Set Returning Functions & JSON and Array Functions

Data Transformation and Aggregation: Solutions

May 19, 2025

Contents

Global Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises. This is the same dataset provided with the exercises.

```sql
-- Dataset for Exercises
-- Drop tables if they exist to ensure a clean setup
DROP TABLE IF EXISTS EventCalendar CASCADE;
DROP TABLE IF EXISTS ProjectAssignments CASCADE;
DROP TABLE IF EXISTS Employees CASCADE;
DROP TABLE IF EXISTS SystemLogs CASCADE;
DROP TABLE IF EXISTS ServiceSubscriptions CASCADE;

-- Create Tables
CREATE TABLE Employees (
    employeeId INT PRIMARY KEY,
    employeeName VARCHAR(100) NOT NULL,
    hireDate DATE NOT NULL,
    department VARCHAR(50),
    skills TEXT[], -- For unnest, array_append, array_length
    performanceReviews JSONB -- For JSON functions, e.g., '[{"year": 2022, "rating": 4, "notes": "Good progress"}, ...]'
);

CREATE TABLE ProjectAssignments (
    assignmentId SERIAL PRIMARY KEY,
    projectId INT NOT NULL,
    projectName VARCHAR(100),
    employeeId INT REFERENCES Employees(employeeId),
    role VARCHAR(50),
    assignmentHours INT,
    assignmentData JSONB -- e.g., '{ "milestones": [{"name": "Phase 1", "status": "completed"}, {"name": "Phase 2", "status": "pending"}], "budget": 5000.00 }'
);

CREATE TABLE SystemLogs (
    logId SERIAL PRIMARY KEY,
    logTimestamp TIMESTAMP NOT NULL,
    serviceName VARCHAR(50),
    logLevel VARCHAR(10), -- e.g., INFO, ERROR, WARN
    logDetails JSONB -- e.g., '{ "clientIp": "192.168.1.10", "requestPath": "/api/data", "statusCode": 200, "userContext": {"userId": 101, "sessionId": "xyz"} }'
);

CREATE TABLE ServiceSubscriptions (
    subscriptionId SERIAL PRIMARY KEY,
    userId INT,
    customerName VARCHAR(100),
    serviceType VARCHAR(50),
    startDate DATE NOT NULL,
    endDate DATE, -- Can be NULL for ongoing subscriptions
    monthlyFee DECIMAL(10,2),
    features JSONB -- e.g., '{ "storageLimitGB": 50, "prioritySupport": true, "addons": ["backup", "monitoring"] }'
);

CREATE TABLE EventCalendar (
    eventId SERIAL PRIMARY KEY,
    eventName VARCHAR(100),
    eventCategory VARCHAR(50),
    eventStartDate DATE,
    eventEndDate DATE,
    expectedAttendees INT,
    bookedResources TEXT[] -- e.g., '{"Room A", "Projector X"}'
);


-- Populate Tables
INSERT INTO Employees (employeeId, employeeName, hireDate, department, skills, performanceReviews) VALUES
(1, 'Alice Wonderland', '2020-01-15', 'Engineering', ARRAY['SQL', 'Python', 'Data Analysis'], '[{"year": 2022, "rating": 4, "notes": "Exceeded expectations in Q3"},
```

```
    {"year": 2023, "rating": 5, "notes": "Top performer"}]'),
(2, 'Bob The Builder', '2019-03-01', 'Engineering', ARRAY['Java', 'Spring', '
    Microservices'], '[{"year": 2022, "rating": 3, "notes": "Met expectations"}, {"year
    ": 2023, "rating": 4, "notes": "Improved significantly"}]'),
(3, 'Charlie Brown', '2021-07-30', 'Sales', ARRAY['Communication', 'Negotiation', 'CRM'
    ], '[{"year": 2023, "rating": 4, "notes": "Good sales figures"}]'),
(4, 'Diana Prince', '2018-05-10', 'HR', ARRAY['Recruitment', 'Employee Relations', '
    Legal Knowledge'], NULL),
(5, 'Edward Scissorhands', '2022-11-01', 'Engineering', ARRAY['Python', 'Machine
    Learning'], '[{"year": 2023, "rating": 5, "notes": "Innovative solutions"}]'),
(6, 'Fiona Gallagher', '2023-02-15', 'Sales', ARRAY['CRM', 'Presentations'], '[]'), --
    Empty JSON array
(7, 'George Jetson', '2017-09-01', 'Management', ARRAY['Leadership', 'Strategy'], '[{"
    year": 2022, "rating": 5, "notes": "Excellent leadership"}, {"year": 2023, "rating":
     4, "notes": "Managed team well through transition"}]');

INSERT INTO ProjectAssignments (projectId, projectName, employeeId, role,
    assignmentHours, assignmentData) VALUES
