Set Returning, JSON, and Array Functions

Data Transformation & Aggregation

Sequential SQL

May 20, 2025

Contents

1 Set Returning Functions (SRFs)

Set Returning Functions (SRFs), or table functions, are a special class of SQL functions. Instead of returning a single value for each input row (like scalar functions), SRFs can return multiple rows. They essentially produce a virtual table that can be used in SQL queries, often in the `FROM` clause. It's like magic; a function call conjures up a whole table!

The primary SRFs we'll focus on are `generate_series` and `unnest`.

1.1 What Are They? (Meanings & Values)

Meaning[a]**:** Set Returning Functions generate a collection of rows (a set) based on their input parameters. They act as a data source within a query. Think of them as functions that "return a table."
Values/Output: The output is a table-like structure. Each row generated by the SRF can have one or more columns, depending on the specific function.

[a]After to this lecture check the official documentation to learn all avaiable features and capabilities

`generate_series(start, stop [, step])`

- **Meaning:** This function generates a series of values, starting from `start` and going up to `stop`, incrementing by `step`. If `step` is omitted, it defaults to 1. It's your go-to for creating sequences of numbers or time periods. Need numbers in line? `generate_series` is fine!
- **Values/Output:** A single-column table where each row contains a value from the generated series. The data type of the column matches the data type of the `start` and `stop` parameters (e.g., integer, numeric, timestamp).

`unnest(array_expression)`

- **Meaning:** This function takes an array as input and expands its elements into a set of rows. Each element of the array becomes a separate row in the output. Array elements break free, with `unnest`, you see!
- **Values/Output:** A single-column table where each row contains one element from the input array. The data type of the column matches the element type of the array.

1.2 Relations: How They Play with Others

SRFs don't live in a vacuum; they interact beautifully with other SQL concepts you've already learned.

Internal Relations (SRFs with SRFs)

- The output of one SRF can sometimes be used as input or in conjunction with another, though this is less common than combining SRFs with table data. For instance, you might generate a series of numbers and then use those numbers to construct array elements which could then be unnested in a more complex scenario.

Relations with Previous SQL Concepts

SRFs are powerful because they integrate seamlessly with established SQL clauses and functions:

- **FROM Clause & LATERAL Joins:** This is the primary playground for SRFs. They are typically placed in the FROM clause. When an SRF's parameters depend on columns from a preceding table in the FROM clause, a LATERAL join is essential. LATERAL allows the SRF to be evaluated for each row of the preceding table.

```
SELECT t.id, s.valueFromSeries FROM myTable t, LATERAL
generate_series(t.startVal, t.endVal) AS s(valueFromSeries);
```

- **Other Joins (INNER, LEFT, etc.):** The set of rows produced by an SRF can be joined to other tables or the output of other SRFs using standard join types.
- **SELECT Clause:** The values generated by SRFs can be used in the SELECT list, either directly or as part of expressions.
- **WHERE Clause:** You can filter the rows generated by an SRF using conditions in the WHERE clause.
- **Date Functions & Arithmetic:** generate_series is exceptionally useful with date/time types.
  - Parameters can be TIMESTAMP or DATE.
  - INTERVAL is used for the step argument (e.g., '1 day'::INTERVAL).
  - Functions like DATE_TRUNC, EXTRACT, and date arithmetic can be applied to the series of dates generated.
- **Aggregators (SUM, COUNT, ARRAY_AGG, etc.):**
  - Values from SRFs can be aggregated. For instance, count the number of days generated in a period.
  - unnest is the conceptual inverse of ARRAY_AGG (an Advanced Aggregate Function). ARRAY_AGG groups rows into an array, and unnest expands an array back into rows.
- **Common Table Expressions (CTEs):** SRFs can be used within a CTE to generate a base set of data which is then processed in subsequent parts of the query. This improves readability and modularity.
- **Casting (::):** Often necessary to ensure parameters for generate_series are of the correct type, e.g., '2023-01-01'::DATE.
- **Subqueries (in FROM clause):** An SRF can be part of a subquery, and its results can be further processed.

1.3 How to Use Them: Structures & Syntax

Let's look at the practical syntax for these functions, primarily in PostgreSQL.

