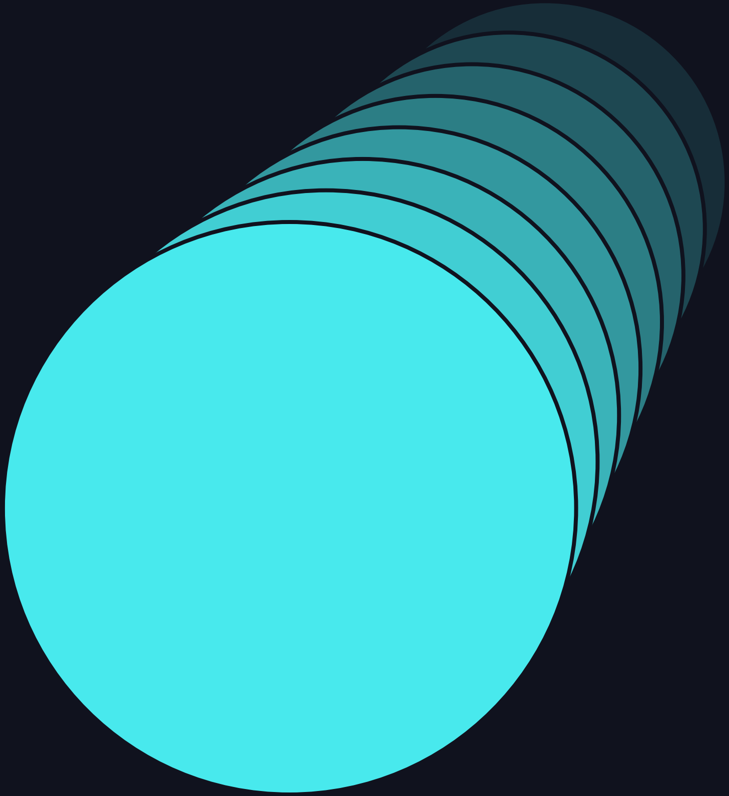
DATA+AI SUMMIT
BY databricks

SQL PROGRAMMING IN DATABRICKS



Serge Rielau & Milan Stefanovic
June 2024

Agenda

- Sorting - your way
  - Sorting points by distance using LAMBDA and SQL UDF
  - Sorting strings, properly (Sneak peek!)
- Short hands
  - GROUP BY, ORDER BY
  - SELECT * - unleashed
- Variables
  - SQL Session Variables
  - What about identifiers?
- Scripting
  - EXECUTE IMMEDIATE
  - SQL/PSM: It's like SQL, but scripted (Sneak peek!)

Sorting - Your way

Quicksort and custom sort expressions

- Task

  “Sort an array of points by distance from (0, 0)”

- Need a custom sort order
  - array_sort() for sorting
  - lambda function for the math
- Hide complexity in a SQL UDF

LAMBDA functions

Anonymous function with one or more named parameters

- Can be passed to a number of builtin map/array functions
- Operates on each element, value of the map/array

```
p -> expr(p)
(p[, ...]) -> expr(p, ...)
```

- `p`: One or more identifiers, as required by the host function.
- `expr(p[, ...])`: A simple (no subqueries, or SQL UDF) expression using p. Result must comply with expectations of the host function.

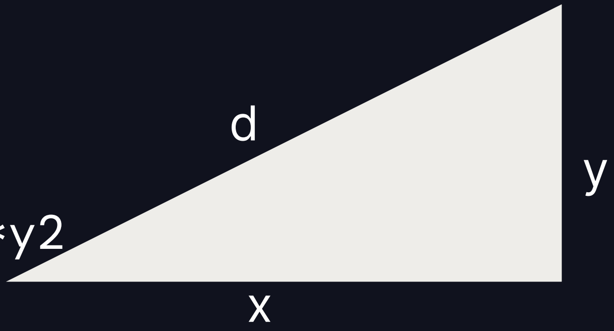
A simple quicksort

lambda(a,b) > 0 => a > b

-
  ```
  SELECT array_sort(array(5, 2, 8, 1, 3),
                    (a, b) -> a - b) AS sorted;
  => [1, 2, 3, 5, 8]
  ```