(101, 'Data Warehouse Migration', 1, 'Lead Data Engineer', 120, '{ "milestones": [{"name
    ": "Schema Design", "status": "completed"}, {"name": "ETL Development", "status": "
    in-progress"}], "budget": 20000.00, "critical": true }'),
(101, 'Data Warehouse Migration', 2, 'Backend Developer', 80, '{ "milestones": [{"name":
     "API Integration", "status": "pending"}], "budget": 15000.00, "critical": true }'),
(102, 'Mobile App Development', 2, 'Lead Mobile Developer', 150, '{ "milestones": [{"
    name": "UI/UX Design", "status": "completed"}, {"name": "Frontend Dev", "status": "
    completed"}, {"name": "Backend Dev", "status": "in-progress"}], "budget": 50000.00,
    "critical": false }'),
(103, 'Sales Platform Upgrade', 3, 'Sales Lead', 100, '{ "milestones": [{"name": "
    Requirement Gathering", "status": "completed"}], "budget": 10000.00, "critical":
    false }'),
(101, 'Data Warehouse Migration', 5, 'ML Engineer', 60, '{ "milestones": [{"name": "
    Model Training", "status": "in-progress"}], "budget": 12000.00, "critical": true }')
    ,
(104, 'HR System Implementation', 4, 'HR Specialist', 90, NULL); -- No JSON data

INSERT INTO SystemLogs (logTimestamp, serviceName, logLevel, logDetails) VALUES
('2023-10-01 10:00:00', 'AuthService', 'INFO', '{ "message": "User login successful", "
    userId": 1, "clientIp": "192.168.0.10" }'),
('2023-10-01 10:05:00', 'OrderService', 'ERROR', '{ "message": "Payment processing
    failed", "orderId": 123, "errorCode": "P5001", "details": {"reason": "Insufficient
    funds"} }'),
('2023-10-01 10:10:00', 'ProductService', 'WARN', '{ "message": "Low stock warning", "
    productId": "XYZ123", "currentStock": 5 }'),
('2023-10-02 11:00:00', 'AuthService', 'INFO', '{ "message": "User login successful", "
    userId": 2, "clientIp": "192.168.0.15" }'),
('2023-10-02 11:15:00', 'OrderService', 'INFO', '{ "message": "Order placed", "orderId":
     124, "items": ["itemA", "itemB"], "totalAmount": 75.50 }'),
(NOW() - INTERVAL '1 day', 'ReportingService', 'DEBUG', '{ "queryId": "q123", "
    executionTimeMs": 1500, "parameters": {"startDate": "2023-01-01", "endDate":
    "2023-01-31"} }');

INSERT INTO ServiceSubscriptions (userId, customerName, serviceType, startDate, endDate,
     monthlyFee, features) VALUES
(101, 'Customer Alpha', 'Premium Cloud Storage', '2023-01-01', '2023-12-31', 20.00, '{ "
    storageLimitGB": 100, "prioritySupport": true, "addons": ["versioning", "encryption
    "] }'),
(102, 'Customer Beta', 'Basic VPN Service', '2023-03-15', NULL, 5.00, '{ "dataCapMB":
    5000, "prioritySupport": false, "serverLocations": ["US", "EU"] }'),
(103, 'Customer Gamma', 'Standard Streaming', '2023-05-01', '2024-04-30', 10.00, '{ "
    resolution": "1080p", "profiles": 4, "offlineDownload": true }'),
(101, 'Customer Alpha', 'Analytics Suite', '2023-06-01', NULL, 50.00, '{ "users": 5, "
    dashboards": 10, "dataSources": ["db1", "s3"] }'),
(104, 'Customer Delta', 'Premium Cloud Storage', '2022-11-01', '2023-11-01', 18.00, '{ "
    storageLimitGB": 100, "prioritySupport": true, "addons": ["backup"] }');

INSERT INTO EventCalendar (eventName, eventCategory, eventStartDate, eventEndDate,
    expectedAttendees, bookedResources) VALUES
('Tech Conference 2024', 'Conference', '2024-03-10', '2024-03-12', 500, ARRAY['Main Hall
    ', 'Audio System', 'Projectors']),
('Product Launch Q1', 'Marketing', '2024-02-15', '2024-02-15', 100, ARRAY['Meeting Room
    Alpha', 'Catering Service']),
('Team Building Workshop', 'HR', '2024-04-05', '2024-04-05', 30, ARRAY['Outdoor Space',
    'Activity Kits']),
```

```
('Quarterly Review Meeting', 'Management', '2024-01-20', '2024-01-20', 15, ARRAY['Board Room']),
('Holiday Party 2023', 'Social', '2023-12-15', '2023-12-15', 150, NULL); -- No resources booked
```

