURRENT_DATE, 'Bank Transfer');

    -- ❌ FAILING 1: negative Amount
    BEGIN
        INSERT INTO Payment (ClaimID, Amount, PaymentDate, Method)
        VALUES (3, -5000, CURRENT_DATE, 'Cheque');
    EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'ERROR: %', SQLERRM;
        ROLLBACK;
        BEGIN;
    END;
    -- ❌ FAILING 2: invalid Method
    BEGIN
        INSERT INTO Payment (ClaimID, Amount, PaymentDate, Method)
        VALUES (4, 120000, CURRENT_DATE, 'Cash');
    EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'ERROR: %', SQLERRM;
        ROLLBACK;
        BEGIN;
    END;

    COMMIT;
END $$;
```

```
ERROR:  new row for relation "payment" violates check constraint "chk_payment_amount_positive"
Failing row contains (3, 3, -5000.00, 2025-10-29, Cheque).
```



```sql
-- Counts
SELECT
  (SELECT COUNT(*) FROM Claim)   AS claim_rows,
  (SELECT COUNT(*) FROM Payment) AS payment_rows;

-- Show new valid rows
SELECT ClaimID, PolicyID, Status, ClaimedAmount FROM Claim ORDER BY ClaimID DESC LIMIT 5;
SELECT PaymentID, ClaimID, Amount, Method FROM Payment ORDER BY PaymentID DESC LIMIT 5;
```

claimid [PK] integer	policyid integer	status character varying (20)	claimedamount numeric (10,2)
3	1	Pending	50000.00
2	2	Approved	80000.00
1	1	Pending	50000.00

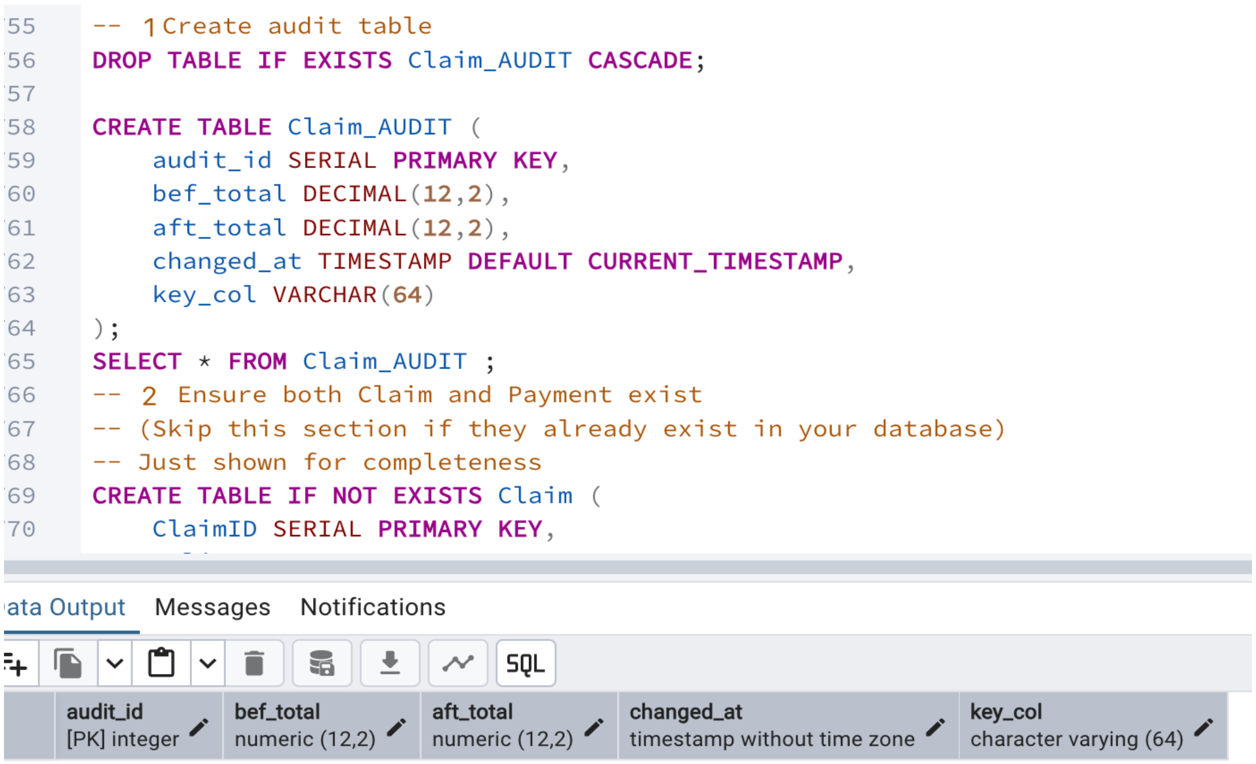
B7 :E–C–A Trigger for Denormalized Totals (small DML set)

WHAT TO DO

1. Create an audit table Claim_AUDIT(bef_total NUMBER, aft_total NUMBER, changed_at TIMESTAMP, key_col VARCHAR2(64)).
2. Implement a statement-level AFTER INSERT/UPDATE/DELETE trigger on Payment that recomputes denormalized totals in Claim once per statement.
3. Execute a small mixed DML script on CHILD affecting at most 4 rows in total; ensure net committed rows across the project remain ≤10.
4. Log before/after totals to the audit table (2–3 audit rows).

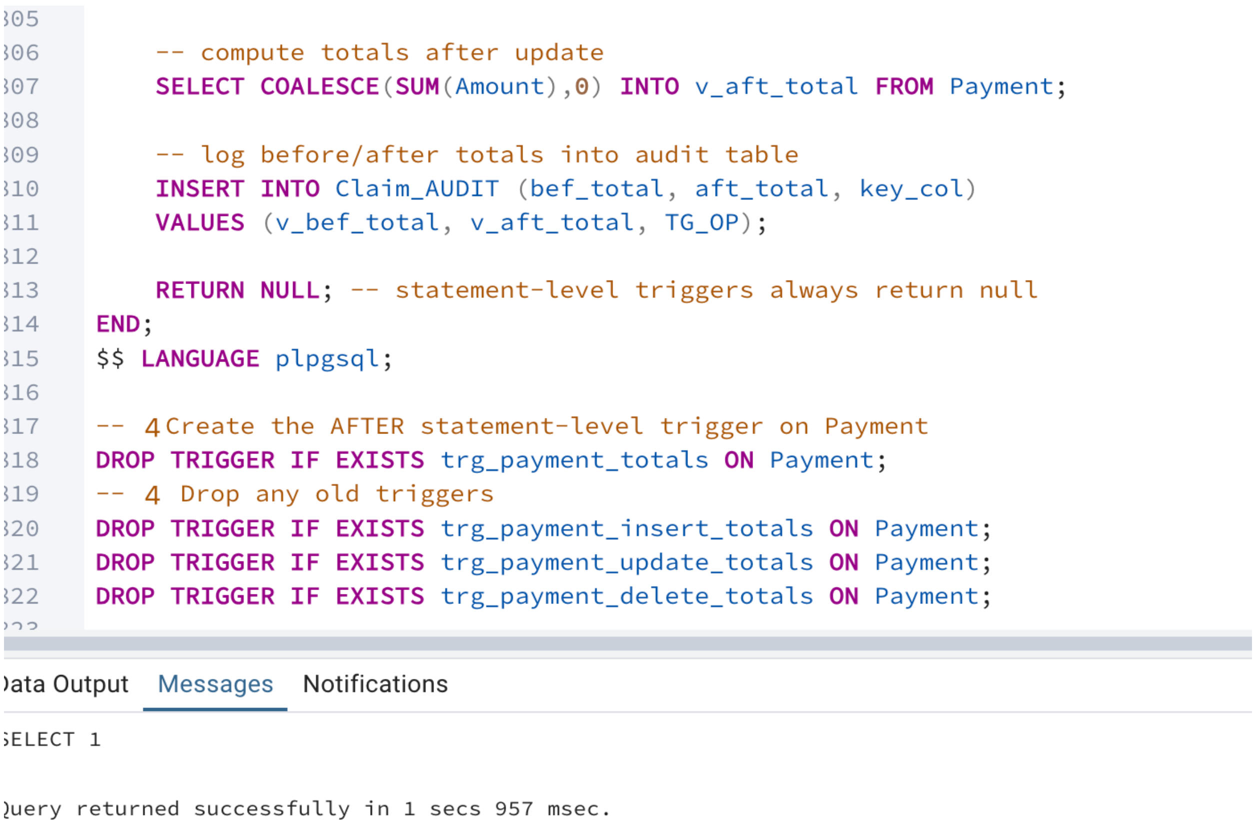
EXPECTED OUTPUT

- ✓ - CREATE TABLE Claim_AUDIT … and CREATE TRIGGER source code.
- ✓ - Mixed DML script and SELECT from totals showing correct recomputation.
- ✓ - SELECT * FROM Claim_AUDIT with 2–3 audit entries.



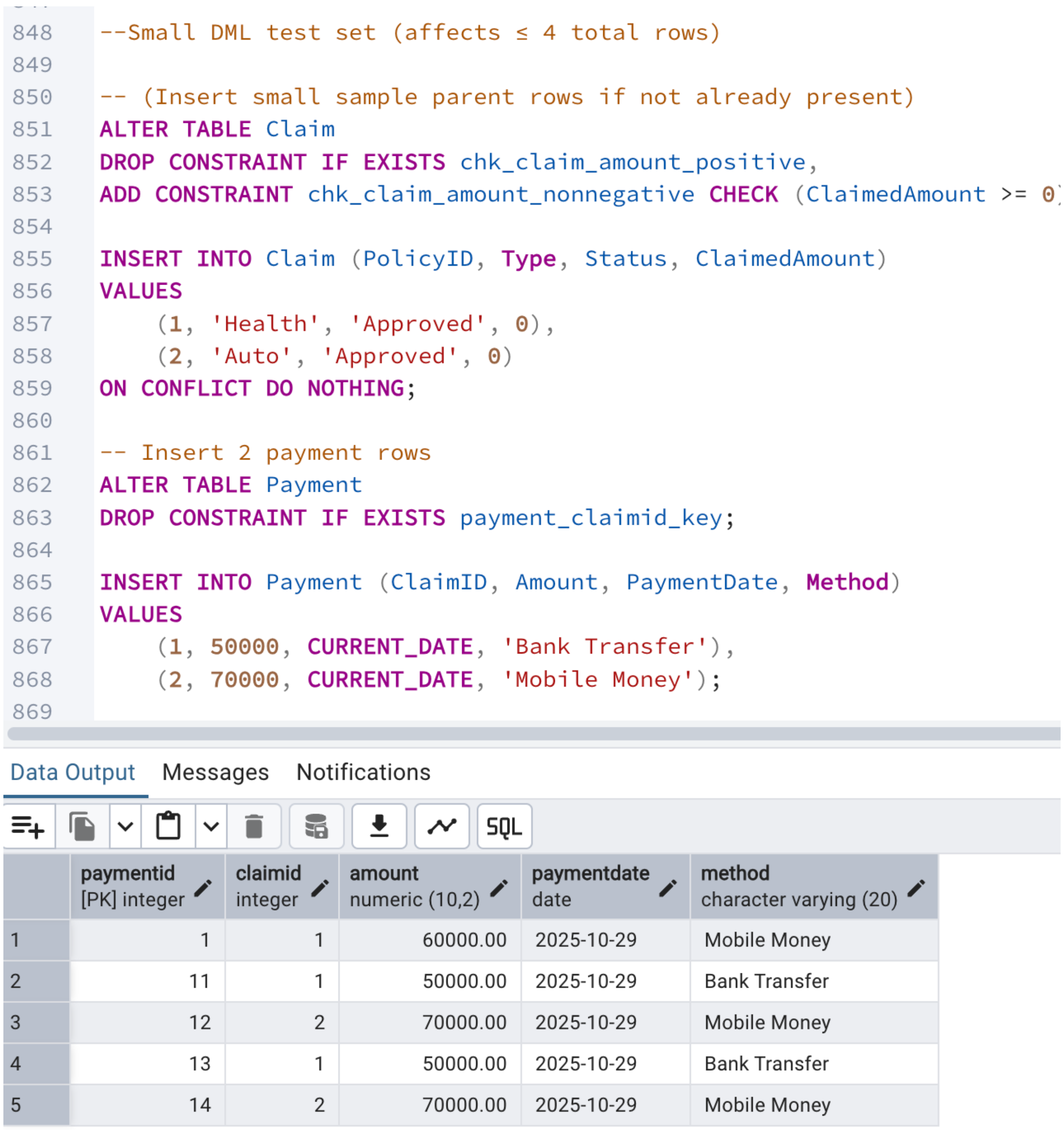
```sql
-- 1Create audit table
DROP TABLE IF EXISTS Claim_AUDIT CASCADE;

