Abstract:

Power to the (SQL) people: Python UDFs in DBSQL

Databricks SQL (DBSQL) allows customers to leverage the simple and powerful Lakehouse architecture with up to 12x better price/performance compared to traditional cloud data warehouses. Analysts can use standard SQL to easily query data and share insights using a query editor, dashboards or a BI tool of their choice, and analytics engineers can build and maintain efficient data pipelines, including with tools like dbt.

While SQL is great at querying and transforming data, sometimes you need to extend its capabilities with the power of Python, a full programming language. Users of Databricks notebooks already enjoy seamlessly mixing SQL, Python and several other programming languages. Use cases include masking or encrypting and decrypting sensitive data, complex transformation logic, using popular open source libraries or simply reusing code that has already been written elsewhere in Databricks. In many cases, it is simply prohibitive or even impossible to rewrite the logic in SQL.

Up to now, there was no way to use Python from within DBSQL. We are removing this restriction with the introduction of Python User Defined Functions (UDFs). DBSQL users can now create, manage and use Python UDFs using standard SQL. UDFs are registered in Unity Catalog, which means they can be governed and used throughout Databricks, including in notebooks.

DATA+AI SUMMIT 2022

Power to the (SQL) people: Python UDFs in DBSQL

Stefania Leone,

Sr. Manager Product Management, Databricks

Martin Grund

Sr Staff Software Engineer

ORGANIZED BY databricks

Product Safe Harbor Statement

This information is provided to outline Databricks' general product direction and is for informational purposes only. Customers who purchase Databricks services should make their purchase decisions relying solely upon services, features, and functions that are currently available. Unreleased features or functionality described in forward-looking statements are subject to change at Databricks discretion and may not be delivered as planned or at all.

Agenda

Data Warehousing on the Lakehouse: Databricks SQL and Unity Catalog

Extensibility in Databricks & Databricks SQL: User-defined functions today

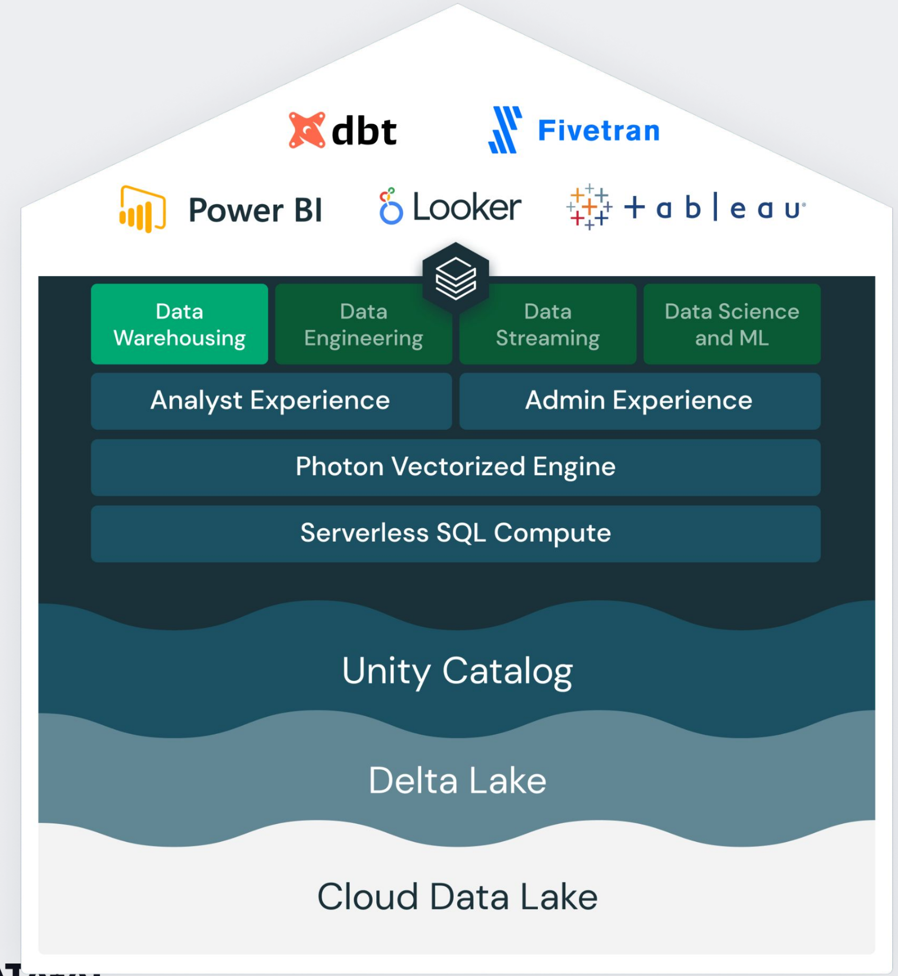
Python UDFs in Databricks SQL incl. demo

