SQL with Python:

Week 4 Workshop Presentation

Workshop Agenda

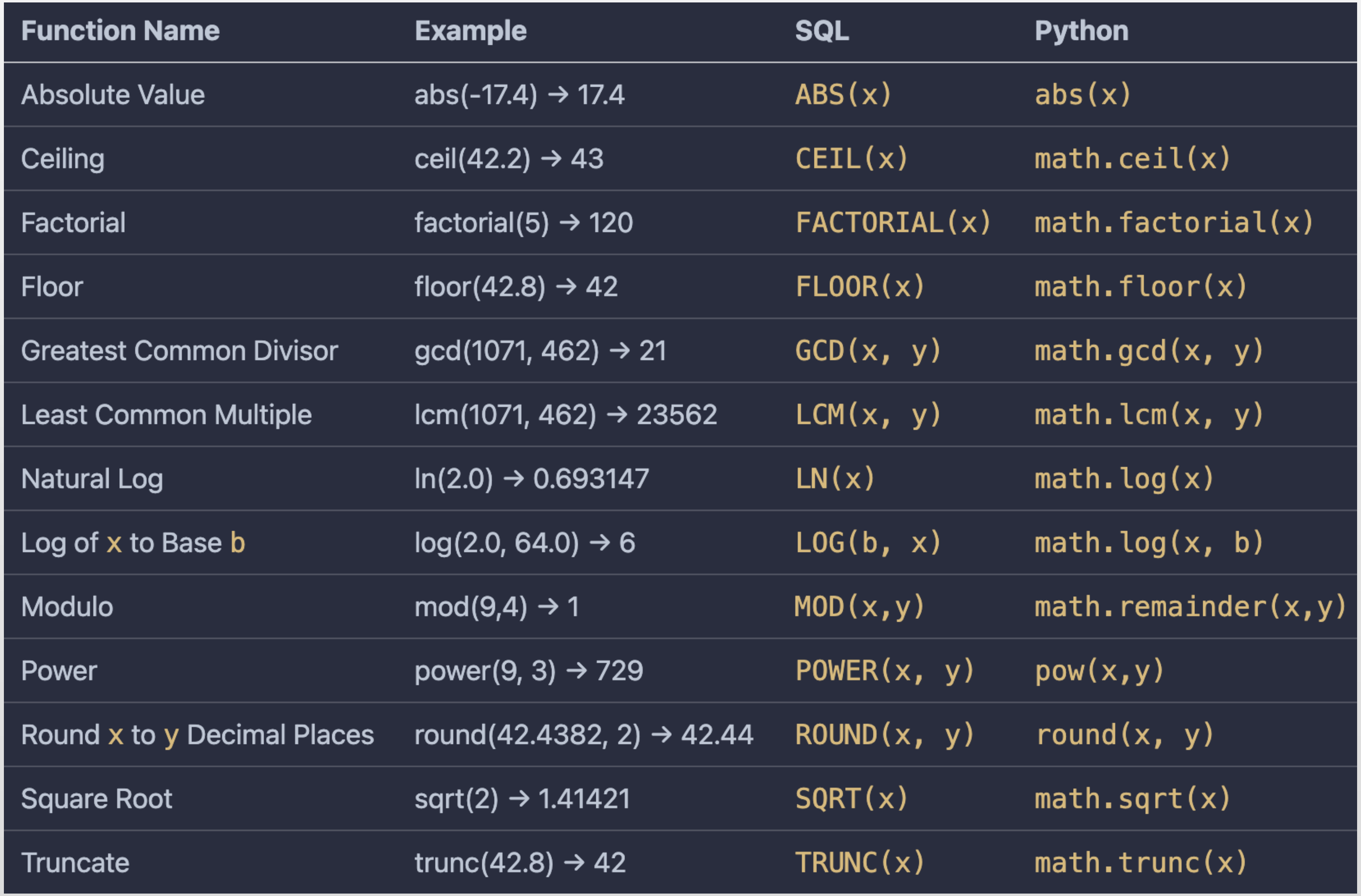
Activity	Estimated Duration
Set up and check in	10 mins
Week 4 Review	60 mins
Assignment Tasks	40 mins
Break	15 mins
Remaining Assignment Tasks	100 mins
Check-Out (Feedback & Wrap-Up)	15 mins

Week 4 Review

Overview

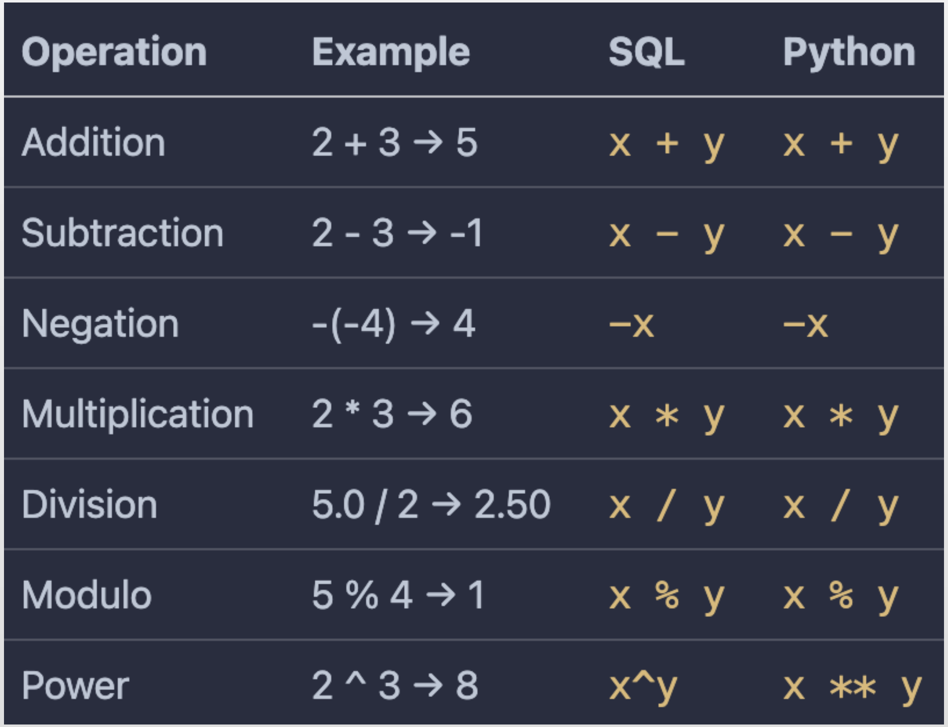
Math Functions & Operators	Query Planning
JSON in SQL	Backups
Triggers	Provisioning
Performance Tips	NumPy
N+1 Query	MatPlotLib
Indexes	Pandas

