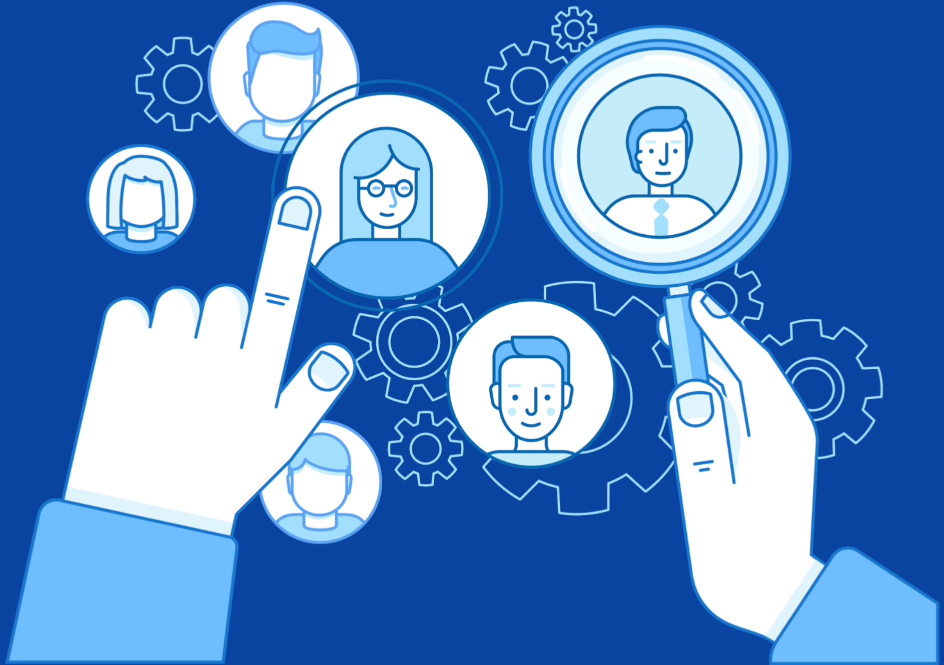
# Digital Career Institute

## Python Course - Advanced SQL



Digital Career Institute

# Goal of the Submodule

The goal of this submodule is to help the student understand how to work with advanced SQL queries and data types. By the end of this submodule, the learners will be able to understand:

- How to use advanced SQL data types.

# Topics

- How to define and use advanced data types.
- How to group and calculate statistics on a set data tables.

# Advanced Data Types
# -
# Enumerated Types

# Enumerated

Types of data that comprise **an ordered set of values**.

- Days of the week
  Monday < Tuesday < Wednesday < Thursday < ...
- Months
  January < February < March < April < May < ...
- Hierarchical position
  Intern < Employed < Coordinator < ...
- Stepped processes
  To define < To start < Doing < Reviewing < ...

# Enumerated

Basic definition of an enumerated type.

```
ENUM(
    “Selling point”,
    “Local office”,
    “Headquarters”
);
```

In PostgreSQL a new type has to be defined first:

```
CREATE TYPE location_type AS ENUM(
    “Selling point”,
    “Local office”,
    “Headquarters”
);
ALTER TABLE Location ADD COLUMN type location_type;
```

In other RDBMS the **`ENUM`** declaration may be used directly in the **`ADD COLUMN`** clause.

# Enumerated

## Validation

```
UPDATE "Location" SET type = 'Something else';

ERROR:  invalid input value for enum location_type:
"Something else"
```

```
UPDATE "Location" SET type = 'SELLING POINT';

ERROR:  invalid input value for enum location_type:
"Something else"
```

Enumerated types are case sensitive.

# Enumerated

## Sorting

```
UPDATE Location SET type = 'Local office';
UPDATE Location SET type = 'Headquarters'
WHERE name = 'Headquarters';
UPDATE Location SET type = 'Selling point'
WHERE name = 'Location 3';

SELECT * FROM Location
ORDER BY type ASC;
```

The enum fields are sorted according to the order each item was given on the field definition.

### Result

| name | city_id | type |
|---|---|---|
| Location 3 | 2 | Selling point |
| Location 2 | 1 | Local office |
| Location 4 | 3 | Local office |
| Location 5 | 4 | Local office |
| Location 6 | 21 | Local office |
| Headquarters | 2 | Headquarters |

# Enumerated

## Relational

```
SELECT * FROM Location
WHERE type > 'Selling point';
```

### Result

| name | city_id | type |
|---|---|---|
| Headquarters | 2 | Headquarters |
| Location 2 | 1 | Local office |
| Location 4 | 3 | Local office |
| Location 5 | 4 | Local office |
| Location 6 | 21 | Local office |

It uses the order of the value in the field definition to evaluate the relational expression.

