Procedures, Functions and Triggers

Кристијан Тримчески 231132
Елена Спасовска 231141
Марија Сергиевска 231048

Procedures:

1. submit_offer - Allows a worker to submit a price offer on an open task request, or enables a client to initiate an offer on behalf of a worker. Performs full validation before inserting the offer record and dispatches a notification to the client. Used when a worker or a client submits an offer.

```
CREATE OR REPLACE PROCEDURE submit_offer(
    p_worker_id      INT,
    p_task_request_id INT,
    p_price          INT,
    p_initiated_by   VARCHAR(10)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status    VARCHAR(20);
    v_category_id    INT;
    v_client_user_id INT;
    v_offer_id       INT;
    v_notif_type_id  INT;
    v_duplicate      INT;
BEGIN

    IF p_initiated_by NOT IN ('CLIENT', 'WORKER') THEN
        RAISE EXCEPTION 'initiated_by must be CLIENT or WORKER, got: %',
p_initiated_by;
    END IF;


    IF p_price <= 0 THEN
        RAISE EXCEPTION 'Price must be greater than 0, got: %', p_price;
    END IF;


    SELECT status, category_id
    INTO v_task_status, v_category_id
    FROM TaskRequest
    WHERE id = p_task_request_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'TaskRequest % does not exist', p_task_request_id;
    END IF;

    IF v_task_status <> 'OPEN' THEN
        RAISE EXCEPTION 'TaskRequest % is not open (status: %)',
p_task_request_id, v_task_status;
    END IF;


    IF NOT EXISTS (
        SELECT 1 FROM WorkerCategory
        WHERE worker_id = p_worker_id
          AND category_id = v_category_id
```

```
    ) THEN
        RAISE EXCEPTION 'Worker % is not registered for the required category',
p_worker_id;
    END IF;


    SELECT COUNT(*) INTO v_duplicate
    FROM Offer
    WHERE worker_id = p_worker_id
      AND task_request_id = p_task_request_id
      AND offer_status IN ('PENDING', 'ACCEPTED');

    IF v_duplicate > 0 THEN
        RAISE EXCEPTION 'Worker % already has an active offer for TaskRequest
%', p_worker_id, p_task_request_id;
    END IF;


    INSERT INTO Offer (worker_id, task_request_id, price, offer_status,
initiated_by)
    VALUES (p_worker_id, p_task_request_id, p_price, 'PENDING', p_initiated_by)
    RETURNING id INTO v_offer_id;

    SELECT u.id
    INTO v_client_user_id
    FROM TaskRequest tr
    JOIN Client c ON tr.client_id = c.id
    JOIN UserAccount u ON c.user_id = u.id
    WHERE tr.id = p_task_request_id;


    SELECT id INTO v_notif_type_id
    FROM NotificationType
    WHERE name = 'NEW_OFFER'
    LIMIT 1;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
offer_id)
    VALUES (
        'New Offer Received',
        'A worker has submitted an offer for your task request #' ||
p_task_request_id,
        v_client_user_id,
        v_notif_type_id,
        v_offer_id
    );

    RAISE NOTICE 'Offer % submitted successfully for TaskRequest %', v_offer_id,
p_task_request_id;
END;
$$;
```

