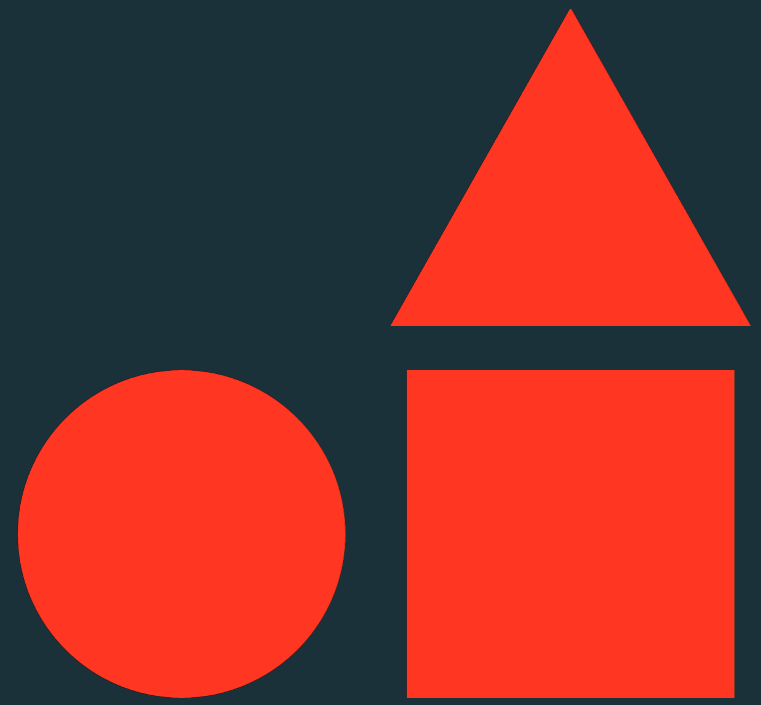
databricks

What's Next for the Upcoming Apache Spark 4.0 Release?

Wenchen Fan cloud-fan
Xiao Li gatorsmile

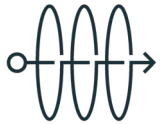
Data + AI Summit 2024



GA

Spark Connect

ANSI Mode



Arrow optimized Python UDF

pandas 2 API parity

Major Features

Structured Logging

[WIP] Variant Data Types

Python Data Source APIs

SQL UDF/UDTF [WIP]

Python

applyInArrow DF.toArrow

Polymorphic Python UDTF

PySpark UDF Unified Profiling

UDF-level Dependency Control [WIP]

SQL

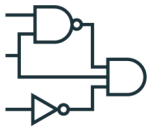
[WIP] Collation Support

[WIP] Stored Procedures

Execute Immediate

View Evolution

Streaming



Arbitrary Stateful Processing V2

State Data Source Reader

Streaming Python Data Sources

New Streaming Doc

More

Error Class Enhancements

Java 21

XML Connectors

Spark K8S operator

Agenda

New Functionalities

Spark Connect, ANSI Mode, Arbitrary Stateful Processing V2, Collation Support, Variant Data Types, pandas 2.x Support

Extensions

Python Data Source APIs, XML/Databricks Connectors and DSV2 Extension, Delta 4.0

Custom Functions and Procedures

SQL UDFs, SQL Scripting, Python UDTF, Arrow optimized Python UDF, PySpark UDF Unified Profiler

Usability

Structured Logging Framework, Error Class Framework, Behavior Change Process

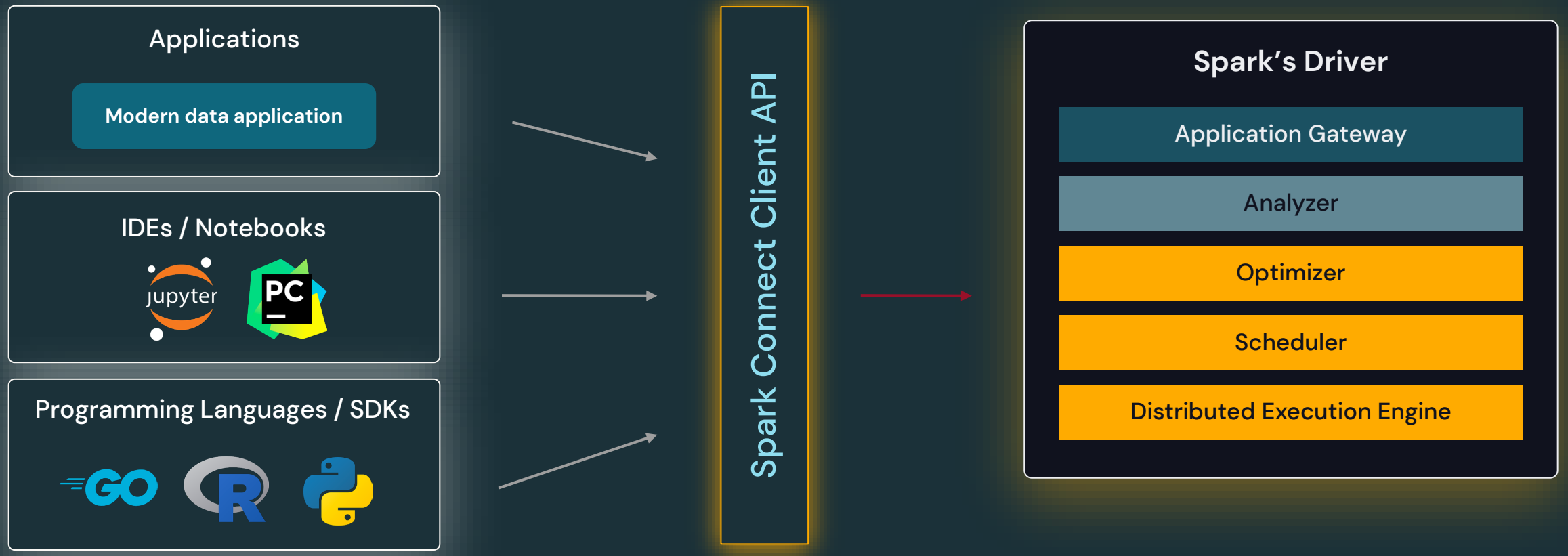
Spark Connect

How to embed Spark in applications?

