Ride Sharing Application

Kakaraparty Srirama Srikar	142301013
Kalava Dheeraj Ram	142301015
Suryadevara Eswar Sai Ramakrishna	142301034

Relations

`users` : `riders` : "1:1 (A rider is a user)"

`users` : `drivers` : "1:1 (A driver is a user)"

`users` : `ratings` : "1:M (A user can give multiple ratings)"

`users` : `payments` : "1:M (A user can make multiple payments)"

`riders` : `ride_requests` : "1:M (A user can request multiple rides)"

`drivers` : `rides` : "1:M (A driver can complete multiple rides)"

`drivers` : `ride_pooling` : "1:M (A driver can pool multiple rides)"

`drivers` : `ratings` : "1:M (A driver can be rated multiple times)"

`drivers` : `driver_location` : "1:1 (Each driver has a current location)"

`riders` : `rides` : "1:M (A rider can take multiple rides)"

`rides` : `ride_pooling` : "M:1 (A ride can belong to a pool)"

`rides` : `ratings` : "1:M (A ride can have multiple ratings)"

`rides` : `payments` : "1:1 (A ride has one payment record)"

`ride_requests` : `rides` : "1:1 (A request turns into a ride)"

Relational Model

1. `users` table

```
CREATE TABLE users (
    user_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    phone VARCHAR(15) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash TEXT NOT NULL,
    role VARCHAR(20) CHECK (role IN ('rider', 'driver', 'admin')) DEFAULT 'rider',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- Stores all users (riders, drivers, admins).
- Role determines whether the user is a rider or driver.

2. `riders` table

```
CREATE TABLE riders (
    rider_id SERIAL PRIMARY KEY,
    user_id INT UNIQUE REFERENCES users(user_id) ON DELETE CASCADE,
    preferred_payment_method VARCHAR(50),
    preferred_ride_type VARCHAR(50),
    ride_count INT DEFAULT 0,
```

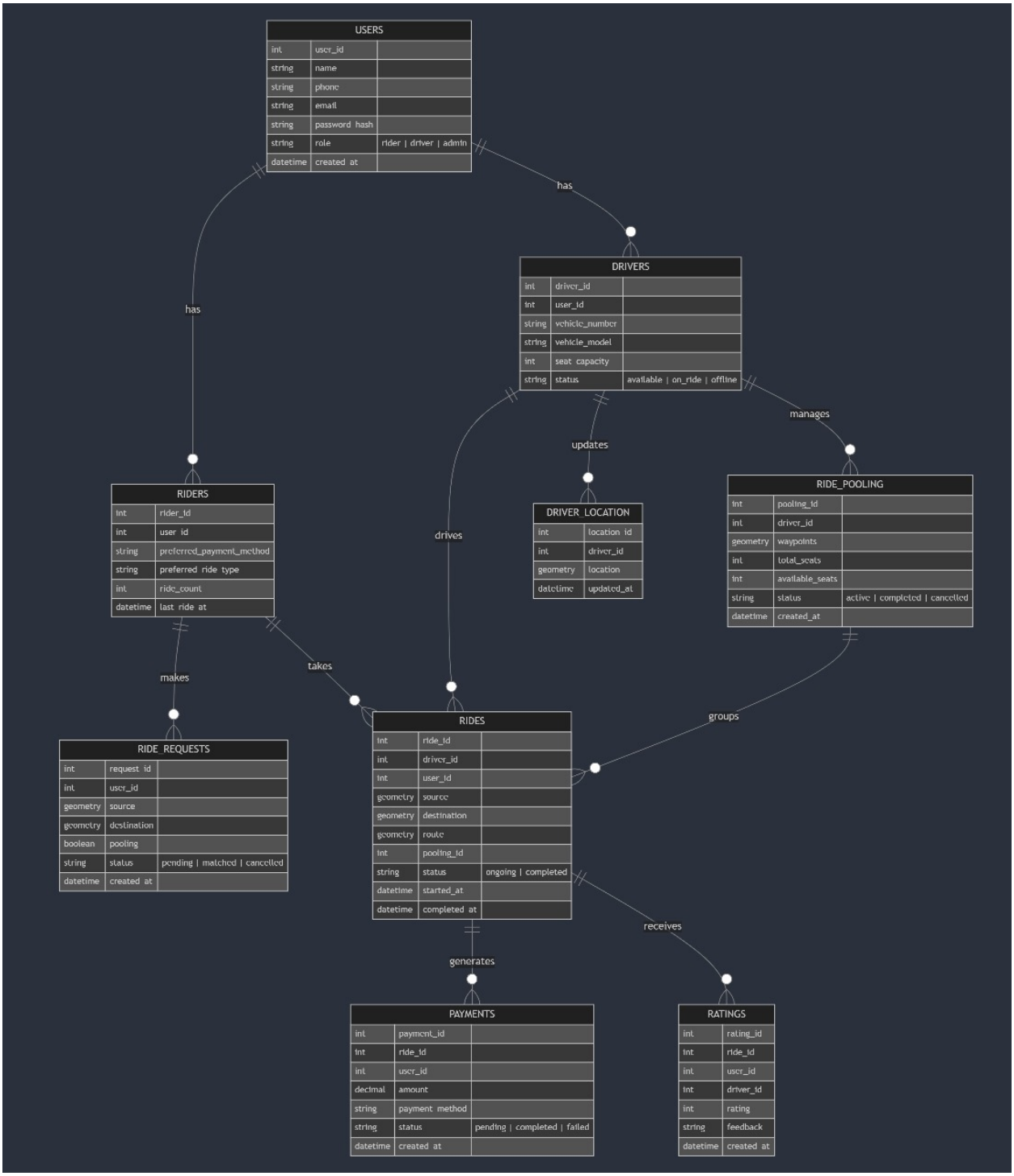

Figure 1: ER Diagram