Categorization of SRFs

- **Series Generators:** Functions that create sequential data.

    - generate_series
- **Expanders/Deconstructors:** Functions that take a composite data type and expand it into rows.
    - unnest (for arrays)

generate_series

Syntax (PostgreSQL):

- generate_series(start INT, stop INT) → SETOF INT
- generate_series(start INT, stop INT, step INT) → SETOF INT
- generate_series(start NUMERIC, stop NUMERIC, step NUMERIC) → SETOF NUMERIC
- generate_series(start TIMESTAMP, stop TIMESTAMP, step INTERVAL) → SETOF TIMESTAMP

Usage Examples:

1. Basic integer series:

```
SELECT s.numValue FROM generate_series(1, 5) AS s(numValue); – Output: Rows with 1, 2, 3, 4, 5
```

2. Date series (e.g., generating all days in a week):

```
SELECT d.dayDate::DATE FROM generate_series( '2024-01-01'::TIMESTAMP, '2024-01-07'::TIMESTAMP, '1 day'::INTERVAL ) AS d(dayDate); – Output: Rows with 2024-01-01, ..., 2024-01-07
```

3. With LATERAL join to generate series per row of another table:

```
– Assume ReportPeriods table with periodStart, periodEnd columns SELECT rp.reportName, gs.monthStart::DATE FROM ReportPeriods rp, LATERAL generate_series(rp.periodStart, rp.periodEnd, '1 month'::INTERVAL) AS gs(monthStart) WHERE rp.reportName = 'Q1 2024 Analysis';
```

unnest

Syntax (PostgreSQL):

- unnest(anyarray) → SETOF anyelement
- unnest(anyarray, anyarray [, ...]) → SETOF record (for parallel unnesting of multiple arrays, each becomes a column in the output set)
- WITH ORDINALITY: Appends an additional column to the output that numbers the elements from the array.

```
SELECT u.skill, u.idx FROM unnest(ARRAY['SQL', 'Python', 'Java']) WITH ORDINALITY AS u(skill, idx); – Output: – skill | idx – ——-|—— – SQL | 1 – Python | 2 – Java | 3
```

Usage Examples:

1. Unnesting a simple array in the SELECT (less common for complex queries but illustrates the concept):

```
SELECT unnest(ARRAY[10, 20, 30]) AS val; – Output: Rows with 10, 20, 30
```

2. Unnesting an array column from a table using LATERAL:

```
– Assume EmployeeSkills table with employeeName and skills TEXT[] columns SELECT es.employeeName, u.skillName FROM EmployeeSkills es, LATERAL unnest(es.skills) AS u(skillName) WHERE es.employeeName = 'Alice Wonderland';
```

3. Parallel unnesting (PostgreSQL specific):

```
– Assume arrays skills and levels are parallel SELECT u.skill, u.level FROM unnest( ARRAY['SQL', 'Python'], ARRAY[5, 4] ) AS u(skill, level); – Output: – skill | level – ——-|——- – SQL | 5 – Python | 4
```

SRFs in Other SQL Engines

While PostgreSQL has robust SRF support, other databases achieve similar results differently:

- **SQL Server:** Often uses Recursive CTEs for series generation. For array-like structures (e.g., in JSON or delimited strings), functions like OPENJSON or string splitting functions are used.
- **MySQL:** Recursive CTEs are the standard for series generation. JSON_TABLE can be used to unnest JSON arrays.
- **Oracle:** Uses CONNECT BY LEVEL for hierarchical queries which can generate series. The TABLE() operator with collection types can unnest array-like structures.

It's good to know these alternatives if you ever find yourself in a different SQL land.

1.4 Why Use Them? (Advantages)

Advantages

SRFs offer significant benefits, making complex tasks simpler and often more efficient:

- **Conciseness & Readability (generate_series):** Dramatically simplifies the generation of sequences (numbers, dates) compared to verbose methods like recursive CTEs (though recursive CTEs are powerful for other things) or application-level loops. "For dates in a row, let 'generate_series' flow."
- **Data Densification (generate_series):** Excellent for creating "scaffolding" data, like a complete list of dates or months for a report, ensuring no periods are missing even if there's no actual data for them. This helps in producing complete and accurate reports.
- **Relational Operations on Array Elements (unnest):** Allows you to treat

array elements as individual rows. This means you can JOIN them, FILTER them using WHERE, GROUP BY them, and apply aggregate functions – all using standard SQL. It "normalizes" array data on-the-fly.