Up until Spark 3.4: Hard to support today's developer experience requirements

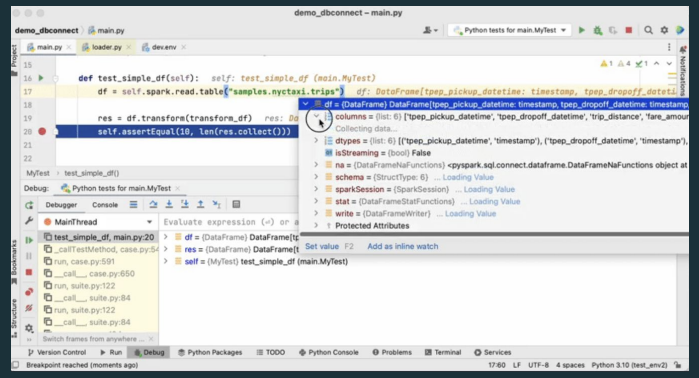
Connect to Spark from Any App

Thin client, with full power of Apache Spark



Spark Connect GA in Apache Spark 4.0

Interactively develop & debug from your IDE



`pip install pyspark>=3.4.0`

in your favorite IDE!

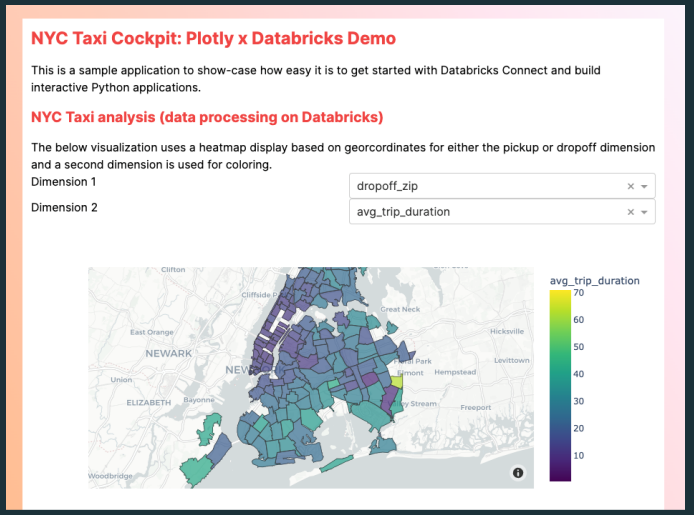
New Connectors and SDKs in any language!

Scala3

Databricks Connect

Check out Databricks Connect, use & contribute the Go client

Build interactive Data Applications

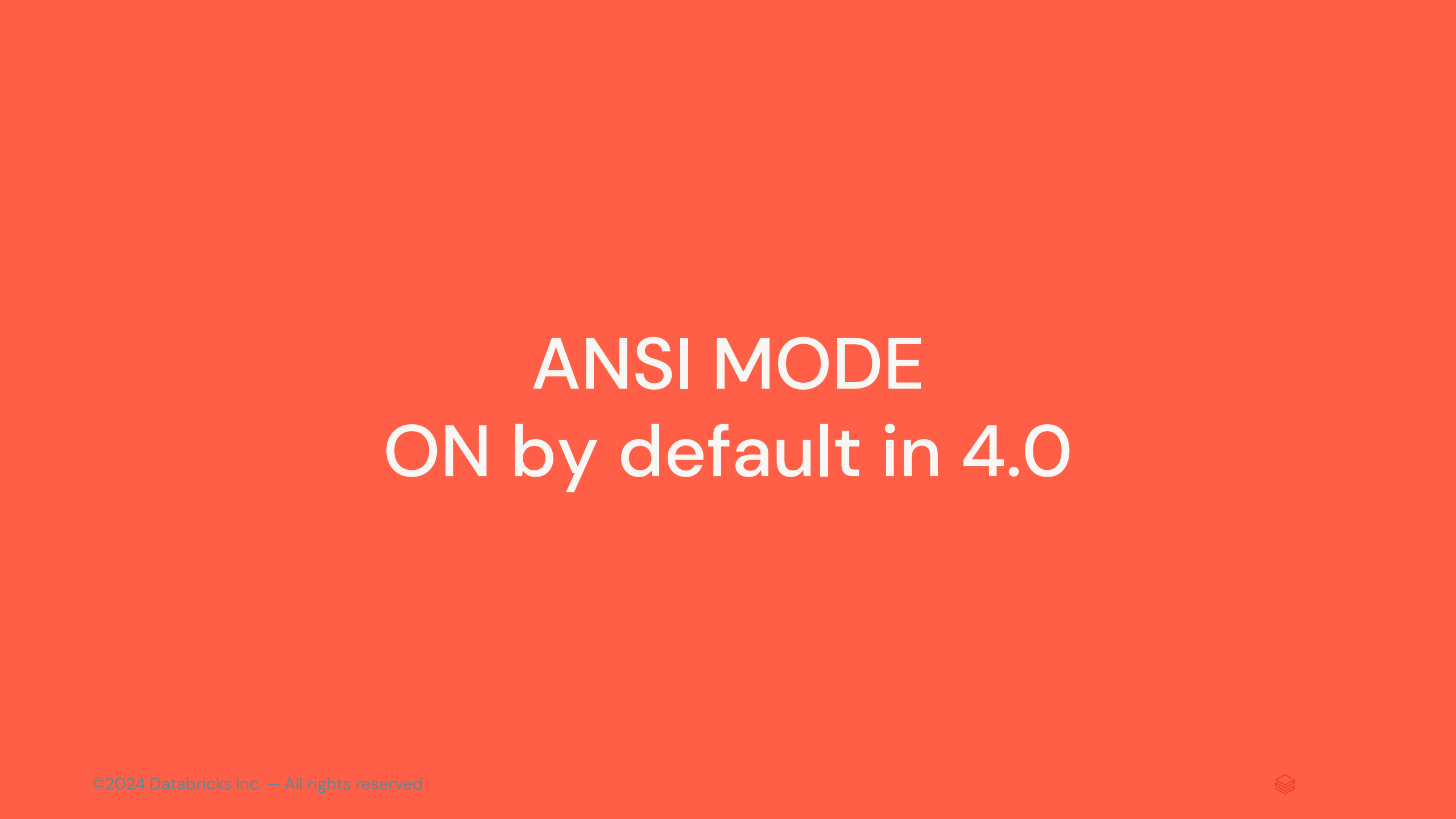


Get started with our github example!