Review: Math Functions & Operators

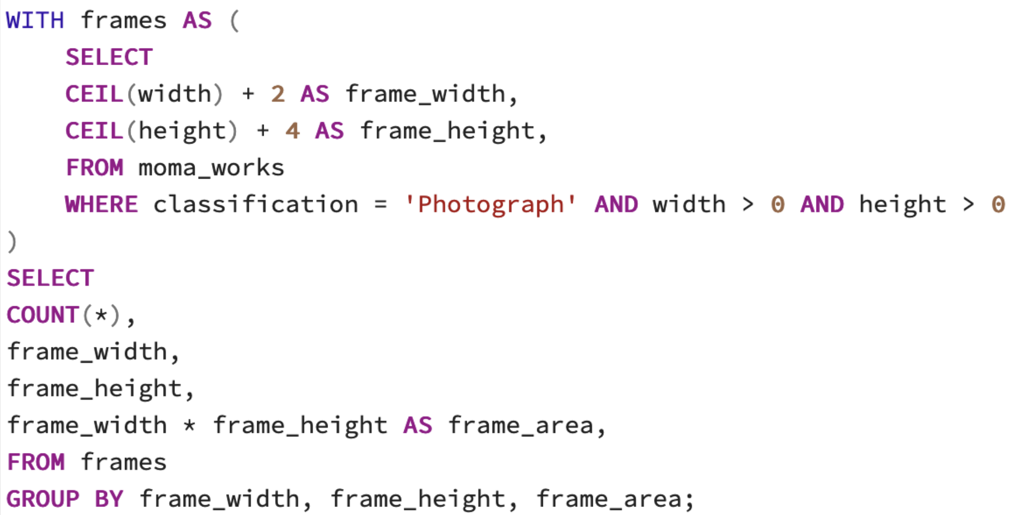


Function Name	Example	SQL	Python
Absolute Value	abs(-17.4) → 17.4	`ABS(x)`	`abs(x)`
Ceiling	ceil(42.2) → 43	`CEIL(x)`	`math.ceil(x)`
Factorial	factorial(5) → 120	`FACTORIAL(x)`	`math.factorial(x)`
Floor	floor(42.8) → 42	`FLOOR(x)`	`math.floor(x)`
Greatest Common Divisor	gcd(1071, 462) → 21	`GCD(x, y)`	`math.gcd(x, y)`
Least Common Multiple	lcm(1071, 462) → 23562	`LCM(x, y)`	`math.lcm(x, y)`
Natural Log	ln(2.0) → 0.693147	`LN(x)`	`math.log(x)`
Log of x to Base b	log(2.0, 64.0) → 6	`LOG(b, x)`	`math.log(x, b)`
Modulo	mod(9,4) → 1	`MOD(x,y)`	`math.remainder(x,y)`
Power	power(9, 3) → 729	`POWER(x, y)`	`pow(x,y)`
Round x to y Decimal Places	round(42.4382, 2) → 42.44	`ROUND(x, y)`	`round(x, y)`
Square Root	sqrt(2) → 1.41421	`SQRT(x)`	`math.sqrt(x)`
Truncate	trunc(42.8) → 42	`TRUNC(x)`	`math.trunc(x)`

Review: Math Functions & Operators



Operation	Example	SQL	Python
Addition	2 + 3 → 5	x + y	x + y
Subtraction	2 - 3 → -1	x − y	x − y
Negation	-(-4) → 4	−x	−x
Multiplication	2 * 3 → 6	x * y	x * y
Division	5.0 / 2 → 2.50	x / y	x / y
Modulo	5 % 4 → 1	x % y	x % y
Power	2 ^ 3 → 8	x^y	x ** y



```sql
WITH frames AS (
    SELECT
    CEIL(width) + 2 AS frame_width,
    CEIL(height) + 4 AS frame_height,
    FROM moma_works
    WHERE classification = 'Photograph' AND width > 0 AND height > 0
)
SELECT
COUNT(*),
frame_width,
frame_height,
frame_width * frame_height AS frame_area,
FROM frames
GROUP BY frame_width, frame_height, frame_area;
```

Review: JSON in SQL

3 ways to store JSON data in Postgres:

TEXT data type – as text in JSON format
JSON data type – also text, but enforces JSON format
JSONB data type – JSON format encoded in binary

Q: Which is generally preferred?