- **Standard SQL Integration:** SRFs fit naturally into SQL queries, particularly in the FROM clause, leveraging the full power of the SQL language for subsequent processing of the generated sets.
- **Performance (often):** Native SRFs are generally well-optimized by the database engine for their specific tasks, often outperforming manual, more complex SQL constructions or application-side logic for the same purpose.

1.5 Watch Out! (Disadvantages)

Disadvantages

While powerful, SRFs come with a few caveats:

- **Resource Consumption (`generate_series`):** Be very careful with the `start`, `stop`, and `step` parameters. Generating an extremely large series (e.g., millions or billions of rows) can consume significant CPU, memory, and time, potentially impacting database performance or even causing queries to fail. A series too grand, can crash the whole land!

```
– SELECT COUNT(*) FROM generate_series(1, 1000000000); – Be careful!
```

- **Row Explosion (`unnest`):** If you `unnest` arrays from many rows, and those arrays are large, the number of resulting rows can increase dramatically (e.g., 1000 rows each with an array of 100 elements becomes 100,000 rows after unnesting). This "row explosion" can lead to slow queries if not anticipated. Unnest with no test, puts performance to the rest.
- **Complexity with LATERAL (for beginners):** The concept of LATERAL joins, often required for SRFs that depend on other tables in the FROM clause, can be a learning curve for those new to advanced SQL.
- **Parallel Unnesting Nuances (`unnest multiple arrays`):** When unnesting multiple arrays in parallel (a PostgreSQL feature), if the arrays have different lengths, PostgreSQL will fill the "shorter" columns with NULLs to match the length of the longest array. This behavior needs to be understood to avoid unexpected NULL values.
- **Portability:** While the concept of table functions exists in many SQL databases, the specific syntax (e.g., `generate_series`, `unnest`) and features (like parallel unnesting or WITH ORDINALITY) can be specific to PostgreSQL. Code using these may not be directly portable to other RDBMSs.

2 JSON and Array Functions

Modern applications often deal with semi-structured data (JSON) and lists of items (arrays). SQL databases, especially PostgreSQL, provide powerful functions to store, manipulate, and query these data types directly within the database. JSON's the key, for data wild and free.

We'll explore functions for:

- **JSON Data:** Extracting values, expanding JSON arrays, and constructing JSON objects.
- **SQL Arrays:** Appending elements and getting array lengths. (Note: SQL arrays are distinct from JSON arrays).

2.1 What Are They? (Meanings & Values)

Meaning[a]**:** These functions provide tools to work with complex data types: JSON (JavaScript Object Notation) for flexible, hierarchical data, and SQL arrays for ordered lists of homogeneous elements.
Values/Output: Outputs vary:

- JSON functions can return JSON values/objects/arrays, scalar text values, or sets of rows (if they are SRFs).
- Array functions typically return modified arrays or scalar information about arrays (like length).

[a]After to this lecture, check all the functions for arrays and jsons as official features of PostgreSQL.

JSON Functions

PostgreSQL offers two JSON data types: `JSON` (stores an exact copy of the input text) and `JSONB` (stores a decomposed binary format, generally more efficient for processing). **JSONB is usually preferred.**

- **`json_extract_path(json, path_elems TEXT[]) / jsonb_extract_path(...)`**
  (Operators: `jsonb -> text`, `jsonb -» text`, `jsonb #> text[]`, `jsonb #» text[]`)
  - **Meaning:** Navigates a JSON structure using a path of keys/indices to extract a specific JSON sub-object, array, or scalar value. Path to the prize, JSON will advise.
  - **Values/Output:** `json_extract_path` (and `->`, `#>`) return a JSON/JSONB value. The `_text` versions (and `-»`, `#»`) return the value as `TEXT`.
- **`json_array_elements(json_array) / jsonb_array_elements(...)`**
  (SRF - Set Returning Function)
  - **Meaning:** Expands the outermost JSON array into a set of rows. Each element of the JSON array becomes a separate row containing that JSON element.
  - **Values/Output:** A set of rows. Each row has a single column (usually named `value`) of type JSON/JSONB. The `_text` versions return elements as `TEXT`.
- **`json_build_object(key1, val1 [, key2, val2 ...]) / jsonb_build_object(...)`**

  - **Meaning:** Constructs a new JSON object from a variadic list of key-value pairs. Keys must be text; values can be any SQL type that's convertible to JSON.
  - **Values/Output:** A JSON/JSONB object.

