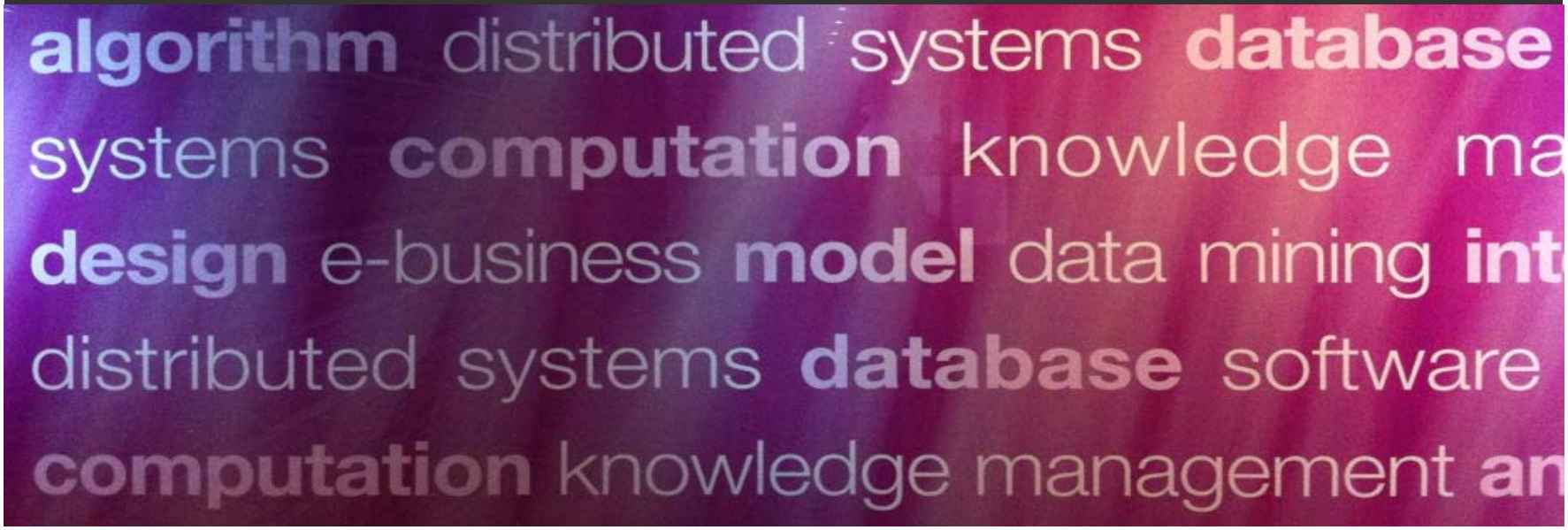
MONASH University

Information Technology

FIT3176 Advanced Database Design

Topic 11: JSON in Oracle Database

Dr. Minh Le
Minh.Le@monash.edu



This unit…

1. Advanced Database Design
   - E/R is not complete enough
   - EER, covering superclass and subclass

2. SQL and PL/SQL
   - Trigger, Procedures/Functions, Packages

3. XML and XML DB
   - XML and XML Schema
   - XML in Oracle (XPath)

4. **JSON and JSON DB**
   - JSON Documents and Schema
   - **JSON in Oracle** (this week)

Learning Objectives

By the end of this week you should be able to:

- Appreciate why storing JSON data in Oracle Database is necessary
- Store and manage JSON data in Oracle Database
- Use JSON Path expressions to traverse and parse JSON documents
- Query JSON data using Oracle constructs such as JSON_EXISTIS, JSON_VALUE, JSON_QUERY, JSON_TABLE
- Generate JSON data from relational data using Oracle functions such as JSON_OBJECT, JSON_ARRAY, JSON_OBJECTAGG, JSON_ARRAYAGG