2. accept_offer - Accepts a specific pending offer, rejects all competing offers on the same task request, closes the task request, creates an active Task, and notifies both the accepted worker and all rejected workers. Used when a user accepts an offer.

```
CREATE OR REPLACE PROCEDURE accept_offer(
    p_offer_id INT
)
LANGUAGE plpgsql
```

```
AS $$
DECLARE
    v_task_request_id INT;
    v_worker_id       INT;
    v_offer_status    VARCHAR(20);
    v_task_id         INT;
    v_notif_type_acc  INT;
    v_notif_type_rej  INT;
    v_worker_user_id  INT;
    rec               RECORD;
BEGIN

    SELECT task_request_id, worker_id, offer_status
    INTO v_task_request_id, v_worker_id, v_offer_status
    FROM Offer
    WHERE id = p_offer_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Offer % does not exist', p_offer_id;
    END IF;

    IF v_offer_status <> 'PENDING' THEN
        RAISE EXCEPTION 'Offer % is not pending (status: %)', p_offer_id,
v_offer_status;
    END IF;


    UPDATE Offer
    SET offer_status = 'ACCEPTED',
        updated_at   = CURRENT_TIMESTAMP
    WHERE id = p_offer_id;


    UPDATE Offer
    SET offer_status = 'REJECTED',
        updated_at   = CURRENT_TIMESTAMP
    WHERE task_request_id = v_task_request_id
      AND id <> p_offer_id
      AND offer_status = 'PENDING';


    UPDATE TaskRequest
    SET status     = 'CLOSED',
        updated_at = CURRENT_TIMESTAMP
    WHERE id = v_task_request_id;

    INSERT INTO Task (offer_id, status)
    VALUES (p_offer_id, 'ACTIVE')
    RETURNING id INTO v_task_id;


    SELECT id INTO v_notif_type_acc FROM NotificationType WHERE name =
'OFFER_ACCEPTED' LIMIT 1;
    SELECT id INTO v_notif_type_rej FROM NotificationType WHERE name =
'OFFER_REJECTED' LIMIT 1;


    SELECT u.id INTO v_worker_user_id
    FROM Worker w
    JOIN UserAccount u ON w.user_id = u.id
    WHERE w.id = v_worker_id;
```

```
    INSERT INTO Notification (title, body, user_id, notification_type_id,
offer_id, task_id)
    VALUES (
        'Your Offer Was Accepted!',
        'Congratulations! Your offer #' || p_offer_id || ' has been accepted.
Task #' || v_task_id || ' is now active.',
        v_worker_user_id,
        v_notif_type_acc,
        p_offer_id,
        v_task_id
    );

    FOR rec IN
        SELECT w.user_id AS uid, o.id AS oid
        FROM Offer o
        JOIN Worker w ON o.worker_id = w.id
        WHERE o.task_request_id = v_task_request_id
          AND o.id <> p_offer_id
          AND o.offer_status = 'REJECTED'
    LOOP
        INSERT INTO Notification (title, body, user_id, notification_type_id,
offer_id)
        VALUES (
            'Offer Not Selected',
            'Unfortunately your offer #' || rec.oid || ' was not selected for
this task.',
            rec.uid,
            v_notif_type_rej,
            rec.oid
        );
    END LOOP;

    RAISE NOTICE 'Offer % accepted. Task % created.', p_offer_id, v_task_id;
END;
$$;
```

3. complete_task - Marks an active task as completed, creates a pending payment record for the agreed offer price, links it via TaskPayment, and notifies both the client and the worker. Used when a task is finished.

```
CREATE OR REPLACE PROCEDURE complete_task(
    p_task_id        INT,
    p_payment_method VARCHAR(50)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status     VARCHAR(20);
    v_offer_id        INT;
    v_offer_price     INT;
    v_worker_id       INT;
    v_client_id       INT;
    v_client_user_id  INT;
    v_worker_user_id  INT;
    v_payment_id      INT;
    v_notif_type_id   INT;
```

```
BEGIN

    IF p_payment_method NOT IN ('CARD', 'CASH', 'PAYPAL') THEN
        RAISE EXCEPTION 'Invalid payment method: %. Must be CARD, CASH, or
PAYPAL', p_payment_method;
    END IF;


    SELECT status, offer_id
    INTO v_task_status, v_offer_id
    FROM Task
    WHERE id = p_task_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Task % does not exist', p_task_id;
    END IF;

    IF v_task_status <> 'ACTIVE' THEN
        RAISE EXCEPTION 'Task % is not active (status: %)', p_task_id,
v_task_status;
    END IF;


    SELECT o.price, o.worker_id, tr.client_id
    INTO v_offer_price, v_worker_id, v_client_id
    FROM Offer o
    JOIN TaskRequest tr ON o.task_request_id = tr.id
    WHERE o.id = v_offer_id;


    UPDATE Task
    SET status       = 'COMPLETED',
        completed_at = CURRENT_TIMESTAMP,
        updated_at   = CURRENT_TIMESTAMP
    WHERE id = p_task_id;


    INSERT INTO Payment (amount, payment_method, status, task_id, client_id,
worker_id)
    VALUES (v_offer_price, p_payment_method, 'PENDING', p_task_id, v_client_id,
v_worker_id)
    RETURNING id INTO v_payment_id;


    INSERT INTO TaskPayment (payment_id, task_id)
    VALUES (v_payment_id, p_task_id);


    SELECT id INTO v_notif_type_id FROM NotificationType WHERE name =
'TASK_COMPLETED' LIMIT 1;


    SELECT u.id INTO v_client_user_id
    FROM Client c JOIN UserAccount u ON c.user_id = u.id
    WHERE c.id = v_client_id;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id, payment_id)
    VALUES (
        'Task Completed',
        'Task #' || p_task_id || ' has been marked as completed. Payment of ' ||
v_offer_price || ' is pending.',
```

```
        v_client_user_id,
        v_notif_type_id,
        p_task_id,
        v_payment_id
    );


    SELECT u.id INTO v_worker_user_id
    FROM Worker w JOIN UserAccount u ON w.user_id = u.id
    WHERE w.id = v_worker_id;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id, payment_id)
    VALUES (
        'Task Completed - Payment Incoming',
        'Task #' || p_task_id || ' is complete. You will receive payment of ' ||
v_offer_price || ' shortly.',
        v_worker_user_id,
        v_notif_type_id,
        p_task_id,
        v_payment_id
    );

    RAISE NOTICE 'Task % completed. Payment % created.', p_task_id,
v_payment_id;
END;
$$;
```