```
    last_ride_at TIMESTAMP
);
```

- Stores riders' preferences and ride history.
- One-to-One (1:1) with `users` (each rider is a user).

3. `drivers` table

```
CREATE TABLE drivers (
    driver_id SERIAL PRIMARY KEY,
    user_id INT UNIQUE REFERENCES users(user_id) ON DELETE CASCADE,
    vehicle_number VARCHAR(20) UNIQUE NOT NULL,
    vehicle_model VARCHAR(50) NOT NULL,
    seat_capacity INT NOT NULL CHECK (seat_capacity > 0),
    status VARCHAR(20) CHECK (status IN ('available', 'on_ride', 'offline')) DEFAULT 'available'
);
```

- Stores driver details, including **car information**.
- One-to-One (1:1) with `users` (each driver is a user).

4. `driver_location` table

```
-- Driver location tracking
CREATE TABLE driver_location (
    location_id SERIAL PRIMARY KEY,
    driver_id INT REFERENCES drivers(driver_id) ON DELETE CASCADE,
    location GEOMETRY(Point, 4326) NOT NULL,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- Tracks real-time driver location.
- One-to-One (1:1) with `drivers`.
- It is updated frequently to get the accurate location.

5. `ride_requests` table

```
CREATE TABLE ride_requests (
    request_id SERIAL PRIMARY KEY,
    rider_id INT REFERENCES riders(rider_id) ON DELETE CASCADE,
    source GEOMETRY(Point, 4326) NOT NULL,
    destination GEOMETRY(Point, 4326) NOT NULL,
    pooling BOOLEAN DEFAULT FALSE,
    status VARCHAR(20) CHECK (status IN ('pending', 'matched', 'cancelled')) DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- Stores ride requests before a ride is assigned.
- One-to-Many (1:M) with `users` (a user can request multiple rides).

6. `rides` table

```
CREATE TABLE rides (
    ride_id SERIAL PRIMARY KEY,
    driver_id INT REFERENCES drivers(driver_id) ON DELETE CASCADE,
    rider_id INT REFERENCES riders(rider_id) ON DELETE CASCADE, -- Linked to riders, not users
    source GEOMETRY(Point, 4326) NOT NULL,
    destination GEOMETRY(Point, 4326) NOT NULL,
    route GEOMETRY(LINESTRING, 4326),
    pooling_id INT REFERENCES ride_pooling(pooling_id) ON DELETE SET NULL DEFAULT -1,
    status VARCHAR(20) CHECK (status IN ('ongoing', 'completed', 'cancelled')) DEFAULT 'ongoing',
```

```
    started_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    completed_at TIMESTAMP
);
```

- Stores actual rides.
- One-to-Many (1:M) with `drivers` (one driver can have multiple rides).
- One-to-Many (1:M) with `users` (one user can take multiple rides).
- Many-to-One (M:1) with `ride_pooling` (multiple rides can belong to a single pool).

7. `ride_pooling` table

```
CREATE TABLE ride_pooling (
    pooling_id SERIAL PRIMARY KEY,
    driver_id INT REFERENCES drivers(driver_id) ON DELETE CASCADE,
    waypoints GEOMETRY(LINESTRING, 4326),
    total_seats INT NOT NULL CHECK (total_seats > 0),
    available_seats INT NOT NULL CHECK (available_seats >= 0),
    status VARCHAR(20) CHECK (status IN ('active', 'completed', 'cancelled')) DEFAULT 'active',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- Stores shared rides where multiple riders share a driver.
- One-to-Many (1:M) with `rides` (one pooling session can have multiple rides).
- One-to-One (1:1) with `drivers` (each pooling is assigned to one driver).

8. `ratings` table

```
CREATE TABLE ratings (
    rating_id SERIAL PRIMARY KEY,
    ride_id INT REFERENCES rides(ride_id) ON DELETE CASCADE,
    rider_id INT REFERENCES riders(rider_id) ON DELETE CASCADE,
    driver_id INT REFERENCES drivers(driver_id) ON DELETE CASCADE,
    rating INT CHECK (rating BETWEEN 1 AND 5),
    feedback TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- Stores ride reviews and ratings.
- One-to-Many (1:M) with `rides` (one ride can have multiple ratings).
- One-to-Many (1:M) with `drivers` (a driver can receive multiple ratings).

9. `payments` table

```
CREATE TABLE payments (
    payment_id SERIAL PRIMARY KEY,
    ride_id INT REFERENCES rides(ride_id) ON DELETE CASCADE,
    rider_id INT REFERENCES riders(rider_id) ON DELETE CASCADE,
    amount DECIMAL(10,2) NOT NULL,
    payment_method VARCHAR(50) NOT NULL,
    status VARCHAR(20) CHECK (status IN ('pending', 'completed', 'failed')) DEFAULT 'pending',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

- Stores payment details for completed rides.
- One-to-One (1:1) with `rides` (each ride has one payment).
- One-to-Many (1:M) with `users` (a user can make multiple payments)

Functions

1. `request_ride` function

```
CREATE OR REPLACE FUNCTION request_ride(
    p_rider_id INT,
    p_source GEOMETRY(Point, 4326),
    p_destination GEOMETRY(Point, 4326),
    p_pooling BOOLEAN
) RETURNS INT AS $$
DECLARE
    v_request_id INT;
BEGIN
    INSERT INTO ride_requests (rider_id, source, destination, pooling, status, created_at)
    VALUES (p_rider_id, p_source, p_destination, p_pooling, 'pending', NOW())
    RETURNING request_id INTO v_request_id;

    RETURN v_request_id;
END;
$$ LANGUAGE plpgsql;
```