The lightweight Spark Connect Package

- `pip install pyspark-connect`
  - Pure Python library, no JVM.
  - Pure Spark Connect client, not entire PySpark
  - Only 1.5MB (PySpark 355 MB)
  - Preferred if your application has fully migrated to the Spark Connect API.

ANSI MODE
ON by default in 4.0



Migration to ANSI ON

Action: Turn On ANSI Mode to fix your data corruptions!

ANSI Compliance

In Spark SQL, there are two options to comply with the SQL standard: `spark.sql.ansi.enabled` and `spark.sql.storeAssignmentPolicy` (See a table below for details).

When `spark.sql.ansi.enabled` is set to `true`, Spark SQL uses an ANSI compliant dialect instead of being Hive compliant. For example, Spark will throw an exception at runtime instead of returning null results if the inputs to a SQL operator/function are invalid. Some ANSI dialect features may be not from the ANSI SQL standard directly, but their behaviors align with ANSI SQL's style.

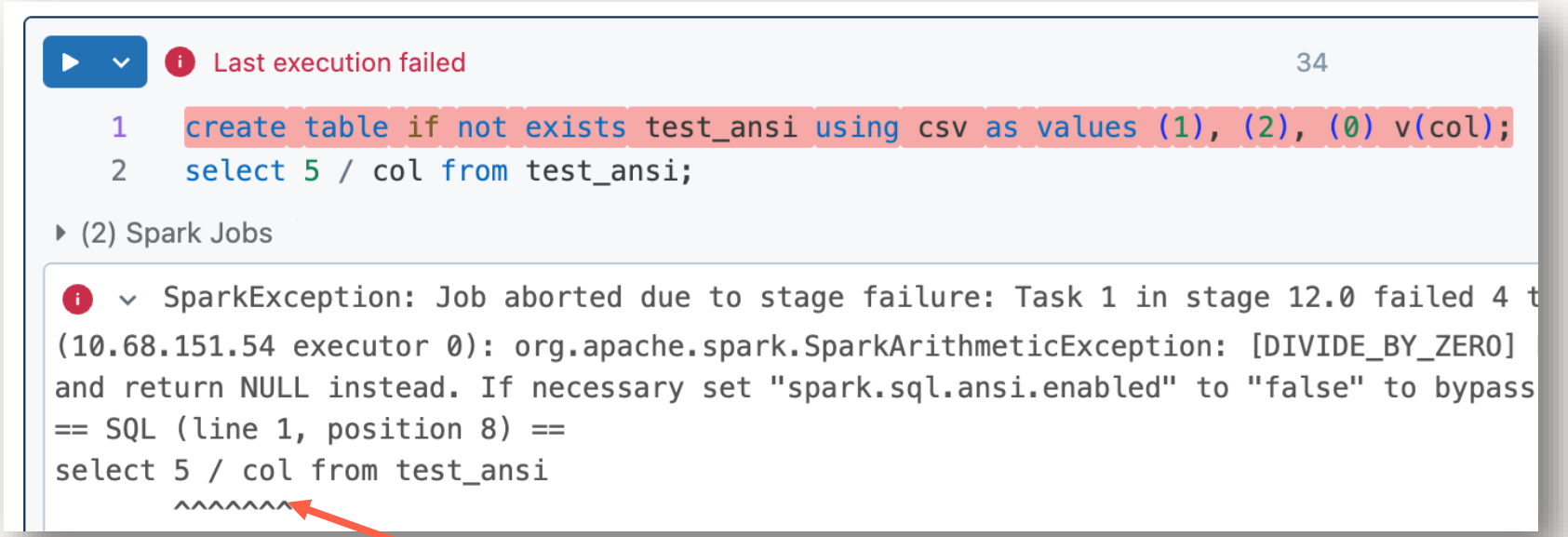
Moreover, Spark SQL has an independent option to control implicit casting behaviours when inserting rows in a table. The casting behaviours are defined as store assignment rules in the standard.

When `spark.sql.storeAssignmentPolicy` is set to `ANSI`, Spark SQL complies with the ANSI store assignment rules. This is a separate configuration because its default value is `ANSI`, while the configuration `spark.sql.ansi.enabled` is disabled by default.

Property Name	Default	Meaning	Since Version
`spark.sql.ansi.enabled`	false	When true, Spark tries to conform to the ANSI SQL specification: 1. Spark SQL will throw runtime exceptions on invalid operations, including integer overflow errors, string parsing errors, etc. 2. Spark will use different type coercion rules for resolving conflicts among data types. The rules are consistently based on data type precedence.	3.0.0
		When inserting a value into a column with different data type, Spark will perform type conversion. Currently, we support 3 policies for the type coercion rules: ANSI, legacy and strict. 1. With ANSI policy, Spark performs the type coercion as per ANSI SQL. In practice, the behavior is mostly the same as PostgreSQL. It disallows certain unreasonable type conversions such as	

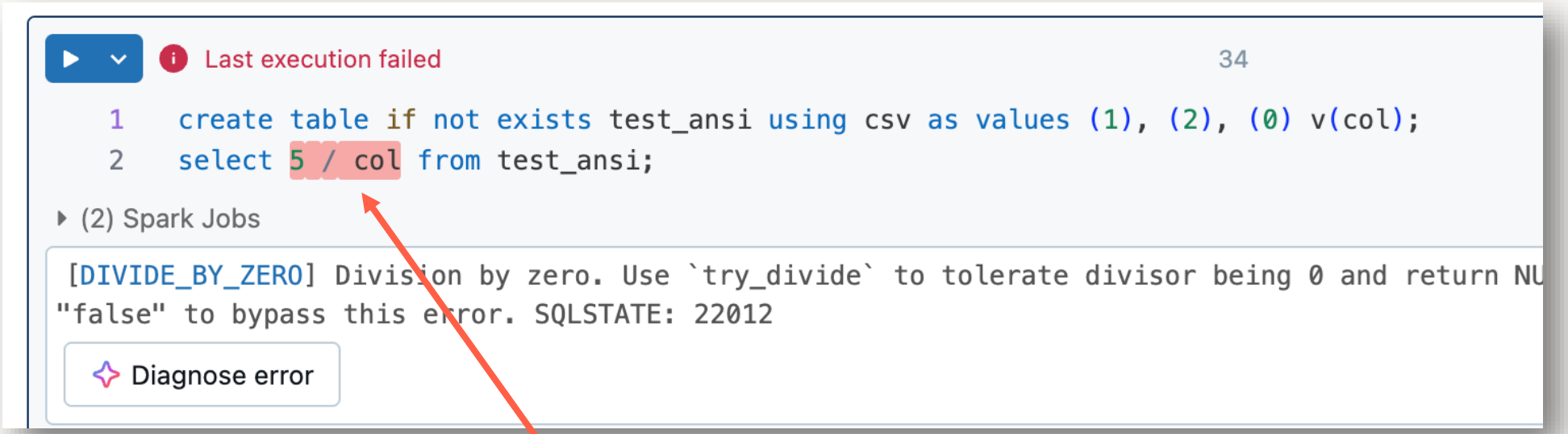
Without ANSI mode

With ANSI mode (3.5)