CREATE TABLE Claim_AUDIT (
    audit_id SERIAL PRIMARY KEY,
    bef_total DECIMAL(12,2),
    aft_total DECIMAL(12,2),
    changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    key_col VARCHAR(64)
);
SELECT * FROM Claim_AUDIT ;
-- 2 Ensure both Claim and Payment exist
-- (Skip this section if they already exist in your database)
-- Just shown for completeness
CREATE TABLE IF NOT EXISTS Claim (
    ClaimID SERIAL PRIMARY KEY,
```

	audit_id [PK] integer	bef_total numeric (12,2)	aft_total numeric (12,2)	changed_at timestamp without time zone	key_col character varying (64)



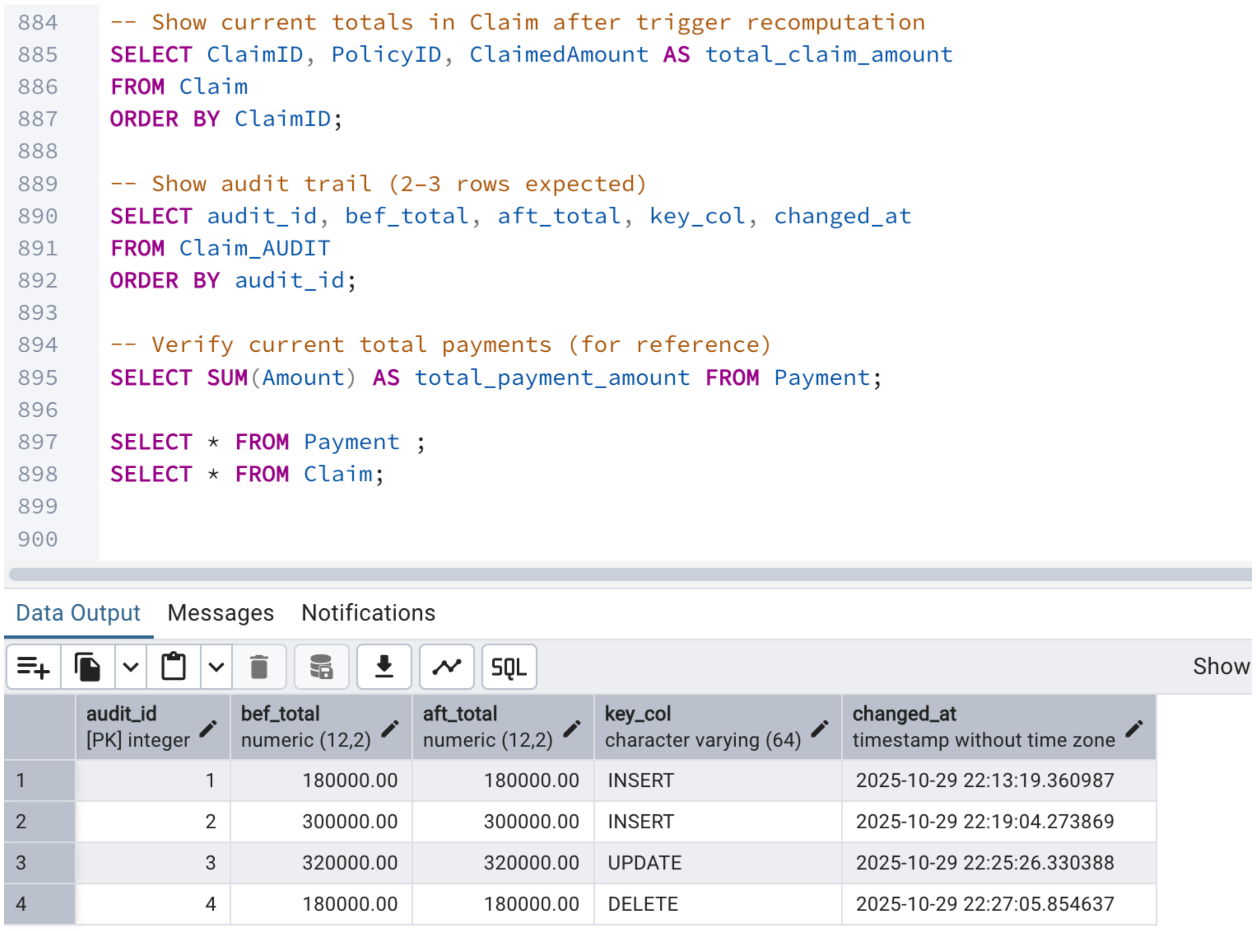
```sql

    -- compute totals after update
    SELECT COALESCE(SUM(Amount),0) INTO v_aft_total FROM Payment;

    -- log before/after totals into audit table
    INSERT INTO Claim_AUDIT (bef_total, aft_total, key_col)
    VALUES (v_bef_total, v_aft_total, TG_OP);

    RETURN NULL; -- statement-level triggers always return null
END;
$$ LANGUAGE plpgsql;

-- 4Create the AFTER statement-level trigger on Payment
DROP TRIGGER IF EXISTS trg_payment_totals ON Payment;
-- 4 Drop any old triggers
DROP TRIGGER IF EXISTS trg_payment_insert_totals ON Payment;
DROP TRIGGER IF EXISTS trg_payment_update_totals ON Payment;
DROP TRIGGER IF EXISTS trg_payment_delete_totals ON Payment;
```

```
;ELECT 1

)uery returned successfully in 1 secs 957 msec.
```



```
--Small DML test set (affects ≤ 4 total rows)

-- (Insert small sample parent rows if not already present)
ALTER TABLE Claim
DROP CONSTRAINT IF EXISTS chk_claim_amount_positive,
ADD CONSTRAINT chk_claim_amount_nonnegative CHECK (ClaimedAmount >= 0)

INSERT INTO Claim (PolicyID, Type, Status, ClaimedAmount)
VALUES
    (1, 'Health', 'Approved', 0),
    (2, 'Auto', 'Approved', 0)
ON CONFLICT DO NOTHING;

-- Insert 2 payment rows
ALTER TABLE Payment
DROP CONSTRAINT IF EXISTS payment_claimid_key;

INSERT INTO Payment (ClaimID, Amount, PaymentDate, Method)
VALUES
    (1, 50000, CURRENT_DATE, 'Bank Transfer'),
    (2, 70000, CURRENT_DATE, 'Mobile Money');
```

	paymentid [PK] integer	claimid integer	amount numeric (10,2)	paymentdate date	method character varying (20)
1	1	1	60000.00	2025-10-29	Mobile Money
2	11	1	50000.00	2025-10-29	Bank Transfer
3	12	2	70000.00	2025-10-29	Mobile Money
4	13	1	50000.00	2025-10-29	Bank Transfer
5	14	2	70000.00	2025-10-29	Mobile Money



```sql
-- Show current totals in Claim after trigger recomputation
SELECT ClaimID, PolicyID, ClaimedAmount AS total_claim_amount
FROM Claim
ORDER BY ClaimID;

-- Show audit trail (2-3 rows expected)
SELECT audit_id, bef_total, aft_total, key_col, changed_at
FROM Claim_AUDIT
ORDER BY audit_id;