4. cancel_task - Cancels an active task, marks any pending payment as FAILED, and notifies both the client and the worker with an optional reason string. Used when a user cancels the active task.

```
CREATE OR REPLACE PROCEDURE cancel_task(
    p_task_id  INT,
    p_reason   VARCHAR(255)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status    VARCHAR(20);
    v_offer_id       INT;
    v_worker_id      INT;
    v_client_id      INT;
    v_client_user_id INT;
    v_worker_user_id INT;
    v_notif_type_id  INT;
    v_payment_id     INT;
BEGIN

    SELECT status, offer_id
    INTO v_task_status, v_offer_id
    FROM Task
    WHERE id = p_task_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Task % does not exist', p_task_id;
    END IF;

    IF v_task_status <> 'ACTIVE' THEN
        RAISE EXCEPTION 'Task % cannot be cancelled (status: %)', p_task_id,
v_task_status;
```

```
    END IF;


    SELECT o.worker_id, tr.client_id
    INTO v_worker_id, v_client_id
    FROM Offer o
    JOIN TaskRequest tr ON o.task_request_id = tr.id
    WHERE o.id = v_offer_id;


    UPDATE Task
    SET status     = 'CANCELLED',
        updated_at = CURRENT_TIMESTAMP
    WHERE id = p_task_id;


    UPDATE Payment
    SET status     = 'FAILED',
        updated_at = CURRENT_TIMESTAMP
    WHERE task_id = p_task_id
      AND status  = 'PENDING'
    RETURNING id INTO v_payment_id;


    SELECT id INTO v_notif_type_id
    FROM NotificationType
    WHERE name = 'TASK_CANCELLED'
    LIMIT 1;


    SELECT u.id INTO v_client_user_id
    FROM Client c
    JOIN UserAccount u ON c.user_id = u.id
    WHERE c.id = v_client_id;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id)
    VALUES (
        'Task Cancelled',
        'Task #' || p_task_id || ' has been cancelled. Reason: ' ||
COALESCE(p_reason, 'No reason provided'),
        v_client_user_id,
        v_notif_type_id,
        p_task_id
    );


    SELECT u.id INTO v_worker_user_id
    FROM Worker w
    JOIN UserAccount u ON w.user_id = u.id
    WHERE w.id = v_worker_id;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id)
    VALUES (
        'Task Cancelled',
        'Task #' || p_task_id || ' has been cancelled. Reason: ' ||
COALESCE(p_reason, 'No reason provided'),
        v_worker_user_id,
        v_notif_type_id,
        p_task_id
    );
```

```
    RAISE NOTICE 'Task % cancelled successfully.', p_task_id;
END;
$$;
```