Error callsite is captured

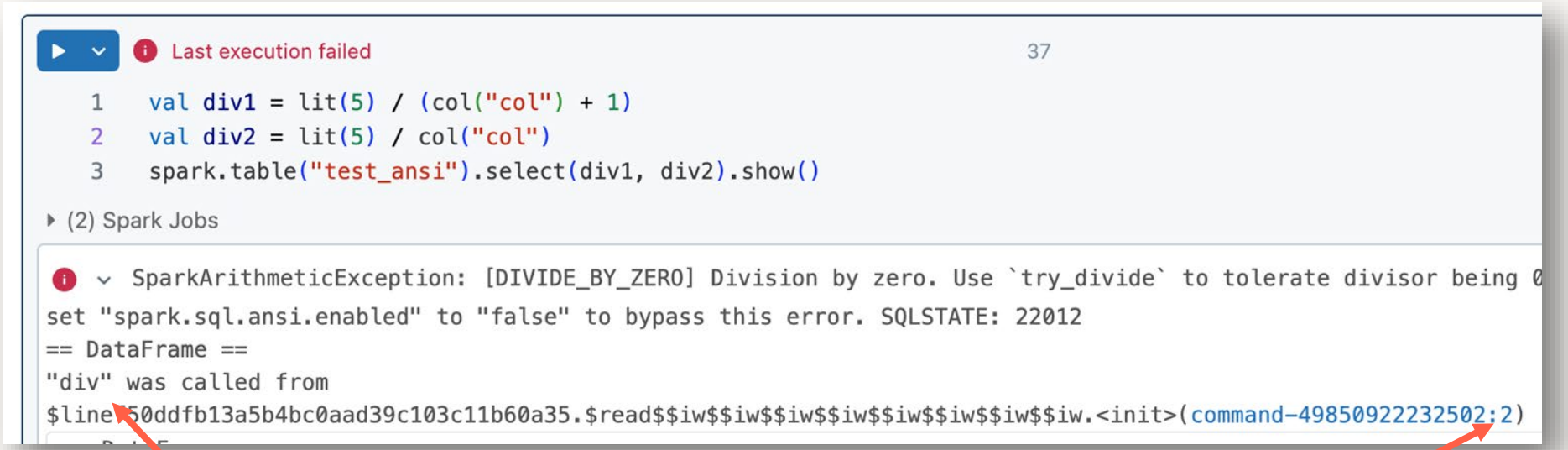
With ANSI mode (4.0)



Error callsite is highlighted

With ANSI mode (4.0)

DataFrame queries with error callsite



Culprit operation

Line number

DataFrame queries with error callsite

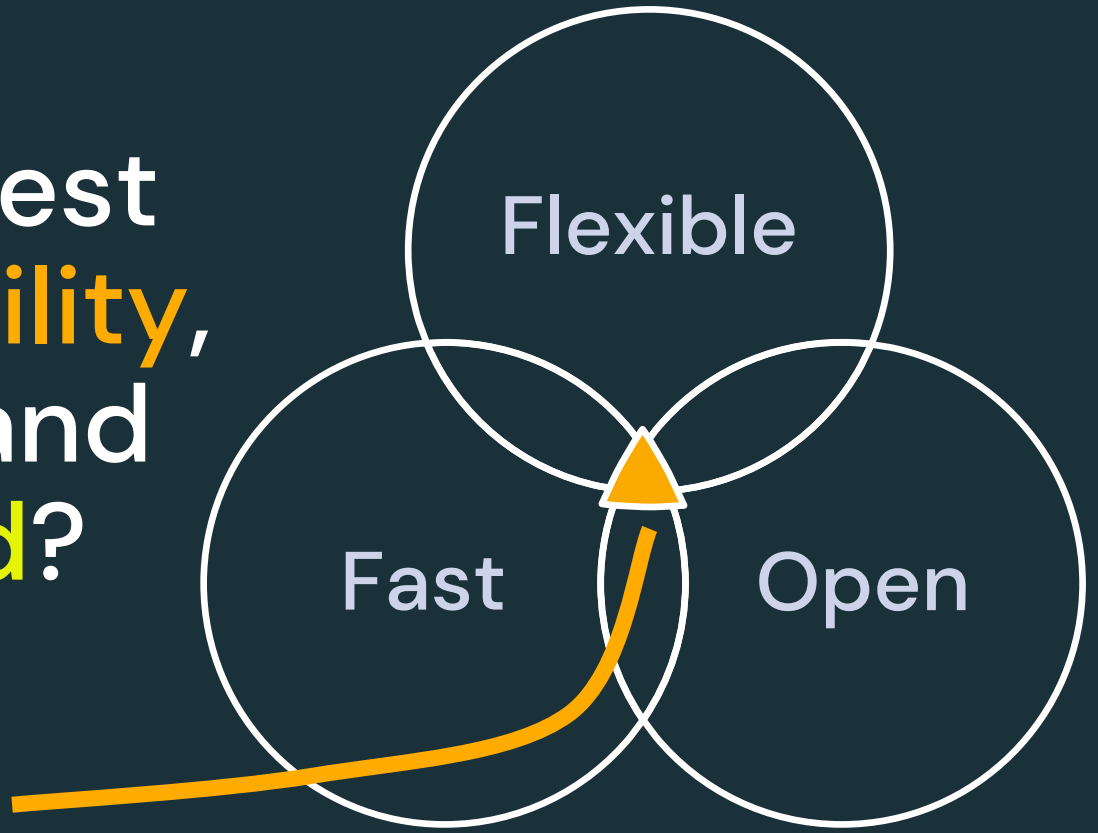
- PySpark support is on the way.
- Spark Connect support is on the way.
- Native notebook integration is on the way (so that you can see highlight).