References

1. Oracle Database JSON Developer's Guide, https://docs.oracle.com/cloud/latest/db122/ADJSN/toc.htm, accessed date: 11 May 2018
2. Beda Hammerschmidt, Simple Queries, https://blogs.oracle.com/database/first-steps-with-json%3a-simple-queries, accessed date: 11 May 2018
3. Beda Hammerschmidt, The new SQL/JSON Query operators (Part1: JSON_VALUE), https://blogs.oracle.com/database/the-new-sqljson-query-operators-part1%3a-json_value, accessed date: 11 May 2018
4. Beda Hammerschmidt, The new SQL/JSON Query operators (Part2: JSON_QUERY), https://blogs.oracle.com/jsondb/the-new-sqljson-query-operators-part2%3a-jsonquery, accessed date: 11 May 2018
5. Beda Hammerschmidt , Storing JSON data in the Oracle database, https://blogs.oracle.com/database/storing-json-data-in-the-oracle-database, accessed date: 11 May 2018
6. Beda Hammerschmidt, Generating JSON data, https://blogs.oracle.com/jsondb/generating-json-data, accessed date: 11 May 2018

Outline

- Introduction to JSON in Oracle Database
  - Why storing JSON documents in Oracle is necessary?
- Storing and managing JSON Data
  - Creating a table with a JSON Column
  - Lax and Strict well-formed JSON Column
- Querying JSON Data
  - Overview of SQL/JSON Path Expressions
  - JSON_EXISTS, JSON_TABLE, JSON_QUERY JSON_VALUE
- Generation of JSON Data
  - JSON_OBJECT, JSON_ARRAY, JSON_OBJECTAGG, JSON_ARRAYAGG

*Source: www.json.org

Why Storing JSON in Oracle Database

- Provide all of benefits of relational database features for use with JSON, including transactions, indexing, declarative querying, and views.
- Use SQL to join JSON data with relational data
- Project JSON data relationally, making it available for relational processes and tools.
- Access JSON data stored in the database the same way as the other database data can be accessed.

➢Storing and managing JSON Data (1)

- JSON data in Oracle Database using columns can be stored using data types: VARCHAR2, CLOB, or BLOB.
  - Use VARCHAR2(4000) if you are sure that your largest JSON documents do not exceed 4000 bytes (or characters)
  - Use VARCHAR2(32767) if you know that some of your JSON documents are larger than 4000 bytes (or characters) and you are sure than none of the documents exceeds 32767 bytes (or characters)
  - Use BLOB (binary large object) or CLOB (character large object) storage if you know that you have some JSON documents that are larger than 32767 bytes (or characters)
- Oracle recommends that you use BLOB, not CLOB storage.
  - choosing BLOB over CLOB storage has the advantage that it avoids the need for character-set conversion when storing the JSON document

➢Storing and managing JSON Data (2)

- A downside of choosing BLOB storage over CLOB (for JSON or any other kind of data) is that it is sometimes more difficult to work with BLOB content using command-line tools such as SQL*Plus
  - When selecting data from a BLOB column, if you want to view it as printable text then you must use SQL function to_clob.
  - When performing insert or update operations on a BLOB column, you must explicitly convert character strings to BLOB format using SQL function rawtohex.
- Ensure That JSON Columns contain Well-Formed JSON Data
  - SQL/JSON condition is json to check whether or not some JSON data is well formed.

➢Storing and managing JSON Data (3)

- You can control which syntax you require a given column of JSON data to conform to: the standard definition (strict syntax) or a JavaScript-like syntax (lax syntax).
- SQL/JSON conditions is json and is not json test whether JSON data is well-formed or not.
- Well-formed data means syntactically correct data. In Oracle, JSON data can be well-formed in two senses, referred to as strict and lax syntax. The default SQL/JSON syntax for Oracle Database is lax.

Example 4-1 Using IS JSON in a Check Constraint to Ensure JSON Data is Well-Formed

```
CREATE TABLE j_purchaseorder
  (id          VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document VARCHAR2 (23767)
   CONSTRAINT ensure_json CHECK (po_document IS JSON));
```

➢Storing and managing JSON Data (4)

- Inserting first JSON data into a VARCHAR2 JSON column.