Review: JSON in SQL

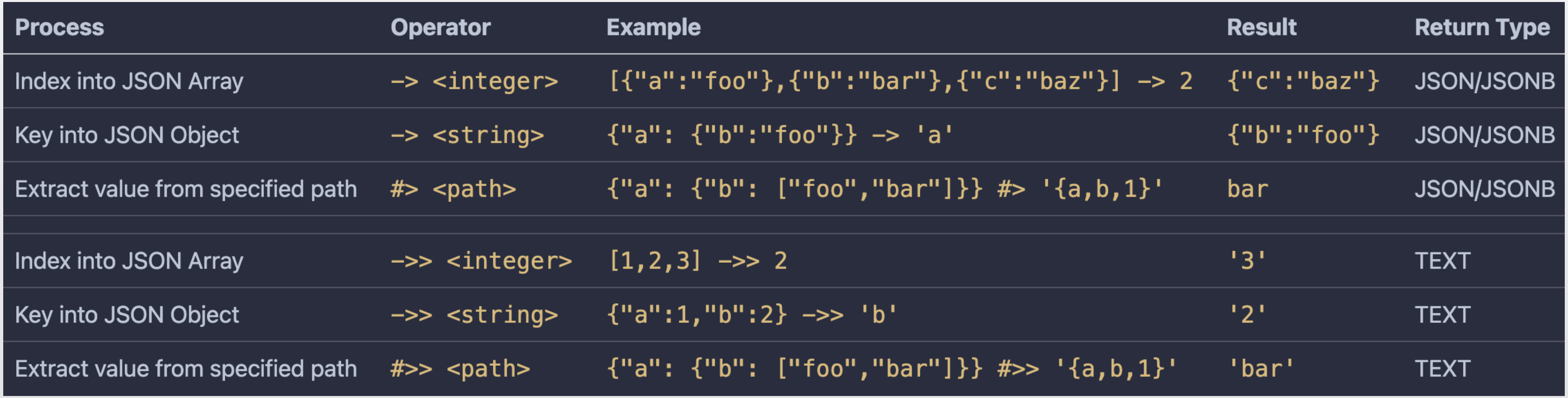
Q: Which is generally preferred?

A: JSONB – most efficient, significantly less time to process

Type	Efficient Storage & Processing	Easily Portable	Validates JSON Rules	JSON Functions Available
TEXT		Yes		
JSON		Yes	Yes	Yes
JSONB	Yes		Yes	Yes

Note: Other relational database systems have different ways of handling JSON. Example: MySQL has a JSON data type but not JSONB, but its JSON data type is in binary format

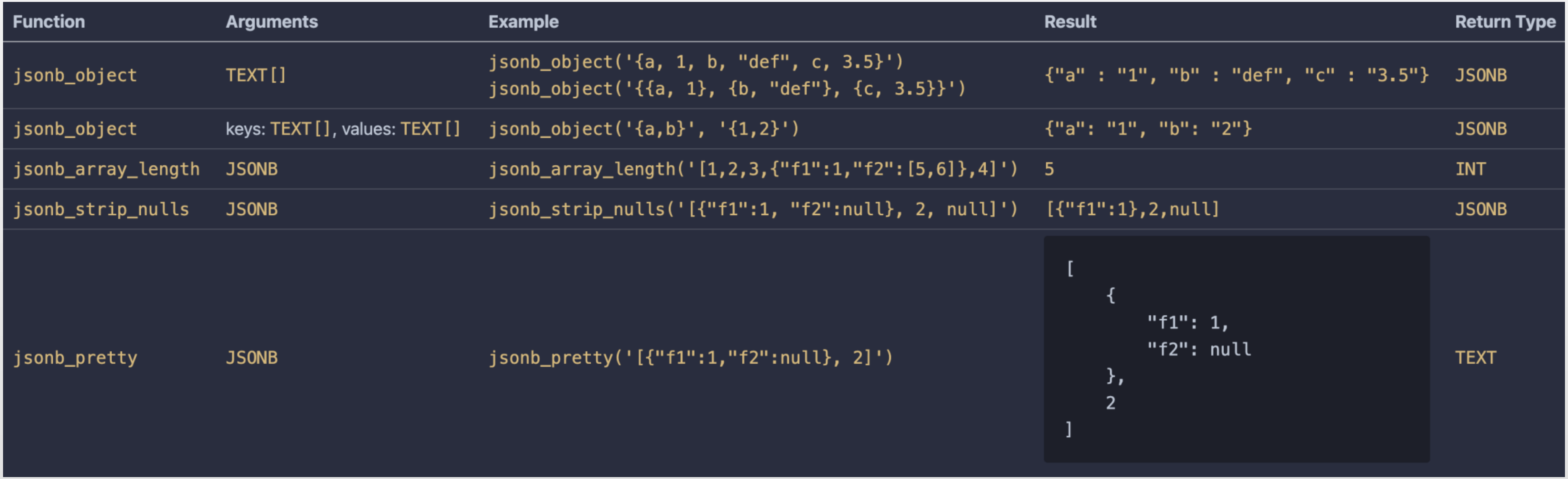
Review: JSON operators



Process	Operator	Example	Result	Return Type
Index into JSON Array	`-> <integer>`	`[{"a":"foo"},{"b":"bar"},{"c":"baz"}] -> 2`	`{"c":"baz"}`	JSON/JSONB
Key into JSON Object	`-> <string>`	`{"a": {"b":"foo"}} -> 'a'`	`{"b":"foo"}`	JSON/JSONB
Extract value from specified path	`#> <path>`	`{"a": {"b": ["foo","bar"]}} #> '{a,b,1}'`	`bar`	JSON/JSONB
Index into JSON Array	`->> <integer>`	`[1,2,3] ->> 2`	`'3'`	TEXT
Key into JSON Object	`->> <string>`	`{"a":1,"b":2} ->> 'b'`	`'2'`	TEXT
Extract value from specified path	`#>> <path>`	`{"a": {"b": ["foo","bar"]}} #>> '{a,b,1}'`	`'bar'`	TEXT

-> operator followed by integer will index into JSON array (like Python list)
-> operator followed by string will key into JSON object (like Python dictionary)
#> operator followed by path will return value from path (JSON array or object)
Use **>>** versions of operators to return data in TEXT format instead of JSON/JSONB

Review: JSON functions



Function	Arguments	Example	Result	Return Type

jsonb_object	TEXT[]	jsonb_object('{a, 1, b, "def", c, 3.5}')
		jsonb_object('{{a, 1}, {b, "def"}, {c, 3.5}}')
			{"a" : "1", "b" : "def", "c" : "3.5"}	JSONB