# Advanced Data Types

# -

# UUID

Digital Career Institute

# UUID

Integer identifiers are good enough for most cases, but sometimes we want our IDs to be unique across different datasets or applications.

## Universally Unique Identifiers

```
a0eebc99-9c0b-4ef8-bb6d-6bb9bd380a11
```

# UUID

```
ALTER TABLE Location ADD COLUMN my_uuid_field uuid;
```

Validation + Relational

PostgreSQL does not provide functions to generate UUIDs.

The module uuid-ossp can be installed for that purpose.

# Advanced Data Types
# -
# JSON Types

# JSON Fields

JavaScript Object Notation

```
{“key1”: “Value 1”, “key2”: 136}
```

# The JSON Type

## Validation

```
ALTER TABLE Location ADD COLUMN info json;

UPDATE Location SET info = 23
WHERE type = 'Headquarters';
ERROR:  invalid input syntax for type json

UPDATE Location SET info = '{"forgot": "a_curly"'
WHERE type = 'Headquarters';
ERROR:  invalid input syntax for type json

UPDATE Location SET info = '{"all": "ok", "none": 2}'
WHERE type = 'Headquarters';
Query returned successfully
```

# The JSON Type

## Basic JSON specific operators

{field} **->** {key}

Returns a JSON data type.

{field} **->>** {key}

Returns a text data type.

# The JSON Type

## Querying

```
SELECT
    info->'all', pg_typeof(info->'all'),
    info->>'all', pg_typeof(info->>'all')
FROM Location
WHERE info->>'all' = 'ok'
```

## Result

| ?column? | pg_typeof | ?column? | pg_typeof |
|---|---|---|---|
| “ok” | json | ok | text |

# The JSON Type

## Querying: nested JSON paths

{field} **#>** {key}

{field} **#>>** {key}

```
UPDATE Location SET info = '{"all": {"ok": true}}'
WHERE type = 'Headquarters';

SELECT
    info#>'{all,ok}', pg_typeof(info#>'{all,ok}'),
    info#>>'{all,ok}', pg_typeof(info#>>'{all,ok}')
FROM Location WHERE info#>>'{all,ok}' = 'true';
```

### Result

| ?column? | pg_typeof | ?column? | pg_typeof |
|---|---|---|---|
| true | json | true | text |

# The JSONB Type

The type `jsonb` is very similar to the type `json` but it is stored in **binary form**.

Advantages over the `json` type:

- more efficient,
- significantly faster to process,
- supports indexing.

# The JSONB Type

## Definition

```
ALTER TABLE Location ADD COLUMN infob jsonb;

UPDATE Location SET infob = '{"all": "ok", "none": 2}'
WHERE type = 'Headquarters';
Query returned successfully
```

# The JSONB Type

## Additional operators: inclusion

{json} **@>** {json}

Returns `True` if the left JSON includes the right JSON.

{json} **<@** {json}

Returns `True` if the right JSON includes the left JSON.

# The JSONB Type

Digital Career Institute

## Additional operators: inclusion

```
SELECT name, infob, infob@>'{"all": "ok"}' FROM Location;
```

| name | infob | ?column? |
|---|---|---|
| Location 2 | | |
| Location 4 | | |
| Location 5 | | |
| Location 6 | | |
| Location 3 | | |
| Headquarters | {“all”: “ok”, “none”: 2} | t |

# The JSONB Type

## Additional operators: key exists

```
SELECT name, infob, infob?'all', infob?'something'
FROM Location;
```

{jsonb} ? {key}

Returns **True** if the left JSON has a key with the name **key**.

| name | infob | ?column? | ?column? |
|---|---|---|---|
| Location 2 | | | |
| Location 4 | | | |
| Location 5 | | | |
| Location 6 | | | |
| Location 3 | | | |
| Headquarters | {“all”: “ok”, “none”: 2} | t | f |

# The JSONB Type

## Additional operators: concatenate

```
SELECT name, infob, infob || jsonb ‘{“all”: “ko”}’
FROM Location;
```

{jsonb} **||** {jsonb}

If keys are duplicated the one on the right prevails.

| name | infob | ?column? |
|---|---|---|
| Location 2 | | |
| Location 4 | | |
| Location 5 | | |
| Location 6 | | |
| Location 3 | | |
| Headquarters | {“all”: “ok”, “none”: 2} | {“all”: “**ko**”, “none”: 2} |

# Advanced Data Types
-
# Array Types

# Array Types

A field can be defined as an array of any other data type.

```
[145, 543, 234]
```

# Arrays

Appending `[]` to a data type will define an array of elements of that type.