Sorting Distances

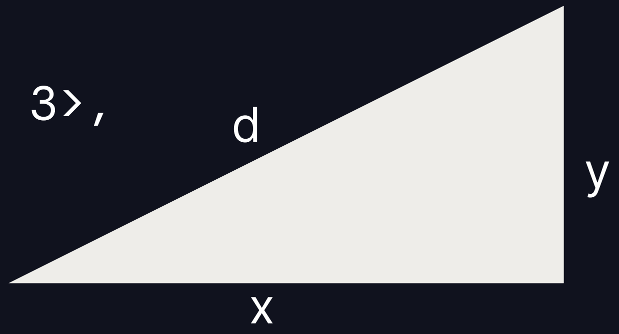
- `d = sqrt(x*x + y*y)`
- `d1 < d2 <=> x1*x1 + y1*y1 < x2*x2 + y2*y2`



Sorting by distance

lambda(p1, p2) > 0 <=> (x1*x1 + y1*y1) > (x2*x2 + y2*y2)

-
  ```
  SELECT array_sort(points,
                    (p1, p2) -> (p1.x*p1.x + p1.y*p1.y)
                              - (p2.x*p2.x + p2.y*p2.y))
         AS points
    FROM point_arrays;
  => [<1,1>, <-2,0>, <3, 0>, <0,-4>, <3, 3>,
      <0,5>, <-5,5>, <6,-4>]
  ```



SQL UDF

Scalar and Table UDFs written in SQL

- Stored in UC as a reusable asset
- Support named parameter invocation and defaulting

Scalar

- Encapsulate (complex) expressions, including subqueries
- May contain subqueries
- Return a scalar value
- Can be used in most places builtin functions go.

Table

- Encapsulate (complex) correlated subqueries aka a parameterized view
- Can be used in the FROM clause

SQL UDF sorting by distance

Hiding all that complexity

```
CREATE FUNCTION points_sort(
   points array<struct<x INT, y INT>>)
  RETURN array_sort(points,
                    (a, b) -> (a.x*a.x + a.y * a.y)
                            - (b.x*b.x + b.y * b.y));


SELECT points_sort(points) AS points
  FROM point_arrays;
=> [<1,1>, <-2,0>, <3, 0>, <0,-4>, <3, 3>,
    <0,5>, <-5,5>, <6,-4>]
```

Announcing Collation Support

ANSI SQL COLLATE (private preview)

Sorting and comparing strings according to locale

- Associate columns, fields, array elements with a collation of choice
  - Case insensitive
  - Accent insensitive
  - Locale aware
- Supported by many string functions such as
  - lower()/upper()
  - substr()
  - locate()
  - like
- Supported by Delta and Photon
- GROUP BY, ORDER BY, comparisons, ...

Private Preview

A look at the default collation

A<Z<a<z<Ā

```sql
SELECT name FROM names ORDER BY name;
```

```
Name
Anthony
Bertha
anthony
bertha
Ānthōnī
```

Is this really what we want here?

COLLATE UNICODE

One size, fits most

```sql
SELECT name FROM names
 ORDER BY name COLLATE unicode;
```

```
Name
Ānthōnī
anthony
Anthony
bertha
Bertha
```

Root collation with decent sort order for most locales

COLLATE UNICODE_CI

Case insensitive comparisons have entered the chat

```sql
SELECT name
 FROM names
 WHERE startswith(name COLLATE unicode_ci, 'a')
 ORDER BY name COLLATE unicode_ci;
```

```
Name
anthony
Anthony
```

Case insensitive is not accent insensitive: We lost Ānthōnī

COLLATE UNICODE_CI_AI

Equality from a to Ź

```sql
SELECT name
 FROM names
 WHERE startswith(name COLLATE unicode_ci_ai, 'a')
 ORDER BY name COLLATE unicode_ci_ai;
```

```
Name
Ānthōnī
anthony
Anthony
```

100s of supported predefined collations across many locales

SQL Shorthands

GROUP BY and ORDER BY

Humble beginnings

- Before

```
SELECT last, first, id, mgr, extract(year FROM workday),
       sum(hours), sum(pay)
  FROM emps
  GROUP BY last, first, id, mgr, extract(year FROM workday)
  ORDER BY last, first, id, mgr, extract(year FROM workday)
```

- After

```
SELECT last, first, id, mgr, extract(year FROM workday),
       sum(hours), sum(pay)
  FROM emps
  GROUP BY 1, 2, 3, 4
  ORDER BY 1, 2, 3
```

Is that the best we can do?!