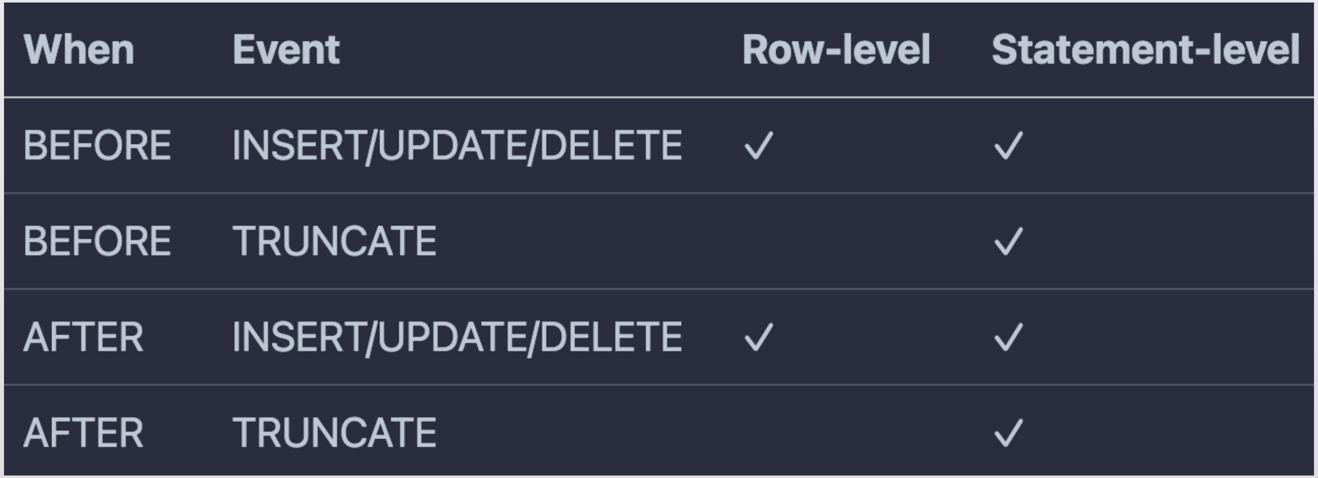
jsonb_object	keys: TEXT[], values: TEXT[]	jsonb_object('{a,b}', '{1,2}')	{"a": "1", "b": "2"}	JSONB
jsonb_array_length	JSONB	jsonb_array_length('[1,2,3,{"f1":1,"f2":[5,6]},4]')	5	INT
jsonb_strip_nulls	JSONB	jsonb_strip_nulls('[{"f1":1, "f2":null}, 2, null]')	[{"f1":1},2,null]	JSONB

jsonb_pretty	JSONB	jsonb_pretty('[{"f1":1,"f2":null}, 2]')	[
			    {
			        "f1": 1,
			        "f2": null
			    },
			    2
			]
				TEXT

jsonb_object(): 3 ways to use it, functionally equivalent
jsonb_array_length(): Retrieve length of array, returns INT
jsonb_strip_nulls(): Recursively removes null entries from JSONB object
jsonb_pretty(): Converts to a prettier format for easier reading, returns TEXT

Review: Triggers

- Functions that execute automatically in response to certain events
- Can be added before/after INSERT, UPDATE, DELETE, TRUNCATE queries
- Only statement-level for TRUNCATE, can be row-level or statement-level for the rest
- For UPDATE triggers, may specify list of columns



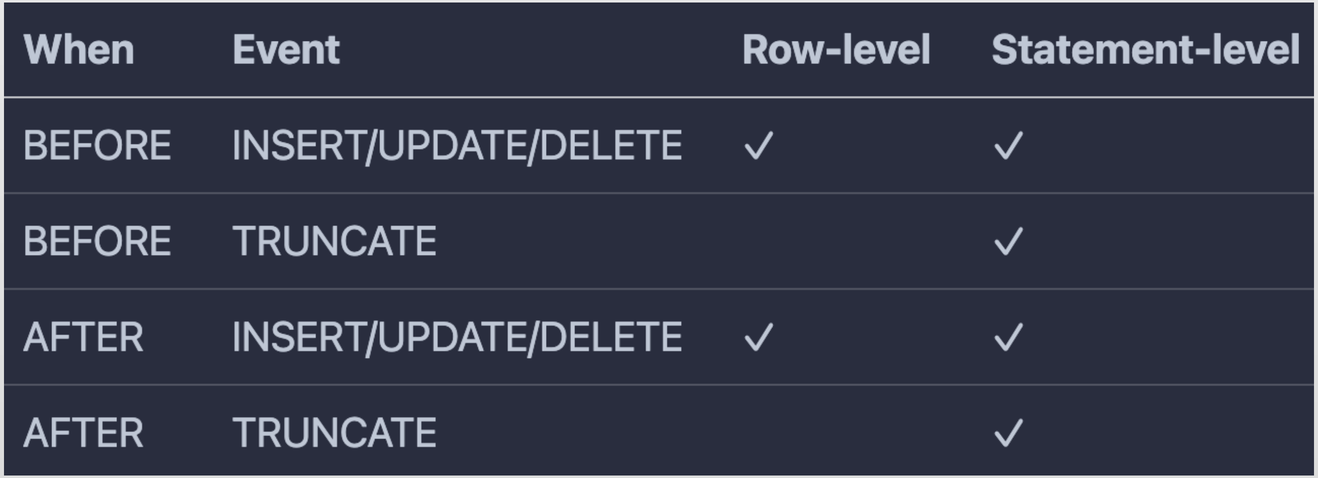
When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	✓	✓
BEFORE	TRUNCATE		✓
AFTER	INSERT/UPDATE/DELETE	✓	✓
AFTER	TRUNCATE		✓

- Q: What is the difference between statement-level and row-level triggers?

Review: Triggers

Q: What is the difference between statement-level and row-level triggers?