Listing 1: Global Dataset for Exercises (Identical to Exercises Document)

1 Set Returning Functions (generate_series, unnest) Solutions

1.1 (i) Meaning, values, relations (with previous concepts), advantages

Solution SRF.1.1: Monthly Active Subscription Report

```
WITH MonthSeries AS (
    SELECT generate_series(
        '2023-01-01'::DATE,
        '2023-12-01'::DATE,
        '1 month'::INTERVAL
    ) AS reportMonth
)
SELECT
    ss.serviceType,
    ms.reportMonth,
    COUNT(ss.subscriptionId) AS activeSubscriptions
FROM
    MonthSeries ms
CROSS JOIN -- Consider all service types for each month initially
    (SELECT DISTINCT serviceType FROM ServiceSubscriptions)
   distinctServiceTypes
LEFT JOIN
    ServiceSubscriptions ss ON ss.serviceType = distinctServiceTypes.
   serviceType
    AND ms.reportMonth >= DATE_TRUNC('month', ss.startDate)
    AND (ms.reportMonth <= DATE_TRUNC('month', ss.endDate) OR ss.
   endDate IS NULL)
GROUP BY
    ss.serviceType,
    ms.reportMonth
HAVING
    ss.serviceType IS NOT NULL -- Only show service types that had
   activity
ORDER BY
    ss.serviceType,
    ms.reportMonth;
```

Solution SRF.1.2: Employee Skills Breakdown

```
SELECT
    e.employeeName,
    e.department,
    unnest(e.skills) AS skill
FROM
    Employees e
WHERE
    e.skills IS NOT NULL AND array_length(e.skills, 1) > 0;
```

1.2 (ii) Disadvantages of all its technical concepts

Solution SRF.2.1: Potential Performance Issue with `generate_series`

```sql
-- Query that would generate the series (limited for safety)
SELECT generate_series(
    '2023-01-01 00:00:00'::TIMESTAMP,
    '2023-01-01 00:00:09'::TIMESTAMP, -- Simulating a small part of a year
    '1 second'::INTERVAL
) AS sensorTimestamp
LIMIT 10;

-- Explanation of disadvantage:
-- A full year has 365 * 24 * 60 * 60 = 31,536,000 seconds.
-- Generating such a large series directly in memory or as an intermediate result set
-- can consume significant server resources (CPU and RAM) and lead to slow query performance
-- or even out-of-memory errors. If this data is to be stored, it would create a very large table.
-- This demonstrates that while powerful, generate_series must be used with an understanding
-- of the scale of data it can produce.
```

Explanation of disadvantage (as provided in the comment within the code block): A full year has 365 * 24 * 60 * 60 = 31,536,000 seconds. Generating such a large series directly in memory or as an intermediate result set can consume significant server resources (CPU and RAM) and lead to slow query performance or even out-of-memory errors. If this data is to be stored, it would create a very large table. This demonstrates that while powerful, `generate_series` must be used with an understanding of the scale of data it can produce.

