Trigger

Our trigger *prevent_conflicts* enforces a critical business rule: users cannot attend two events on the same date. The trigger fires BEFORE INSERT on the WantsToAttend table and uses an IF statement to check whether the user already has an event scheduled on the new event's date. If a conflict exists (event_count > 0), the trigger signals an error with a clear message and prevents the insert. This has direct real-world value for our concert travel planning app because it's physically impossible to attend two events simultaneously, so the trigger automatically prevents users from making invalid schedules. By enforcing this constraint at the database level rather than in application code, we guarantee data integrity regardless of which interface or feature a user uses to add events, providing immediate transactional feedback that stops bad data before it enters the system.

```
DELIMITER //
CREATE TRIGGER prevent_conflicts
BEFORE INSERT ON WantsToAttend
FOR EACH ROW
BEGIN
  DECLARE event_date DATE;
  DECLARE event_count INT;

  SELECT date INTO event_date
  FROM Event
  WHERE event_id = NEW.event_id;

  SELECT COUNT(*) INTO event_count
  FROM WantsToAttend w
  JOIN Event e ON w.event_id = e.event_id
  WHERE w.user_id = NEW.user_id
    AND e.date = event_date;

  IF event_count > 0 THEN
    SIGNAL SQLSTATE '45000'
    SET MESSAGE_TEXT = 'You already have an event scheduled on
this date';
  END IF;
END//
DELIMITER ;
```

Stored Procedure

Our stored procedure *sp_event_airbnb_summary()* combines Query 1 (cheapest Airbnb per event within 1 mile) and Query 3 (events with most available listings within 5 miles) into a single efficient database call. It uses control structures to populate two temporary tables with aggregated results from multiple JOINs and GROUP BY operations using MIN(), COUNT(), AVG(), and ROUND() functions. This powers Endpoints #1 and #3 in our application, reducing network overhead by executing both complex queries server-side in one call instead of two separate requests. For users comparing lodging options across many events, this optimization delivers fast results without repeatedly running expensive aggregations.

```
DELIMITER $$

CREATE PROCEDURE sp_event_airbnb_summary()
BEGIN
    -- Temporary table for cheapest Airbnb per event (<=1 mile)
    CREATE TEMPORARY TABLE IF NOT EXISTS tmp_cheapest_airbnb (
        event_id INT,
        event_name VARCHAR(255),
        city_name VARCHAR(255),
        state VARCHAR(50),
        cheapest_total_cost DECIMAL(10,2)
    );
    TRUNCATE TABLE tmp_cheapest_airbnb;

    INSERT INTO tmp_cheapest_airbnb (event_id, event_name,
city_name, state, cheapest_total_cost)
    SELECT
        e.event_id,
        e.name AS event_name,
        c.city_name,
        c.state,
        MIN(n.total_cost) AS cheapest_total_cost
    FROM Event e
    JOIN Venue v ON e.venue_id = v.venue_id
    JOIN Nearby n ON e.event_id = n.event_id
    JOIN AirbnbListing a ON n.listing_id = a.listing_id
    JOIN City c ON v.city_id = c.city_id
    WHERE n.distance <= 1
    GROUP BY e.event_id, e.name, c.city_name, c.state
    ORDER BY cheapest_total_cost ASC
    LIMIT 15;
```

```
    -- Temporary table for events with most available Airbnb
listings (<= 5 miles)
    CREATE TEMPORARY TABLE IF NOT EXISTS tmp_most_available (
        event_id INT,
        event_name VARCHAR(255),
        city_name VARCHAR(255),
        state VARCHAR(50),
        num_available_listings INT,
        avg_price_per_night DECIMAL(10,2),
        closest_listing_distance DECIMAL(5,2)
    );
    TRUNCATE TABLE tmp_most_available;

    INSERT INTO tmp_most_available (event_id, event_name,
city_name, state, num_available_listings, avg_price_per_night,
closest_listing_distance)
    SELECT
        e.event_id,
        e.name AS event_name,
        c.city_name,
        c.state,
        COUNT(a.listing_id) AS num_available_listings,
        ROUND(AVG(a.price_per_night), 2) AS avg_price_per_night,
        MIN(n.distance) AS closest_listing_distance
    FROM Event e
    JOIN Venue v ON e.venue_id = v.venue_id
    JOIN City c ON v.city_id = c.city_id
    JOIN Nearby n ON e.event_id = n.event_id
    JOIN AirbnbListing a ON n.listing_id = a.listing_id
    WHERE n.distance <= 5 AND a.availability_365 > 0
    GROUP BY e.event_id, e.name, c.city_name, c.state
    ORDER BY num_available_listings DESC, avg_price_per_night
ASC
    LIMIT 15;
    -- Return results of both queries
    SELECT * FROM tmp_cheapest_airbnb;
    SELECT * FROM tmp_most_available;

END$$
DELIMITER ;
```

Transaction

Our transaction implementation powers the "Add 5 Upcoming Events from City" feature, which atomically adds multiple events to a user's schedule while checking for date conflicts and duplicate entries. The transaction executes two advanced queries: (1) a JOIN with ORDER BY and LIMIT to find the 5 soonest upcoming events in a specified city, and (2) a JOIN with GROUP BY aggregation to identify existing user commitments and prevent conflicts. If any part of the operation fails, whether due to a date conflict detected by our trigger, a duplicate event ID, or a database error, the entire transaction rolls back, ensuring no partial data is inserted. This is critical because without it, a concurrent request could insert a conflicting event between our conflict-check and insert operations, resulting in double bookings.

```
db.beginTransaction(err => {
    if (err) {
      console.error("Transaction start error:", err);
      return res.status(500).json({ error: "Database error" });
    }

    const query = `INSERT INTO WantsToAttend (user_id, event_id)
VALUES (?, ?)`;
    db.query(query, [userId, eventId], (err, result) => {
      if (err) {
        // rollback transaction on error
        return db.rollback(() => {
          if (err.sqlMessage && err.sqlMessage.includes('already
have an event')) {
            return res.status(409).json({ error: "You already
have an event scheduled on this date" });
          }
          console.error("Error adding event:", err);
          return res.status(500).json({ error: "Database error"
});
        });
      }

      db.commit(commitErr => {
        if (commitErr) {
          return db.rollback(() => {
            console.error("Transaction commit error:",
commitErr);
            return res.status(500).json({ error: "Database
error" });
```

```
          });
        }
        res.json({ message: "Event added successfully", eventId
});
      });
    });
  });
```