5. submit_complaint - Allows a client to file a complaint against a worker on a completed task or a worker to file a complaint against a client on a completed task. Validates ownership, prevents duplicate open complaints, inserts the complaint record, and notifies the one that filed the complaint. Used when a user issues a complaint.

```
CREATE OR REPLACE PROCEDURE submit_complaint(
    p_task_id     INT,
    p_client_id   INT,
    p_worker_id   INT,
    p_reason      TEXT,
    p_description TEXT,
    p_filed_by    VARCHAR(10)
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_task_status    VARCHAR(20);
    v_offer_id       INT;
    v_real_client_id INT;
    v_real_worker_id INT;
    v_complaint_id   INT;
    v_filer_user_id  INT;
    v_notif_type_id  INT;
BEGIN
    IF p_filed_by NOT IN ('CLIENT', 'WORKER') THEN
        RAISE EXCEPTION 'p_filed_by must be CLIENT or WORKER, got: %',
p_filed_by;
    END IF;

    SELECT status, offer_id
    INTO   v_task_status, v_offer_id
    FROM   Task
    WHERE  id = p_task_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'Task % does not exist', p_task_id;
    END IF;

    IF v_task_status <> 'COMPLETED' THEN
        RAISE EXCEPTION 'Complaints can only be filed on COMPLETED tasks
(status: %)', v_task_status;
    END IF;

    SELECT tr.client_id, o.worker_id
    INTO   v_real_client_id, v_real_worker_id
    FROM   Offer o
    JOIN   TaskRequest tr ON o.task_request_id = tr.id
    WHERE  o.id = v_offer_id;

    IF v_real_client_id <> p_client_id THEN
        RAISE EXCEPTION 'Client % is not the client for Task %', p_client_id,
p_task_id;
    END IF;
```

```
    IF v_real_worker_id <> p_worker_id THEN
        RAISE EXCEPTION 'Worker % is not the worker for Task %', p_worker_id,
p_task_id;
    END IF;

    IF p_filed_by = 'CLIENT' AND EXISTS (
        SELECT 1 FROM Complaint
        WHERE task_id   = p_task_id
          AND client_id = p_client_id
          AND status    = 'OPEN'
    ) THEN
        RAISE EXCEPTION 'Client % already has an open complaint for Task %',
p_client_id, p_task_id;
    END IF;

    IF p_filed_by = 'WORKER' AND EXISTS (
        SELECT 1 FROM Complaint
        WHERE task_id   = p_task_id
          AND worker_id = p_worker_id
          AND status    = 'OPEN'
    ) THEN
        RAISE EXCEPTION 'Worker % already has an open complaint for Task %',
p_worker_id, p_task_id;
    END IF;

    INSERT INTO Complaint (reason, description, status, task_id, client_id,
worker_id)
    VALUES (p_reason, p_description, 'OPEN', p_task_id, p_client_id,
p_worker_id)
    RETURNING id INTO v_complaint_id;


    IF p_filed_by = 'CLIENT' THEN
        SELECT u.id INTO v_filer_user_id
        FROM   Client c
        JOIN   UserAccount u ON c.user_id = u.id
        WHERE  c.id = p_client_id;
    ELSE
        SELECT u.id INTO v_filer_user_id
        FROM   Worker w
        JOIN   UserAccount u ON w.user_id = u.id
        WHERE  w.id = p_worker_id;
    END IF;

    SELECT id INTO v_notif_type_id
    FROM   NotificationType
    WHERE  name = 'COMPLAINT_FILED'
    LIMIT  1;

    INSERT INTO Notification (title, body, user_id, notification_type_id,
task_id)
    VALUES (
        'Complaint Submitted',
        'Your complaint for Task #' || p_task_id || ' has been received and is
under review. Reason: ' || p_reason,
        v_filer_user_id,
        v_notif_type_id,
        p_task_id
    );

    RAISE NOTICE 'Complaint % filed for Task % by %.', v_complaint_id,
p_task_id, p_filed_by;
```

```
END;
$$;
```

Functions:

1. clean_text - Transliterates Cyrillic characters to their Latin equivalents using a character-by-character mapping. The function is declared IMMUTABLE, making it safe for use in index expressions and constant-folding optimizations by the query planner. Used when filtering a worker or client by name.

```
CREATE OR REPLACE FUNCTION clean_text(input TEXT)
RETURNS TEXT AS $$
BEGIN
    RETURN TRANSLATE(input,
        'абвгдѓежзсијклљмнњопрстќуфхцчџшАБВГДЃЕЖЗСИЈКЛЉМНЊОПРСТЌУФХЦЧЏШ',
        'abvgdgezzsijkljmnjoprstkufhccdsABVGDGEZZSIJKLJMJNJOPRSTKUFHCCDS'
    );
END;
$$ LANGUAGE plpgsql IMMUTABLE;
```