Solution SRF.2.2: Row Explosion with `unnest`

```sql
-- Hypothetical scenario explanation (no direct query needed to demonstrate explosion,
-- but we can illustrate a join that would cause it if data were larger):
SELECT
    e.employeeName,
    s.skill,
    pa.projectName
FROM
    Employees e,
    unnest(e.skills) AS s(skill) -- Generates N rows for N skills
LEFT JOIN
    ProjectAssignments pa ON e.employeeId = pa.employeeId -- Could multiply by M projects
WHERE e.employeeId = 1; -- Example for one employee

-- Explanation of disadvantage:
-- If an employee has 100 skills, `unnest(e.skills)` for that employee creates 100 rows.
-- If this employee is also assigned to 10 projects, a subsequent JOIN on `employeeId`
```

```
-- (like with 'ProjectAssignments') would result in 100 (skills) * 10 (projects) = 1000 rows
-- for this single employee *before* any filtering or aggregation specific to skill/project combinations.
-- This "row explosion" can significantly increase the size of intermediate datasets,
-- leading to higher memory consumption, slower processing for joins, aggregations, and sorting,
-- thus degrading overall query performance. It's crucial when using 'unnest' in conjunction
-- with joins to be aware of this multiplicative effect and to apply filters as early as possible.
```

Explanation of disadvantage (as provided in the comment within the code block): If an employee has 100 skills, `unnest(e.skills)` for that employee creates 100 rows. If this employee is also assigned to 10 projects, a subsequent JOIN on `employeeId` (like with `ProjectAssignments`) would result in 100 (skills) * 10 (projects) = 1000 rows for this single employee *before* any filtering or aggregation specific to skill/project combinations. This "row explosion" can significantly increase the size of intermediate datasets, leading to higher memory consumption, slower processing for joins, aggregations, and sorting, thus degrading overall query performance. It's crucial when using `unnest` in conjunction with joins to be aware of this multiplicative effect and to apply filters as early as possible.

1.3 (iii) Cases where people lose advantages due to inefficient solutions

Solution SRF.3.1: Inefficiently Generating Date Sequences

```
WITH RECURSIVE DateSequenceCTE AS (
    SELECT '2024-01-01'::DATE AS aDate
    UNION ALL
    SELECT (aDate + INTERVAL '1 day')::DATE
    FROM DateSequenceCTE
    WHERE aDate < '2024-01-31'::DATE
)
SELECT aDate FROM DateSequenceCTE;
```

```
SELECT generate_series(
    '2024-01-01'::DATE,
    '2024-01-31'::DATE,
    '1 day'::INTERVAL
) AS aDate;
```

Advantage Explanation: The `generate_series` approach is more advantageous because:

1. **Conciseness:** It's a single function call, making the SQL much shorter and easier to read and understand.

2. **Performance:** `generate_series` is typically implemented as a highly optimized internal function, often outperforming recursive CTEs for simple sequence generation.

3. **Simplicity:** It directly expresses the intent of generating a series without the boilerplate of setting up a recursive query (anchor member, recursive member, termination condition).

People might use recursive CTEs if they are unaware of `generate_series` or if they need more complex, conditional series generation that `generate_series` alone can't handle (though often, `generate_series` can be combined with other logic).

Solution SRF.3.2: Inefficiently Handling Array Elements

```
SELECT
    employeeName,
    skills
FROM
    Employees
WHERE
    employeeId = 1 AND
    array_to_string(skills, ',') LIKE '%Python%'; -- Prone to errors (e.g. skill 'Pythonista') and inefficient
```

```
-- Using array contains operator @>
SELECT
    employeeName,
    skills
FROM
    Employees
WHERE
    employeeId = 1 AND
    skills @> ARRAY['Python'];

-- Alternative using ANY with unnest (conceptually, though often optimized without full unnesting)
SELECT
    e.employeeName,
    e.skills
FROM
    Employees e
WHERE
    e.employeeId = 1 AND
    'Python' = ANY(e.skills);
-- Or with explicit unnest for checking:
SELECT DISTINCT e.employeeName, e.skills
FROM Employees e, unnest(e.skills) s
WHERE e.employeeId = 1 AND s = 'Python';
```

Advantage Explanation: The SQL-native array approaches (`@>`, `ANY`, or `unnest` with `WHERE`) are advantageous because:

1. **Accuracy:** They correctly match whole array elements, avoiding partial matches or issues with delimiters that `array_to_string` and `LIKE` might suffer from (e.g., searching for 'Py' would match 'Python').

2. **Performance:** PostgreSQL has optimized operators and indexing capabilities (GIN indexes) for arrays, making these searches much faster than string operations on concatenated array content, especially for large arrays or tables.