Variant Data Type for Semi-Structured Data

Motivation

Data Engineer's Dilemma: Only pick 2 out of 3...

What if you could ingest JSON, maintain flexibility, boost performance, and use an open standard?

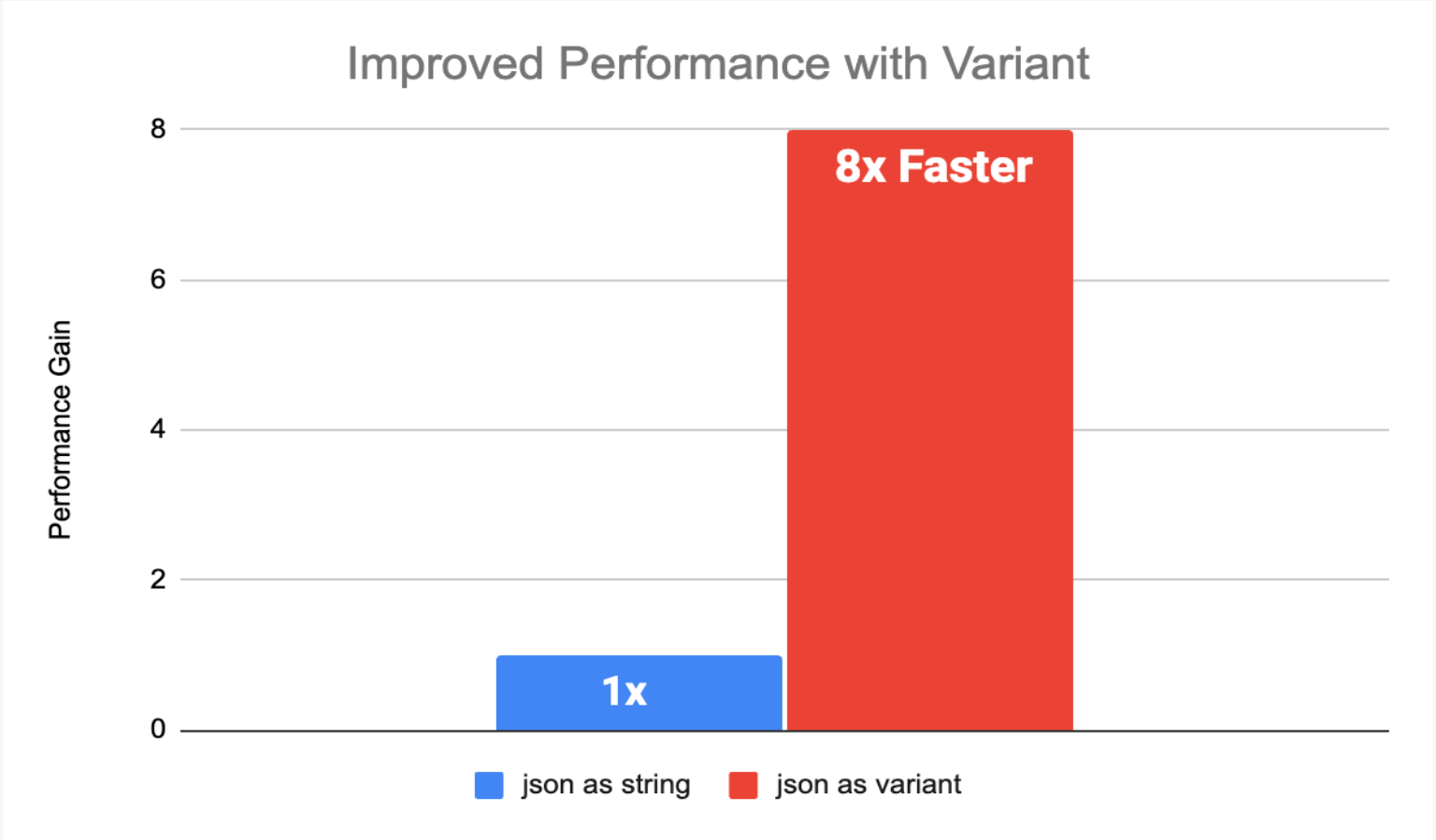


Variant is flexible

SQL

```sql
INSERT INTO variant_tbl (event_data)
VALUES
  (
    PARSE_JSON(
      '{"level": "warning",
        "message": "invalid request",
        "user_agent": "Mozilla/5.0 ..."}'
    )
  );

SELECT
  *
FROM
  variant_tbl
WHERE
  event_data:user_agent ilike '%Mozilla%';
```


Performance



String Collation Support

ANSI SQL COLLATE

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
- GROUP BY, ORDER BY, comparisons, …
- Supported by Delta and Photon

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
```

SQL

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
```