SQL Array Functions

These operate on native SQL array types like `INT[]`, `TEXT[]`, etc.

- **`array_append(array, element)`**
  - **Meaning:** Appends a new element to the end of an existing SQL array. The element must be compatible with the array's base type. Add to the end, your array will extend.
  - **Values/Output:** A new SQL array with the element added.
- **`array_length(array, dimension_number)`**
  - **Meaning:** Returns the length (number of elements) of the specified dimension of an SQL array. For one-dimensional arrays, the dimension number is 1. To know its might, `array_length` sheds light.
  - **Values/Output:** An integer representing the length.
  - **Note:** `cardinality(array)` is often a more convenient way to get the total number of elements in an array, regardless of dimensions, or the length of a 1D array.

2.2 Relations: How They Play with Others

JSON and Array functions integrate with the SQL ecosystem, building upon previous concepts.

Internal Relations (Function Chaining)

- The output of `json_extract_path` can be a JSON array, which can then be passed to `json_array_elements`.
- Values extracted using `json_extract_path_text` can be used as arguments when constructing new objects with `json_build_object`.
- `array_length` might be used to check an array's size before deciding to `array_append` or use `unnest` (an SRF discussed previously).

Relations with Previous SQL Concepts

- **Data Types:** Understanding the difference between `JSON`, `JSONB`, and SQL arrays (e.g., `TEXT[]`, `INTEGER[]`) is fundamental. JSONB is usually recommended over JSON for efficiency in PostgreSQL.
- **`WHERE` Clause:**
  - **JSON:** Operators like `->>`, `jsonb_extract_path_text`, and containment operators (`@>`, `<@`) are vital for filtering rows based on JSON content.
  - **SQL Arrays:** Array operators like `ANY`, `ALL`, element access (`myArray[1]`), and containment (`@>`, `<@`) are used for filtering.
- **`SELECT` Clause:**
  - Used to extract parts of JSON/arrays for display or further computation.
  - `json_build_object` is primarily used here to create JSON outputs.

- **Casting (::):** Values extracted from JSON (often as `TEXT` via `-»` or `_text` functions) frequently need to be cast to specific SQL types (`INTEGER`, `BOOLEAN`, `DATE`, `NUMERIC`) for comparisons or calculations.
- **Aggregators (Advanced: `JSON_AGG`, `ARRAY_AGG`; Basic: `SUM`, `AVG`):**
  - `JSON_AGG` and `ARRAY_AGG` construct JSON arrays or SQL arrays from sets of rows. `json_array_elements` and `unnest` are their conceptual opposites (deconstructors).
  - Standard aggregators can operate on numeric values extracted from JSON or SQL arrays (after unnesting/element extraction and casting).
- **SRFs (`json_array_elements`, `unnest`):** `json_array_elements` is an SRF. Like `unnest`, it's typically used with a `LATERAL` join in the `FROM` clause to expand JSON array elements into rows.
- **String Functions (`LOWER`, `LIKE`, `SUBSTRING`, etc.):** Can be applied to textual data extracted from JSON fields.
- **Common Table Expressions (CTEs):** Extremely helpful for organizing complex JSON or array processing steps, improving query readability and maintainability.
- **Joins:** Queries can join tables based on values extracted from JSON columns or elements from SQL array columns, often after casting.

2.3 How to Use Them: Structures & Syntax

Here's a practical look at using these functions in PostgreSQL.