3. **Readability and Intent:** Using array-specific operators clearly conveys the intent to operate on array elements.

People might resort to string manipulation if they are unfamiliar with array functions and operators, leading to less robust and slower queries.

1.4 (iv) Hardcore problem combining previous concepts

Solution SRF.4.1: Comprehensive Project Health and Skill Utilization Report

```
WITH MonthlySeries AS (
    SELECT generate_series(
        '2023-01-01'::DATE,
        '2023-06-01'::DATE,
        '1 month'::INTERVAL
    ) AS reportMonth
),
ProjectEmployeeSkills AS (
    SELECT
        pa.projectId,
        pa.projectName,
        e.employeeId,
        e.hireDate,
        s.skill,
        pa.assignmentHours,
        (pa.assignmentData ->> 'critical')::BOOLEAN AS isCritical
    FROM
        ProjectAssignments pa
    JOIN
        Employees e ON pa.employeeId = e.employeeId
    CROSS JOIN
        unnest(e.skills) AS s(skill) -- unnest skills
),
ProjectsWithPythonSkill AS ( -- Identify projects that utilize 'Python' skill
    SELECT DISTINCT projectId
    FROM ProjectEmployeeSkills
    WHERE skill = 'Python'
)
SELECT
    ms.reportMonth,
    pes.projectName,
    SUM(pes.assignmentHours) AS totalAssignedHours,
    MAX(CASE WHEN pes.isCritical THEN 1 ELSE 0 END)::BOOLEAN AS criticalProjectFlag, -- True if any assignment is critical
    STRING_AGG(DISTINCT pes.skill, ', ' ORDER BY pes.skill) AS listOfDistinctSkillsUtilized,
    ROUND(AVG(
        EXTRACT(YEAR FROM AGE(ms.reportMonth, pes.hireDate)) +
        EXTRACT(MONTH FROM AGE(ms.reportMonth, pes.hireDate)) / 12.0 +
```

```sql
        EXTRACT(DAY FROM AGE(ms.reportMonth, pes.hireDate)) / 365.25 -- More precise age calculation
    ), 2) AS averageYearsOfService
FROM
    MonthlySeries ms
CROSS JOIN -- To consider each project for each month
    (SELECT DISTINCT projectId, projectName FROM ProjectAssignments) uniqueProjects
JOIN
    ProjectEmployeeSkills pes ON uniqueProjects.projectId = pes.projectId
WHERE
    pes.hireDate <= ms.reportMonth -- Employee active in the report month
    AND pes.projectId IN (SELECT projectId FROM ProjectsWithPythonSkill) -- Filter for projects with Python skill
GROUP BY
    ms.reportMonth,
    pes.projectName,
    pes.projectId -- Added projectId to GROUP BY for correctness if projectName is not unique across IDs
ORDER BY
    ms.reportMonth,
    pes.projectName;
```

2 JSON and Array Functions Solutions

2.1 (i) Meaning, values, relations (with previous concepts), advantages

Solution JAF.1.1: Extracting Specific Log Information

```sql
SELECT
    logTimestamp,
    logDetails ->> 'clientIp' AS clientIp,
    logDetails -> 'userContext' ->> 'userId' AS userId
    -- Alternative for userId: jsonb_extract_path_text(logDetails, 'userContext', 'userId')
FROM
    SystemLogs
WHERE
    serviceName = 'AuthService' AND logLevel = 'INFO';
```

Solution JAF.1.2: Expanding Performance Review Details

```sql
SELECT
    e.employeeName,
    (review.reviewData ->> 'year')::INT AS reviewYear,
    (review.reviewData ->> 'rating')::INT AS reviewRating,
    review.reviewData ->> 'notes' AS reviewNotes
FROM
    Employees e,
    jsonb_array_elements(e.performanceReviews) AS review(reviewData) -- reviewData is each JSON object in the array
WHERE
    e.performanceReviews IS NOT NULL AND jsonb_typeof(e.performanceReviews) = 'array' AND jsonb_array_length(e.performanceReviews) > 0;
```