SQL

Case insensitive is not accent insensitive: We lost Ānthōnī

COLLATE UNICODE_CI_AI

Equality from a to Ź

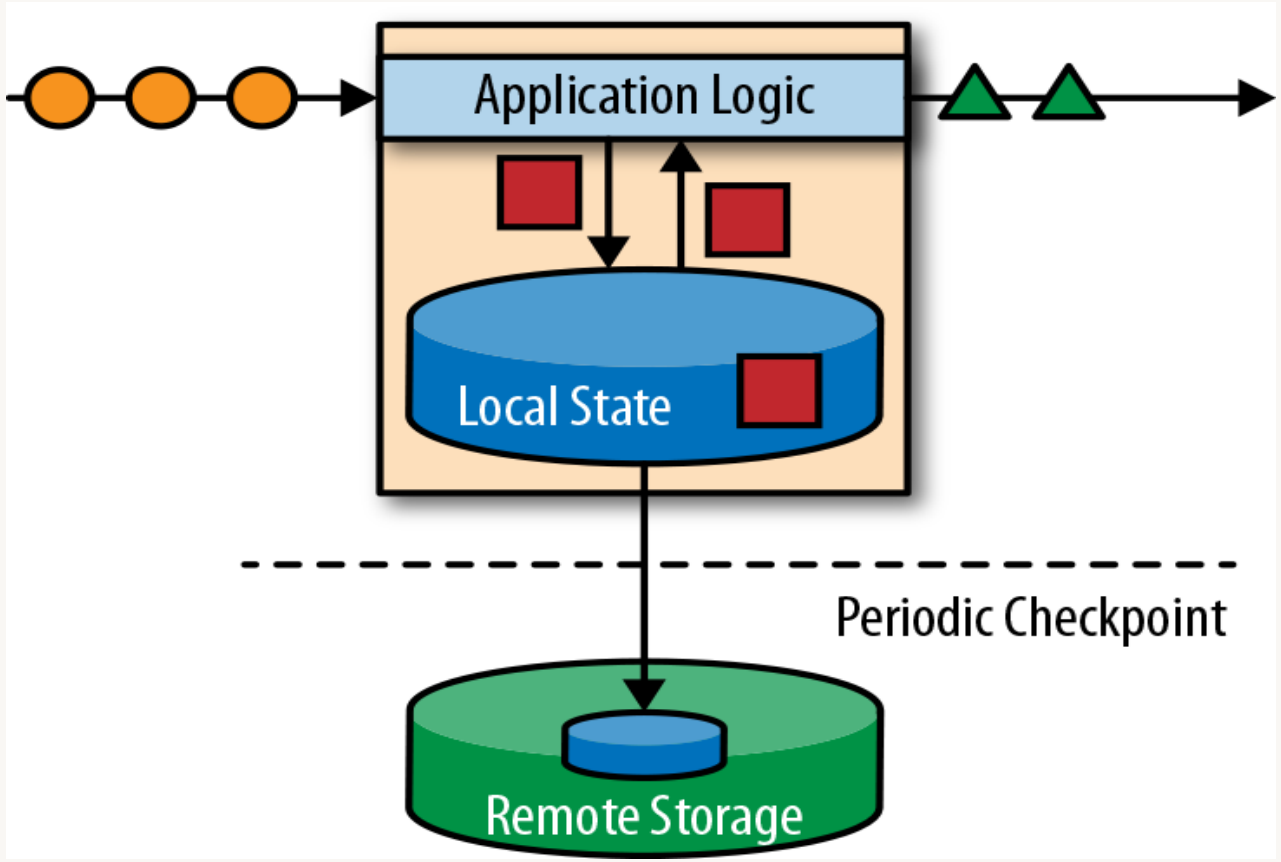
```sql
> SELECT name
FROM names
WHERE startswith(name COLLATE unicode_ci_ai, 'a')
ORDER BY name COLLATE unicode_ci_ai;

name
Ānthōnī
anthony
Anthony
```

SQL

100s of supported predefined collations across many locales

Streaming State Data Source

Stateful Stream Processing



Streaming State data source

- Allows you to inspect the internal states of streaming applications, for debugging, profiling, testing, troubleshooting, etc.
- Allows you to manipulate the internal states for quick workaround to recover urgent issues.
- All with your familiar data source APIs.

State Reader API: state-metadata

High-level statestore info

```
display(spark.read.format("state-metadata").load(ad_clicks_checkpoint))
```

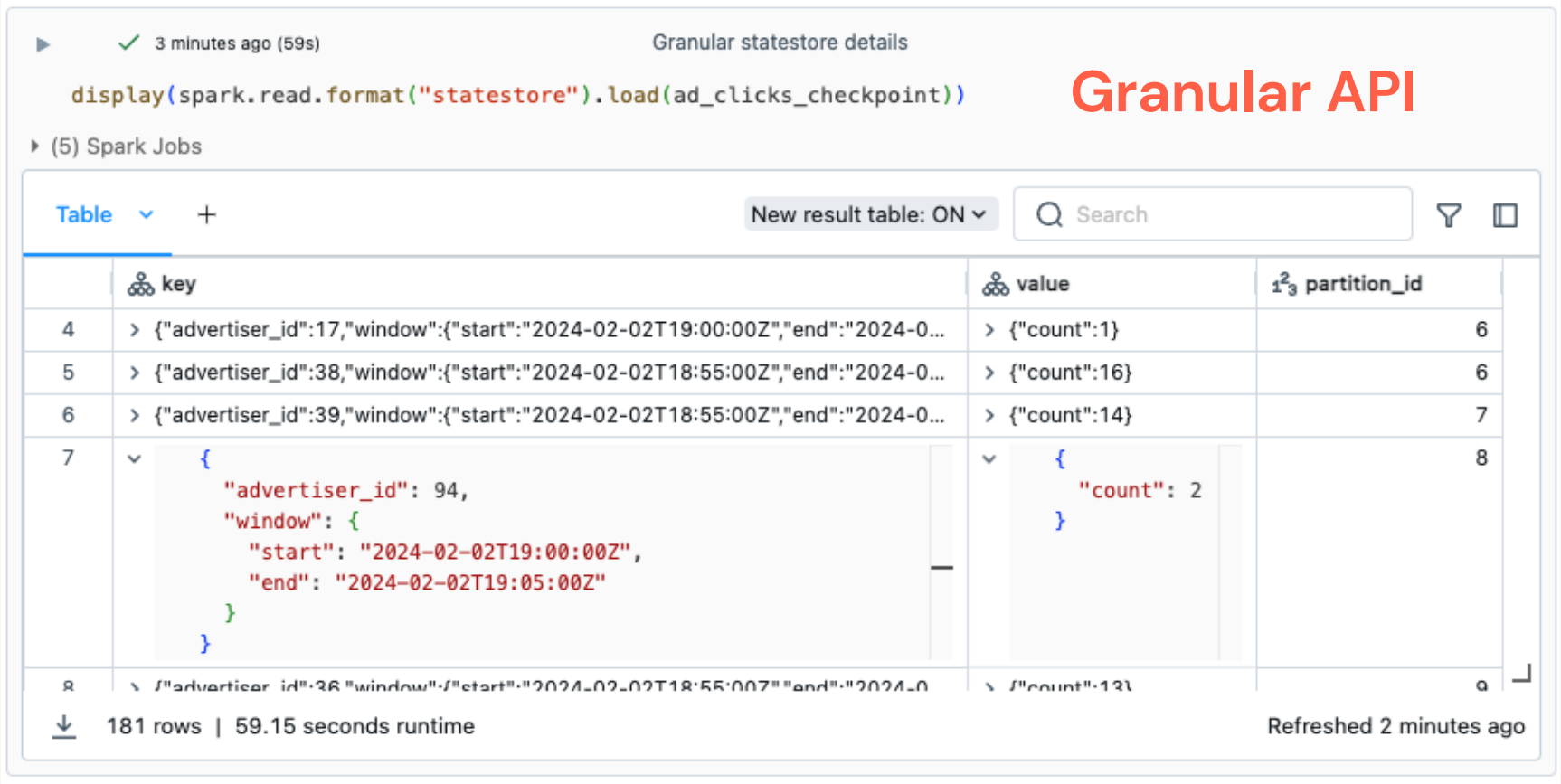
High-level API

Table + New result table: ON Search

	operatorId	operatorName	stateStoreName	numPartitions	minBatchId	maxBatchId
1	0	stateStoreSave	default	200	0	13
2	1	dedupeWithinWatermark	default	200	0	13