Example 4-2 Inserting JSON Data Into a VARCHAR2 JSON Column

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-DEC-2014'),
    '{"PONumber"             : 1600,
      "Reference"            : "ABULL-20140421",
      "Requestor"            : "Alexis Bull",
      "User"                 : "ABULL",
      "CostCenter"           : "A50",
      "ShippingInstructions" : {"name"    : "Alexis Bull",
                                "Address" : {"street"  : "200 Sporting Green",
                                             "city"    : "South San Francisco",
                                             "state"   : "CA",
                                             "zipCode" : 99236,
                                             "country" : "United States of America"},
                                "Phone"   : [{"type" : "Office", "number" : "909-555-7307"},
                                             {"type" : "Mobile", "number" : "415-555-1234"}]},
      "Special Instructions" : null,
      "AllowPartialShipment" : true,
      "LineItems"            : [{"ItemNumber" : 1,
                                 "Part"       : {"Description" : "One Magic Christmas",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 13131092899},
                                 "Quantity"   : 9.0},
                                {"ItemNumber" : 2,
                                 "Part"       : {"Description" : "Lethal Weapon",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 85391628927},
                                 "Quantity"   : 5.0}]}');
```

➢Storing and managing JSON Data (5)

- Inserting second JSON data into a VARCHAR2 JSON column.

```
INSERT INTO j_purchaseorder
  VALUES (
    SYS_GUID(),
    to_date('30-DEC-2014'),
    '{"PONumber"             : 672,
      "Reference"            : "SBELL-20141017",
      "Requestor"            : "Sarah Bell",
      "User"                 : "SBELL",
      "CostCenter"           : "A50",
      "ShippingInstructions" : {"name"    : "Sarah Bell",
                                "Address" : {"street"  : "200 Sporting Green",
                                             "city"    : "South San Francisco",
                                             "state"   : "CA",
                                             "zipCode" : 99236,
                                             "country" : "United States of America"},
                                "Phone"   : "983-555-6509"},
      "Special Instructions" : "Courier",
      "LineItems"            : [{"ItemNumber" : 1,
                                 "Part"       : {"Description" : "Making the Grade",
                                                 "UnitPrice"   : 20,
                                                 "UPCCode"     : 27616867759},
                                 "Quantity"   : 8.0},
                                {"ItemNumber" : 2,
                                 "Part"       : {"Description" : "Nixon",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 717951002396},
                                 "Quantity"   : 5}
                                {"ItemNumber" : 3,
                                 "Part"       : {"Description" : "Eric Clapton: Best Of 1981-1999",
                                                 "UnitPrice"   : 19.95,
                                                 "UPCCode"     : 75993851120},
                                 "Quantity"   : 5.0}
                                ]}');
```

Lax and Strict Well-formed JSON Data (1)

- Examples of JASON Objects that are considered lax and/ore strict well-formed. Note: Oracle calling a Field as a Key in a Key-Value pair

Table 5-1 JSON Object Field Syntax Examples

Example	Well-Formed?
`"part number": 1234`	Lax and strict: yes. Space characters are allowed.
`part number: 1234`	Lax (and strict): **no**. Whitespace characters, including space characters, are not allowed in unquoted names.
`"part\tnumber": 1234`	Lax and strict: yes. Escape sequence for tab character is allowed.
`"part  number": 1234`	Lax and strict: **no**. Unescaped tab character is not allowed. Space is the only unescaped whitespace character allowed.
`"\"part\"number": 1234`	Lax and strict: yes. Escaped double quotation marks are allowed, if name is quoted.
`\"part\"number: 1234`	Lax and strict: **no**. Name must be quoted.
`'\"part\"number': 1234`	Lax: yes, **strict: no**. Single-quoted names (object fields and strings) are allowed for lax syntax only. Escaped double quotation mark is allowed in a quoted name.
`"pärt : number":1234`	Lax and strict: yes. Any Unicode character is allowed in a quoted name. This includes whitespace characters and characters, such as colon (`:`), that are structural in JSON.
`part:number:1234`	Lax (and strict): **no**. Structural characters are not allowed in unquoted names.

Lax and Strict Well-formed JSON Data (2)

- To ensure that JSON data is strictly well-formed, (STRICT) should be appending after IS JSON condition as shown in the below example.

Example 5-1 Using IS JSON in a Check Constraint to Ensure JSON Data is Strictly Well-Formed (Standard)