Data Warehousing on the Lakehouse:

DBSQL & Unity Catalog

The best data warehouse is a lakehouse

Powered by Databricks SQL



Seamless Integration with the Ecosystem

Ease of Use

Real-world Performance

Centralized Governance

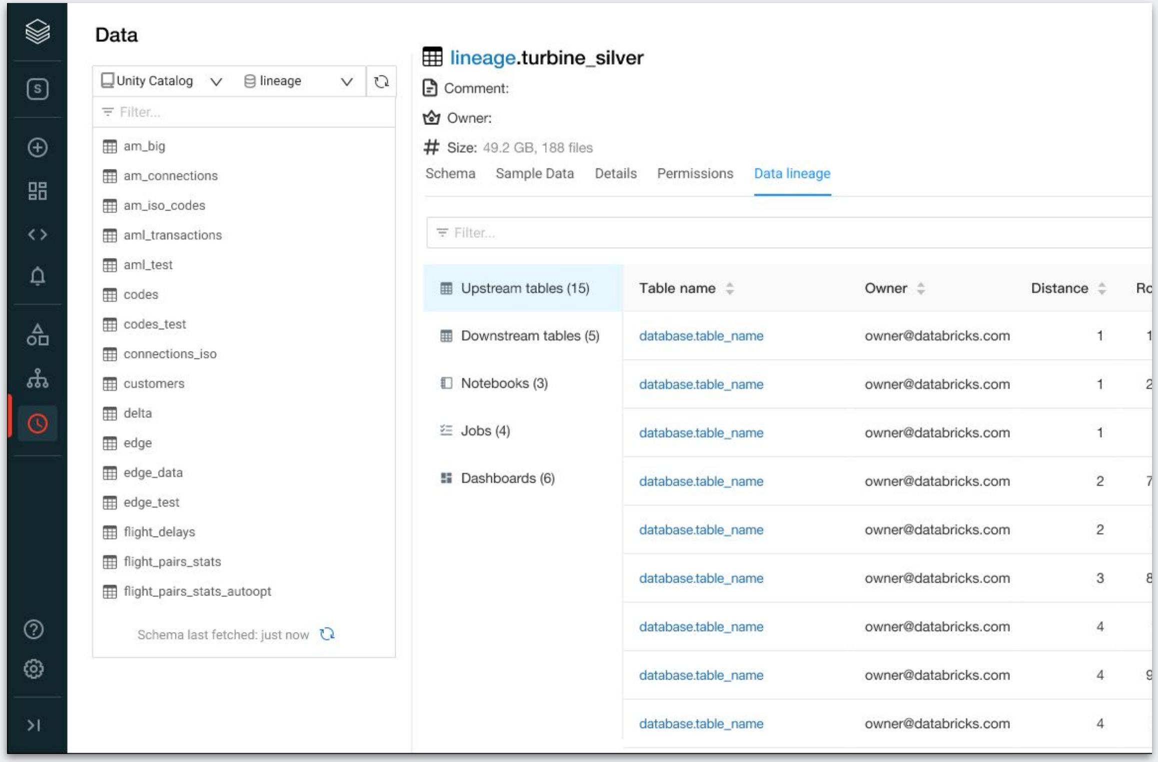
Open and Reliable Data Lake as the Foundation

GA Coming Soon

Centrally govern all your data with standard SQL

Unity Catalog + Databricks SQL

- **Standardize** with a unified fine-grained governance model
- **Easily search, discover and access** all data assets from data explorer
- **Securely share** live data across platforms with Delta Sharing
- Built-in **data lineage** across tables, columns, notebooks, workflows, dashboards
- Captured in real time across all workloads—**SQL, Python, Scala, and R**