2. `find_nearest_driver` function

```
CREATE OR REPLACE FUNCTION find_nearest_driver(v_source GEOMETRY(Point, 4326))
RETURNS INT AS $$
DECLARE
    v_driver_id INT;
BEGIN
    SELECT d.driver_id
    INTO v_driver_id
    FROM drivers d
    JOIN driver_location dl ON d.driver_id = dl.driver_id
    WHERE d.status = 'available'
    ORDER BY ST_Distance(dl.location, v_source) ASC
    LIMIT 1;

    IF NOT FOUND THEN
        RETURN NULL;
    END IF;

    RETURN v_driver_id;
END;
$$ LANGUAGE plpgsql;
```

3. `match_ride` function:

```
CREATE OR REPLACE FUNCTION match_ride(v_rider_id INT)
RETURNS INT AS $$
DECLARE
    v_driver_id INT;
    v_source GEOMETRY(Point, 4326);
    v_destination GEOMETRY(Point, 4326);
    v_pooling BOOLEAN;
    v_pooling_id INT;
    v_existing_pooling INT;
    v_available_seats INT;
    v_ride_id INT;
```

```
BEGIN
    -- Get ride request details
    SELECT source, destination, pooling
    INTO v_source, v_destination, v_pooling
    FROM ride_requests
    WHERE rider_id = v_rider_id AND status = 'pending'
    LIMIT 1;

    -- Check if the ride request exists
    IF NOT FOUND THEN
        RAISE EXCEPTION 'No pending ride request found for rider %', v_rider_id;
    END IF;

    -- Case 1: Handling Ride Pooling
    IF v_pooling THEN
        -- Check for an active pooling session with available seats
        SELECT pooling_id, driver_id, available_seats
        INTO v_existing_pooling, v_driver_id, v_available_seats
        FROM ride_pooling
        WHERE status = 'active' AND available_seats > 0
        ORDER BY created_at ASC  -- Prioritize the oldest active pooling session
        LIMIT 1;

        -- If no existing pooling session is found, create a new one
        IF v_existing_pooling IS NULL THEN
            -- Find an available driver for pooling
            SELECT driver_id
            INTO v_driver_id
            FROM drivers
            WHERE status = 'available'
            ORDER BY random()  -- Pick a random available driver
            LIMIT 1;

            -- If no available driver found
            IF v_driver_id IS NULL THEN
                RAISE EXCEPTION 'No available drivers for ride pooling';
            END IF;

            -- Create a new pooling session
            INSERT INTO ride_pooling (driver_id, waypoints, total_seats, available_seats, status)
            VALUES (v_driver_id, ST_GeomFromText('LINESTRING EMPTY', 4326), 4, 3, 'active') -- Assuming
            RETURNING pooling_id INTO v_pooling_id;
        ELSE
            -- Assign to existing pooling session
            v_pooling_id := v_existing_pooling;

            -- Reduce available seats
            UPDATE ride_pooling
            SET available_seats = available_seats - 1
            WHERE pooling_id = v_pooling_id;
        END IF;
    ELSE
        -- Case 2: Handling Regular (Non-Pooling) Rides
        v_pooling_id := NULL;
```

```
        -- Find the nearest available driver
        SELECT driver_id
        INTO v_driver_id
        FROM driver_location
        WHERE ST_DWithin(location, v_source, 5000) -- Find drivers within 5km range
        ORDER BY ST_Distance(location, v_source) ASC
        LIMIT 1;

        -- If no nearby driver found
        IF v_driver_id IS NULL THEN
            RAISE EXCEPTION 'No available drivers nearby';
        END IF;

        -- Update driver status
        UPDATE drivers SET status = 'on_ride' WHERE driver_id = v_driver_id;
    END IF;

    -- Insert the ride into the rides table
    INSERT INTO rides (driver_id, rider_id, source, destination, pooling_id, status, started_at)
    VALUES (v_driver_id, v_rider_id, v_source, v_destination, v_pooling_id, 'ongoing', NOW())
    RETURNING ride_id INTO v_ride_id;

    -- Mark ride request as matched
    UPDATE ride_requests
    SET status = 'matched'
    WHERE rider_id = v_rider_id;

    RETURN v_ride_id;
END;
$$ LANGUAGE plpgsql;
```

4. `complete_ride` function

```
CREATE OR REPLACE FUNCTION complete_ride(v_ride_id INT)
RETURNS VOID AS $$
DECLARE
    v_driver_id INT;
    v_pooling_id INT;
    v_remaining_rides INT;
    v_ride_status VARCHAR(20);
BEGIN
    -- Get ride details
    SELECT driver_id, pooling_id, status
    INTO v_driver_id, v_pooling_id, v_ride_status
    FROM rides
    WHERE ride_id = v_ride_id;

    -- Check if ride exists
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Ride ID % not found', v_ride_id;
    END IF;

    -- Ensure the ride is ongoing before completing
    IF v_ride_status <> 'ongoing' THEN
```

```
        RAISE EXCEPTION 'Ride % is already completed or cancelled', v_ride_id;
    END IF;

    -- Mark the ride as completed
    UPDATE rides
    SET status = 'completed', completed_at = NOW()
    WHERE ride_id = v_ride_id;

    -- If the ride was part of pooling, check if it's the last ride in that pool
    IF v_pooling_id IS NOT NULL THEN
        SELECT COUNT(*)
        INTO v_remaining_rides
        FROM rides
        WHERE pooling_id = v_pooling_id AND status = 'ongoing';

        -- If no more active rides remain in this pool, mark the pooling session as completed
        IF v_remaining_rides = 0 THEN
            UPDATE ride_pooling
            SET status = 'completed'
            WHERE pooling_id = v_pooling_id;
        END IF;
    END IF;

    -- Check if the driver has any ongoing rides
    SELECT COUNT(*)
    INTO v_remaining_rides
    FROM rides
    WHERE driver_id = v_driver_id AND status = 'ongoing';

    -- If no more ongoing rides, mark the driver as available
    IF v_remaining_rides = 0 THEN
        UPDATE drivers
        SET status = 'available'
        WHERE driver_id = v_driver_id;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

Triggers and Trigger Functions

Update Rider Stats After Ride