-- Verify current total payments (for reference)
SELECT SUM(Amount) AS total_payment_amount FROM Payment;

SELECT * FROM Payment ;
SELECT * FROM Claim;
```

	audit_id [PK] integer	bef_total numeric (12,2)	aft_total numeric (12,2)	key_col character varying (64)	changed_at timestamp without time zone
1	1	180000.00	180000.00	INSERT	2025-10-29 22:13:19.360987
2	2	300000.00	300000.00	INSERT	2025-10-29 22:19:04.273869
3	3	320000.00	320000.00	UPDATE	2025-10-29 22:25:26.330388
4	4	180000.00	180000.00	DELETE	2025-10-29 22:27:05.854637

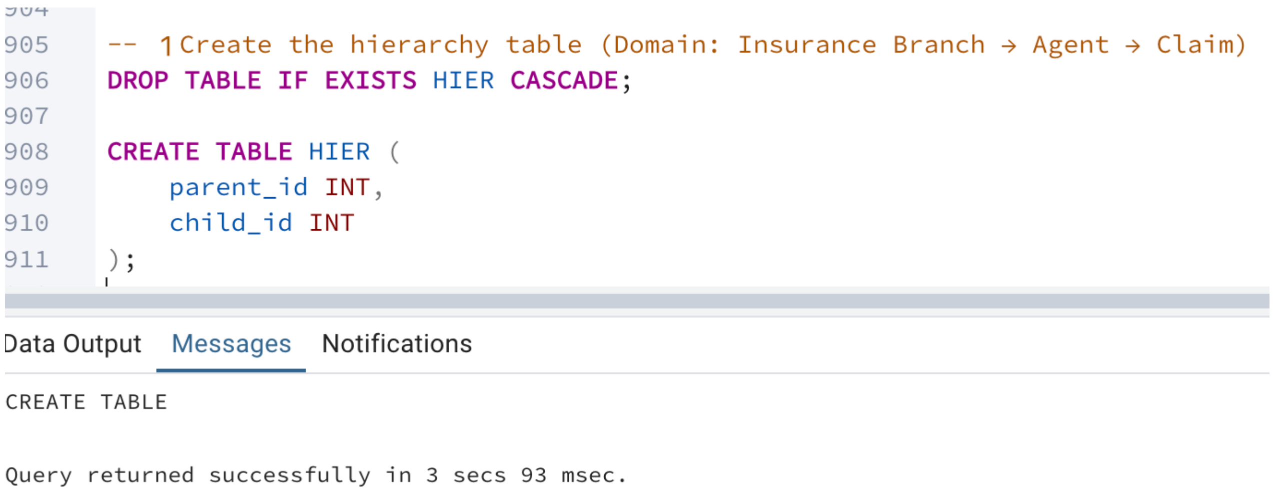
B8 :Recursive Hierarchy Roll-Up (6–10 rows)

WHAT TO DO

1. Create table HIER(parent_id, child_id) for a natural hierarchy (domain-specific).
2. Insert 6–10 rows forming a 3-level hierarchy.
3. Write a recursive WITH query to produce (child_id, root_id, depth) and join to Claim or its parent to compute rollups; return 6–10 rows total.
4. Reuse existing seed rows; do not exceed the ≤10 committed rows budget.

EXPECTED OUTPUT

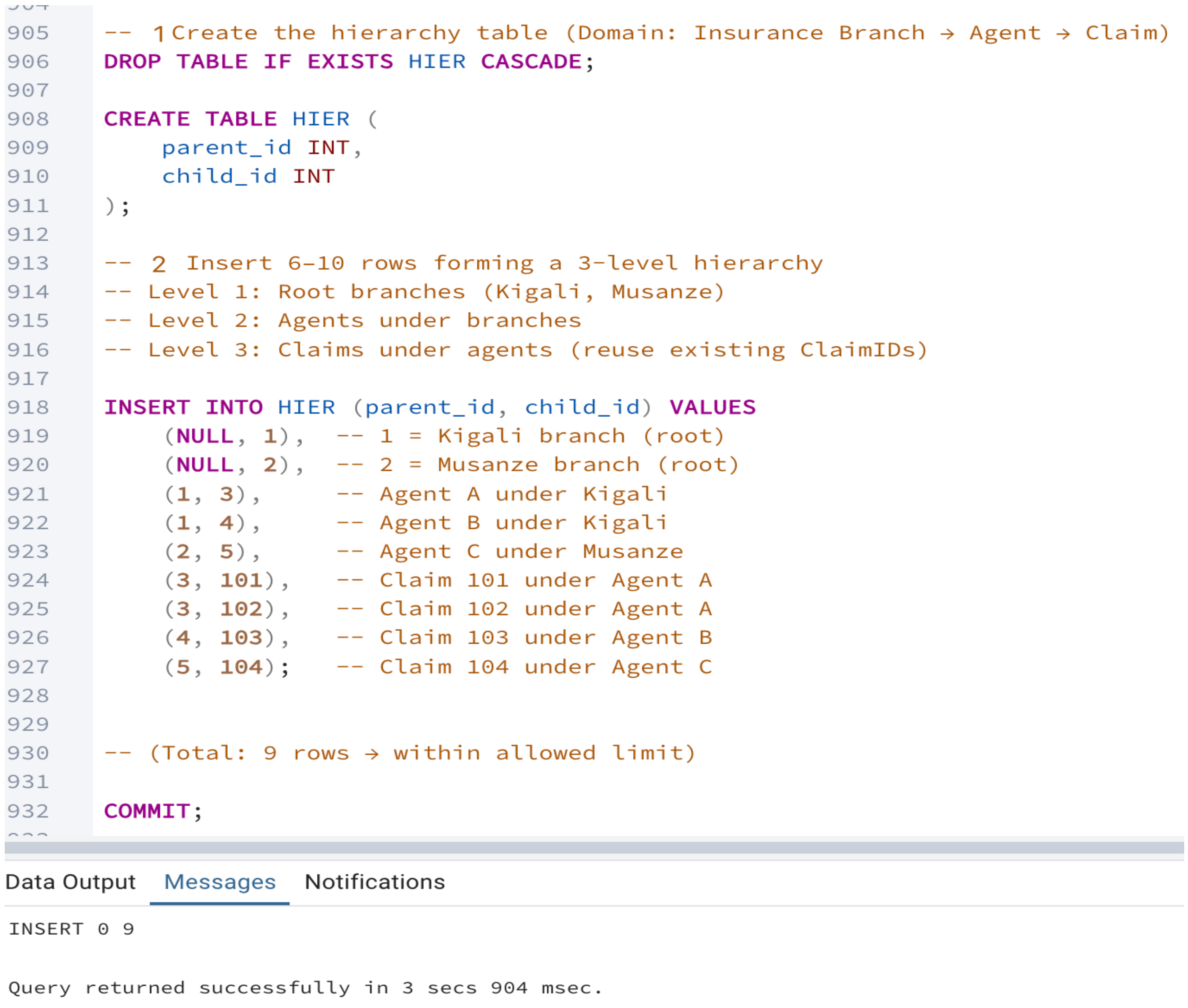
- ✓ - DDL + INSERTs for HIER (6–10 rows).
- ✓ - Recursive WITH SQL and sample output rows (6–10).
- ✓ - Control aggregation validating rollup correctness.



```sql
-- 1Create the hierarchy table (Domain: Insurance Branch → Agent → Claim)
DROP TABLE IF EXISTS HIER CASCADE;

CREATE TABLE HIER (
    parent_id INT,
    child_id INT
);
```

Data Output   Messages   Notifications

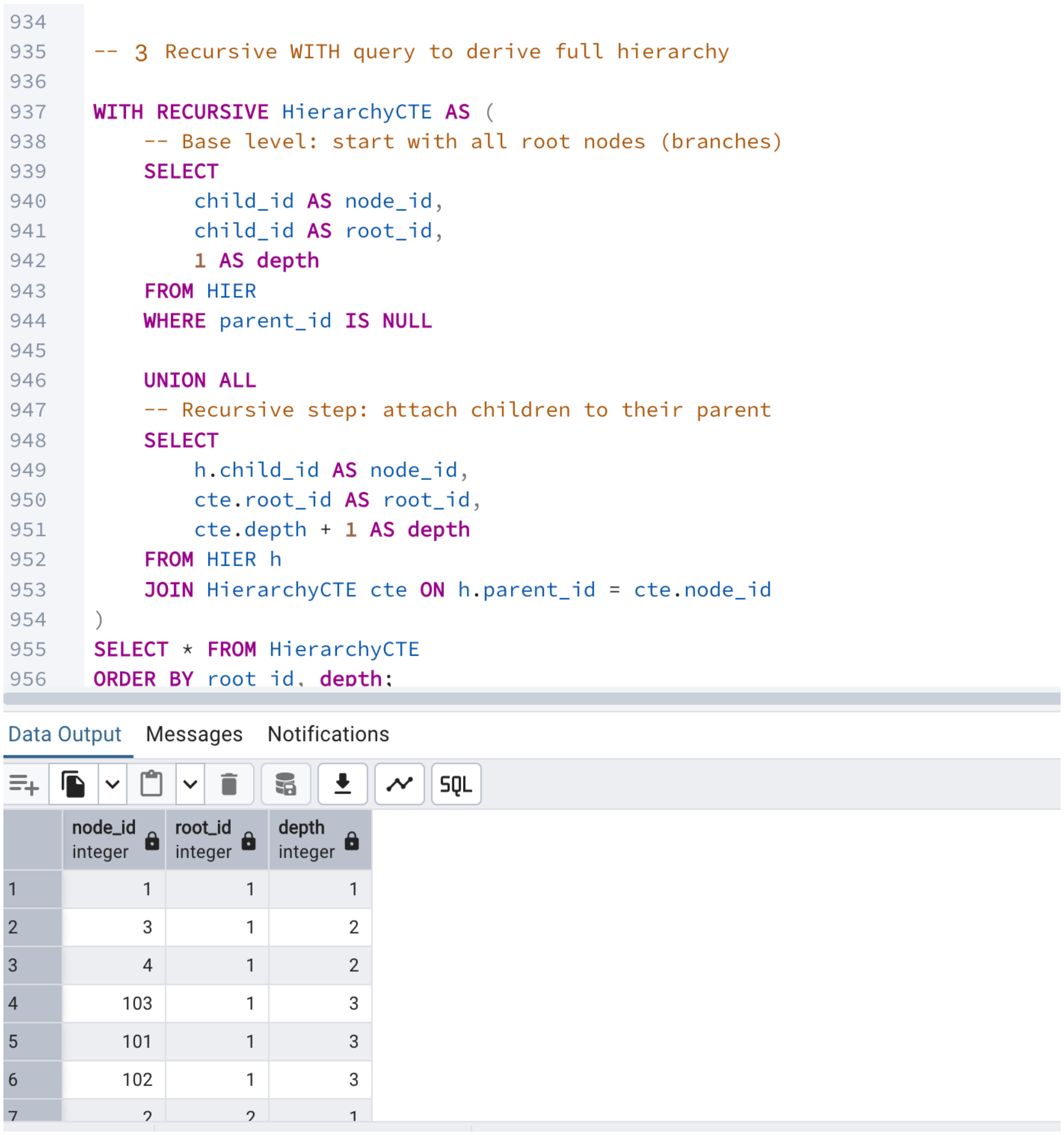
```
CREATE TABLE