```
CREATE TABLE j_purchaseorder
  (id          VARCHAR2 (32) NOT NULL PRIMARY KEY,
   date_loaded TIMESTAMP (6) WITH TIME ZONE,
   po_document VARCHAR2 (32767)
   CONSTRAINT ensure_json CHECK (po_document IS JSON (STRICT)));
```

Querying JSON Data- Path Expressions (1)

- Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.
- When JSON data is stored in the database you can query it using path expressions that are somewhat similar to XQuery or XPath expressions for XML data.
- SQL/JSON path expressions have a simple syntax. A path expression selects zero or more JSON values that match, or satisfy, it.
- A JSON Path Expression can target: an entire object, a scalar value, an array or a specific object.
- Generally, a SQL/JSON path expression and some JSON data are passed to a SQL/JSON query function (e.g. JSON_VALUE or JSON_QUERY) to retrieve desired JSON data. They can also be passed to a condition function (e.g. JSON_EXISTS) to return true or false.

Querying JSON Data- Path Expressions (2)

- For example, passing a SQL/JSON path expression to JSON_EXISTS, it returns true if at least one value matches and false if no value matches.
  - If a single value matches (that is, JSON_EXISTS returns true), then SQL/JSON function JSON_VALUE returns that value if it is scalar and raises an error if it is non-scalar.
  - If no value matches the path expression then JSON_VALUE returns SQL NULL.
  - If there is more than one match, using SQL/JSON function JSON_QUERY, it can return all of the matching values, that is, it can return multiple values.
- Matching is case-sensitive.

Querying JSON Data- Path Expressions (3)

- All JSON Path Expressions begin with $ and are followed by zero or more steps, each of which can be an object step or an array step.
- An object step is a period (.), followed by an object key name or the *
- An array step is started with left bracket ([) followed by the * or one or more array indexes and followed by a right bracket (]).
- JSON Path Expressions are still in the development stage.

Querying JSON Data- Path Expressions (4)

JSON Path Operators

Operator	Description
`$`	The root element to query. This starts all path expressions.
`@`	The current node being processed by a filter predicate.
`*`	Wildcard. Available anywhere a name or numeric are required.
`..`	Deep scan. Available anywhere a name is required.
`.<name>`	Dot-notated child
`['<name>' (, '<name>')]`	Bracket-notated child or children
`[<number> (, <number>)]`	Array index or indexes
`[start:end]`	Array slice operator
`[?(<expression>)]`	Filter expression. Expression must evaluate to a boolean value.

*Source: https://github.com/json-path/JsonPath

Querying JSON Data- Path Expressions (5)

JSON Path Functions

Function	Description	Output
min()	Provides the min value of an array of numbers	Double
max()	Provides the max value of an array of numbers	Double
avg()	Provides the average value of an array of numbers	Double
stddev()	Provides the standard deviation value of an array of numbers	Double
length()	Provides the length of an array	Integer

*Source: https://github.com/json-path/JsonPath

Querying JSON Data- Path Expressions (6)

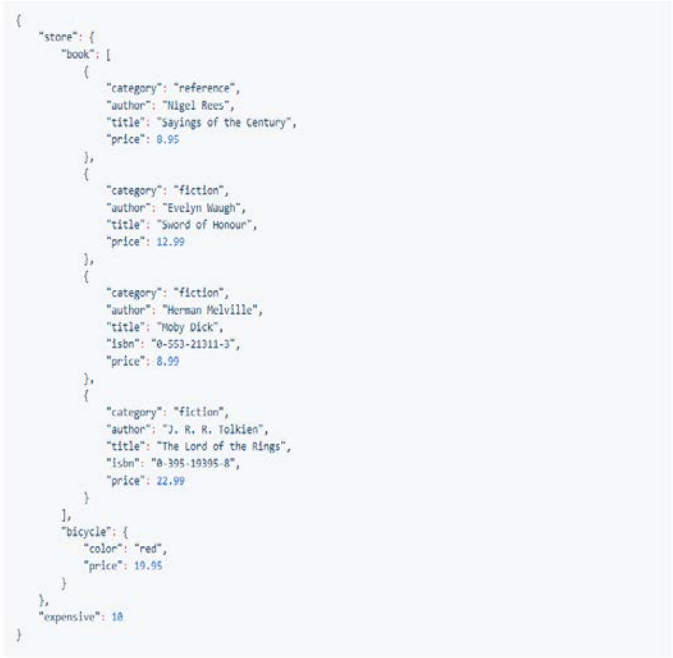
Example of a JSON Document