2 rows

State Reader API: statestore



Arbitrary Stateful Processing V2

(flat)MapGroupsWithState: current V1 version

- Supports a single user defined state object per grouping key
- State object can be updated while evaluating the current group, and updated value will be available in next trigger.

```scala
val ds = spark.readStream.json(path)
  .as[CreditCardTransaction]

ds.groupByKey(_.cardId)
  .flatMapGroupsWithState[
    CreditCardTransactionState,
    CreditCardTransaction
  ](
    OutputMode.Append(),
    GroupStateTimeout.NoTimeout())
  (
    (_, txns, groupState) => {
      // read state, compute new average, and save to state
      ...
  }
)
```

Scala

Limitations of (flat)MapGroupsWithState

Lack of Data Modelling Flexibility

Prevents users from splitting state (for a grouping key) into multiple logical instances, which can be read/updated independently.

Lack of Composite Types

Values stored in GroupState are single types and cannot support data structures like List, Map etc efficiently. Current approach requires users to read/update the entire data structure.

Lack of State Eviction Support

No support for eventual state cleanup using TTL

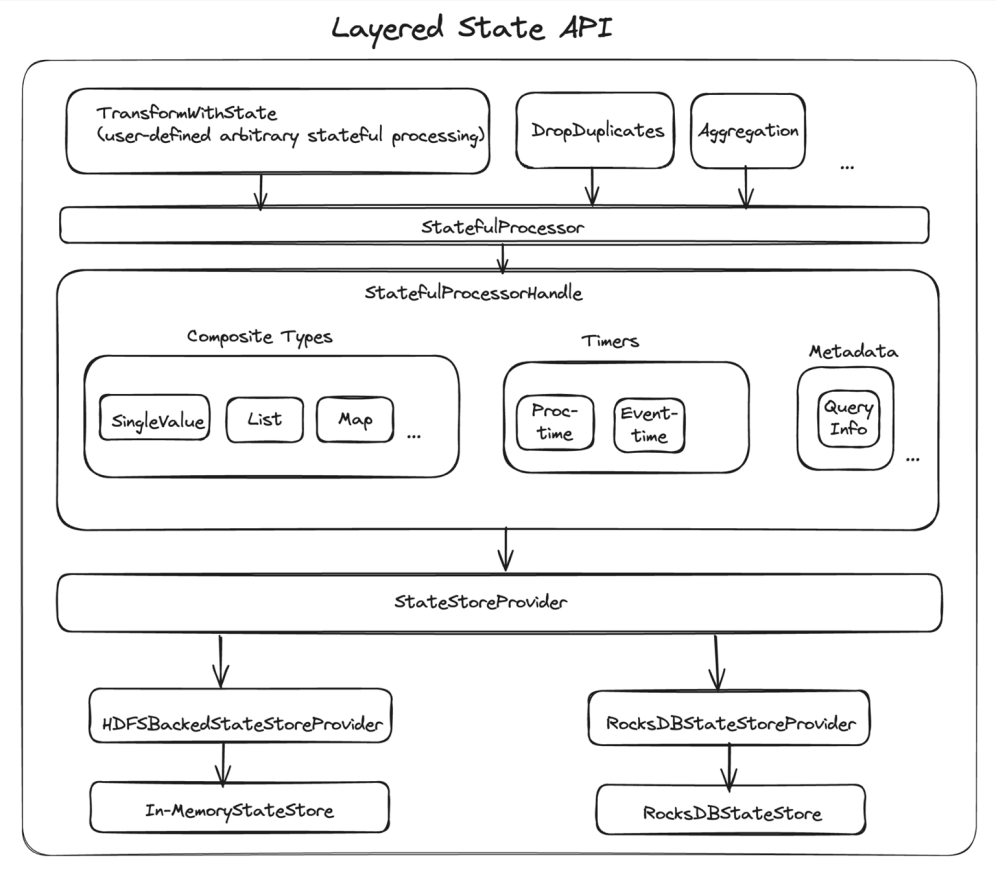
Lack of State Schema Evolution

Does not support changes to state schema once the streaming query has started.

transformWithState: the V2 version

Layered, Flexible,

Extensible State API



pandas 2 support/API parity

Pandas API on Spark (Koalas)

pandas

```python
import pandas as pd
df = pd.read_csv("my_data.csv")
df.columns = ['x', 'y', 'z1']
df['x2'] = df.x * df.x
```

Python

Pandas API on Spark

```python
import pyspark.pandas as ps
df = ps.read_csv("my_data.csv")
df.columns = ['x', 'y', 'z1']
df['x2'] = df.x * df.x
```