Query returned successfully in 3 secs 93 msec.
```



```sql
-- 1Create the hierarchy table (Domain: Insurance Branch → Agent → Claim)
DROP TABLE IF EXISTS HIER CASCADE;

CREATE TABLE HIER (
    parent_id INT,
    child_id INT
);

-- 2 Insert 6–10 rows forming a 3-level hierarchy
-- Level 1: Root branches (Kigali, Musanze)
-- Level 2: Agents under branches
-- Level 3: Claims under agents (reuse existing ClaimIDs)

INSERT INTO HIER (parent_id, child_id) VALUES
    (NULL, 1),  -- 1 = Kigali branch (root)
    (NULL, 2),  -- 2 = Musanze branch (root)
    (1, 3),     -- Agent A under Kigali
    (1, 4),     -- Agent B under Kigali
    (2, 5),     -- Agent C under Musanze
    (3, 101),   -- Claim 101 under Agent A
    (3, 102),   -- Claim 102 under Agent A
    (4, 103),   -- Claim 103 under Agent B
    (5, 104);   -- Claim 104 under Agent C


-- (Total: 9 rows → within allowed limit)

COMMIT;
```

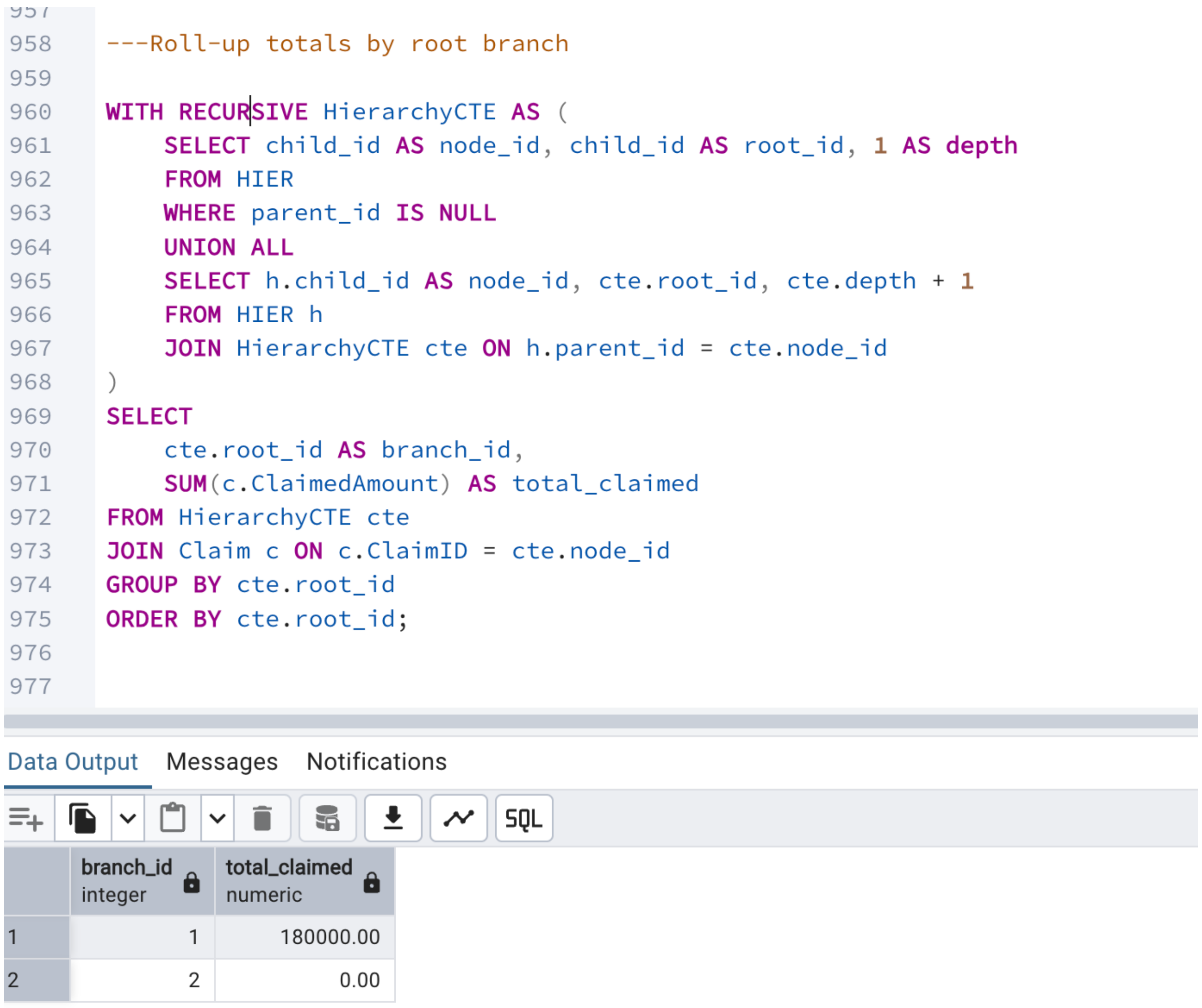
Data Output   Messages   Notifications

```
INSERT 0 9

Query returned successfully in 3 secs 904 msec.
```



```sql

-- 3 Recursive WITH query to derive full hierarchy

WITH RECURSIVE HierarchyCTE AS (
    -- Base level: start with all root nodes (branches)
    SELECT
        child_id AS node_id,
        child_id AS root_id,
        1 AS depth
    FROM HIER
    WHERE parent_id IS NULL

    UNION ALL
    -- Recursive step: attach children to their parent
    SELECT
        h.child_id AS node_id,
        cte.root_id AS root_id,
        cte.depth + 1 AS depth
    FROM HIER h
    JOIN HierarchyCTE cte ON h.parent_id = cte.node_id
)
SELECT * FROM HierarchyCTE
ORDER BY root_id, depth;
```

	node_id integer	root_id integer	depth integer
1	1	1	1
2	3	1	2
3	4	1	2
4	103	1	3
5	101	1	3
6	102	1	3
7	2	2	1


