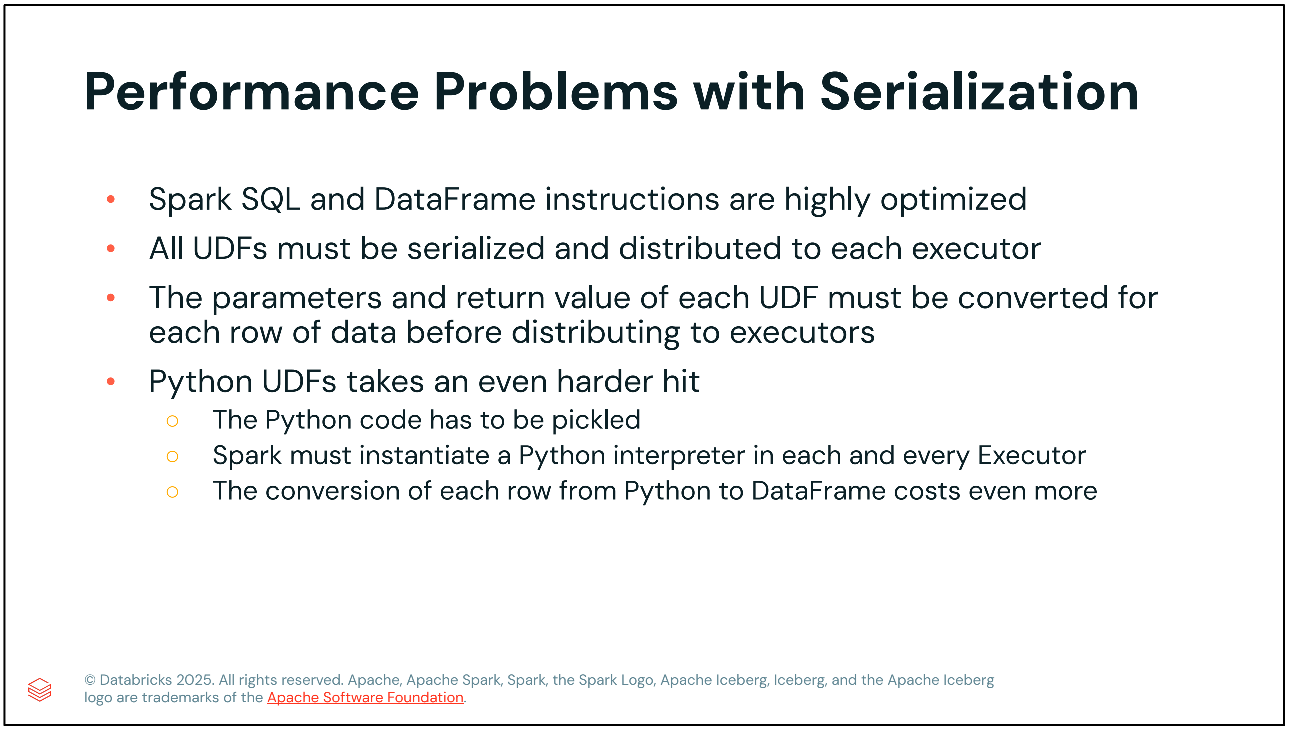
databricks

Code Optimization

LECTURE

Serialization

This lecture discusses performance issues caused by serialization in Spark and techniques to mitigate them.