Categorization of Functions

- **JSON Accessors/Extractors:** Get data out of JSON.
  - `jsonb_extract_path(jsonb, TEXT...)`
  - `jsonb_extract_path_text(jsonb, TEXT...)`
  - Operators: `->` (get JSONB object field/array element), `-»` (get as TEXT), `#>` (get JSONB at path), `#»` (get as TEXT at path).
- **JSON Deconstructors (SRFs):** Expand JSON structures into rows.
  - `jsonb_array_elements(jsonb_array)`
  - `jsonb_array_elements_text(jsonb_array)`
  - Others (not in primary list but related): `jsonb_each()`, `jsonb_object_keys()`.
- **JSON Constructors:** Build new JSON values.
  - `jsonb_build_object(key TEXT, val ANY [, ...])`
  - `jsonb_build_array(val ANY [, ...])`
  - `to_jsonb(ANY)`
- **SQL Array Modifiers:** Change SQL arrays.
  - `array_append(array ANYARRAY, element ANYELEMENT)`
  - `array_prepend(element ANYELEMENT, array ANYARRAY)`
  - `array_cat(array1 ANYARRAY, array2 ANYARRAY)`
- **SQL Array Information/Access:** Get metadata or elements.
  - `array_length(array ANYARRAY, dim INT)`
  - `cardinality(array ANYARRAY)` (total elements)
  - Element access: `myArray[index]`, Slicing: `myArray[lower:upper]`

JSON Function Examples (using JSONB)

1. **Extracting values using `jsonb_extract_path_text` and operators:** Assume a table `ProductCatalog` with a JSONB column `specifications`. `specifications`: `{"display": {"type": "AMOLED", "size": "1.4"}, "batteryLife": "7 days"}`

```
SELECT productName, – Using function jsonb_extract_path_text(specifications, 'display', 'type') AS displayTypeFunc, – Using -» operator for top-level key specifications -» 'batteryLife' AS batteryLifeOp, – Using -> for nested object, then -» for text value specifications -> 'display' -» 'size' AS displaySizeOp FROM ProductCatalog WHERE productId = 1001;
```

2. **Expanding a JSON array using `jsonb_array_elements_text`:** Assume `specifications` contains `"sensors": ["heart rate", "gps", "spo2"]`.

```
SELECT pc.productName, sensorElement.value AS sensor FROM ProductCatalog pc, LATERAL jsonb_array_elements_text(pc.specifications -> 'sensors') AS sensorElement WHERE pc.productId = 1001; – Output: Multiple rows for product 1001, one for each sensor.
```

3. **Constructing a JSON object using `jsonb_build_object`:**

```
SELECT productId, productName, jsonb_build_object( 'category', category, 'primaryTag', tags[1] – Assuming 'tags' is a TEXT[] SQL array ) AS productSummary FROM ProductCatalog WHERE productId = 1001;
```

SQL Array Function Examples

Assume a table `EmployeeSkills` with `TEXT[]` column `skills`.

1. **Appending an element using `array_append`:**

```
– To show the result (UPDATE would persist it) SELECT employeeId, skills AS originalSkills, array_append(skills, 'Communication') AS updatedSkills FROM EmployeeSkills WHERE employeeId = 101;
– Actual update: – UPDATE EmployeeSkills – SET skills = array_append(skills, 'Communication') – WHERE employeeId = 101;
```

2. **Getting array length using `array_length` or `cardinality`:**

```
SELECT employeeId, skills, array_length(skills, 1) AS skillCountLength, – For 1st dimension cardinality(skills) AS skillCountCardinality – Total elements FROM EmployeeSkills WHERE employeeId = 101;
```

3. **Accessing an array element:**

```
SELECT employeeId, skills[1] AS firstSkill – Arrays are 1-indexed by default in
PostgreSQL FROM EmployeeSkills WHERE employeeId = 101;
```

Usage in Other SQL Engines (JSON)

Most modern RDBMSs have robust JSON support, though syntax varies:

- **SQL Server:** Uses `JSON_VALUE` (extract scalar), `JSON_QUERY` (extract object/array), `OPENJSON` (shred JSON to rows), `FOR JSON PATH/AUTO` (construct JSON).
- **MySQL:** Functions like `JSON_EXTRACT`, `JSON_OBJECT`, `JSON_ARRAY`, `JSON_TABLE` (shred to rows). Operators like `->`, `-»`.
- **Oracle:** Functions like `JSON_VALUE`, `JSON_QUERY`, `JSON_TABLE`, `JSON_OBJECT`, `JSON_ARRAYAGG`.