```
---Roll-up totals by root branch

WITH RECURSIVE HierarchyCTE AS (
    SELECT child_id AS node_id, child_id AS root_id, 1 AS depth
    FROM HIER
    WHERE parent_id IS NULL
    UNION ALL
    SELECT h.child_id AS node_id, cte.root_id, cte.depth + 1
    FROM HIER h
    JOIN HierarchyCTE cte ON h.parent_id = cte.node_id
)
SELECT
    cte.root_id AS branch_id,
    SUM(c.ClaimedAmount) AS total_claimed
FROM HierarchyCTE cte
JOIN Claim c ON c.ClaimID = cte.node_id
GROUP BY cte.root_id
ORDER BY cte.root_id;
```

Data Output Messages Notifications

	branch_id integer	total_claimed numeric
1	1	180000.00
2	2	0.00

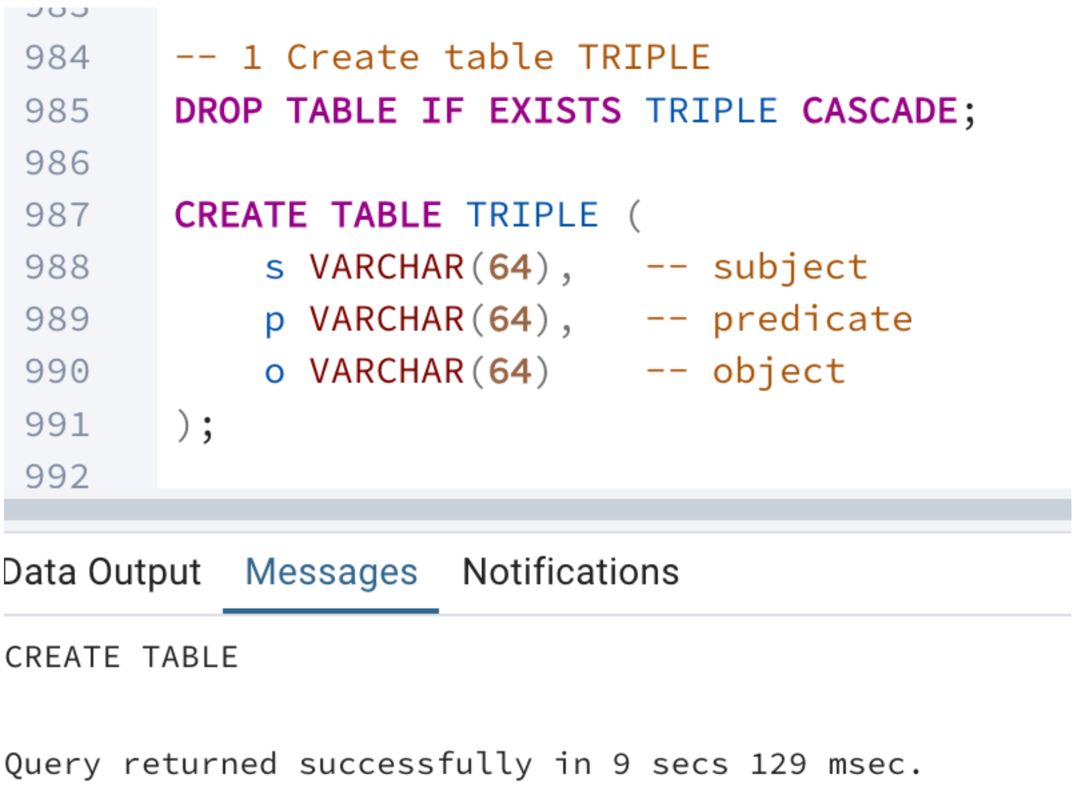
B9 :Mini-Knowledge Base with Transitive Inference (≤10 facts)

WHAT TO DO

1. Create table TRIPLE(s VARCHAR2(64), p VARCHAR2(64), o VARCHAR2(64)).
2. Insert 8–10 domain facts relevant to your project (e.g., simple type hierarchy or rule implications).
3. Write a recursive inference query implementing transitive isA*; apply labels to base records and return up to 10 labeled rows.
4. Ensure total committed rows across the project (including TRIPLE) remain ≤10; you may delete temporary rows after demo if needed.

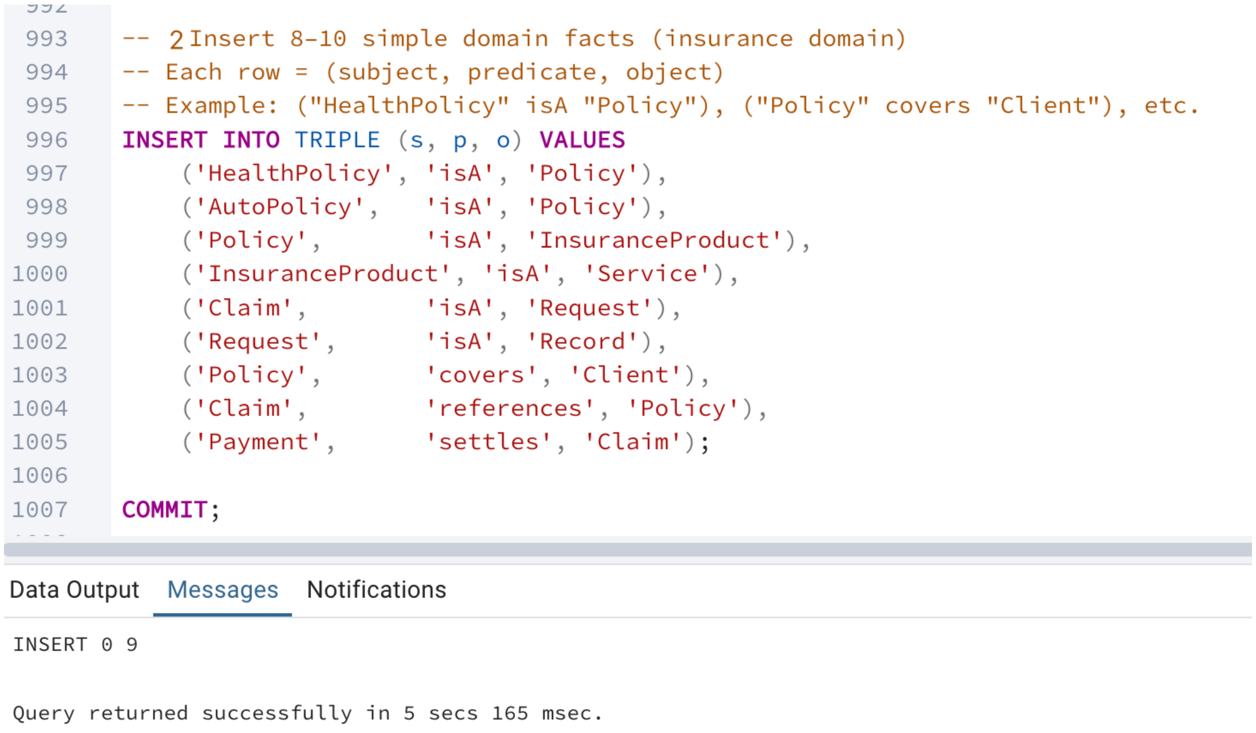
EXPECTED OUTPUT

- ✓ - DDL for TRIPLE and INSERT scripts for 8–10 facts.
- ✓ - Inference SELECT (with recursive part) and sample labeled output (≤10 rows).
- ✓ - Grouping counts proving inferred labels are consistent.



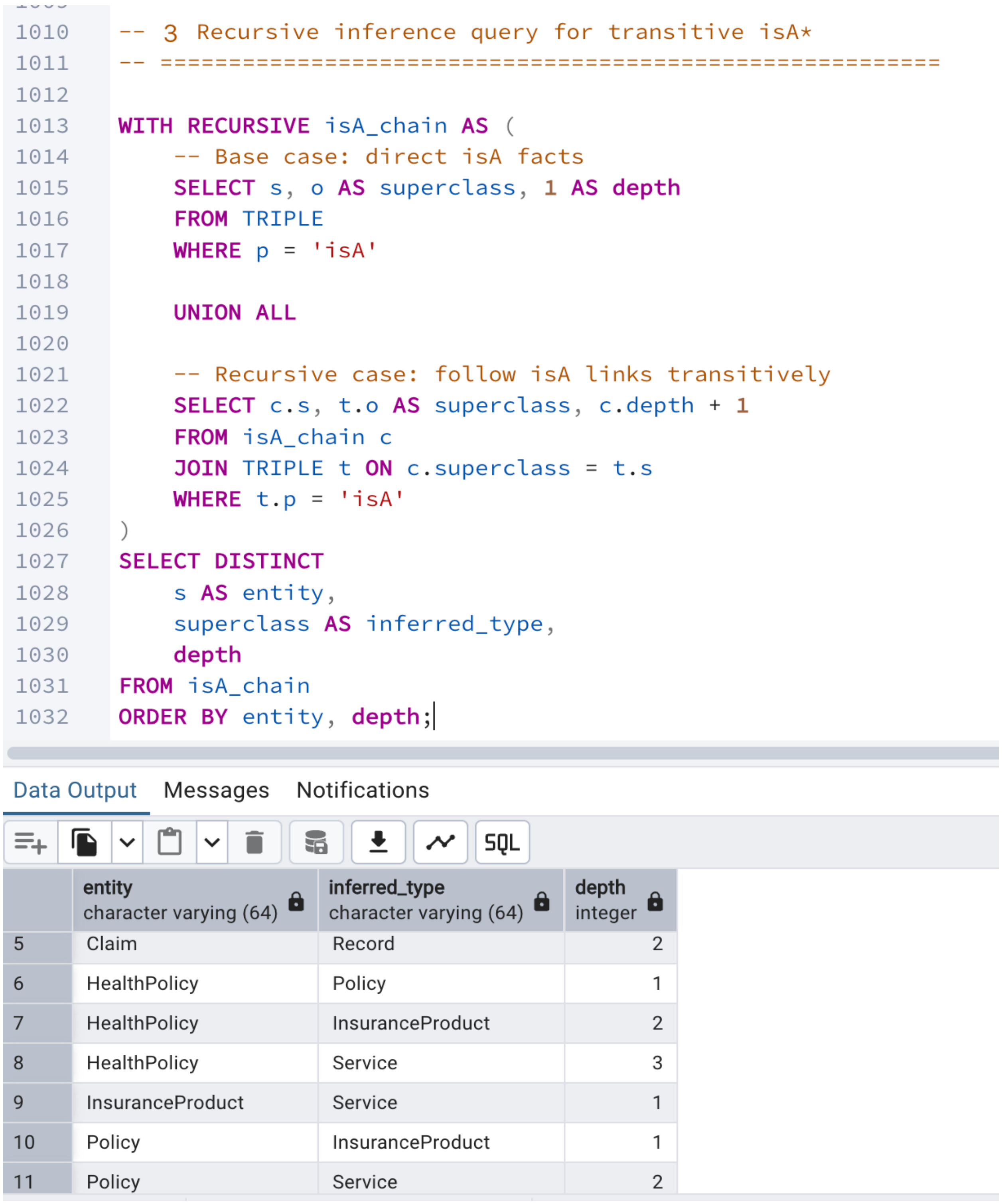
```
-- 1 Create table TRIPLE
DROP TABLE IF EXISTS TRIPLE CASCADE;

CREATE TABLE TRIPLE (
      s VARCHAR(64),    -- subject
      p VARCHAR(64),    -- predicate
      o VARCHAR(64)     -- object
);
```

```
CREATE TABLE

Query returned successfully in 9 secs 129 msec.
```



```
-- 2 Insert 8-10 simple domain facts (insurance domain)
-- Each row = (subject, predicate, object)
-- Example: ("HealthPolicy" isA "Policy"), ("Policy" covers "Client"), etc.
INSERT INTO TRIPLE (s, p, o) VALUES
    ('HealthPolicy', 'isA', 'Policy'),
    ('AutoPolicy',   'isA', 'Policy'),
    ('Policy',       'isA', 'InsuranceProduct'),
    ('InsuranceProduct', 'isA', 'Service'),
    ('Claim',        'isA', 'Request'),
    ('Request',      'isA', 'Record'),
    ('Policy',       'covers', 'Client'),
    ('Claim',        'references', 'Policy'),
    ('Payment',      'settles', 'Claim');