2. get_client_tasks - Returns a result set of task requests belonging to a specific client, enriched with category name and city. An optional status filter restricts results to OPEN or CLOSED task requests. Results are ordered by creation date descending (newest first) and soft-deleted records are excluded. Used when a client wants to see his task requests.

```
CREATE OR REPLACE FUNCTION get_client_tasks(
    p_client_id INT,
    p_status    VARCHAR(20) DEFAULT NULL
)
RETURNS TABLE (
    task_request_id INT,
    client_id       INT,
    description     VARCHAR,
    work_mode       VARCHAR,
    status          VARCHAR,
    category_name   VARCHAR,
    city            VARCHAR
)
LANGUAGE plpgsql
AS $$
```

```
BEGIN

    IF p_status IS NOT NULL AND p_status NOT IN ('OPEN', 'CLOSED') THEN
        RAISE EXCEPTION 'Invalid status filter: %. Must be OPEN or CLOSED',
p_status;
    END IF;

    IF NOT EXISTS (SELECT 1 FROM Client WHERE id = p_client_id) THEN
        RAISE EXCEPTION 'Client % does not exist', p_client_id;
    END IF;

    RETURN QUERY
    SELECT
        tr.id          AS task_request_id,
        tr.client_id,
        tr.description,
        tr.work_mode,
        tr.status,
        c.category_name,
        l.city
    FROM TaskRequest tr
    JOIN Category c  ON tr.category_id  = c.id
    JOIN Location l  ON tr.location_id  = l.id
    WHERE tr.client_id = p_client_id
      AND tr.deleted_at IS NULL
      AND (p_status IS NULL OR tr.status = p_status)
    ORDER BY tr.created_at DESC;
END;
$$;
```