```
CREATE OR REPLACE FUNCTION update_rider_stats_on_ride()
RETURNS TRIGGER AS $$
BEGIN
  UPDATE core_rider
  SET
    ride_count = ride_count + 1,
    last_ride_at = CURRENT_TIMESTAMP
  WHERE rider_id = NEW.rider_id;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_update_rider_stats
```

```
AFTER INSERT ON core_ride
FOR EACH ROW
EXECUTE PROCEDURE update_rider_stats_on_ride();
```

Explanation: This trigger automatically updates the core_rider table whenever a new ride is recorded in the core_ride table. It increments the rider's ride_count and stores the timestamp of their last ride.Benefit: This ensures that rider statistics are always up-to-date, which can be useful for loyalty programs, user profiles, and data analysis.Validate Pooling SeatsCREATE OR REPLACE FUNCTION validate_pooling_seats()

Validating the number of seats

```
RETURNS TRIGGER AS $$
BEGIN
  IF NEW.available_seats > NEW.total_seats THEN
    RAISE EXCEPTION 'Available seats cannot exceed total seats';
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_validate_seats
BEFORE INSERT OR UPDATE ON core_ridepooling
FOR EACH ROW
EXECUTE FUNCTION validate_pooling_seats();
```

Explanation: This trigger prevents invalid data from being inserted or updated in the core_ridepooling table. It checks if the number of available_seats exceeds the total_seats and raises an exception if it does.Benefit: This enforces data integrity, ensuring that the database always reflects a realistic state of seat availability in ride pooling scenarios.

Log Ride Status Changes

```
CREATE TABLE ride_status_log (
  log_id SERIAL PRIMARY KEY,
  ride_id INT REFERENCES core_ride(id),
  old_status VARCHAR(20),
  new_status VARCHAR(20),
  changed_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE OR REPLACE FUNCTION log_ride_status_change()
RETURNS TRIGGER AS $$
BEGIN
  INSERT INTO ride_status_log (ride_id, old_status, new_status)
  VALUES (NEW.id, OLD.status, NEW.status);
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_log_ride_status_change
AFTER UPDATE ON core_ride
FOR EACH ROW
EXECUTE FUNCTION log_ride_status_change();
```

Purpose: To maintain a history of how a ride's status changes over time. It is useful for auditing, debugging, and analyzing ride progression. You can track how often rides are canceled, delayed, or reassigned.Enforce

Business Rules

Calculate Ride Fare

```
CREATE OR REPLACE FUNCTION calculate_ride_fare()
RETURNS TRIGGER AS $$
DECLARE
  calculated_fare DECIMAL(10, 2);
  distance_in_km DECIMAL(10, 2);  --  You'd need a way to store distance
  start_time TIMESTAMP;
  end_time TIMESTAMP;
  duration_in_minutes INTEGER;
BEGIN
  IF NEW.status = 'completed' AND OLD.status <> 'completed' THEN

    --  distance_in_km := get_distance(OLD.source, NEW.destination);
    distance_in_km := 10.5;  --  Placeholder

    -- Get ride start and end times
    SELECT start_time, end_time INTO start_time, end_time
    FROM core_ride
    WHERE id = NEW.id;

    duration_in_minutes := EXTRACT(EPOCH FROM (end_time - start_time)) / 60;

    calculated_fare := (distance_in_km * 2.00) + (duration_in_minutes * 0.25);

    UPDATE core_ride
    SET
      fare = calculated_fare
    WHERE id = NEW.id;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trg_calculate_ride_fare
AFTER UPDATE ON core_ride
FOR EACH ROW
EXECUTE FUNCTION calculate_ride_fare();
```

Purpose: To automatically calculate the ride fare when the ride is completed. It ensures that fares are calculated consistently and accurately, reducing the risk of errors. Offloads fare calculation from the application to the database.

Indices, Views and Roles

Here are the index definitions in the Django models, the corresponding SQL index creation statements (created in the Django migrations), database views for simplified data access, and the implementation of role-based access control for enhanced uses.

A. Django Model Index Definitions

The following index definitions are implemented within the Django models to optimize database performance for specific query patterns.