COMMIT;
```

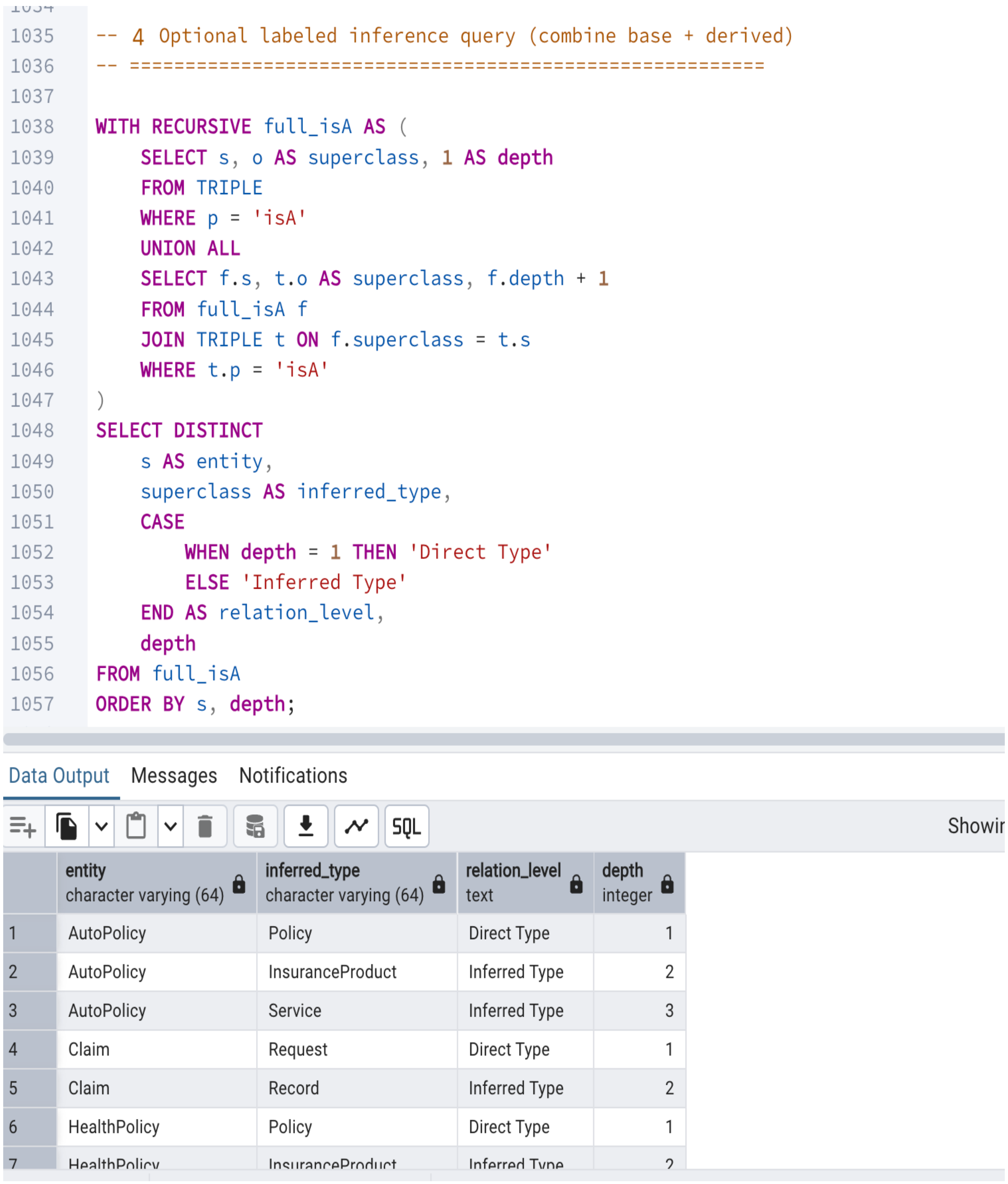
```
INSERT 0 9

Query returned successfully in 5 secs 165 msec.
```



```
-- 3 Recursive inference query for transitive isA*
-- ============================================================

WITH RECURSIVE isA_chain AS (
    -- Base case: direct isA facts
    SELECT s, o AS superclass, 1 AS depth
    FROM TRIPLE
    WHERE p = 'isA'

    UNION ALL

    -- Recursive case: follow isA links transitively
    SELECT c.s, t.o AS superclass, c.depth + 1
    FROM isA_chain c
    JOIN TRIPLE t ON c.superclass = t.s
    WHERE t.p = 'isA'
)
SELECT DISTINCT
    s AS entity,
    superclass AS inferred_type,
    depth
FROM isA_chain
ORDER BY entity, depth;
```

Data Output Messages Notifications

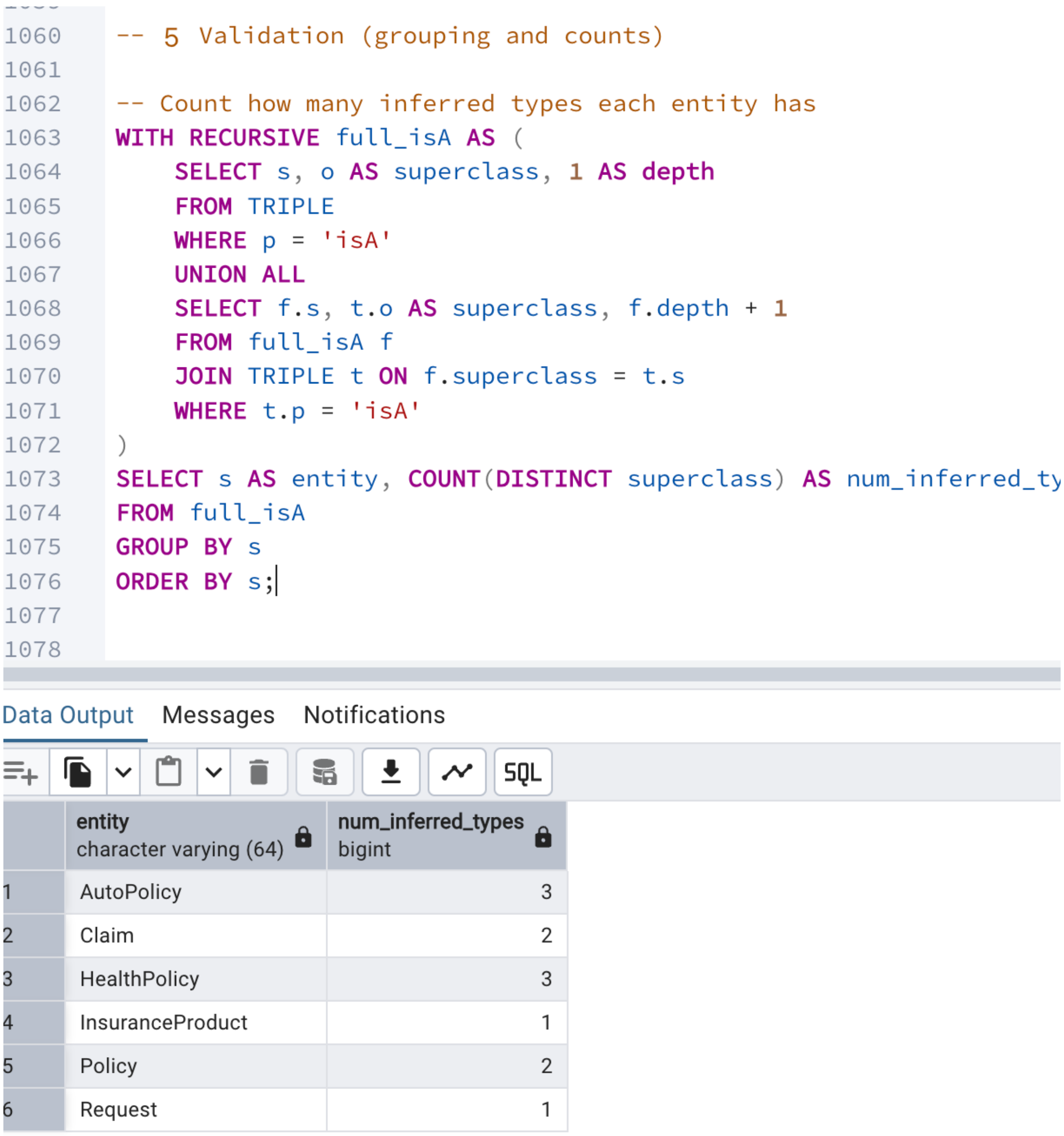
	entity character varying (64)	inferred_type character varying (64)	depth integer
5	Claim	Record	2
6	HealthPolicy	Policy	1
7	HealthPolicy	InsuranceProduct	2
8	HealthPolicy	Service	3
9	InsuranceProduct	Service	1
10	Policy	InsuranceProduct	1
11	Policy	Service	2



```
-- 4 Optional labeled inference query (combine base + derived)
-- ==========================================================

WITH RECURSIVE full_isA AS (
    SELECT s, o AS superclass, 1 AS depth
    FROM TRIPLE
    WHERE p = 'isA'
    UNION ALL
    SELECT f.s, t.o AS superclass, f.depth + 1
    FROM full_isA f
    JOIN TRIPLE t ON f.superclass = t.s
    WHERE t.p = 'isA'
)
SELECT DISTINCT
    s AS entity,
    superclass AS inferred_type,
    CASE
        WHEN depth = 1 THEN 'Direct Type'
        ELSE 'Inferred Type'
    END AS relation_level,
    depth
FROM full_isA
ORDER BY s, depth;
```

Data Output Messages Notifications

	entity character varying (64)	inferred_type character varying (64)	relation_level text	depth integer
1	AutoPolicy	Policy	Direct Type	1
2	AutoPolicy	InsuranceProduct	Inferred Type	2
3	AutoPolicy	Service	Inferred Type	3
4	Claim	Request	Direct Type	1
5	Claim	Record	Inferred Type	2
6	HealthPolicy	Policy	Direct Type	1
7	HealthPolicy	InsuranceProduct	Inferred Type	2



```sql
-- 5 Validation (grouping and counts)