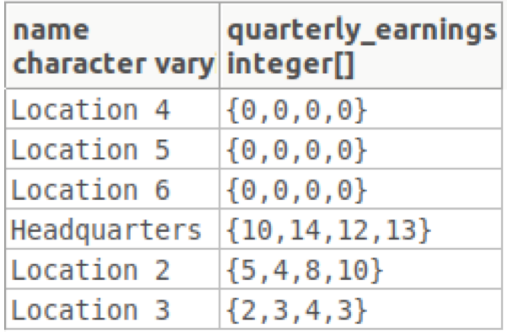
```
ALTER TABLE Location
ADD COLUMN quarterly_earnings integer[];
```

```
ALTER TABLE City
ADD COLUMN alternate_name varchar[];
```

```
ALTER TABLE Country
ADD COLUMN boundaries jsonb[];
```

# Arrays

```
UPDATE Location
SET quarterly_earnings = ARRAY[0, 0, 0, 0];

UPDATE Location
SET quarterly_earnings = ARRAY[10, 14, 12, 13]
WHERE name = 'Headquarters';

UPDATE Location
SET quarterly_earnings = ‘{5, 4, 8, 10}’
WHERE name = 'Location 2';

UPDATE Location
SET quarterly_earnings = ARRAY[2, 3, 4, 1]
WHERE name = 'Location 3';

UPDATE Location
SET quarterly_earnings[4] = 3
WHERE name = 'Location 3';
```



| name<br>character vary | quarterly_earnings<br>integer[] |
|---|---|
| Location 4 | {0,0,0,0} |
| Location 5 | {0,0,0,0} |
| Location 6 | {0,0,0,0} |
| Headquarters | {10,14,12,13} |
| Location 2 | {5,4,8,10} |
| Location 3 | {2,3,4,3} |

Array indexes start at **1**.

# Arrays

## Accessing elements in the array

```
SELECT
    name,
    type,
    quarterly_earnings[1] AS Q1,
    quarterly_earnings[2] AS Q2
FROM Location
WHERE quarterly_earnings[2] < quarterly_earnings[1];
```

| name | type | Q1 | Q2 |
|---|---|---|---|
| Location 2 | Local office | 5 | 4 |

# Multidimensional Arrays

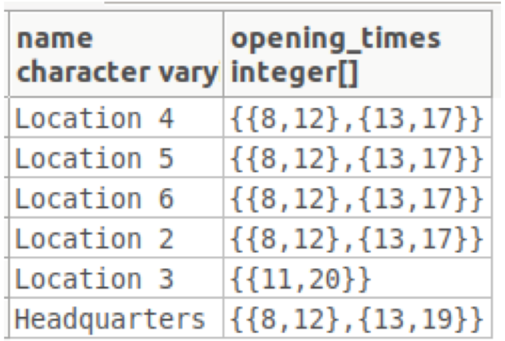
Append as many `[]` as dimensions in the array.

```
ALTER TABLE Location
ADD COLUMN opening_times integer[][];
```

```
[[8, 12],[13, 17]]
```

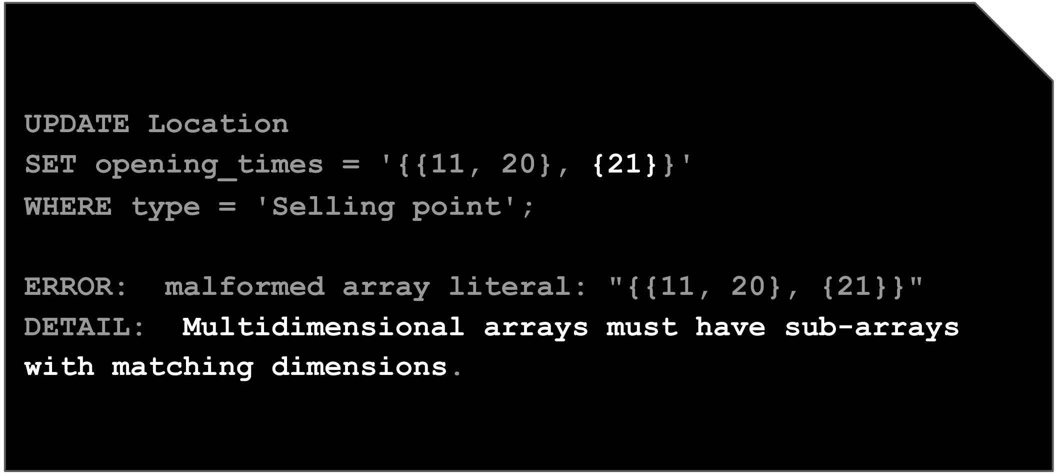
# Multidimensional Arrays

```
UPDATE Location
SET opening_times = ARRAY[[8, 12], [13, 17]];

UPDATE Location
SET opening_times = '{{11, 20}}'
WHERE type = 'Selling point';

UPDATE Location
SET opening_times[2][2] = 19
WHERE name = 'Headquarters';
```