```python
from django.contrib.postgres.indexes import GistIndex
from django.db import models

class Ride(models.Model):
    ## ... other fields ...
    class Meta:
        indexes = [
            GistIndex(fields=["source"], name="core_ride_source_36a219_gist"),
            GistIndex(fields=["destination"], name="core_ride_destina_3c7ee9_gist"),
        ]

class RideRequest(models.Model):
    ## ... other fields ...
    class Meta:
        indexes = [
            models.Index(fields=["status"], name="core_ridere_status_17c131_idx"),
        ]
```

Details:

1. **`Ride` Model Indices:**
   - **`GistIndex(fields=["source"], name="core_ride_source_36a219_gist")`**: This defines a GiST (Generalized Search Tree) index on the `source` field of the `Ride` model. The `source` field is likely a geospatial data type (e.g., `PointField` from `django.contrib.gis.db.models`).
     - **Purpose:** GiST indices are highly efficient for geospatial queries such as finding rides within a certain radius of a given point or determining if a point lies within a specific area. This index significantly speeds up searches based on the ride's origin.
     - **Technology:** GiST is a balanced tree structure that allows for the implementation of various indexing strategies, making it suitable for non-scalar data types like geometric shapes.
   - **`GistIndex(fields=["destination"], name="core_ride_destina_3c7ee9_gist")`**: Similarly, this creates a GiST index on the `destination` field of the `Ride` model.
     - **Purpose:** This index optimizes geospatial queries related to the ride's drop-off location, enabling efficient searches for rides ending in a particular area.
2. **`RideRequest` Model Index:**
   - **`models.Index(fields=["status"], name="core_ridere_status_17c131_idx")`**: This defines a standard B-tree index on the `status` field of the `RideRequest` model.
     - **Purpose:** B-tree indices are highly effective for equality and range-based queries. In this case, it allows for rapid filtering of ride requests based on their status (e.g., selecting all 'pending' requests). This is crucial for managing and processing ride requests efficiently.
     - **Technology:** B-trees are self-balancing tree structures that maintain sorted data, allowing for logarithmic time complexity for search, insertion, and deletion operations.

B. Corresponding SQL Index Creation Statements

These SQL statements are the direct translation of the Django model index definitions and are executed on the PostgreSQL database.

```sql
-- 1. GIST Index on Ride.source (for fast geospatial queries on pickup location)
CREATE INDEX core_ride_source_36a219_gist
ON core_ride
USING GIST (source);

-- 2. GIST Index on Ride.destination (for fast geospatial queries on drop-off location)
CREATE INDEX core_ride_destina_3c7ee9_gist
ON core_ride
USING GIST (destination);
```

```sql
-- 3. B-tree Index on RideRequest.status (for efficient status-based filtering)
CREATE INDEX core_ridere_status_17c131_idx
ON core_riderequest (status);
```

Details:

- `CREATE INDEX <index_name> ON <table_name> USING <index_type> (<column_name>);`: This is the standard SQL syntax for creating an index.
- `USING GIST`: Specifies that a GiST index should be created. This is essential for indexing geospatial data types.
- `USING (default)`: When no specific index type is mentioned (as in the third statement), PostgreSQL defaults to using a B-tree index.

C. Views

Database views are virtual tables based on the result of an SQL statement. They simplify complex queries and provide a more abstract way to access data.

```sql
-- 1. Pending Ride Requests
CREATE OR REPLACE VIEW pending_ride_requests_view AS
SELECT rr.id as request_id, u.name AS rider_name, rr.source, rr.destination, rr.pooling, rr.created_at
FROM core_riderequest rr
JOIN core_rider r ON rr.rider_id = r.id
JOIN core_user u ON r.user_id = u.id
WHERE rr.status = 'pending';

-- 2. Active Pooling Summary
CREATE OR REPLACE VIEW active_pooling_summary AS
SELECT rp.id as pooling_id, d.id, u.name AS driver_name, rp.available_seats, rp.total_seats, rp.status,
FROM core_ridepooling rp
JOIN core_driver d ON rp.driver_id = d.id
JOIN core_user u ON d.user_id = u.id
WHERE rp.status = 'active';

-- 3. Driver Status Board
CREATE OR REPLACE VIEW driver_status_board AS
SELECT d.id, u.name AS driver_name, d.vehicle_model, d.vehicle_number, d.status
FROM core_driver d
JOIN core_user u ON d.user_id = u.id;
```

Details:

1. `pending_ride_requests_view`:
   - **Purpose:** This view provides a simplified way to retrieve all pending ride requests along with the rider's name.
   - **Functionality:** It joins the `core_riderequest`, `core_rider`, and `core_user` tables to select relevant information and filters the results to include only requests where the `status` is 'pending'.
   - **Benefits:** This view avoids the need to write complex JOIN statements every time pending ride requests and rider names are required, making queries cleaner and easier to understand.
2. `active_pooling_summary`:
   - **Purpose:** This view offers a summary of all active ride pooling instances, including driver details and seat availability.
   - **Functionality:** It joins the `core_ridepooling`, `core_driver`, and `core_user` tables to gather information about active pooling rides (where `status` is 'active').
   - **Benefits:** This view centralizes the logic for retrieving active pooling information, useful for dashboards or matching algorithms that consider active pools.

3. `driver_status_board`:
    - **Purpose:** This view presents a snapshot of the current status of all drivers, including their name, vehicle details, and operational status.
    - **Functionality:** It joins the `core_driver` and `core_user` tables to display driver-specific information.
    - **Benefits:** This view provides a readily accessible overview of the driver fleet's status, which can be valuable for monitoring and dispatching.

D. Roles

Role-Based Access Control (RBAC) is implemented using PostgreSQL roles to manage permissions and ensure data security.