Extensibility in Databricks: User-defined functions

User-defined functions (UDFs)

Extensibility in Spark

User defined functions allow to extend Spark with custom business logic:

- Define functions as UDFs
- Use UDFs in Spark

```python
#define UDF
@udf
def redact_json(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
     if k in keys:
         obj[k] = "REDACTED"
  return json.dumps(obj);
 ...
#use UDF
df.select(redact_json(df.json_dump))
```

PySpark UDF used on a Dataframe

Extensibility in Spark

`(User-defined) functions in the language of your choice`

PySpark	Python UDFs Pandas UDFs, Pandas API
Scala/Java	UDFs
SQL	Built-in & Lambda functions SQL UDFs *Use registered Python/Pandas/Scala UDFs*

Python UDF example

Redacting PII data from JSON fields

Python UDF

```
@udf
def redact_json(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
     if k in keys:
         obj[k] = "REDACTED"
  return json.dumps(obj);
```

```
spark.udf.register('redact', redact_json)
```

```
SELECT redact(json_dump) as redacted
FROM default.rawdata
```

Characteristics

Define UDF:

- Arbitrary code as Python functions
- PySpark: annotate as UDF
- Spark SQL: register in Spark session

→ UDFs are session-based (not cataloged)

Run UDF:

- Use in SQL, PySpark
- Row-at-a-time processing

Pandas UDF example

Redacting PII data from JSON fields

Pandas UDF

```
def redact_json(a):
  ...
  return json.dumps(obj);
```

```
@pandas_udf('string', PandasUDFType.SCALAR)
def redact_json_pd(batch: pd.Series) -> pd.Series:
  return batch.apply(redact_json);
```

```
spark.udf.register('redact_pd', redact_json_pandas)
```

```
SELECT redact(json_dump) as redacted
FROM default.rawdata
```

Characteristics

- Arbitrary Python code
- Session-based
- Vectorized UDF: runs the UDF on batches (pandas.Series)
- Faster than Python UDFs, especially for row-independent state

DBSQL Extensibility

DBSQL Extensibility

- Support for SQL built-in and Lambda functions
- SQL UDFs
- No support for non-SQL UDFs

SQL UDF Example

Example: Email Masking

- Create a reusable SQL expression to mask emails.
- SQL UDFs are cataloged
- Created by a user with USAGE and CREATE permission on the schema, USAGE on the catatalog

email
john.doe@laview.com
martin.grund@databricks.com

mask_email()

email
jo...oe@la...ew.com
ma...nd@da...ks.com

```
CREATE OR REPLACE FUNCTION mycatalog.finance.mask_email(email string)
RETURNS STRING LANGUAGE SQL
RETURN SELECT substring(split_part(email, "@", 1), 1, 2) || '...'
 || substring(split_part(email, "@", 1), -2)
 || '@' || substring(split_part(email, "@", 2), 1, 2) ||  '...'
 || substring(split_part(split_part(email, "@", 2), '.', 1), -2) || '.'
 || split_part(split_part(email, "@", 2), '.', -1);
```

SQL UDF Example

Example: Email Masking

- GRANT SQL analyst permission to run the function
- Use mask_email() as part of a query

```sql
GRANT EXECUTE on mycatalog.finance_db.mask_email
TO 'finance_analysts'
```

```sql
SELECT first_name, last_name, mask_email(email)
FROM account_info;
```

email
john.doe@laview.com
martin.grund@databricks.com

mask_email()

email
jo...oe@la...ew.com
ma...nd@da...ks.com

More power to the SQL people

Beyond SQL UDFs

- Some logic is hard or impossible to express in SQL

Redact example in Python

```python
def redact_json(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
     if k in keys:
         obj[k] = "REDACTED"
  return json.dumps(obj);
```

Redact example in SQL

```sql
with surrogate as (
    select
    ROW_NUMBER() OVER (order by json_dump) as rn,
    json_dump
    from default.rawdata
),
 exploded as (
  select
    rn,
    explode(from_json(json_dump, 'MAP<STRING,STRING>'))
  from surrogate
),
redacted as (
  select
    rn,
    collect_list(
      struct(
        key,
        if(key in ('email', 'phone'), 'REDACTED', value)
      )
    ) as attr
  from
    exploded
  group by 1
)
select
  map_from_entries(attr) as json_dump
from
  redacted;
```

More power to the SQL people

Beyond SQL UDFs

- Some logic is hard or impossible to express in SQL
- A lot of business logic has already been implemented in Python,

by all of you!