-- Count how many inferred types each entity has
WITH RECURSIVE full_isA AS (
    SELECT s, o AS superclass, 1 AS depth
    FROM TRIPLE
    WHERE p = 'isA'
    UNION ALL
    SELECT f.s, t.o AS superclass, f.depth + 1
    FROM full_isA f
    JOIN TRIPLE t ON f.superclass = t.s
    WHERE t.p = 'isA'
)
SELECT s AS entity, COUNT(DISTINCT superclass) AS num_inferred_ty
FROM full_isA
GROUP BY s
ORDER BY s;
```

	entity character varying (64)	num_inferred_types bigint
1	AutoPolicy	3
2	Claim	2
3	HealthPolicy	3
4	InsuranceProduct	1
5	Policy	2
6	Request	1

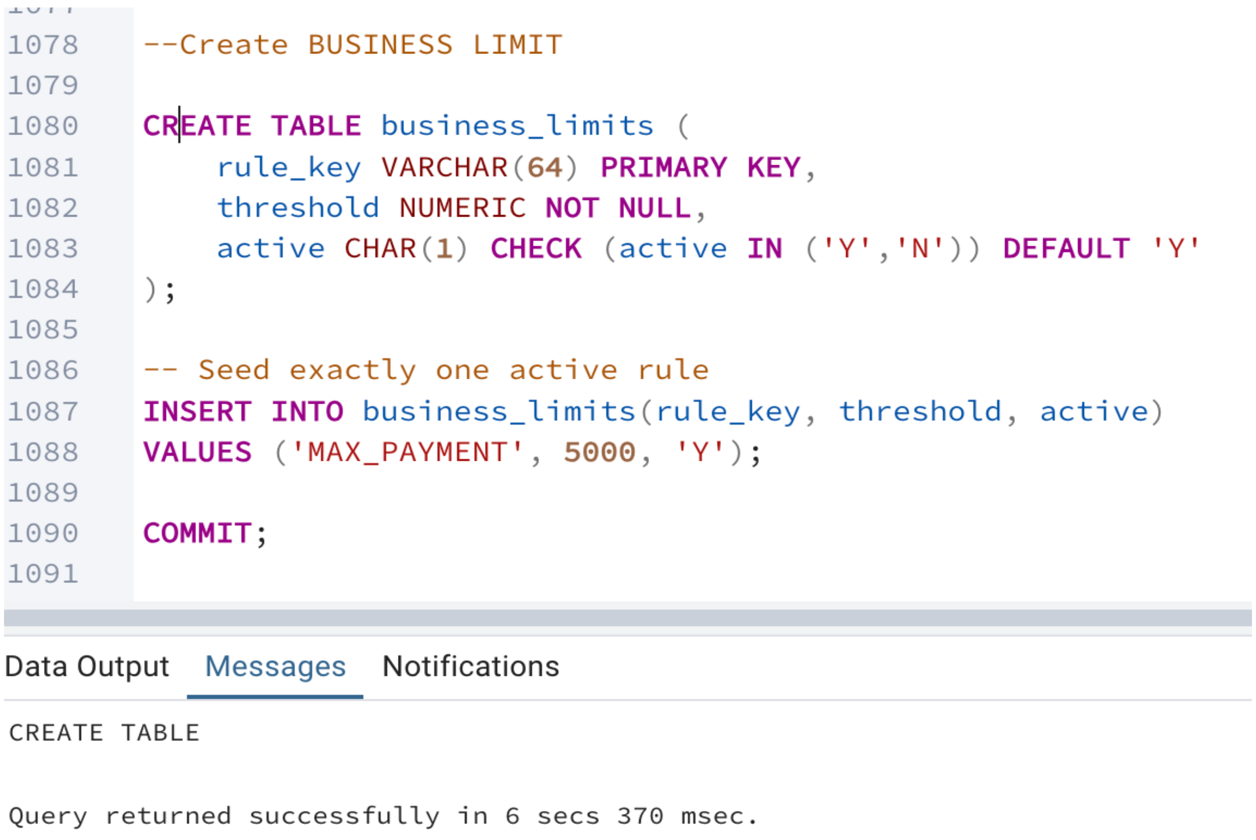
B10 :Business Limit Alert (Function + Trigger) (row-budget safe)

WHAT TO DO

1. Create BUSINESS_LIMITS(rule_key VARCHAR2(64), threshold NUMBER, active CHAR(1) CHECK(active IN('Y','N'))) and seed exactly one active rule.
2. Implement function fn_should_alert(...) that reads BUSINESS_LIMITS and inspects current data in Payment or Claim to decide a violation (return 1/0).
3. Create a BEFORE INSERT OR UPDATE trigger on Payment (or relevant table) that raises an application error when fn_should_alert returns 1.
4. Demonstrate 2 failing and 2 passing DML cases; rollback the failing ones so total committed rows remain within the ≤10 budget.

EXPECTED OUTPUT

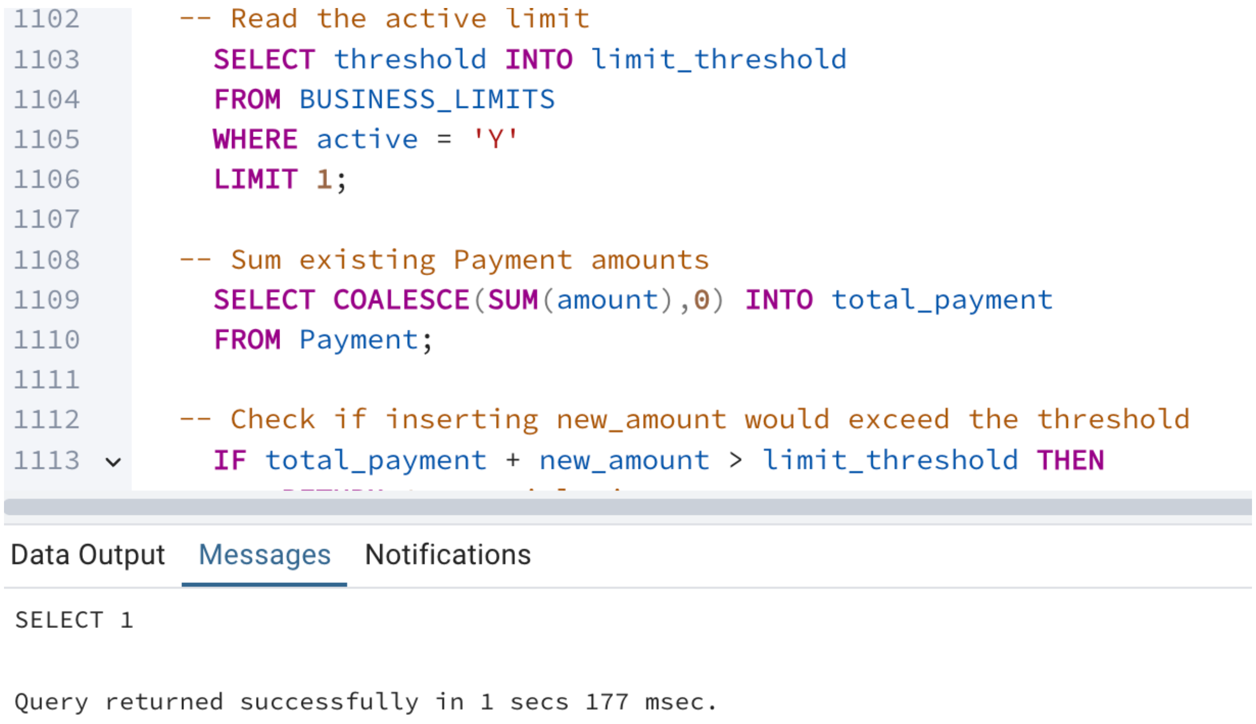
- ✓ - DDL for BUSINESS_LIMITS, function source, and trigger source.
- ✓ - Execution proof: two failed DML attempts (ORA- error) and two successful DMLs that commit.
- ✓ - SELECT showing resulting committed data consistent with the rule; row budget respected.



```
--Create BUSINESS LIMIT

CREATE TABLE business_limits (
    rule_key VARCHAR(64) PRIMARY KEY,
    threshold NUMERIC NOT NULL,
    active CHAR(1) CHECK (active IN ('Y','N')) DEFAULT 'Y'
);

-- Seed exactly one active rule
INSERT INTO business_limits(rule_key, threshold, active)
VALUES ('MAX_PAYMENT', 5000, 'Y');