3. fn_match_workers – Matches qualified workers to a specific task request based on multiple criteria including category expertise, work mode compatibility, geographic proximity, performance metrics, and current workload. Returns a ranked list of the best-matching workers. Used when a client is searching for workers for the task.

```
CREATE OR REPLACE FUNCTION fn_match_workers(
    p_task_request_id INT,
    p_limit           INT DEFAULT 20
)
RETURNS TABLE (
    out_worker_id         INT,
    out_user_id           INT,
    out_name              VARCHAR,
    out_surname           VARCHAR,
    out_city              VARCHAR,
    out_work_mode         VARCHAR,
    out_service_radius_km INT,
    out_distance_km       NUMERIC,
    out_categories        TEXT[],
    out_avg_rating        NUMERIC,
```

```
    out_total_reviews     BIGINT,
    out_performance_level VARCHAR,
    out_active_badge      VARCHAR,
    out_badge_tier        INT,
    out_active_tasks      BIGINT,
    out_match_score       NUMERIC
)
LANGUAGE plpgsql
AS $$
DECLARE
    v_category_id   INT;
    v_parent_cat_id INT;
    v_work_mode     VARCHAR(20);
    v_location_id   INT;
    v_task_lat      DECIMAL(9,6);
    v_task_lon      DECIMAL(9,6);
    v_task_created  TIMESTAMP;
BEGIN

    SELECT tr.category_id,
           tr.work_mode,
           tr.location_id,
           tr.created_at
    INTO   v_category_id, v_work_mode, v_location_id, v_task_created
    FROM   TaskRequest tr
    WHERE  tr.id = p_task_request_id;

    IF NOT FOUND THEN
        RAISE EXCEPTION 'TaskRequest % does not exist', p_task_request_id;
    END IF;


    SELECT l.latitude, l.longitude
    INTO   v_task_lat, v_task_lon
    FROM   Location l
    WHERE  l.id = v_location_id;

    SELECT c.parent_category_id
    INTO   v_parent_cat_id
    FROM   Category c
    WHERE  c.id = v_category_id;


    RETURN QUERY
    WITH

    cat_workers AS (
        SELECT
            w.id                AS cw_worker_id,
            w.user_id           AS cw_user_id,
            w.work_mode         AS cw_work_mode,
            w.service_radius_km AS cw_radius,
            w.location_id       AS cw_location_id,
            w.created_at        AS cw_created
        FROM  Worker         w
        JOIN  WorkerCategory wc ON wc.worker_id  = w.id
        WHERE wc.category_id = v_category_id
          AND w.created_at   < v_task_created
    ),

    mode_workers AS (
        SELECT cw.*
```

```
        FROM   cat_workers cw
        WHERE  cw.cw_work_mode = v_work_mode
           OR  cw.cw_work_mode = 'HYBRID'
           OR  v_work_mode     = 'HYBRID'
    ),

    location_workers AS (
        SELECT
            mw.cw_worker_id,
            mw.cw_user_id,
            mw.cw_work_mode,
            mw.cw_radius,
            l.city              AS lw_city,
            CASE
                WHEN v_work_mode = 'REMOTE' THEN NULL
                ELSE ROUND((
                    6371 * 2 * ASIN(SQRT(
                        POWER(SIN(RADIANS(v_task_lat - l.latitude)  / 2), 2)
                        + COS(RADIANS(l.latitude))
                        * COS(RADIANS(v_task_lat))
                        * POWER(SIN(RADIANS(v_task_lon - l.longitude) / 2), 2)
                    ))
                )::NUMERIC, 2)
            END                 AS lw_distance_km
        FROM  mode_workers mw
        JOIN  Location     l  ON l.id = mw.cw_location_id
        WHERE v_work_mode = 'REMOTE'
           OR (
               6371 * 2 * ASIN(SQRT(
                   POWER(SIN(RADIANS(v_task_lat - l.latitude)  / 2), 2)
                   + COS(RADIANS(l.latitude))
                   * COS(RADIANS(v_task_lat))
                   * POWER(SIN(RADIANS(v_task_lon - l.longitude) / 2), 2)
               )) <= mw.cw_radius
           )
    ),

    worker_categories AS (
    SELECT wc.worker_id AS wcat_worker_id,
           ARRAY_AGG(c.category_name::TEXT ORDER BY c.category_name) AS
wcat_categories
    FROM WorkerCategory wc
    JOIN Category c ON c.id = wc.category_id
    WHERE wc.worker_id IN (
        SELECT w.id FROM Worker w
        JOIN WorkerCategory wc2 ON wc2.worker_id = w.id
        WHERE wc2.category_id = v_category_id
    )
    GROUP BY wc.worker_id
),

    rating_stats AS (
        SELECT
            r.reviewed_id           AS rs_user_id,
            ROUND(AVG(r.rating), 2) AS rs_avg_rating,
            COUNT(r.id)             AS rs_total_reviews
        FROM  Review r
        GROUP BY r.reviewed_id
    ),

    best_badge AS (
        SELECT DISTINCT ON (wb.worker_id)
```

```
            wb.worker_id    AS bb_worker_id,
            b.badge_name    AS bb_badge_name,
            b.tier_level    AS bb_tier_level
        FROM  WorkerBadge wb
        JOIN  Badge       b  ON b.id = wb.badge_id
        WHERE wb.is_active   = TRUE
          AND b.category_id  = v_parent_cat_id
        ORDER BY wb.worker_id, b.tier_level DESC
    ),

    active_load AS (
    SELECT o.worker_id AS al_worker_id, COUNT(t.id) AS al_active_tasks
    FROM Task t
    JOIN Offer o ON o.id = t.offer_id
    WHERE t.status = 'ACTIVE'
      AND o.worker_id IN (
          SELECT w.id FROM Worker w
          JOIN WorkerCategory wc ON wc.worker_id = w.id
          WHERE wc.category_id = v_category_id
      )
    GROUP BY o.worker_id
),



    scored AS (
        SELECT
            lw.cw_worker_id                                 AS sc_worker_id,
            lw.cw_user_id                                   AS sc_user_id,
            u.name                                          AS sc_name,
            u.surname                                       AS sc_surname,
            lw.lw_city                                      AS sc_city,
            lw.cw_work_mode                                 AS sc_work_mode,
            lw.cw_radius                                    AS sc_radius,
            lw.lw_distance_km                               AS sc_distance_km,
            COALESCE(wcat.wcat_categories,  '{}')           AS sc_categories,
            COALESCE(rs.rs_avg_rating,      0)              AS sc_avg_rating,
            COALESCE(rs.rs_total_reviews,   0)              AS sc_total_reviews,
            CASE
                WHEN COALESCE(rs.rs_avg_rating, 0) >= 4.5 THEN 'TOP'
                WHEN COALESCE(rs.rs_avg_rating, 0) >= 3.5 THEN 'MEDIUM'
                ELSE 'LOW'
            END::VARCHAR                                    AS sc_performance_level,
            bb.bb_badge_name                                AS sc_active_badge,
            bb.bb_tier_level                                AS sc_badge_tier,
            COALESCE(al.al_active_tasks,    0)              AS sc_active_tasks,

            ROUND((

                COALESCE(rs.rs_avg_rating, 0) / 5.0 * 50.0

                + LEAST(LN(COALESCE(rs.rs_total_reviews, 0) + 1) * 2.5, 10.0)

                + COALESCE(bb.bb_tier_level, 0) * 3.0



                + CASE
```

```
                    WHEN v_work_mode = 'REMOTE' OR lw.lw_distance_km IS NULL
THEN 10.0
                    WHEN lw.cw_radius = 0 THEN 0.0
                    ELSE GREATEST(0.0,
                        (1.0 - lw.lw_distance_km / NULLIF(lw.cw_radius, 0)) *
10.0
                    )
                END


                - LEAST(COALESCE(al.al_active_tasks, 0) * 3.0, 15.0)

            )::NUMERIC, 2)                                AS sc_match_score

        FROM       location_workers   lw
        JOIN       UserAccount        u    ON u.id             = lw.cw_user_id
        LEFT JOIN  worker_categories  wcat ON wcat.wcat_worker_id =
lw.cw_worker_id
        LEFT JOIN  rating_stats       rs   ON rs.rs_user_id       =
lw.cw_user_id
        LEFT JOIN  best_badge         bb   ON bb.bb_worker_id     =
lw.cw_worker_id
        LEFT JOIN  active_load        al   ON al.al_worker_id     =
lw.cw_worker_id
    )

    SELECT
        sc_worker_id,
        sc_user_id,
        sc_name,
        sc_surname,
        sc_city,
        sc_work_mode,
        sc_radius,
        sc_distance_km,
        sc_categories,
        sc_avg_rating,
        sc_total_reviews,
        sc_performance_level,
        sc_active_badge,
        sc_badge_tier,
        sc_active_tasks,
        sc_match_score
    FROM  scored
    ORDER BY sc_match_score DESC
    LIMIT p_limit;

END;
$$;
```