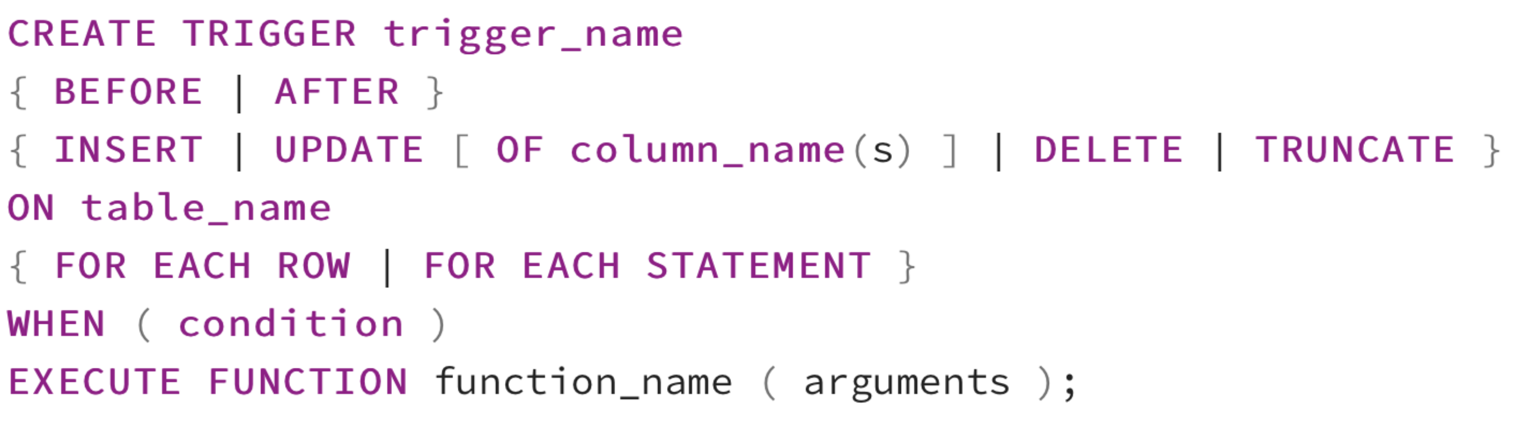
A:
A row-level trigger runs the trigger function once for each row that is modified by the query that triggered it.
A statement-level trigger runs the trigger function once for the entire query that triggered it.



When	Event	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	✓	✓
BEFORE	TRUNCATE		✓
AFTER	INSERT/UPDATE/DELETE	✓	✓
AFTER	TRUNCATE		✓

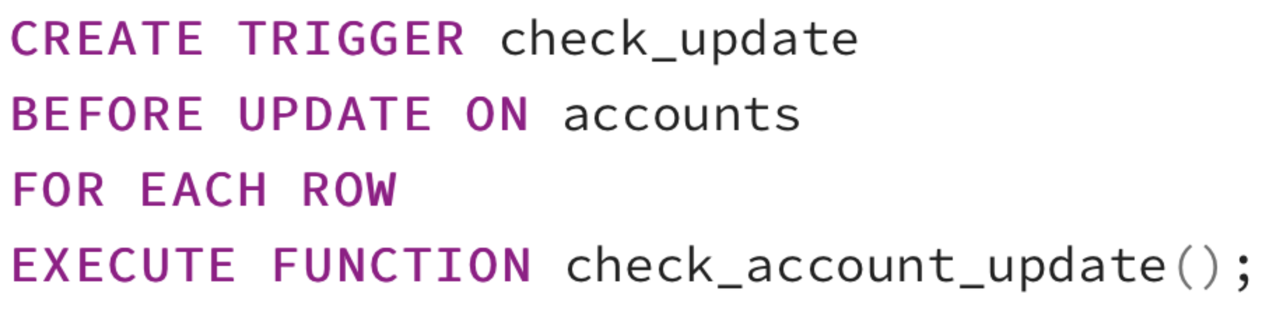
Review: CREATE TRIGGER

Syntax:



```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER }
{ INSERT | UPDATE [ OF column_name(s) ] | DELETE | TRUNCATE }
ON table_name
{ FOR EACH ROW | FOR EACH STATEMENT }
WHEN ( condition )
EXECUTE FUNCTION function_name ( arguments );
```

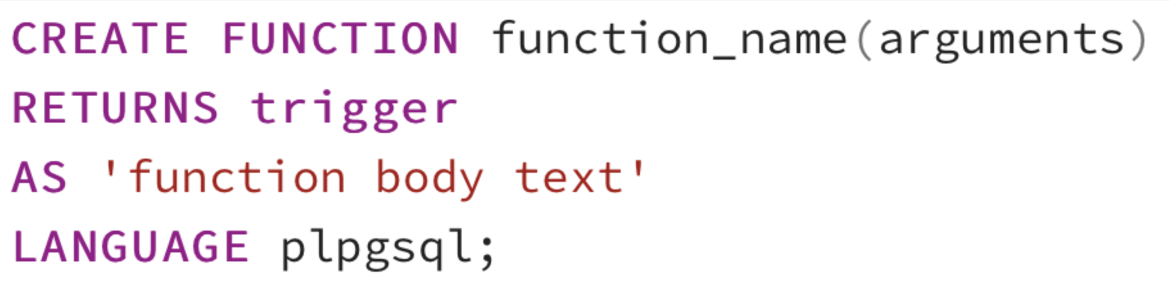
Example:



```
CREATE TRIGGER check_update
BEFORE UPDATE ON accounts
FOR EACH ROW
EXECUTE FUNCTION check_account_update();
```