```
DO $$
BEGIN
IF NOT EXISTS (SELECT FROM pg_roles WHERE rolname = 'rider_role') THEN
CREATE ROLE rider_role;
END IF;
IF NOT EXISTS (SELECT FROM pg_roles WHERE rolname = 'driver_role') THEN
CREATE ROLE driver_role;
END IF;
IF NOT EXISTS (SELECT FROM pg_roles WHERE rolname = 'admin_role') THEN
CREATE ROLE admin_role;
END IF;
END
$$;

GRANT SELECT, INSERT ON core_riderequest TO rider_role;
GRANT SELECT ON pending_ride_requests_view TO rider_role;
GRANT SELECT, UPDATE ON core_driverlocation TO driver_role;
GRANT SELECT ON driver_status_board TO driver_role;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO admin_role;
GRANT ALL ON ALL SEQUENCES IN SCHEMA public TO admin_role;
```

Details:

1. **Role Creation:**
    - The `DO $$ BEGIN ... END $$;` block ensures that the roles (`rider_role`, `driver_role`, `admin_role`) are created only if they do not already exist in the PostgreSQL database. This prevents errors if the script is executed multiple times.
    - `CREATE ROLE <role_name>;`: This SQL command creates a new database role. Roles can be granted specific permissions on database objects.
2. **Granting Permissions:**
    - `GRANT <privileges> ON <database_object> TO <role>;`: This command grants the specified privileges to the given role on the specified database object (table, view, sequence, etc.).
    - `rider_role`:
        - `GRANT SELECT, INSERT ON core_riderequest TO rider_role;`: Riders are allowed to view and create new ride requests.
        - `GRANT SELECT ON pending_ride_requests_view TO rider_role;`: Riders can view the list of pending ride requests.
    - `driver_role`:
        - `GRANT SELECT, UPDATE ON core_driverlocation TO driver_role;`: Drivers can view and update their location information.
        - `GRANT SELECT ON driver_status_board TO driver_role;`: Drivers can view the driver status board.
    - `admin_role`:

– `GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public TO admin_role;`: Administrators have full CRUD (Create, Read, Update, Delete) access to all tables within the public schema.
– `GRANT ALL ON ALL SEQUENCES IN SCHEMA public TO admin_role;`: Administrators have all privileges on all sequences in the public schema, which are typically used for generating unique identifiers.

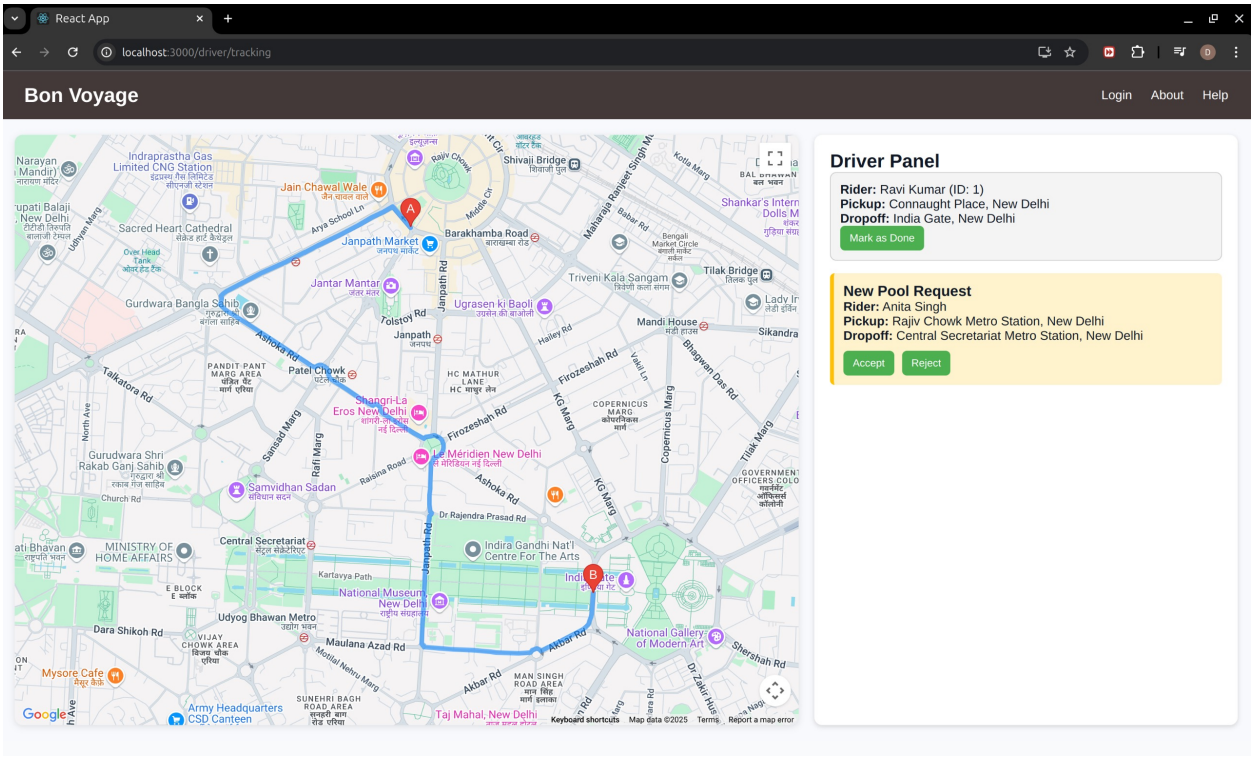

Figure 2: Route before accepting

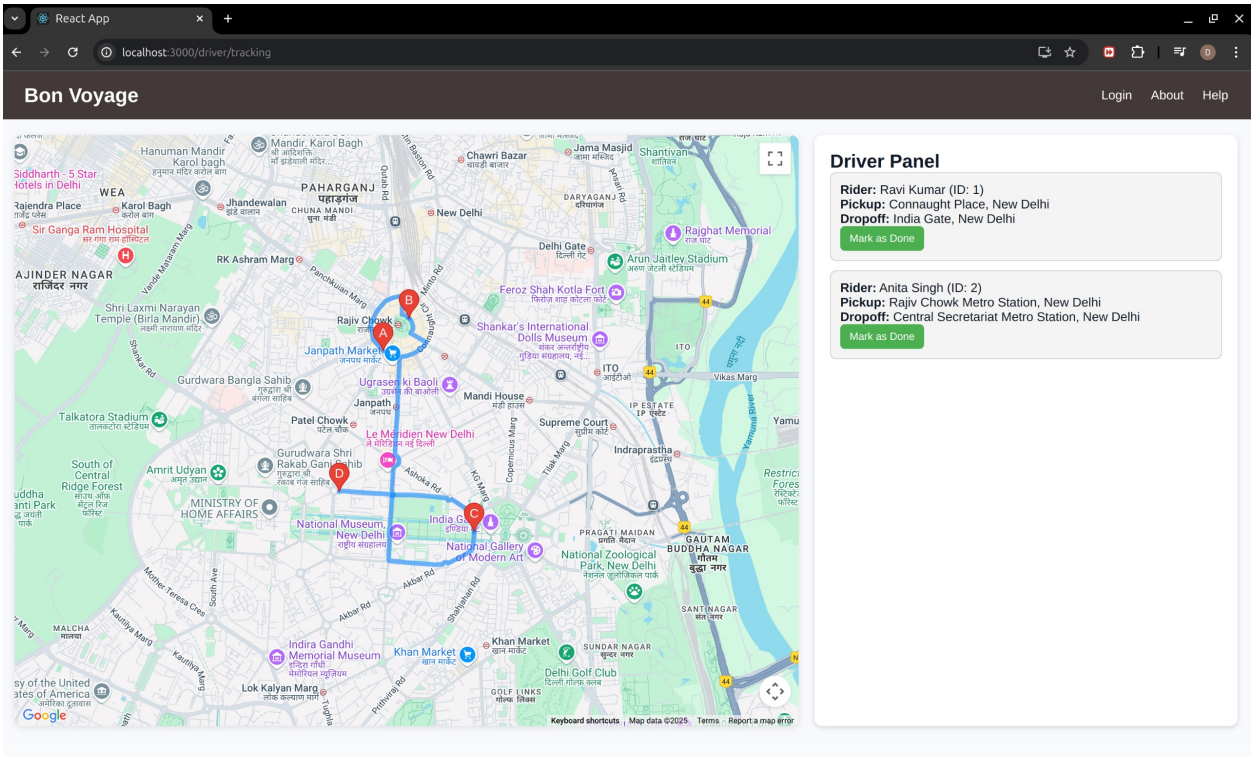

Figure 3: Route after accepting

Function Output