SQL Array support is less standardized across databases compared to JSON.

2.4 Why Use Them? (Advantages)

Advantages

JSON and Array functions bring flexibility and power to your SQL toolkit:

JSON Functions:

- **Schema Flexibility:** JSONB allows you to store and query data that doesn't have a fixed, predefined structure. This is great for evolving requirements, sparse attributes, or data from external APIs. It's like having a data field that says, "bring what you got!"
- **Rich Hierarchical Data Representation:** JSON naturally handles nested objects and arrays, ideal for complex configurations, product specifications, or document-like data.
- **Powerful Querying Capabilities:** PostgreSQL provides a rich set of operators and functions to dissect, transform, and filter JSON data directly within the database, reducing the need to pull large JSON objects into application code for processing.
- **Interoperability:** JSON is a ubiquitous format for data exchange on the web. Storing and generating JSON directly in the database simplifies integration with APIs and web services.

SQL Array Functions:

- **Storing Ordered Collections:** SQL arrays are excellent for storing ordered lists of simple, homogeneous items (like tags, historical values, multiple phone numbers) directly within a column. An array so neat, makes lists complete.
- **Conciseness for Simple Lists:** For simple one-to-many relationships where the "many" side is just a list of values, an array column can be more concise and sometimes easier to manage than a separate related table.
- **Atomic Array Operations:** Functions like `array_append`, `array_prepend`, and `array_cat` provide convenient ways to modify arrays.
- **Efficient Searching (with GIN indexes):** When indexed properly (e.g.,

with GIN indexes in PostgreSQL), searching for elements within arrays (e.g., using `ANY`, `ALL`, `@>`) can be very performant.

2.5 Watch Out! (Disadvantages)

Disadvantages

With great power comes great responsibility. Here are things to be mindful of:
JSON Functions:

- **Performance on Unindexed/Complex Queries:** Querying JSONB without appropriate GIN indexes, or using very complex/deep path extractions on large JSON documents, can lead to slow performance. The database might need to parse many large JSON objects. JSON deep and wide, performance may hide.
- **Loss of Strict Schema Enforcement:** The flexibility of JSON means the database doesn't enforce the structure *within* the JSON document itself (beyond validating it's well-formed JSON). This responsibility shifts to the application or requires careful query construction.
- **Query Verbosity:** Accessing deeply nested JSON data can sometimes lead to longer, more complex SQL queries compared to accessing data in normalized relational tables.
- **Overuse / Misuse:** JSON is not a universal replacement for relational design. If data is inherently structured and relational, using traditional tables and columns is often more efficient and maintainable. Don't use a JSON hammer for every data nail.
- **Type Handling:** Data extracted from JSON (especially with `-»` or `_text` functions) is text. It often needs explicit casting to the correct SQL type for comparisons, calculations, or joins, which adds a bit of overhead and potential for errors if not handled carefully.

SQL Array Functions:

- **Querying Complexity for Certain Tasks:** While simple element checks are easy, more complex queries on array elements (e.g., "find rows where at least 3 elements meet a condition") can be less straightforward than querying normalized related tables. Often requires `unnest`-ing.
- **Normalization Concerns (1NF):** Storing multiple distinct pieces of information that could each be their own attribute or row in an array can arguably violate First Normal Form (1NF), especially if those elements themselves have internal structure or if you frequently need to query individual elements independently. Use for genuine lists of similar items.
- **Modification Overhead for Large Arrays:** While functions like `array_append` are convenient, operations like updating or removing an element from the middle of a very large array can be less efficient than targeted DML on a normalized table structure, as it might involve rewriting the entire array.
- **Search Performance (without GIN indexes):** Searching for elements within arrays without a GIN index will typically result in a sequential scan of the array for each row, which is inefficient for large tables or large arrays.

- **Limited Dimensionality/Structure:** SQL arrays are best for simple lists or rectangular multi-dimensional arrays. They are not as flexible as JSON for representing complex, jagged, or deeply nested hierarchical structures.