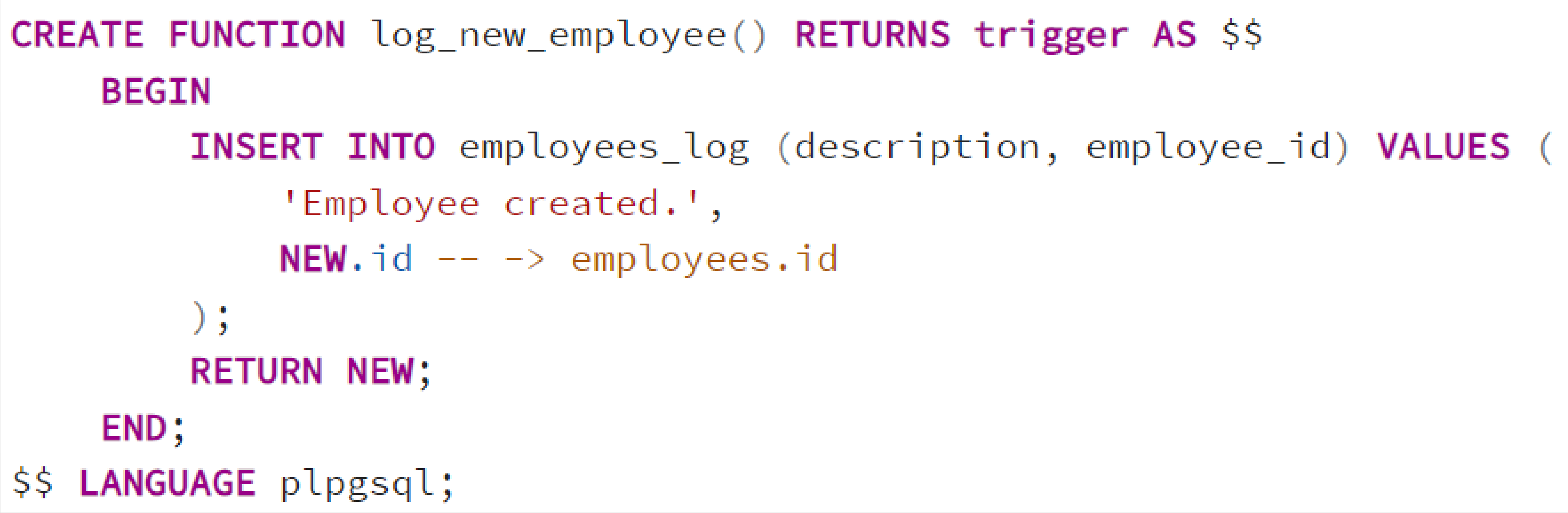
Review: CREATE FUNCTION

Syntax:



```
CREATE FUNCTION function_name(arguments)
RETURNS trigger
AS 'function body text'
LANGUAGE plpgsql;
```

Example:



```
CREATE FUNCTION log_new_employee() RETURNS trigger AS $$
    BEGIN
        INSERT INTO employees_log (description, employee_id) VALUES (
            'Employee created.',
            NEW.id -- -> employees.id
        );
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;
```

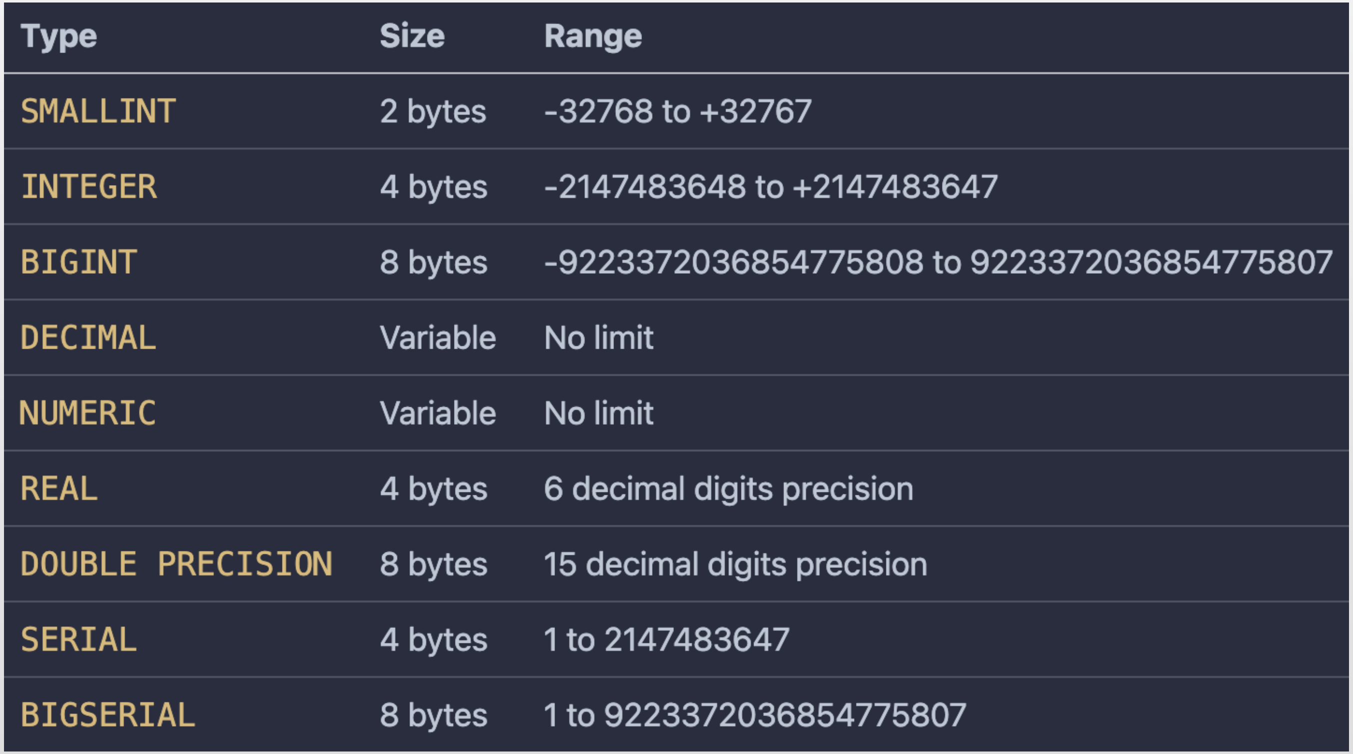
Review: Performance Tips

Ways to improve database performance:

Choose your data types economically
Design your database wisely
Write elegant, optimized queries
Reduce round trips
Avoid the N+1 query antipattern
...can you name others?