GROUP BY and ORDER BY

Just do it!

- Expectation
  - GROUP BY all column in the select list which are not aggregated!
  - ORDER BY all columns left to right (or enough to guarantee uniqueness)
- Let Databricks figure it out

```
SELECT last, first, id, mgr, extract(year FROM workday),
       sum(hours), sum(pay)
  FROM emps
  GROUP BY ALL
  ORDER BY ALL
```

- Better, ...

SELECT * to ALL

Hold my beer!

- SELECT * is the bad boy of SQL!
  - What if the schema changes?
  - No one knows what the SQL is doing!
- We all hate it... But we all use it... why?

  Carry over from OLTP where schema evolution is tightly controlled.

In the Lakehouse, schema evolution is expected!

SELECT *

The early years

```
SELECT * FROM t, s;
```

- Select all columns available in FROM

```
SELECT t.* FROM t, s;
```

- Select all columns available in t

```
SELECT * EXCEPT (col1, col2) FROM t, s;
```

- Select all column except col1 and col2
- Can also exclude fields in a struct!

* unleashed

Unnesting fields in a struct

```
WITH person(name, first, address) AS
     (VALUES('Coyote', 'Wiley',
              named_struct('street', '123 Canyon Rd',
                           'city', 'Grand Canyon',
                           'zip', 12345)))
SELECT * EXCEPT (address),
       address.* EXCEPT(street)
  FROM person;
```

```
name   first city         zip
Coyote Wiley Grand Canyon 12345
```

* unleashed

Nesting columns into a struct

```
WITH person(name, first, street, city, zip) AS
     (VALUES('Coyote', 'Wiley', '123 Canyon Rd',
             'Grand Canyon', 12345))
SELECT name, first,
       struct(* EXCEPT (name, first)) AS address
  FROM person;
```

```
name   first address
Coyote Wiley {street:"123 Canyon Rd",city:"Grand Canyon",
              zip: 12345}
```

* unleashed

Transposing columns into an array

```
WITH contact(name, work, home) AS
  (VALUES('Coyote', '905-555-1234', '408-555-1234'))
SELECT name,
       array(* EXCEPT (name)) AS numbers
  FROM contact;
```

```
name   numbers
Coyote ['905-555-1234', '408-555-1234']
```

* unleashed

Can be used just about anywhere.

For fixed & variable length argument functions

- LEAST(*)
- GREATEST(*)
- COALESCE(*)
- CONCAT(*), CONCAT_WS(*)
- ... and with UDFs, too

Even in the WHERE clause e.g. IN(*)

* unleashed

Finally: Serializing a row into a string

```
WITH person(name, first, street, city, zip) AS
  (VALUES('Coyote', 'Wiley', '123 Canyon Rd',
          'Grand Canyon', 12345))
SELECT concat_ws(', ', *) AS result
  FROM person;
```

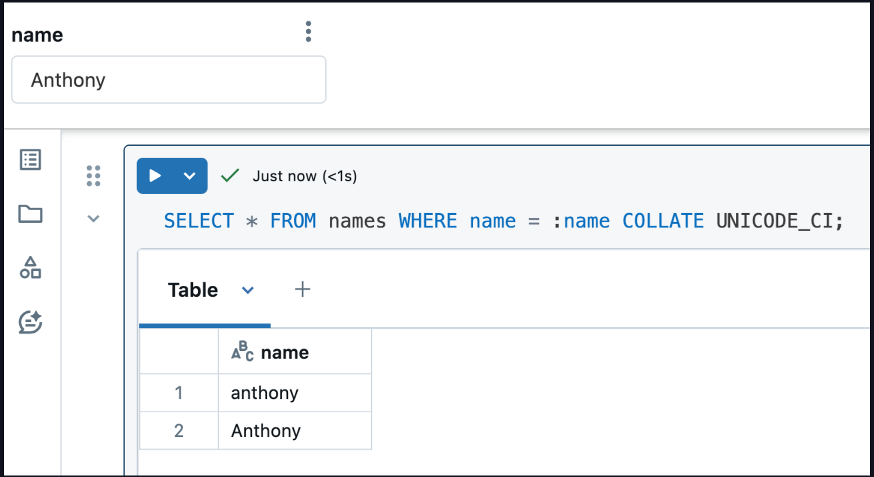
```
result
Coyote, Wiley, 123 Canyon Rd, Grand Canyon, 12345
```

Variables et al.