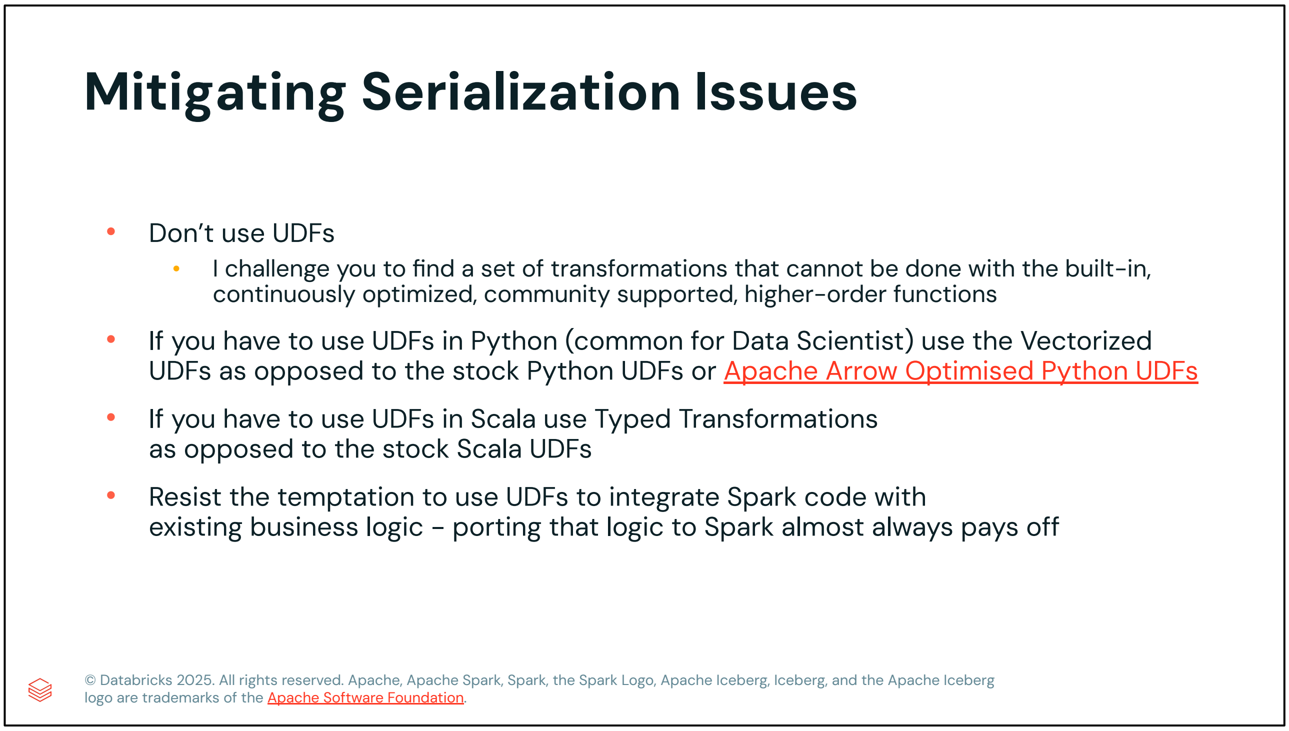


Performance Problems with Serialization

- Spark SQL and DataFrame instructions are highly optimized
- All UDFs must be serialized and distributed to each executor
- The parameters and return value of each UDF must be converted for each row of data before distributing to executors
- Python UDFs takes an even harder hit
  - The Python code has to be pickled
  - Spark must instantiate a Python interpreter in each and every Executor
  - The conversion of each row from Python to DataFrame costs even more

Serialization is an important code optimization topic, especially when working with user-defined functions (UDFs). Serialization refers to the process needed when a UDF is created—this function must be serialized and sent out to each executor across the cluster. While Spark SQL and DataFrame operations are highly optimized and can take advantage of Spark's internal efficiencies, this level of optimization only applies when those operations are used directly. If a custom UDF is written, it has to be serialized and distributed to all executors, which adds overhead in both time and resources. Additionally, the parameters and return values for each invocation of the UDF need to be converted for every row of data distributed to the executors, which further increases the computational cost.

Python UDFs are the least optimized in this context—Python code must be pickled (serialized in a format suitable for Python), and then Spark must start a Python interpreter in every executor. Each row's conversion back and forth between Python and the DataFrame adds another layer of overhead, making these UDFs much less efficient compared to native Spark SQL or DataFrame expressions.



Mitigating Serialization Issues

- Don't use UDFs
  - I challenge you to find a set of transformations that cannot be done with the built-in, continuously optimized, community supported, higher-order functions
- If you have to use UDFs in Python (common for Data Scientist) use the Vectorized UDFs as opposed to the stock Python UDFs or Apache Arrow Optimised Python UDFs
- If you have to use UDFs in Scala use Typed Transformations as opposed to the stock Scala UDFs
- Resist the temptation to use UDFs to integrate Spark code with existing business logic - porting that logic to Spark almost always pays off

- UDFs create an analysis barrier for the Catalyst Optimizer
- The Catalyst Optimizer cannot connect code before and after UDF
- The UDF is a black box which means optimizations are limited to the code before and after, excluding the UDF and how all the code works together

databricks

Thank you for completing this lesson and continuing your journey to develop your skills with us.