```
139  select * from users;
140
```

Data Output | Messages | Notifications

	user_id [PK] integer	name character varying (100)	phone character varying (15)	email character varying (100)	password_hash text	role character varying (20)	created_at timestamp without time zone
1	1	Rider 1	9111111111	rider1@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
2	2	Rider 2	9222222222	rider2@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
3	3	Rider 3	9333333333	rider3@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
4	4	Rider 4	9123456789	rider4@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
5	5	Rider 5	9876543210	rider5@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
6	6	Rider 6	9087654321	rider6@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
7	7	Rider 7	8901234567	rider7@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
8	8	Rider 8	8765432109	rider8@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
9	9	Rider 9	9999999999	rider9@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
10	10	Rider 10	9000000000	rider10@example.com	hashed_password	rider	2025-03-03 23:03:44.050009
11	11	Driver 1	8111111111	driver1@example.com	hashed_password	driver	2025-03-03 23:03:44.050009
12	12	Driver 2	8222222222	driver2@example.com	hashed_password	driver	2025-03-03 23:03:44.050009
13	13	Driver 3	8911111111	driver3@example.com	hashed_password	driver	2025-03-03 23:03:44.050009
14	14	Driver 4	8722222222	driver4@example.com	hashed_password	driver	2025-03-03 23:03:44.050009
15	15	Driver 5	8611111111	driver5@example.com	hashed_password	driver	2025-03-03 23:03:44.050009
16	16	Driver 6	8522222222	driver6@example.com	hashed_password	driver	2025-03-03 23:03:44.050009

This is the initial users table

```
156  select * from riders;
```

Data Output | Messages | Notifications

	rider_id [PK] integer	user_id integer	preferred_payment_method character varying (50)	preferred_ride_type character varying (50)	ride_count integer	last_ride_at timestamp without time zone
1	1	1	credit_card	standard	0	[null]
2	2	2	upi	standard	0	[null]
3	3	3	upi	pooled	0	[null]
4	4	4	upi	pooled	0	[null]
5	5	6	upi	pooled	0	[null]
6	6	9	cash	pooled	0	[null]

This is the riders table. We have a few riders from the users listed for testing purposes.

```
169  select * from drivers;
170
```

Data Output | Messages | Notifications

	driver_id [PK] integer	user_id integer	vehicle_number character varying (20)	vehicle_model character varying (50)	seat_capacity integer	status character varying (20)
1	1	11	ABC123	Toyota Prius	4	available
2	2	12	ABC567	Ford Figo	4	available
3	3	13	ABC890	Honda i20	4	available
4	4	14	XYZ123	Suzuki Ciaz	4	available
5	5	15	ABC321	Toyota Prius	4	available
6	6	16	XYZ789	Honda Civic	4	available

This is the drivers table

```
206  select * from driver_location;
207
```

Data Output | Messages | Notifications

	location_id [PK] integer	driver_id integer	location geometry	updated_at timestamp without time zone
1	1	1	0101000020E61000005E4BC8073D5B4440AAF1D24D628052…	2025-03-03 23:13:05.649244
2	2	2	0101000020E6100000B610E4A0845D44401FF64201DB7B52C0	2025-03-03 23:13:05.649244
3	3	3	0101000020E6100000D42AFA4333534440149337C0CC7C52C0	2025-03-03 23:13:05.649244

This is the driver_location table. And we have inserted only 3 values for testing purposes.