Triggers:

1. fn_set_updated_at / trg_*_updated_at - A shared trigger function that automatically sets the updated_at column to the current timestamp before any UPDATE on the registered tables. Applied to five tables via individual trigger definitions.

```sql
CREATE OR REPLACE FUNCTION fn_set_updated_at()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    NEW.updated_at = CURRENT_TIMESTAMP;
    RETURN NEW;
END;
$$;

CREATE OR REPLACE TRIGGER trg_worker_updated_at
    BEFORE UPDATE ON Worker
    FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

CREATE OR REPLACE TRIGGER trg_useraccount_updated_at
    BEFORE UPDATE ON UserAccount
    FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

CREATE OR REPLACE TRIGGER trg_taskrequest_updated_at
    BEFORE UPDATE ON TaskRequest
    FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();


CREATE OR REPLACE TRIGGER trg_offer_updated_at
    BEFORE UPDATE ON Offer
    FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();

CREATE OR REPLACE TRIGGER trg_payment_updated_at
    BEFORE UPDATE ON Payment
    FOR EACH ROW EXECUTE FUNCTION fn_set_updated_at();
```

2. trg_notification_created_at - Enforces temporal integrity by ensuring that a Notification's created_at timestamp is strictly after the created_at timestamp of the associated UserAccount. Prevents back-dated notifications from being inserted or updated into the table.

```sql
CREATE OR REPLACE FUNCTION check_notification_created_at()
RETURNS TRIGGER AS $$
BEGIN
    IF NEW.created_at <= (SELECT created_at FROM UserAccount WHERE id =
NEW.user_id) THEN
        RAISE EXCEPTION 'notification created_at must be after user created_at';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_notification_created_at
    BEFORE INSERT OR UPDATE ON Notification
    FOR EACH ROW
    EXECUTE FUNCTION check_notification_created_at();
```