| name<br>character vary | opening_times<br>integer[] |
|---|---|
| Location 4 | {{8,12},{13,17}} |
| Location 5 | {{8,12},{13,17}} |
| Location 6 | {{8,12},{13,17}} |
| Location 2 | {{8,12},{13,17}} |
| Location 3 | {{11,20}} |
| Headquarters | {{8,12},{13,19}} |

# Multidimensional Arrays



```
UPDATE Location
SET opening_times = '{{11, 20}, {21}}'
WHERE type = 'Selling point';

ERROR:  malformed array literal: "{{11, 20}, {21}}"
DETAIL:  Multidimensional arrays must have sub-arrays
with matching dimensions.
```

All arrays of the main array must have the same length.

# Array Operators

Digital Career Institute

**INCLUDES**

{array} **@>** {array}
{array} **<@** {array}

**CONCATENATE**

{array} **||** {array}

**OVERLAPS**

{array} **&&** {array}

# Array Functions

### APPEND ()

```
array = array_append(
    array,
    element
)
```

### REMOVE ()

```
array = array_remove(
    array,
    element
)
```

### LENGTH ()

```
array = array_length(
    array,
    integer
)
```

The `integer` indicates the depth of the array hierarchy, starting at 1 as the first dimension of the array.

# Advanced Data Types

# -

# Binary Types

# Binary Types

A binary field allows the storage of files.

`image.png`

# Managing Files

There are 2 general ways of storing files as attributes of a record in a table.

REFERENCE

The file is stored on the file system and there is a field in the database storing the path as `text`.

VALUE

The file is stored directly on the table field as a `binary` object.

# Storing Files as Text References

```
ALTER TABLE Country
ADD COLUMN flag varchar;

UPDATE Country
SET flag = 'usa_flag.png'
WHERE name = 'USA';
```

## PROS

- Faster
- Smaller database size
- Easier maintenance

## CONS

- More complex architecture
- Reduced consistency

# Storing Files as Binary Values

```
ALTER TABLE Country
ADD COLUMN flag bytea;

UPDATE Country
SET flag = '\xDEADBEEF...'
WHERE name = 'USA';
```

## PROS

- Consistency guaranteed
- Simple use of SQL

## CONS

- Slower
- Larger database size
- Higher maintenance

# Text References vs. Binary Values

Digital Career Institute

## Architecture

### Text Reference

The SQL can only provide the reference and the **the application must know** how to use it.

### Binary Value

The SQL can provide the actual file and **the application does not need to know** how the data is stored.

# Advanced Data Types
-
# Time Types

Digital Career Institute

# Temporal Data

SQL has specific types to manage time data.

```
0001-01-01 00:00:00
```

# Temporal Data

There are 4 basic types to work with temporal data.

DATE

TIME

INTERVAL

TIMESTAMP

# Time Fields

## Defining Time Fields

```
ALTER TABLE Location
ADD COLUMN opened_on date;
ALTER TABLE Location
ALTER COLUMN opened_on TYPE time;
ALTER TABLE Location
ALTER COLUMN opened_on TYPE timestamp;

ALTER TABLE Location
ALTER COLUMN opened_on TYPE time with time zone;
ALTER TABLE Location
ALTER COLUMN opened_on TYPE timestamp with time zone;
```

Time and timestamp can also be made aware of the time zone.