Let's bring the Power of Python &

your existing business logic to Databricks SQL

as fully cataloged and governed UDFs!

Introducing Python UDFs for Databricks SQL

Scalar Python UDFs in Databricks SQL

Power to the SQL People

Bring Python's expressive power to Databricks SQL.

```
CREATE FUNCTION redact(a STRING)
RETURNS STRING
LANGUAGE PYTHON
AS $$
  return "Hello World"
$$;
```

- Permanent, first-class object in **Unity Catalog**.
  - UDFs can be governed using GRANT/REVOKE syntax.
  - Accessible using the standard three level namespace syntax.
- Fully **sandboxed** and **isolated** execution mode without cross-query interference.

Scalar Python UDFs in Databricks SQL

Syntax Composition

Mapping between Python and SQL code.

```python
def function_name(arg1, arg2):
    if arg1 > 2:
        return arg2
    else:
        return arg1
```

```sql
CREATE FUNCTION function_name(
    arg1 DOUBLE, arg2 DOUBLE)
RETURNS DOUBLE LANGUAGE PYTHON
AS $$
  if arg1 > 2:
    return arg1
  else:
    return arg2
$$
```

Python UDF Example

Redacting PII data from JSON fields

ID	...	Attributes
2022-05-01	...	{"phone": "555-123-3412", "project": "silver", "email": "geronimo@galiato.ab"}
2022-05-01	...	{"phone": "555-372-7482", "project": "gold", "email": "goldenrules@bretteck.co"}

Example: Redact all fields in the JSON string where the keys are in a deny-list with "REDACTED"

```
CREATE FUNCTION redact(a STRING)
RETURNS STRING
LANGUAGE PYTHON
AS $$
import json
keys = ["email", "phone"]
obj = json.loads(a)
for k in obj:
   if k in keys:
       obj[k] = "REDACTED"
return json.dumps(obj)

$$;
```

redact()

ID	...	Attributes
2022-05-01	...	{"phone": "REDACTED", "project": "silver", "email": "REDACTED"}
2022-05-01	...	{"phone": "REDACTED", "project": "gold", "email": "REDACTED"}

Seamless transition from Spark UDFs to Python UDFs

From Spark UDFs to Python UDFs in DBSQL

Step 1: CREATE FUNCTION (instead of spark.udf.register)

```python
import json

def my_redact(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
    if k in keys:
      obj[k] = "REDACTED"
  return json.dumps(obj)

spark.udf.register("redact", my_redact)
```

1

```sql
CREATE FUNCTION redact(a STRING)
RETURNS STRING
LANGUAGE PYTHON
AS $$
import json
keys = ["email", "phone"]
obj = json.loads(a)
for k in obj:
    if k in keys:
        obj[k] = "REDACTED"
return json.dumps(obj)
$$;
```

From Spark UDFs to Python UDFs in DBSQL

Step 2: Function body

```python
import json

def my_redact(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
    if k in keys:
      obj[k] = "REDACTED"
  return json.dumps(obj)

spark.udf.register("redact", my_redact)
```

2

```sql
CREATE FUNCTION redact(a STRING)
RETURNS STRING
LANGUAGE PYTHON
AS $$
import json
keys = ["email", "phone"]
obj = json.loads(a)
for k in obj:
   if k in keys:
       obj[k] = "REDACTED"
return json.dumps(obj)
$$;
```

Multiple functions: Keep all definitions, and only inline the outermost function (or call `return outermost()` from global scope)

From Spark UDFs to Python UDFs in DBSQL

Step 3: Import dependencies

3

```python
import json

def my_redact(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
    if k in keys:
      obj[k] = "REDACTED"
  return json.dumps(obj)

spark.udf.register("redact", my_redact)
```

Spark UDF in Notebook

```sql
CREATE FUNCTION redact(a STRING)
RETURNS STRING
LANGUAGE PYTHON
AS $$
import json
keys = ["email", "phone"]
obj = json.loads(a)
for k in obj:
   if k in keys:
       obj[k] = "REDACTED"
return json.dumps(obj)
$$;
```