```
{
    "store": {
        "book": [
            {
                "category": "reference",
                "author": "Nigel Rees",
                "title": "Sayings of the Century",
                "price": 8.95
            },
            {
                "category": "fiction",
                "author": "Evelyn Waugh",
                "title": "Sword of Honour",
                "price": 12.99
            },
            {
                "category": "fiction",
                "author": "Herman Melville",
                "title": "Moby Dick",
                "isbn": "0-553-21311-3",
                "price": 8.99
            },
            {
                "category": "fiction",
                "author": "J. R. R. Tolkien",
                "title": "The Lord of the Rings",
                "isbn": "0-395-19395-8",
                "price": 22.99
            }
        ],
        "bicycle": {
            "color": "red",
            "price": 19.95
        }
    },
    "expensive": 10
}
```


*Source: https://github.com/json-path/JsonPath

Querying JSON Data- Path Expressions (6)

Example of JSON Path Expressions



```
{
    "store": {
        "book": [
            {
                "category": "reference",
                "author": "Nigel Rees",
                "title": "Sayings of the Century",
                "price": 8.95
            },
            {
                "category": "fiction",
                "author": "Evelyn Waugh",
                "title": "Sword of Honour",
                "price": 12.99
            },
            {
                "category": "fiction",
                "author": "Herman Melville",
                "title": "Moby Dick",
                "isbn": "0-553-21311-3",
                "price": 8.99
            },
            {
                "category": "fiction",
                "author": "J. R. R. Tolkien",
                "title": "The Lord of the Rings",
                "isbn": "0-395-19395-8",
                "price": 22.99
            }
        ],
        "bicycle": {
            "color": "red",
            "price": 19.95
        }
    },
    "expensive": 10
}
```

JsonPath (click link to try)	Result
$.store.book[*].author	The authors of all books
$..author	All authors
$.store.*	All things, both books and bicycles
$.store..price	The price of everything
$..book[2]	The third book
$..book[-2]	The second to last book
$..book[0,1]	The first two books
$..book[:2]	All books from index 0 (inclusive) until index 2 (exclusive)
$..book[1:2]	All books from index 1 (inclusive) until index 2 (exclusive)
$..book[-2:]	Last two books
$..book[2:]	Book number two from tail
$..book[?(@.isbn)]	All books with an ISBN number
$.store.book[?(@.price < 10)]	All books in store cheaper than 10
$..book[?(@.price <= $['expensive'])]	All books in store that are not "expensive"
$..book[?(@.author =~ /.*REES/i)]	All books matching regex (ignore case)
$..*	Give me every thing
$..book.length()	The number of books

*Source: https://github.com/json-path/JsonPath

Querying JSON Data- JSON Operators

- JSON_EXISTS
  - Filter rows based on JSON-PATH expressions.
  - It is used in the WHERE clause
- JSON_VALUE
  - Returns a scalar value from a JSON Document
  - It is often used in the SELECT clause.
  - It can also be used in the WHERE and ORDER BY clauses.
- JSON_QUERY
  - Returns fragments of a JSON document. A fragment can be a JSON object or a JSON array.
  - It is often used in the SELECT clause.
- JSON_TABLE
  - Projects specific JSON data into VARCHAR2, NUMBER, DATE, TIMESTAMP columns, etc. That is, it creates a virtual relational table from JSON Data for views.
  - It is often used in the FROM clause

JSON Operators - JSON_EXISTS (1)

- JSON_EXISTS
  - Filter rows based on JSON-PATH expressions
  - SQL/JSON condition json_exists returns true for documents containing data that matches a SQL/JSON path expression

Example 15-1 JSON_EXISTS: Path Expression Without Filter

This example selects purchase-order documents that have a line item whose part description contains a UPC code entry.

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$.LineItems.Part.UPCCode');
```

JSON Operators - JSON_EXISTS (2)

Example 15-2 JSON_EXISTS: Current Item and Scope in Path Expression Filters

This example shows three *equivalent* ways to select documents that have a line item whose part contains a UPC code with a value of `85391628927`.

```
SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$?(@.LineItems.Part.UPCCode == 85391628927)');

SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$.LineItems?(@.Part.UPCCode == 85391628927)');

SELECT po.po_document FROM j_purchaseorder po
  WHERE json_exists(po.po_document, '$.LineItems.Part?(@.UPCCode == 85391628927)');
```