```
190  SELECT request_ride(1, ST_GeomFromText('POINT(40.7165 -73.9546)', 4326),
191                         ST_GeomFromText('POINT(40.6352 -73.9245)', 4326), TRUE);
192
193
194  select * from ride_requests;
195
```

Data Output | Messages | Notifications

	request_id [PK] integer	rider_id integer	source geometry	destination geometry	pooling boolean	status character varying (20)	created_at timestamp without time
1	1	1	0101000020E6100000C1CAA145B65B4440BE30992A187D52…	0101000020E6100000A835CD3B4E514440BA490C022B7B52C0	true	pending	2025-03-03 23:10:55.34

Here we have called the `request_ride` function. And as we can see, it has been added to the ride_requests table.

```
312  SELECT match_ride(1);
313
314
315  select * from rides;
316
317
```

Data Output | Messages | Notifications

	ride_id [PK] integer	driver_id integer	rider_id integer	source geometry	destination geometry	route geometry	pooling_id integer	status character vary
1	1	2	1	0101000020E6100000C1CAA145B65B4440BE30992A187D52…	0101000020E6100000A835CD3B4E514440BA490C022B7B52C0	[null]	1	ongoing

Now we've called the match_ride function which calls the find_nearest_driver function and as we can see we have matched rider 1 with driver 2, who is the nearest from the pickup location and the ride has been added to the rides table.

```
318  select * from drivers;
319
```

Data Output | Messages | Notifications

	driver_id [PK] integer	user_id integer	vehicle_number character varying (20)	vehicle_model character varying (50)	seat_capacity integer	status character varying (20)
1	1	11	ABC123	Toyota Prius	4	available
2	3	13	ABC890	Honda i20	4	available
3	4	14	XYZ123	Suzuki Ciaz	4	available
4	5	15	ABC321	Toyota Prius	4	available
5	6	16	XYZ789	Honda Civic	4	available
6	2	12	ABC567	Ford Figo	4	on_ride

And as we can see Driver 2 has been labelled on_ride in the status after the match is done.

Now we call the complete_ride function when the ride is completed.

```
384  select complete_ride(1);
385  select * from rides;
386
387
388
389
390
```

Data Output | Messages | Notifications

	ride_id [PK] integer	driver_id integer	rider_id integer	source geometry	destination geometry	route geometry	pooling_id integer	status character var
1	1	2	1	0101000020E6100000C1CAA145B65B4440BE30992A187D52…	0101000020E6100000A835CD3B4E514440BA490C022B7B52C0	[null]	1	completed

```
387  select * from drivers;
388
389
390
```

Data Output | Messages | Notifications

	driver_id [PK] integer	user_id integer	vehicle_number character varying (20)	vehicle_model character varying (50)	seat_capacity integer	status character varying (20)
1	1	11	ABC123	Toyota Prius	4	available
2	3	13	ABC890	Honda i20	4	available
3	4	14	XYZ123	Suzuki Ciaz	4	available
4	5	15	ABC321	Toyota Prius	4	available
5	6	16	XYZ789	Honda Civic	4	available
6	2	12	ABC567	Ford Figo	4	available

And now the driver has been set back to available

Now we check the case for pooled rides.

```
507  select * from ride_requests;
508
```

Data Output | Messages | Notifications

	request_id [PK] integer	rider_id integer	source geometry	destination geometry	pooling boolean	status character varying (20)
1	1	1	0101000020E6100000C1CAA145B65B4440BE30992A187D52...	0101000020E6100000A835CD3B4E514440BA490C022B7B52...	true	matched
2	2	4	0101000020E61000005E4BC8073D5B4440AAF1D24D628052...	0101000020E6100000B610E4A0845D44401FF64201DB7B52C0	true	pending
3	3	5	0101000020E61000002041F163CC5D44401B2FDD24068152C0	0101000020E6100000978BF84ECC5E44408F334DD87E7C52C0	true	pending

Now riders 4 and 5 have requested for rides

```
509  select match_ride(4);
510  select * from rides;
```

Data Output | Messages | Notifications

	ride_id [PK] integer	driver_id integer	rider_id integer	source geometry	destination geometry	route geometry	pooling_id integer	status character var
1	1	2	1	0101000020E6100000C1CAA145B65B4440BE30992A187D52...	0101000020E6100000A835CD3B4E514440BA490C022B7B52...	[null]	1	completed
2	2	3	4	0101000020E61000005E4BC8073D5B4440AAF1D24D628052C0	0101000020E6100000B610E4A0845D44401FF64201DB7B52C0	[null]	2	ongoing

We initially create a ride for rider 4.

Then we match a ride for rider 5.

```
512  select match_ride(5);
513  select * from rides;
```

Data Output | Messages | Notifications

	ride_id [PK] integer	driver_id integer	rider_id integer	source geometry	destination geometry	route geometry	pooling_id integer	status character var
1	1	2	1	0101000020E6100000C1CAA145B65B4440BE30992A187D52...	0101000020E6100000A835CD3B4E514440BA490C022B7B52...	[null]	1	completed
2	2	3	4	0101000020E61000005E4BC8073D5B4440AAF1D24D628052...	0101000020E6100000B610E4A0845D44401FF64201DB7B52C0	[null]	2	ongoing
3	3	3	5	0101000020E61000002041F163CC5D44401B2FDD24068152C0	0101000020E6100000978BF84ECC5E44408F334DD87E7C52C0	[null]	2	ongoing

As we can see we have the same driver_id and pooling_id for rider 5 as well. So we have successfully pooled riders 4 and 5.