Named parameters

The mustache ‘{{ }}’ is dead, long live the colon ‘:’!

- Uniquely named placeholder for a typed literal
- Safe from SQL injection
- Adjust type and input through automatically generated notebook widget
- Reference most places literals go.



name

Anthony

Just now (<1s)

```
SELECT * FROM names WHERE name = :name COLLATE UNICODE_CI;
```

Table

	name
1	anthony
2	Anthony

Cannot fill value from SQL ...

Session variables

Flowing data through a session, using SQL only.

- Declarative, with type and default.

  ```
  DECLARE VARIABLE name STRING DEFAULT 'anthony';
  ```

  or

  ```
  DECLARE name = 'anthony';
  ```

- Reference anywhere a query literal can go.

  ```
  SELECT * FROM names
   WHERE name = session.name COLLATE unicode_ci;
  ```

  ```
  name
  anthony
  Anthony
  ```

Session variables

Flowing data through a session, using SQL only.

- Set using SQL expressions

```
SET VAR name = (SELECT min(name) FROM names);
```

- Set multiple variables at once

```
SET VAR (first, last) =
  (SELECT first, last FROM person WHERE id = :id)
```

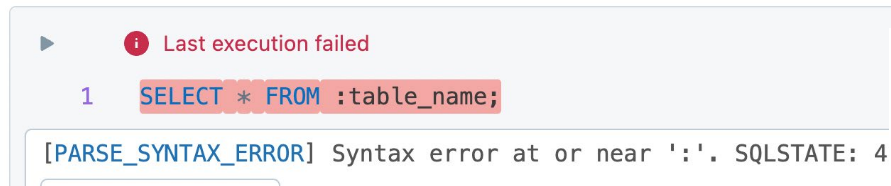
- Reset to default

```
SET VAR name = DEFAULT;
```

- Private to the session (like a temp view)

How about table parameters/variables?

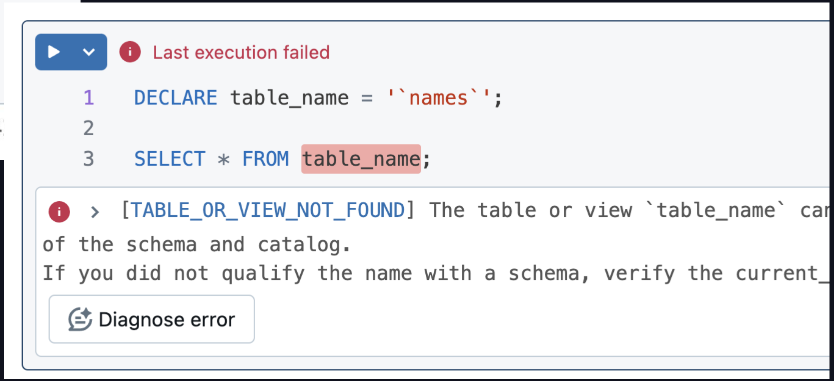
Can I pass a table name as a parameter?



Last execution failed

```sql
SELECT * FROM :table_name;
```

```
[PARSE_SYNTAX_ERROR] Syntax error at or near ':'. SQLSTATE: 4
```



Last execution failed

```sql
DECLARE table_name = '`names`';

SELECT * FROM table_name;
```

```
[TABLE_OR_VIEW_NOT_FOUND] The table or view `table_name` car
of the schema and catalog.
If you did not qualify the name with a schema, verify the current_
```

- Values are not names
- Need to teach Databricks to
  - Evaluate value during parsing
  - Turn values into a name

Parameterizing names

Using the IDENTIFIER clause

```
IDENTIFIER(constStr)
```

constStr: an expression that can be evaluated as a string before query runs.