Solution JAF.1.3: Constructing a Simplified Project Overview JSON

```sql
SELECT
    pa.projectId,
    pa.projectName, -- Keep projectName separate or include in jsonb_build_object
    jsonb_build_object(
        'projectNameForJson', pa.projectName,
        'assignedEmployeeIds', jsonb_agg(DISTINCT pa.employeeId ORDER BY pa.employeeId) -- Using jsonb_agg for a JSON array
    ) AS projectOverviewJson
FROM
    ProjectAssignments pa
GROUP BY
    pa.projectId, pa.projectName
ORDER BY
    pa.projectId;
-- If a SQL array within the JSON is desired:
-- SELECT
--     pa.projectId,
--     pa.projectName,
```

```
--      jsonb_build_object(
--          'projectNameForJson', pa.projectName,
--          'assignedEmployeeIds', TO_JSONB(ARRAY_AGG(DISTINCT pa.
    employeeId ORDER BY pa.employeeId))
--      ) AS projectOverviewJson
-- FROM
--      ProjectAssignments pa
-- GROUP BY
--      pa.projectId, pa.projectName
-- ORDER BY
--      pa.projectId;
```

Solution JAF.1.4: Updating Event Resources and Checking Count

```
-- This exercise implies an update then select, or a select simulating
    the update.
-- For an exercise, simulating the update in a SELECT is safer.
-- If an actual UPDATE is desired:
-- UPDATE EventCalendar
-- SET bookedResources = array_append(bookedResources, 'WiFi Access
    Point')
-- WHERE eventName = 'Tech Conference 2024';

SELECT
    eventName,
    array_append(COALESCE(bookedResources, '{}'), 'WiFi Access Point')
    AS updatedResources,
    array_length(array_append(COALESCE(bookedResources, '{}'), 'WiFi
    Access Point'), 1) AS totalResources
FROM
    EventCalendar
WHERE
    eventName = 'Tech Conference 2024';
```

2.2 (ii) Disadvantages of all its technical concepts

Solution JAF.2.1: Performance of Complex JSON Queries vs. Normalized Data

Explanation: While JSONB is efficient for storage and flexible, querying deeply nested values (e.g., `logDetails -> 'path1' -> 'path2' -> ... -> 'targetValue'`) has disadvantages compared to dedicated relational columns:

1. **Query Performance:** Accessing deeply nested elements requires traversing the JSON structure. Even with GIN/GiST indexes on the JSONB column, which significantly help, direct access to a value in its own indexed column is typically faster, especially for highly selective queries on that specific piece of data. The overhead of JSON parsing/traversal, though optimized, can be higher than a direct column read.

2. **Indexing Complexity:** While GIN/GiST indexes can index JSONB paths or all keys, designing and maintaining these indexes for optimal performance across various deep-path queries can be more complex than indexing simple scalar columns. Full JSONB indexing can also lead to larger index sizes.

3. **Readability and Maintenance:** Queries involving multiple chained `->` or `->>` operators for deep paths can become less readable and harder to maintain than queries on flat table structures.

4. **Data Integrity and Typing:** Relational columns enforce data types at the column level. While JSONB preserves types, extracting and casting them correctly from deep within JSON requires careful query construction and offers less schema enforcement than separate columns.

If certain deeply nested values are frequently queried and critical for performance, it might be more advantageous to extract them into their own relational columns, even if it means some data duplication or a hybrid model.

Solution JAF.2.2: Array Overuse and Normalization

Explanation: Storing complex skill information (skill name, level, experience) as encoded strings within a single `TEXT[]` array (e.g., `ARRAY['Python:Expert:5yrs', 'SQL:Intermediate:` has several disadvantages compared to a normalized approach (e.g., a separate `EmployeeSkills` table with columns `employeeId`, `skillName`, `skillLevel`, `yearsExperience`):

1. **Querying Complexity:** To query for employees with 'Expert' level in 'Python', you'd need complex string parsing (e.g., `SUBSTRING`, `SPLIT_PART`) within the array elements, which is inefficient and error-prone. A separate table allows simple `WHERE skillName = 'Python' AND skillLevel = 'Expert'`.

2. **Data Integrity and Consistency:** There's no easy way to enforce that the skill level is always one of 'Beginner', 'Intermediate', 'Expert', or that years of experience is a number. Encoded strings are free-form. A separate table can have foreign keys, check constraints, and appropriate data types.