Review: Performance Tips

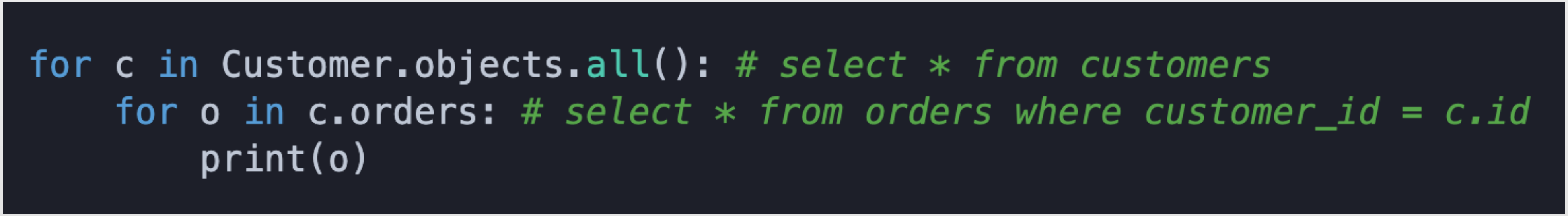
Data Types: Don't waste space using a larger data type than you need!



Type	Size	Range
SMALLINT	2 bytes	-32768 to +32767
INTEGER	4 bytes	-2147483648 to +2147483647
BIGINT	8 bytes	-9223372036854775808 to 9223372036854775807
DECIMAL	Variable	No limit
NUMERIC	Variable	No limit
REAL	4 bytes	6 decimal digits precision
DOUBLE PRECISION	8 bytes	15 decimal digits precision
SERIAL	4 bytes	1 to 2147483647
BIGSERIAL	8 bytes	1 to 9223372036854775807

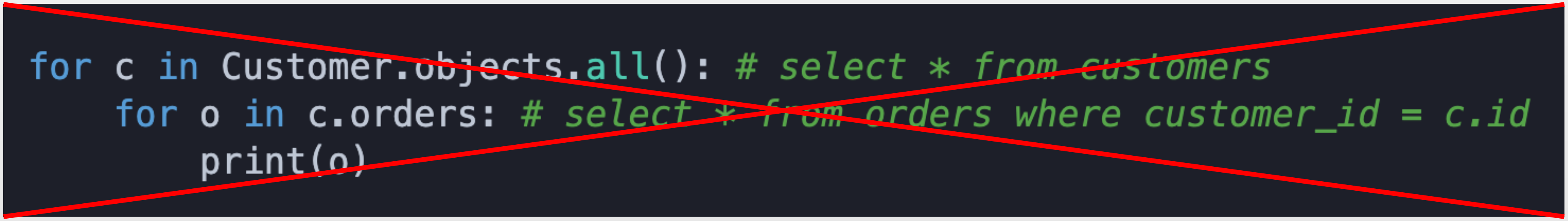
Review: N+1 Query

- The N+1 Query is an antipattern – a pattern you want to avoid
- Commonly caused by use of ORM that obfuscates actual query
- Example:

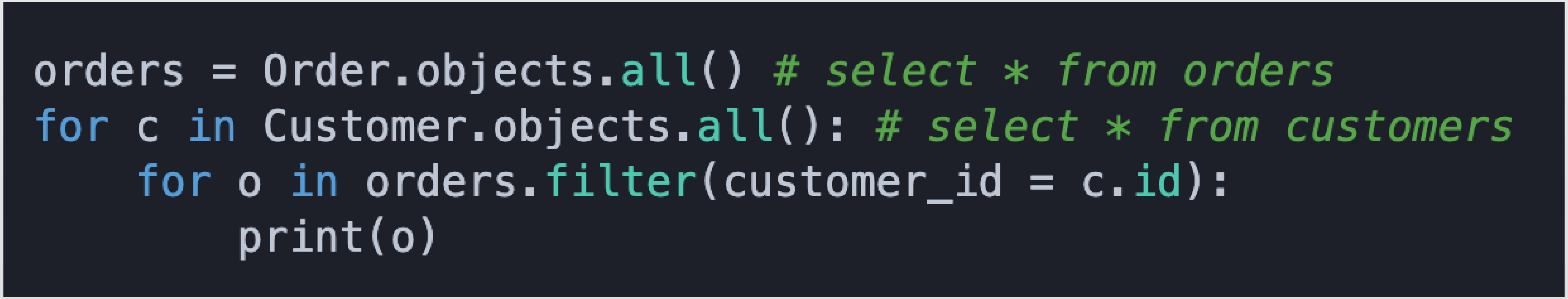


```
for c in Customer.objects.all(): # select * from customers
    for o in c.orders: # select * from orders where customer_id = c.id
        print(o)
```

Review: N+1 Query



```
for c in Customer.objects.all(): # select * from customers
    for o in c.orders: # select * from orders where customer_id = c.id
        print(o)
```



```
orders = Order.objects.all() # select * from orders
for c in Customer.objects.all(): # select * from customers
    for o in orders.filter(customer_id = c.id):
        print(o)
```

Review: Indexes

- Pre-built indexes on a database speed up searches on indexed columns
- Indexes use optimized data structures, such as B-Tree and Hash Tables
- Postgres uses B-Tree indexes by default
- Hash indexes are faster and should be preferred when possible

Q: What makes it possible to use a hash table index?

Review: Indexes

Q: What makes it possible to use a hash table index?