JSON Operators – JSON_VALUE

- JSON_VALUE
  - Returns a scalar value from a JSON Document
  - Has two required arguments and accepts optional returning and error clauses.
  - The first argument to json_value is a SQL expression that returns an instance of a scalar SQL data type (that is, not an object or collection data type (e.g., a tuple)).
  - The second argument to json_value is a SQL/JSON path expression followed by optional clauses RETURNING, ON ERROR, and ON EMPTY

Example 16-1 JSON_VALUE: Two Ways to Return a JSON Boolean Value in SQL

```
SELECT json_value(po_document, '$.AllowPartialShipment')
 FROM j_purchaseorder;

SELECT json_value(po_document, '$.AllowPartialShipment' RETURNING NUMBER)
  FROM j_purchaseorder;
```

JSON Operators – JSON_QUERY (1)

- JSON_QUERY
  - Returns fragments of a JSON document. A fragment can be a JSON object or a JSON array.
  - The first argument to json_query is a SQL expression that returns an instance of a scalar SQL data type.
  - The second argument to json_query is a SQL/JSON path expression followed by optional clauses RETURNING, WRAPPER, ON ERROR, and ON EMPTY. The path expression can target any number of JSON values.
  - In the RETURNING clause you can specify only data type VARCHAR2; you cannot specify NUMBER.

JSON Operators - JSON_QUERY (2)

- Example 17-1 shows an example of the use of SQL/JSON function json_query with an array wrapper. For each document it returns a VARCHAR2 value whose contents represent a JSON array with elements the phone types, in an unspecified order. For the document in Example 4-2 the phone types are "Office" and "Mobile", and the array returned is either ["Mobile", "Office"] or ["Office", "Mobile"].

Example 17-1 Selecting JSON Values Using JSON_QUERY

```
SELECT json_query(po_document, '$.ShippingInstructions.Phone[*].type'
                               WITH WRAPPER)
  FROM j_purchaseorder;
```

JSON Operators - JSON_TABLE (1)

- JSON_TABLE
  - Projects specific JSON data into VARCHAR2, NUMBER, DATE, TIMESTAMP columns, etc. That is, it creates a virtual relational table from JSON Data for views.
  - Allows us to insert this virtual table into a pre-existing database table, or to query it using SQL
  - A common use of json_table is to create a view of JSON data. This lets applications, tools, and programmers operate on JSON data without consideration of the syntax of JSON or JSON path expressions.

JSON Operators - JSON_TABLE - EXAMPLES

Example 18-3 Projecting an Entire JSON Array as JSON Data

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(po_document, '$'
         COLUMNS (requestor VARCHAR2(32 CHAR) PATH '$.Requestor',
                  ph_arr    VARCHAR2(100 CHAR) FORMAT JSON
                            PATH '$.ShippingInstructions.Phone')) AS "JT";
```

Example 18-4 Projecting Elements of a JSON Array

```
SELECT jt.*
  FROM j_purchaseorder,
       json_table(po_document, '$.ShippingInstructions.Phone[*]'
         COLUMNS (phone_type VARCHAR2(10) PATH '$.type',
                  phone_num  VARCHAR2(20) PATH '$.number')) AS "JT";

PHONE_TYPE      PHONE_NUM
----------      ---------
Office          909-555-7307
Mobile          415-555-1234
```

Generation of JSON Data (1)

- SQL/JSON functions json_object, json_array, json_objectagg, and json_arrayagg are used to construct JSON data from non-JSON data in the database. The JSON data is returned as a SQL VARCHAR2 value.
- By nesting these functions, complex, hierarchical JSON documents can be generated
- These functions allow us to
  - Construct well-formed JSON documents
  - Generate an entire set of JSON documents using a single SQL statement.
  - Because only the generated documents are returned to a client, network overhead is minimized: there is at most one round trip per document generated.

Generation of JSON Data (2)

- Functions json_object and json_array construct a JSON object or array, respectively, given as arguments SQL name–value pairs and values, respectively.
- Functions json_objectagg, and json_arrayagg are aggregate SQL functions. They transform information that is contained in the rows of a grouped SQL query into JSON objects and arrays, respectively.
- For json_objectagg, the order of object members is unspecified. For json_arrayagg, the order of array elements reflects the query result order. You can use SQL ORDER BY in the query to control the array element order.

Generation of JSON Data (3) - Examples

Examples of USING JSON_OBJECT

Example 20-1 Declaring an Input Value To Be JSON

This example specifies `FORMAT JSON` for SQL string values `'true'` and `'false'`, in order that the JSON Boolean values `true` and `false` are used.