# Time Fields

## Using Time Fields

```
ALTER TABLE Location
ADD COLUMN opened_on date;
UPDATE Location SET opened_on = ‘1999-01-23’;

ALTER TABLE Location
ALTER COLUMN opened_on TYPE time;
UPDATE Location SET opened_on = ‘14:21:02’;

ALTER TABLE Location
ALTER COLUMN opened_on TYPE timestamp;
UPDATE Location SET opened_on = ‘2004-10-19 10:23:54’;
```

# Time Fields

## Using Time Fields

```
ALTER TABLE Location
ALTER COLUMN opened_on TYPE time with time zone;
UPDATE Location SET opened_on = ‘14:21:02 PST’;

ALTER TABLE Location
ALTER COLUMN opened_on TYPE timestamp with time zone;
UPDATE Location
SET opened_on = ‘2004-10-19 10:23:54+02’;

ALTER TABLE Location
ALTER COLUMN opened_on TYPE time[];
UPDATE Location
SET opened_on = [time ‘14:21:02’, time ‘15:34:21’];
```

# Time Fields

## Using Time Fields

```
SELECT * FROM Location
WHERE opened_on >= ‘2000-01-01’;

SELECT * FROM Location
WHERE opened_on BETWEEN ‘2000-01-01’ AND ‘2003-01-01’;

SELECT CURRENT_TIMESTAMP - opened_on AS “Days open”
FROM Location;
```

Subtracting a `timestamp` from a `timestamp` produces a value of type `interval`.

# Interval Fields

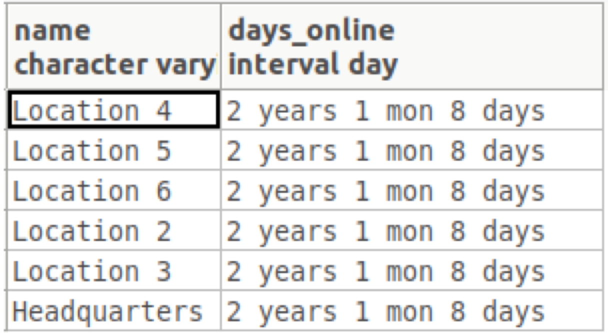
Digital Career Institute

## Defining Interval Fields

We can specify the resolution we desire.

```
ALTER TABLE Location
ADD COLUMN days_online interval day;

UPDATE Location SET days_online = 'P2Y1M1W1DT1H1M1S';
```

- `P` indicates the formatting used.
- `2Y1M1W1D` adds 2 years, 1 month, 1 week and 1 day.
- `T` indicates the following is referring to time
- And `1H1M1S` adds 1 hour, 1 minute and 1 second.



| name<br>character vary | days_online<br>interval day |
|---|---|
| Location 4 | 2 years 1 mon 8 days |
| Location 5 | 2 years 1 mon 8 days |
| Location 6 | 2 years 1 mon 8 days |
| Location 2 | 2 years 1 mon 8 days |
| Location 3 | 2 years 1 mon 8 days |
| Headquarters | 2 years 1 mon 8 days |

# We learned ...

- That enumerate types are data types that are ordered lists in nature and that we can use this to sort the records.
- That there is a data type called UUID to store and validate universal identifiers.
- That we can store JSON objects in a field and we can query them using specific SQL operators.
- That any type can be used as an array of any dimension.
- That we can store binary files directly into a field.
- That there are specific data types to manage time-related data.

# Documentation

# Documentation

- JOIN
https://www.w3schools.com/sql/sql_join.asp
https://www.postgresql.org/docs/current/tutorial-join.html
https://www.postgresqltutorial.com/postgresql-joins/
- Data types
https://www.postgresql.org/docs/current/datatype-enum.html
https://www.postgresql.org/docs/current/datatype-uuid.html
https://www.postgresql.org/docs/current/datatype-json.html
https://www.postgresql.org/docs/current/functions-json.html
https://www.postgresql.org/docs/current/arrays.html
https://www.postgresql.org/docs/current/datatype-binary.html
https://www.postgresql.org/docs/current/datatype-datetime.html
https://www.w3schools.com/sql/sql_dates.asp

# Documentation

- Subqueries
https://www.w3resource.com/sql/subqueries/understanding-sql-subqueries.php
https://cloud.google.com/bigquery/docs/reference/standard-sql/subqueries
https://www.postgresql.org/docs/current/functions-subquery.html
- Group & aggregate
https://www.w3schools.com/sql/sql_groupby.asp
https://www.w3schools.com/sql/sql_having.asp
https://cloud.google.com/bigquery/docs/reference/standard-sql/aggregate_functions
https://www.postgresql.org/docs/current/functions-aggregate.html

# THANK YOU

**Contact Details**
**DCI Digital Career Institute gGmbH**