Python UDF in DBSQL

From Spark UDFs to Python UDFs in DBSQL

Alternative: Functions with multiple dependencies

```python
import json

def my_hash(a):
  return hash(a)

def my_filter(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
    if k in keys:
      obj[k] = my_hash(obj[k])
  return json.dumps(obj)

spark.register("redact", my_filter)
```

1

```sql
CREATE FUNCTION redact(a STRING)
RETURNS STRING
LANGUAGE PYTHON
AS $$
import json

def my_hash(a):
  return hash(a)

def my_filter(a):
  keys = ["email", "phone"]
  obj = json.loads(a)
  for k in obj:
    if k in keys:
      obj[k] = my_hash(obj[k])
  return json.dumps(obj)
return my_filter(a)
$$;
```

2

Demo

Online Model Scoring In
Databricks SQL

ML Example

Leveraging Model Scoring in Databricks SQL

Goal: Define a UDF that leverages an integrated Scikit-Learn model for predicting housing prices in Berlin.

Path to UDF:

- Training data (blog) from a publicly available dataset.
- Train the model, serialize the model, create UDF.

ML Example

Training the model

Execution in a regular Notebook or Python REPL using Scikit-Learn, PySpark, Pandas.

1. Load the data
2. Train the Model
3. Generate the SQL Code for the UDF

1

```
# Load Data from a table stored in Unity Catalog.
df = spark.read.table("berlin_housing_data")
# Convert result to Pandas DataFrame, selecting only the features
# to use for training.
X = df.select(df.living_space.cast(FloatType()),
              df.number_rooms.cast(IntegerType())).toPandas().to_numpy()
# Select target column.
Y = df.select(df.cold_price.cast(FloatType())).toPandas()
y = Y["cold_price"].values

# Prepare the training data
X_train, X_test, y_train, y_test = train_test_split(
   X, y, test_size=0.1, random_state=13
)
params = { "n_estimators": 500, "max_depth": 4, "min_samples_split": 5,
           "learning_rate": 0.01, "loss": "ls"}
```

2

```
# Run the training.
reg = ensemble.GradientBoostingRegressor(**params)
reg.fit(X_train, y_train)
```

```
# Build a UDF based on the serialized model.
data = base64.b64encode(zlib.compress(pickle.dumps(reg)))
template = f"""CREATE OR REPLACE FUNCTION score(sqm float, rooms int)
RETURNS FLOAT
LANGUAGE PYTHON
RETURN $$
import zlib, pickle, base64
data = {data}
pred = pickle.loads(zlib.decompress(base64.b64decode(data)))
return float(pred.predict([[sqm, rooms]])[0])
$$
"""
```

3

```
display(template)
```

ML Example

Create the UDF

⚠ This shows the power of UDFs, but computation cost is high on every iteration.

- From the SQL Query editor in Databricks SQL paste the previously generated query.

Python UDF with UC Preview (L) | Share | Save | Run (limit 1000)