Applies to

- Most identifiers in a query, or DML statement function/column/table name
- Subject of many DDL statements ALTER/CREATE/DROP
- Subject of auxiliary statements
- USE

Parameterizing names

Using session variables

```
DECLARE table_name = 'names';
DECLARE col_name   = 'name';
DECLARE func_name  = 'count';

SELECT IDENTIFIER(func_name)(IDENTIFIER(col_name))
  FROM IDENTIFIER(table_name) AS t
  WHERE IDENTIFIER('t.' || col_name) = 'Anthony';
result
1
```

Parameterizing names

Using named parameters

```
SELECT IDENTIFIER(:func_name)(IDENTIFIER(:col_name))
  FROM IDENTIFIER(:table_name) AS t
  WHERE IDENTIFIER('t.' || :col_name) = 'Anthony';
result
1
```

SQL Scripting

SQL Scripting

It’s SQL, but with control flow!

SQL is nice but I need python for procedural stuff

Not anymore

- Support for control flow, iterators & error handling Natively in SQL
  - Control flow → `IF/ELSE, CASE`
  - Looping → `WHILE, REPEAT, ITERATE`
  - Resultset iterator → `FOR`
  - Exception handling → `CONTINUE/EXIT`
  - Parameterized queries → `EXECUTE IMMEDIATE`
- Following the SQL/PSM standard

Private Preview

SQL Scripting - real world example

Renaming all columns with spelling errors

I have a common column in many tables that has a spelling error and I want to rename it in all tables.

colour

SQL Scripting - real world example

How do I find all my tables?

- Use the information schema

```
-- Fetch all tables in desired catalog and schema
-- and store them into array
SELECT
    array_agg(table_name)
FROM INFORMATION_SCHEMA.columns
WHERE column_name = oldColName
```

SQL Scripting - real world example

Loop through the tables

- Iterate with WHILE loop

```
WHILE i < array_size(tableArray)
DO
.
.
END WHILE;
```

SQL Scripting - real world example

Conditional logic to special case views

- But you cannot rename column in VIEWs
- Solution: NEED IF branch

```
IF tableType != 'VIEW'
THEN
…
ELSE -- it’s a view
…
END IF;
```

SQL Scripting - real world example

Dynamically generate SQL

- Finally we need to construct alter statement based on table and column names.
- Solution: EXECUTE IMMEDIATE

```
EXECUTE IMMEDIATE
'ALTER TABLE ' || tableName ||
' RENAME COLUMN ' || oldColName || ' TO ' || newColName
```

SQL Scripting - real world example

Glueing it all together!

```
-- parameters
DECLARE oldColName = 'ColoUr';
DECLARE newColName = 'color';

BEGIN
 DECLARE tableArray Array<STRING>;
 DECLARE tableType STRING;
 DECLARE i INT = 0;
 DECLARE alterQuery STRING;

 SET tableArray = (
  SELECT array_agg(table_name)
   FROM INFORMATION_SCHEMA.columns
   WHERE column_name COLLATE UNICODE_CI
     = oldColName);

 WHILE i < array_size(tableArray) DO
  SET tableType = (
   SELECT table_type
    FROM INFORMATION_SCHEMA.tables
    WHERE table_name = tableArray[i]);

  IF tableType != 'VIEW' COLLATE UNICODE_CI
  THEN
   SET alterQuery =
    'ALTER TABLE ' || tableArray[i] ||
    ' RENAME COLUMN ' || oldColName ||
    ' TO ' || newColName;
   EXECUTE IMMEDIATE alterQuery;
  END IF;

  SET i = i + 1;
 END WHILE;
END;
```

Summary

Databricks supports interesting SQL features with many more to come

- Lambda functions
- SQL UDF
- String collation — In Private Preview
- Named Parameter Markers
- SQL Session variables
- IDENTIFIER clause
- EXECUTE IMMEDIATE
- SQL Scripting — In Private Preview

Q&A

Private Preview Signup form

https://forms.gle/qXMG2NKj3DbHg1Lh7