COMMIT;

```

Data Output Messages Notifications

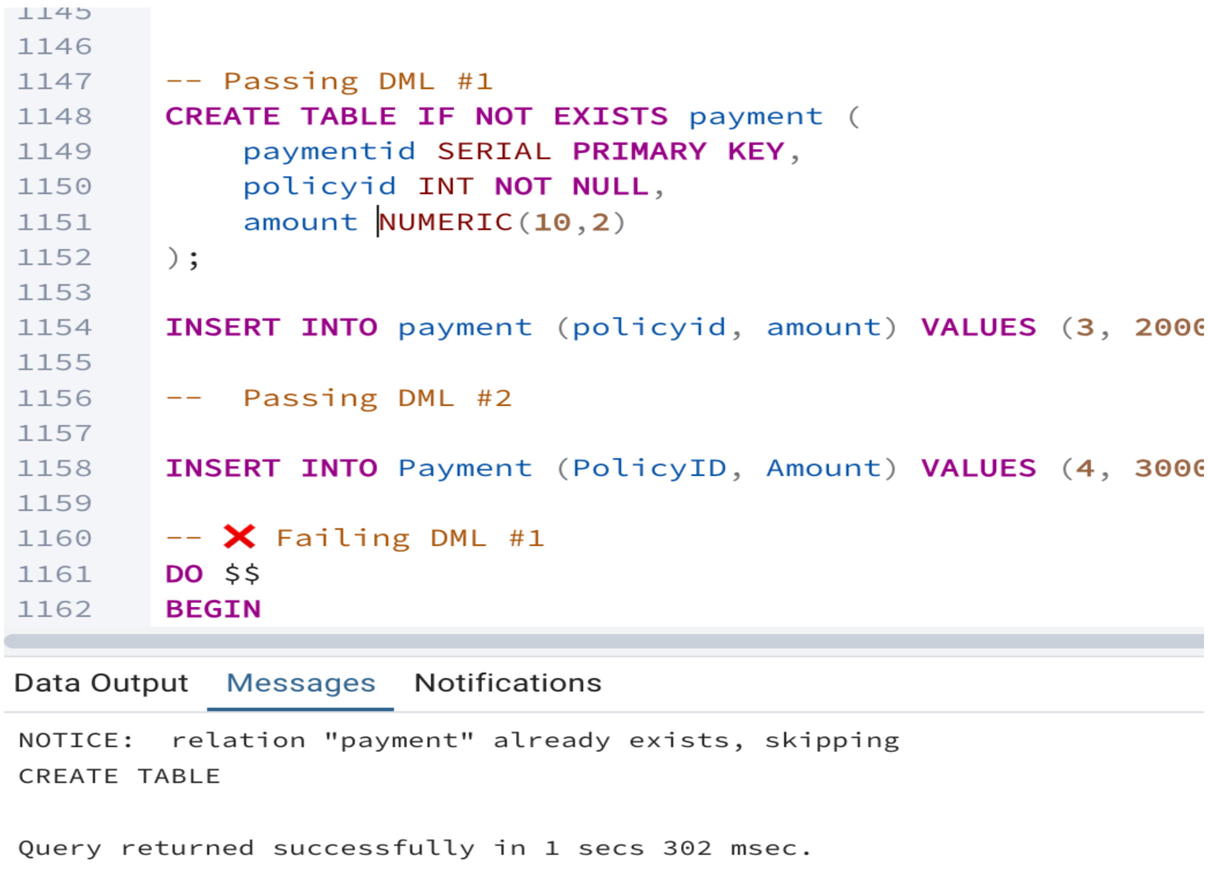
```
CREATE TABLE

Query returned successfully in 6 secs 370 msec.
```



```
  -- Read the active limit
    SELECT threshold INTO limit_threshold
    FROM BUSINESS_LIMITS
    WHERE active = 'Y'
    LIMIT 1;

  -- Sum existing Payment amounts
    SELECT COALESCE(SUM(amount),0) INTO total_payment
    FROM Payment;

  -- Check if inserting new_amount would exceed the threshold
    IF total_payment + new_amount > limit_threshold THEN
```

Data Output Messages Notifications

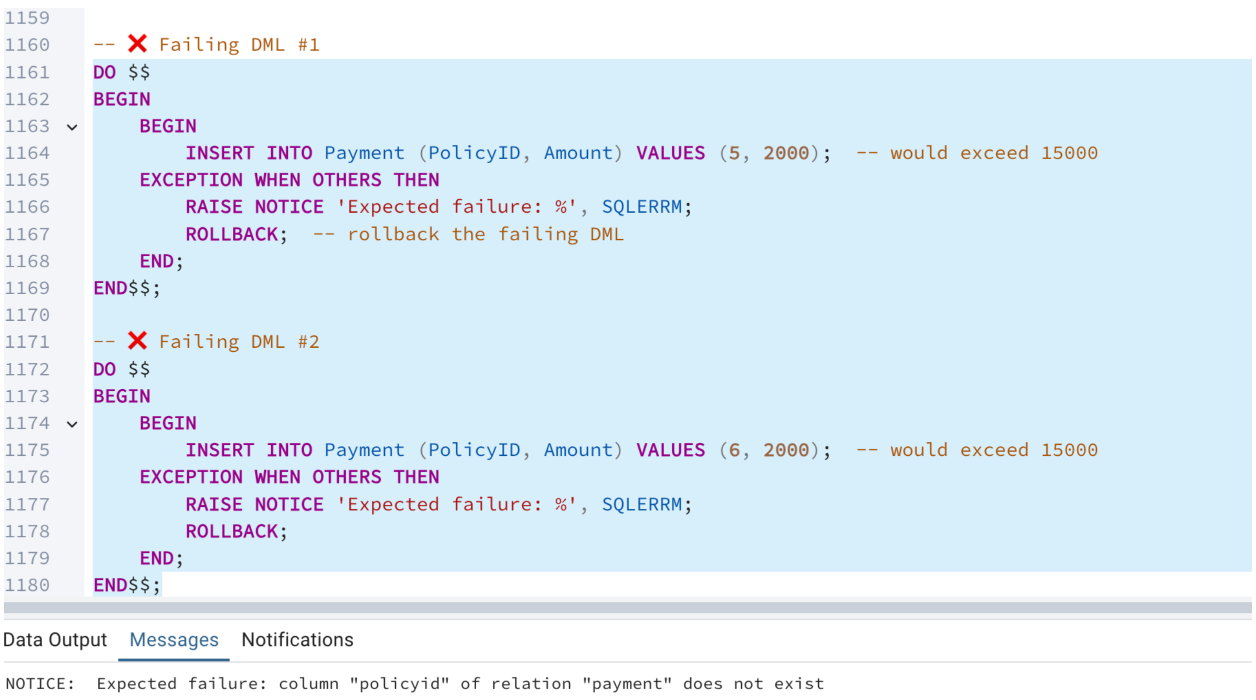
```
SELECT 1

Query returned successfully in 1 secs 177 msec.
```



```
-- Passing DML #1
CREATE TABLE IF NOT EXISTS payment (
    paymentid SERIAL PRIMARY KEY,
    policyid INT NOT NULL,
    amount NUMERIC(10,2)
);

INSERT INTO payment (policyid, amount) VALUES (3, 200

--  Passing DML #2

INSERT INTO Payment (PolicyID, Amount) VALUES (4, 300

-- ❌ Failing DML #1
DO $$
BEGIN
```

Data Output Messages Notifications

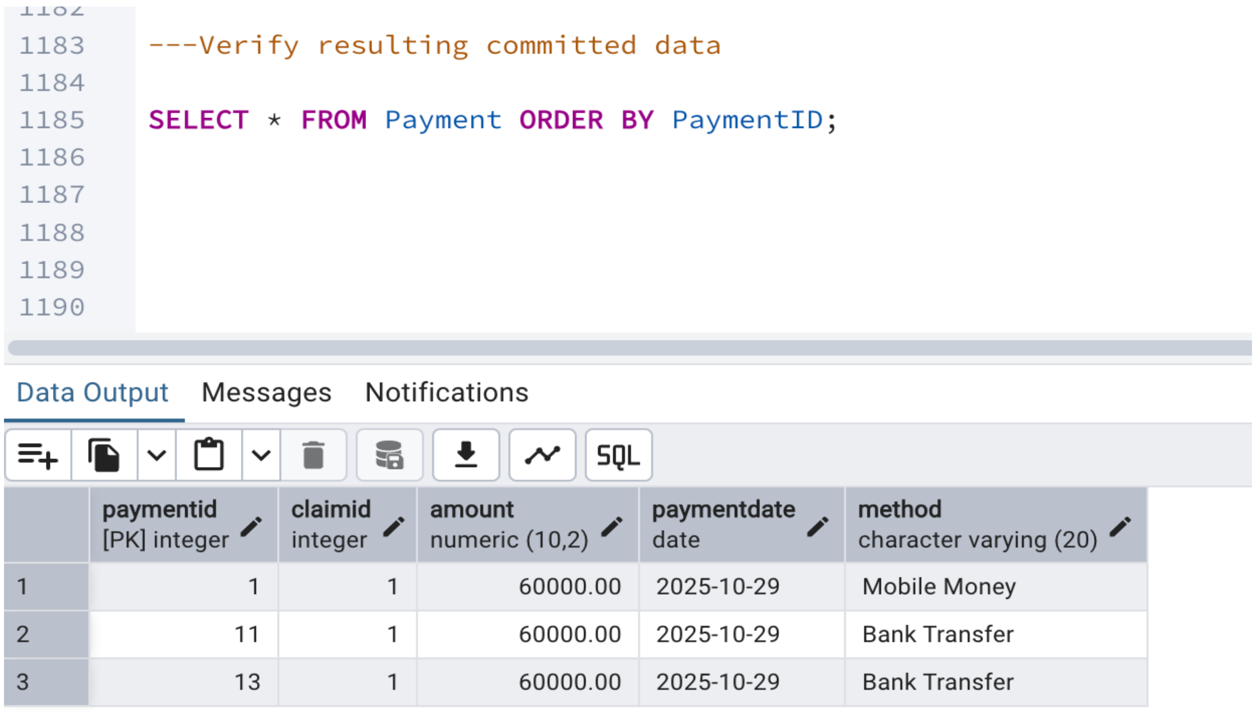
```
NOTICE:  relation "payment" already exists, skipping
CREATE TABLE

Query returned successfully in 1 secs 302 msec.
```



```

-- ❌ Failing DML #1
DO $$
BEGIN
    BEGIN
        INSERT INTO Payment (PolicyID, Amount) VALUES (5, 2000);  -- would exceed 15000
    EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'Expected failure: %', SQLERRM;
        ROLLBACK;  -- rollback the failing DML
    END;
END$$;

-- ❌ Failing DML #2
DO $$
BEGIN
    BEGIN
        INSERT INTO Payment (PolicyID, Amount) VALUES (6, 2000);  -- would exceed 15000
    EXCEPTION WHEN OTHERS THEN
        RAISE NOTICE 'Expected failure: %', SQLERRM;
        ROLLBACK;
    END;
END$$;
```

Data Output Messages Notifications

```
NOTICE:  Expected failure: column "policyid" of relation "payment" does not exist
```



```sql
---Verify resulting committed data

SELECT * FROM Payment ORDER BY PaymentID;
```

	paymentid [PK] integer	claimid integer	amount numeric (10,2)	paymentdate date	method character varying (20)
1	1	1	60000.00	2025-10-29	Mobile Money
2	11	1	60000.00	2025-10-29	Bank Transfer
3	13	1	60000.00	2025-10-29	Bank Transfer