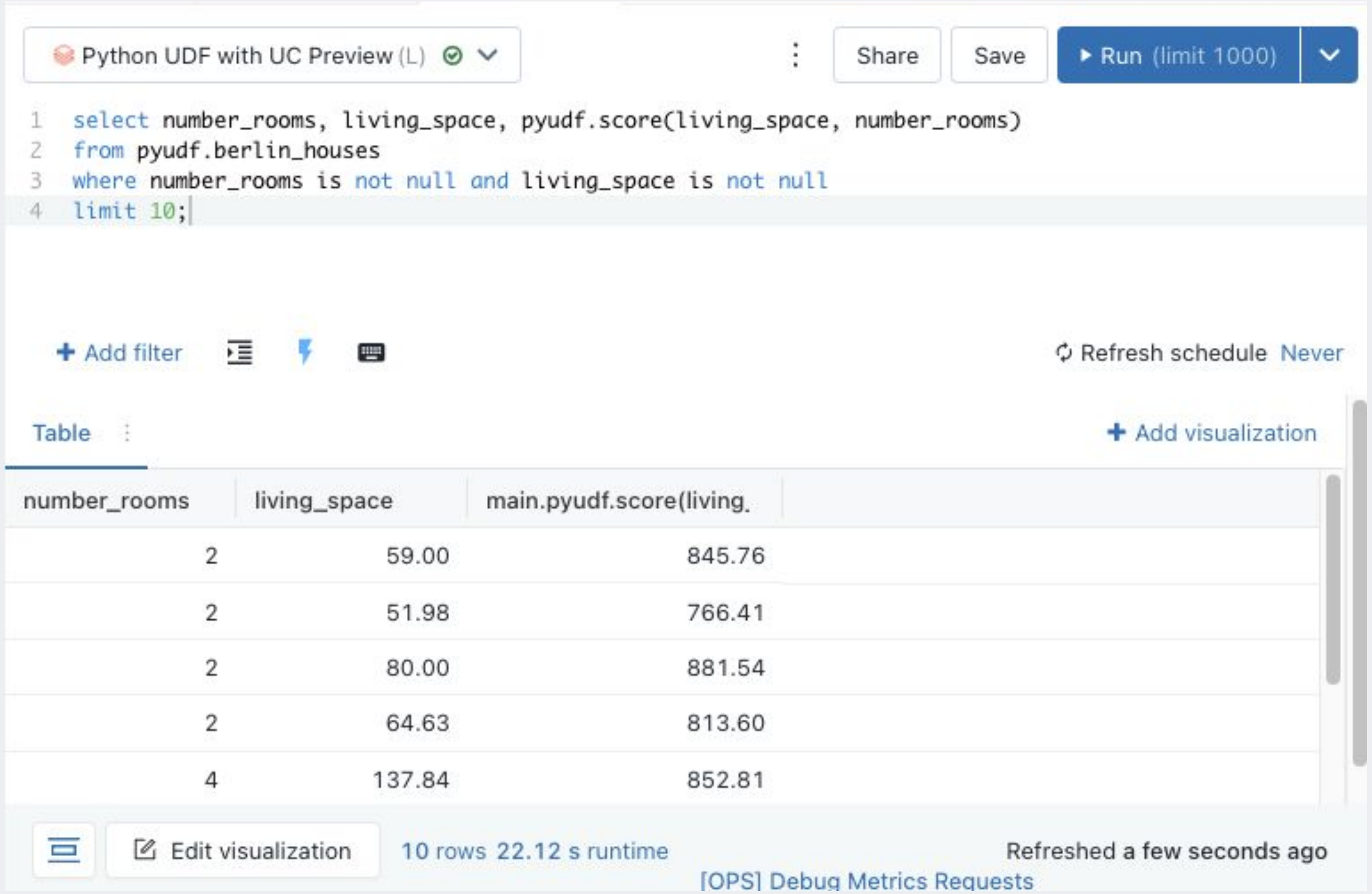
```
CREATE OR REPLACE FUNCTION pyudf.score2(sqm float, rooms int)
RETURNS FLOAT
LANGUAGE PYTHON
AS $$
import zlib, pickle, base64
data = b'eJzknQdUFMnasAFBQXJUARUDOWckFjnnnHPOGQEFCSJBkKCSJSmCCigKCAgloCKCIkFFUJIEERWRoJh+dN29eN17udV7jufbf8cjM0zP29Xd08NTz/tW1
pred = pickle.loads(zlib.decompress(base64.b64decode(data)))
return float(pred.predict([[sqm, rooms]])[0])
$$
```

ML Example

Run predictions!

```sql
select number_rooms, living_space, pyudf.score(living_space, number_rooms)
from pyudf.berlin_houses
where number_rooms is not null and living_space is not null
limit 10;
```

- Run the predictions in batch directly in Databricks SQL.
- Use the UDF like any other built-in function.
- Consume the result in custom visualizations and dashboards.



Conclusion and Outlook

Summary

- Power to the SQL people: Python UDFs in DBSQL bring the expressive power of Python to Databricks SQL
- UDFs are registered as UC objects with fine-grained access control
- Existing code and application logic in Python UDFs can be seamlessly created in Databricks SQL

Outlook

- Private Preview Sign-Up: https://dbricks.co/udfpreview
- Public preview planned for Q3

Roadmap

- Pandas UDFs in DB SQL
- User-defined dependencies
- Governed UDFs in Notebooks
- Remote functions

DATA+AI SUMMIT 2022

Thank you

Stefania Leone

Senior Manager, Product Management

Martin Grund

Sr Staff Software Engineer