```
SELECT json_object('name'          VALUE first_name || ' ' || last_name,
                   'hasCommission' VALUE
                     CASE WHEN commission_pct IS NULL THEN 'false' ELSE 'true'
                     END FORMAT JSON)
  FROM employees WHERE first_name LIKE 'W%';

JSON_OBJECT('NAME'ISFIRST_NAME||''||LAST_NAME,'
-----------------------------------------------
{"name":"William Gietz","hasCommission":false}
{"name":"William Smith","hasCommission":true}
{"name":"Winston Taylor","hasCommission":false}
```

Generation of JSON Data (4) - Examples

Examples of USING JSON_OBJECT

Example 20-2 Using JSON_OBJECT to Construct JSON Objects

This example constructs a JSON object for each employee of table `hr.employees` (from standard database schema `HR`) whose salary is less than 15000. The object includes, as the value of its field `contactInfo`, an object with fields `mail` and `phone`.

Because the return value of `json_object` is JSON data, `FORMAT JSON` is deduced for the input format of field `contactInfo` — the explicit `FORMAT JSON` here is not needed.

```
SELECT json_object('id'          VALUE employee_id,
                   'name'        VALUE first_name || ' ' || last_name,
                   'hireDate'    VALUE hire_date,
                   'pay'         VALUE salary,
                   'contactInfo' VALUE json_object('mail'  VALUE email,
                                                   'phone' VALUE phone_number)
                                       FORMAT JSON)
  FROM employees
  WHERE salary > 15000;

-- The query returns rows such as this (pretty-printed here for clarity):

{"id":101,
 "name":"Neena Kochhar",
 "hireDate":"21-SEP-05",
 "pay":17000,
 "contactInfo":{"mail":"NKOCHHAR",
                "phone":"515.123.4568"}}
```

Generation of JSON Data (5) - Examples

Examples of USING JSON_OBJECT

Example 20-3 Using JSON_OBJECT With ABSENT ON NULL

This example queries table `hr.locations` from standard database schema `HR` to create JSON objects with fields `city` and `province`.

The default `NULL`-handling behavior for json_object is `NULL ON NULL`.

In order to prevent the creation of a field with a `null` JSON value, the example uses `ABSENT ON NULL`. The `NULL` SQL value for column `state_province` when column `city` has value `'Singapore'` means that no `province` field is created for that location.

```
SELECT JSON_OBJECT('city'     VALUE city,
                   'province' VALUE state_province ABSENT ON NULL)
  FROM locations
  WHERE city LIKE 'S%';

JSON_OBJECT('CITY'ISCITY,'PROVINCE'ISSTATE_PROVINCEABSENTONNULL)
--------------------------------------------------------------------
{"city":"Southlake","province":"Texas"}
{"city":"South San Francisco","province":"California"}
{"city":"South Brunswick","province":"New Jersey"}
{"city":"Seattle","province":"Washington"}
{"city":"Sydney","province":"New South Wales"}
{"city":"Singapore"}
{"city":"Stretford","province":"Manchester"}
{"city":"Sao Paulo","province":"Sao Paulo"}
```

Generation of JSON Data (6) - Examples

An Example of USING JSON_ARRAY

Example 20-4 Using JSON_ARRAY to Construct a JSON Array

This example constructs a JSON object for each job in database table `hr.jobs` (from standard database schema `HR`). The fields of the objects are the job title and salary range. The salary range (field `salaryRange`) is an array of two numeric values, the minimum and maximum salaries for the job. These values are taken from SQL columns `min_salary` and `max_salary`.