A: Searches involving matches based on equality only

Any >, >=, <, <= comparisons cannot be used by hash index

Example:

```
SELECT artist FROM moma_works WHERE artist = 'Frank Lloyd Wright';
```

B-tree indexes can be used for searches that match on =, >, >=, <, <=

Q: Can a b-tree index be used for searches like this one?:

```
SELECT name FROM users WHERE birthyear BETWEEN '1948' AND '1979';
```

Review: Indexes

B-tree indexes can be used for searches that match on =, >, >=, <, <=

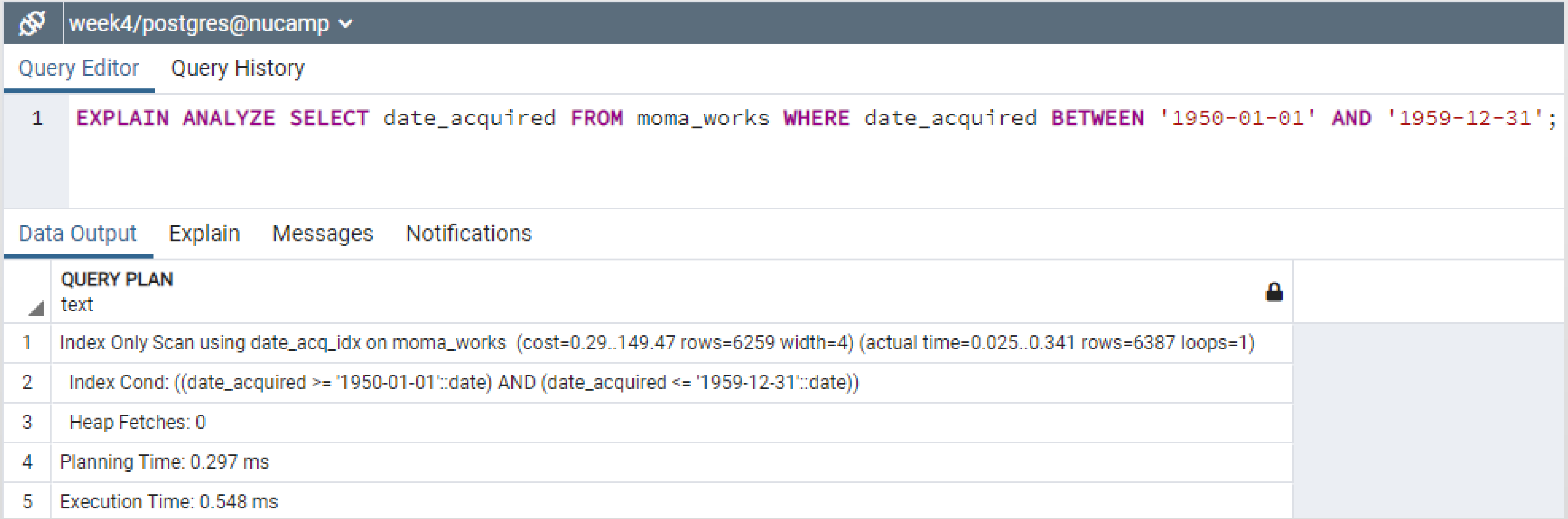
Q: Can a b-tree index be used for searches like this one?:

```
SELECT name FROM users WHERE birthyear BETWEEN '1948' AND '1979';
```

A: Yes, because under the surface, BETWEEN is a combination of >= and <=

Review: Query Planning

- The Postgres Query Planning tools are developer tools that help with understanding how Postgres queries are executed.
- Any query can be prepended with **EXPLAIN** to show the steps Postgres takes to run that query.
- Prepending with **EXPLAIN ANALYZE** shows actual runtimes and other statistics.
- Example:



```sql
EXPLAIN ANALYZE SELECT date_acquired FROM moma_works WHERE date_acquired BETWEEN '1950-01-01' AND '1959-12-31';
```

	QUERY PLAN text
1	Index Only Scan using date_acq_idx on moma_works (cost=0.29..149.47 rows=6259 width=4) (actual time=0.025..0.341 rows=6387 loops=1)
2	Index Cond: ((date_acquired >= '1950-01-01'::date) AND (date_acquired <= '1959-12-31'::date))
3	Heap Fetches: 0
4	Planning Time: 0.297 ms
5	Execution Time: 0.548 ms

Review: Backups

Database administrators must have a backup plan:
How often will databases be backed up, and where?
How long will backups be stored before deletion?
Plans will vary depending on use case.

Backup options:

- pg_dump
- pgAdmin backup tool
- Heroku and other cloud platforms have their own backup tools

Review: Provisioning

Popular cloud platforms with Postgres hosting options include:
Heroku, AWS, Google Cloud, Microsoft Azure, EDB (EnterpriseDB)

More listed at:
https://www.postgresql.org/support/professional_hosting/northamerica/
(see Intro to Provisioning: Additional Resources)

Review: NumPy

Numeric Python

- Library for scientific computing with Python
- Uses custom array data structure called **ndarray**
  - **nd** stands for n-dimensional, meaning multi-dimensional arrays (1 or more dimension)
  - ndarray is a high-performance data structure optimized for advanced math
- Comprehensive math functions, random number generators, linear algebra routines, Fourier transforms, and more
- Used by *many* other Python libraries

Review: MatPlotLib

MatPlotLib

- "a comprehensive library for creating static, animated, and interactive visualizations in Python" - matplotlib.com
- Submodule MatPlotLib.pyplot contains plotting functions that work similar to MATLAB (a very popular data science programming language)
- Uses NumPy ndarray

Review: Pandas

Python Data Analysis Library

- "fast, powerful, flexible and easy to use open source data analysis and manipulation tool" – pandas.pydata.org
- Also uses NumPy ndarrays
- Also integrated with basic MatPlotLib plotting functions
- Useful for working with SQL data
- **DataFrame** object – tabular data structure with rows & columns (axes)
- **Series** object - an indexed 1-dimensional ndarray, used as columns for DataFrame

Workshop 4 Assignment

Goal: MoMa has requested your help in analyzing data from its database.

Task 1: Break down its artworks by department.
Task 2: Break down its artworks by classification.
Task 3: Analyze the diversity of its collection.
Task 4: Come up with a gender breakdown of artists.
Task 5: Bonus Task – Describe in words what is represented by a graph visualizing data from MoMa's database.

You will be split up into groups to work on the assignment together. Talk through each step out loud with each other, code collaboratively. If your team spends more than 10 minutes trying to solve one problem, ask your instructor for help!