3. **Updating Difficulty:** Updating just the skill level for one skill for an employee becomes a complex array manipulation task (find element, parse, change part, reconstruct string, update array). In a separate table, it's a simple `UPDATE` on a specific row.

4. **Indexing Limitations:** Indexing attributes within these encoded strings is very difficult. A separate table allows indexing on `skillName`, `skillLevel`, etc., for fast lookups.

5. **Violation of First Normal Form (1NF):** While the array itself might contain atomic text values, if those text values are composite (like `'Python:Expert:5yrs'`), it's a way of smuggling multiple logical attributes into a single array element. True 1NF would suggest each attribute (skill name, level, experience) be in its own column, often necessitating a separate table for a one-to-many or many-to-many relationship.

Using arrays for simple lists of atomic values is fine (like current `skills` which are just names), but for structured data within each "element," a normalized approach is generally superior for data integrity, querying, and maintenance.

2.3 (iii) Cases where people lose advantages due to inefficient solutions

Solution JAF.3.1: Inefficiently Querying JSON Data with String Matching

```
SELECT
    logId,
    logDetails
FROM
    SystemLogs
WHERE
    logDetails::TEXT LIKE '%"orderId": 123%'; -- Inefficient and error-prone (e.g. whitespace, "orderId": "123")
```

```
SELECT
    logId,
    logDetails
FROM
    SystemLogs
WHERE
    logDetails ->> 'orderId' = '123'; -- Assumes orderId value is stored as a number or string '123'
    -- If orderId is always a number in JSONB:
    -- WHERE (logDetails -> 'orderId')::TEXT = '123'; -- JSONB numbers need casting to text for = with text
    -- Or, the most robust way for JSONB if orderId is a number:
    -- WHERE logDetails @> '{"orderId": 123}'::JSONB;
```

Advantage Explanation: The JSONB-specific approach (`->>`, `@>`) is advantageous because:

1. **Accuracy and Type Safety:** JSONB operators understand JSON structure and types. `logDetails ->> 'orderId' = '123'` correctly extracts the value associated with the key 'orderId' and compares it as text. `logDetails @> '{"orderId": 123}'::JSONB` correctly checks for the key and number value. `LIKE '%"orderId": 123%'` is a fragile string match; it could falsely match if `123` appeared as part of another value or if there was different whitespace (e.g., `"orderId" : 123`). It also doesn't distinguish between the number `123` and the string `"123"` in the JSON if not careful with the `LIKE` pattern.

2. **Performance:** PostgreSQL's JSONB operators are highly optimized and can leverage GIN/GiST indexes on JSONB columns. Casting to `TEXT` and using `LIKE` prevents effective use of JSONB-specific indexes and involves costly string operations on potentially large JSON structures.

3. **Readability and Intent:** Using JSONB operators clearly communicates that the query is interacting with JSON data structure, not just arbitrary text.

Solution JAF.3.2: Storing Multiple Flags as a Comma-Separated String Instead of JSON/Array

```
-- Hypothetical table structure: CREATE TABLE Users (userId INT, userTags VARCHAR(255));
-- INSERT INTO Users VALUES (1, 'active,premium,verified'), (2, 'active,verified'), (3, 'premium');

-- Assuming 'Users' table exists and is populated as above:
-- SELECT userId, userTags
-- FROM Users -- Hypothetical table
-- WHERE
--     (',' || userTags || ',') LIKE '%,premium,%'; -- Common workaround for whole-word matching
```

Advantageous Approaches (Array or JSONB):

1. Using `TEXT[]` (Array): If `userTags` was `TEXT[]` (e.g., `ARRAY['active', 'premium', 'verified']`):

```
-- Query for 'premium' tag:
-- SELECT userId, userTags FROM Users WHERE userTags @> ARRAY['premium'];
-- Or: SELECT userId, userTags FROM Users WHERE 'premium' = ANY(userTags);
```

Advantages:

- **Atomicity:** Each tag is a distinct element.
- **Efficient Searching:** Array operators (`@>`, `&&`, `ANY`) are optimized and can use GIN indexes.
- **Easier Manipulation:** Adding/removing tags is cleaner with `array_append`, `array_remove`.