```
SELECT json_object('title'       VALUE job_title,
                   'salaryRange' VALUE json_array(min_salary, max_salary))
  FROM jobs;

JSON_OBJECT('TITLE'ISJOB_TITLE,'SALARYRANGE'ISJSON_ARRAY(MIN_SALARY,MAX_SALARY))
--------------------------------------------------------------------------------
{"title":"President","salaryRange":[20080,40000]}
{"title":"Administration Vice President","salaryRange":[15000,30000]}
{"title":"Administration Assistant","salaryRange":[3000,6000]}
{"title":"Finance Manager","salaryRange":[8200,16000]}
{"title":"Accountant","salaryRange":[4200,9000]}
{"title":"Accounting Manager","salaryRange":[8200,16000]}
{"title":"Public Accountant","salaryRange":[4200,9000]}
{"title":"Sales Manager","salaryRange":[10000,20080]}
{"title":"Sales Representative","salaryRange":[6000,12008]}
{"title":"Purchasing Manager","salaryRange":[8000,15000]}
{"title":"Purchasing Clerk","salaryRange":[2500,5500]}
{"title":"Stock Manager","salaryRange":[5500,8500]}
{"title":"Stock Clerk","salaryRange":[2008,5000]}
{"title":"Shipping Clerk","salaryRange":[2500,5500]}
{"title":"Programmer","salaryRange":[4000,10000]}
{"title":"Marketing Manager","salaryRange":[9000,15000]}
{"title":"Marketing Representative","salaryRange":[4000,9000]}
{"title":"Human Resources Representative","salaryRange":[4000,9000]}
{"title":"Public Relations Representative","salaryRange":[4500,10500]}
```

Generation of JSON Data (7) - Examples

An Example of USING JSON_OBJECTAGG

```
SELECT json_objectagg(department_name VALUE department_id) FROM departments;

-- The returned object is pretty-printed here for clarity.
-- The order of the object members is arbitrary.

JSON_OBJECTAGG(DEPARTMENT_NAMEISDEPARTMENT_ID)
----------------------------------------------
{"Administration":        10,
 "Marketing":             20,
 "Purchasing":            30,
 "Human Resources":       40,
 "Shipping":              50,
 "IT":                    60,
 "Public Relations":      70,
 "Sales":                 80,
 "Executive":             90,
 "Finance":              100,
 "Accounting":           110,
 "Treasury":             120,
 "Corporate Tax":        130,
 "Control And Credit":   140,
 "Shareholder Services": 150,
 "Benefits":             160,
 "Manufacturing":        170,
 "Construction":         180,
 "Contracting":          190,
 "Operations":           200,
 "IT Support":           210,
 "NOC":                  220,
 "IT Helpdesk":          230,
 "Government Sales":     240,
 "Retail Sales":         250,
 "Recruiting":           260,
 "Payroll":              270}
```

Generation of JSON Data (8) - Examples

An Example of USING JSON_ARRAYAGG

```
SELECT json_object('id'         VALUE mgr.employee_id,
                   'manager'    VALUE (mgr.first_name || ' '|| mgr.last_name),
                   'numReports' VALUE count(rpt.employee_id),
                   'reports'    VALUE json_arrayagg(rpt.employee_id
                                                    ORDER BY rpt.employee_id))
  FROM  employees mgr, employees rpt
  WHERE mgr.employee_id = rpt.manager_id
  GROUP BY mgr.employee_id, mgr.last_name, mgr.first_name
  HAVING count(rpt.employee_id) > 6;

-- The returned object is pretty-printed here for clarity.

JSON_OBJECT('ID'ISMGR.EMPLOYEE_ID,'MANAGER'VALUE(MGR.FIRST_NAME||''||MGR.LAST_NAME)
--------------------------------------------------------------------------------
{"id":         100,
 "manager":    "Steven King",
 "numReports": 14,
 "reports":    [101,102,114,120,121,122,123,124,145,146,147,148,149,201]}

{"id":         120,
 "manager":    "Matthew Weiss",
 "numReports": 8,
 "reports":    [125,126,127,128,180,181,182,183]}

{"id":         121,
 "manager":    "Adam Fripp",
 "numReports": 8,
 "reports":    [129,130,131,132,184,185,186,187]}

{"id":         122,
 "manager":    "Payam Kaufling",
 "numReports": 8,
 "reports":    [133,134,135,136,188,189,190,191]}

{"id":         123,
 "manager":    "Shanta Vollman",
 "numReports": 8,
 "reports":    [137,138,139,140,192,193,194,195]}

{"id":         124,
 "manager":    "Kevin Mourgos",
 "numReports": 8,
 "reports":    [141,142,143,144,196,197,198,199]}
```