2. Using `JSONB` (Object for flags): If `userTags` was `JSONB` (e.g., `{"active": true, "premium": true, "verified": true}` or `{"tags": ["active", "premium", "verified"]}`):

```
-- Query for 'premium' flag (if keys are flags):
-- SELECT userId, userTags FROM Users WHERE (userTags ->> 'premium')::BOOLEAN = TRUE;
-- Or using existence operator if premium key means it's set:
-- SELECT userId, userTags FROM Users WHERE userTags ? 'premium';

-- If JSONB stores an array: {"tags": ["active", "premium", "verified"]}
-- SELECT userId, userTags FROM Users WHERE userTags -> 'tags' @> '["premium"]'::JSONB;
```

Advantages:

- **Structured Data:** Can store key-value pairs (e.g., flag name and its boolean state) or arrays within JSON.
- **Powerful Querying:** JSON operators and path expressions offer flexible querying, also benefiting from GIN indexes.
- **Schema Flexibility:** Easily add new flags/tags without altering table structure.

Overall Advantages of Array/JSONB over Comma-Separated Strings:

- **Data Integrity:** Avoids issues with delimiters within tag names or inconsistent formatting.
- **Search Performance:** SQL-native array/JSONB operations are significantly faster and can be indexed effectively. String `LIKE` on concatenated strings is slow and cannot use specialized indexes well for this purpose.
- **Maintainability:** Easier to add, remove, or modify individual tags/flags without complex string parsing and reconstruction.
- **Clarity of Intent:** Using appropriate data types makes the data model and queries more understandable.

People often use comma-separated strings due to perceived simplicity initially or when migrating from systems that use such formats, but they lose out on the robustness, performance, and maintainability offered by PostgreSQL's array and JSON types.

2.4 (iv) Hardcore problem combining previous concepts

Solution JAF.4.1: Advanced Customer Subscription Feature Analysis and Aggregation

```
WITH CustomerSubscriptionDetails AS (
    SELECT
        ss.subscriptionId,
        ss.customerName,
        ss.serviceType,
        ss.monthlyFee,
        ss.startDate,
        ss.endDate,
        ss.features,
        CASE
            WHEN ss.endDate IS NULL OR ss.endDate > CURRENT_DATE THEN 'Active'
            ELSE 'Expired'
        END AS status,
        EXTRACT(YEAR FROM AGE(COALESCE(ss.endDate, CURRENT_DATE), ss.startDate)) * 12 +
        EXTRACT(MONTH FROM AGE(COALESCE(ss.endDate, CURRENT_DATE), ss.startDate)) AS durationMonths,
        COALESCE((ss.features ->> 'prioritySupport')::BOOLEAN, FALSE) AS hasPrioritySupport,
        -- Extract addons array; ensure it's an array and handle NULLs/non-arrays gracefully
        CASE
            WHEN jsonb_typeof(ss.features -> 'addons') = 'array'
            THEN (SELECT jsonb_agg(elem) FROM jsonb_array_elements_text(ss.features -> 'addons') AS elem)
            ELSE '[]'::JSONB
        END AS featureListJsonArray
    FROM
        ServiceSubscriptions ss
),
CustomersWithPrioritySupport AS (
    SELECT DISTINCT customerName
```

```sql
    FROM CustomerSubscriptionDetails
    WHERE hasPrioritySupport = TRUE
)
SELECT
    csd.customerName, -- Included for clarity, though it's in the JSON
    jsonb_build_object(
        'customerName', csd.customerName,
        'totalMonthlyFee', COALESCE(SUM(csd.monthlyFee) FILTER (WHERE csd.status = 'Active'), 0.00),
        'activeServicesCount', COUNT(*) FILTER (WHERE csd.status = 'Active'),
        'serviceDetails', jsonb_agg(
            jsonb_build_object(
                'serviceType', csd.serviceType,
                'status', csd.status,
                'durationMonths', csd.durationMonths,
                'featureList', csd.featureListJsonArray,
                'hasPrioritySupport', csd.hasPrioritySupport
            ) ORDER BY csd.startDate DESC
        )
    ) AS customerReport
FROM
    CustomerSubscriptionDetails csd
WHERE
    csd.customerName IN (SELECT customerName FROM CustomersWithPrioritySupport)
GROUP BY
    csd.customerName
ORDER BY
    SUM(csd.monthlyFee) FILTER (WHERE csd.status = 'Active') DESC NULLS LAST;
```