Python

Pandas 2.x Support

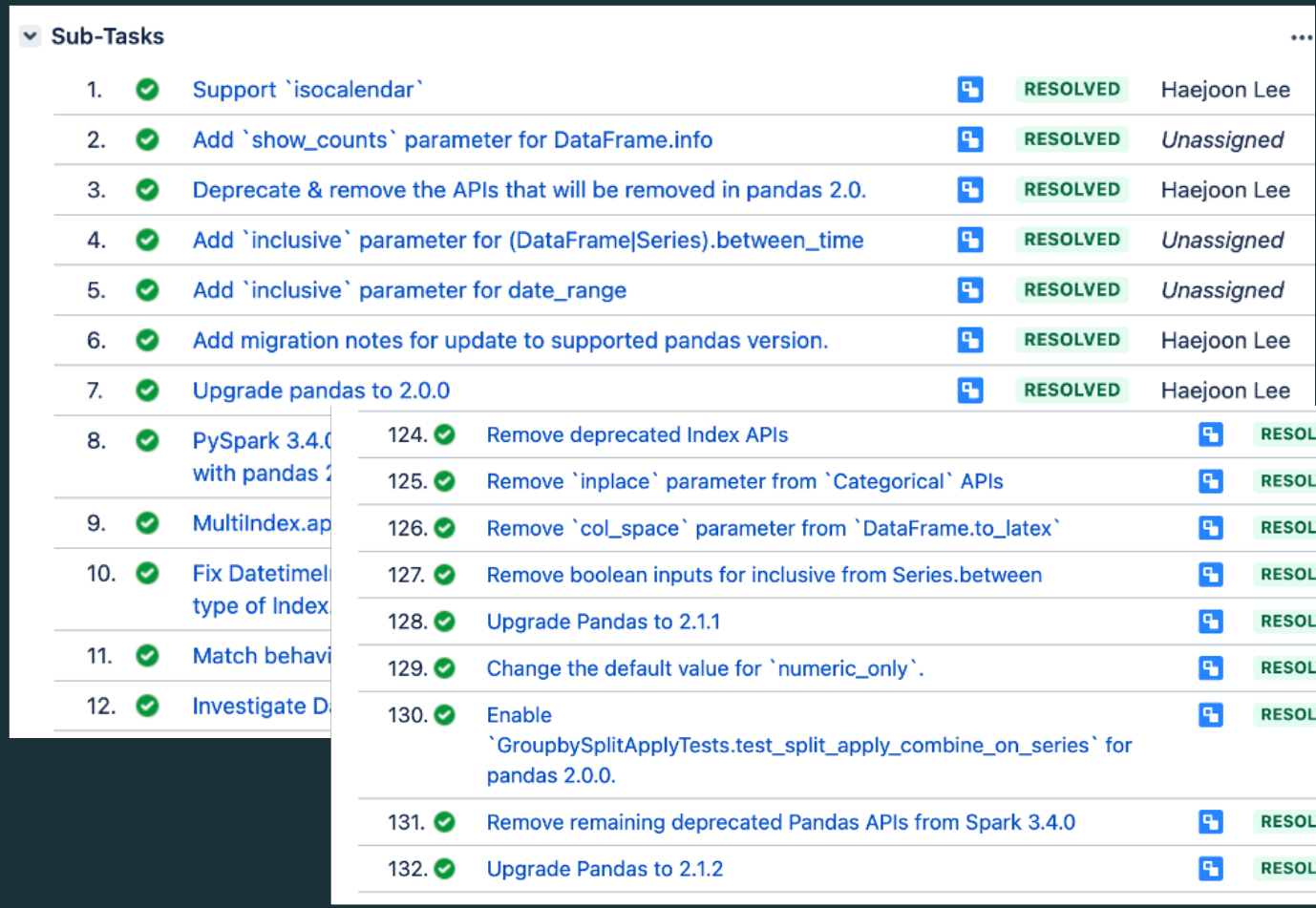
API change parity with Pandas 2.2.2

Backwards incompatible API changes

- Pandas 2.0.0
- Pandas 2.1.0

SPARK-44101

Spark migration guide



Agenda

New Functionalities

Spark Connect, ANSI Mode, Arbitrary Stateful Processing V2, Collation Support, Variant Data Types, pandas 2.x Support

Extensions

Python Data Source APIs, XML/Databricks Connectors and DSV2 Extension, Delta 4.0

Custom Functions and Procedures

SQL UDFs, SQL Scripting, Python UDTF, Arrow optimized Python UDF, PySpark UDF Unified Profiler

Usability

Structured Logging Framework, Error Class Framework, Behavior Change Process

Streaming and Batching Python Data Sources

Why Python Data Source?

- People like writing Python!
- `pip install` is so convenient.
- Simplified API without complicated performance features in Data Source V2.

Spark 💔 Python

Python Data Source APIs

- SPIP: Python Data Source API SPARK-44076
- Available in Spark 4.0 preview version and Databricks Runtime 15.2
- Support both batch and streaming, read and write

master spark / python / pyspark / sql / datasource.py ↑ Top

Code Blame 815 lines (652 loc) · 27.2 KB Raw

```
class DataSource(ABC):
    """
    A base class for data sources.

    This class represents a custom data source that allows for reading from and/or
    writing to it. The data source provides methods to create readers and writers
    for reading and writing data, respectively. At least one of the methods
```

Python Data Source Overview

Easy Three Steps to create and use your custom data sources

Step 1: Create a Data Source

```
class MySource(DataSource):
    …
```

Step 2: Register the Data Source

Register the data source in the current Spark session using the Python data source class:

```
spark
.dataSource
.register(MySource)
```

Step 3: Read from or write to the data source

```
spark.read
.format("my-source")
.load(...)


df.write
.format("my-source")
.mode("append")
.save(...)
```

DataFrame.toArrow
GroupedData.applyInArrow

DataFrame.toArrow

- An simply API to convert PySpark DataFrame to PyArrow Table.
- Make it easier to integrate with Arrow ecosystem.
- Note, all the data is loaded into the driver's memory. It may cause out-of-memory errors for large data

```
>>> df.toArrow()
pyarrow.Table
age: int64
name: string
----
age: [[2,5]]
name: [["Alice","Bob"]]
```


GroupedData.applyInArrow

- Utilizes Apache Arrow to map functions over DataFrame groups
- Returns the result as a DataFrame
- Supports functions taking pyarrow.Table or tuple of grouping keys and pyarrow.Table
- Note: Function requires a full shuffle and may cause out-of-memory errors for large data groups

```
from pyspark.sql.functions import ceil
>>> import pyarrow
>>> import pyarrow.compute as pc
>>> df = spark.createDataFrame(
...     [(1, 1.0), (1, 2.0), (2, 3.0), (2, 5.0), (2, 10.0)],
...     ("id", "v"))
>>> def normalize(table):
...     v = table.column("v")
...     norm = pc.divide(pc.subtract(v, pc.mean(v)),
pc.stddev(v, ddof=1))
...     return table.set_column(1, "v", norm)
>>> df.groupby("id").applyInArrow(
...     normalize, schema="id long, v double").show()
+---+-------------------+
| id|                  v|
+---+-------------------+
|  1|-0.7071067811865475|
|  1| 0.7071067811865475|
|  2|-0.8320502943378437|
|  2|-0.2773500981126146|
|  2| 1.1094003924504583|
+---+-------------------+
```


XML Connectors

Reading XML files out of the box

```python
spark.read.xml("/path/to/my/file.xml").show()
```

```
+-------+-----+
| name  | age |
+-------+-----+
| Alice | 23  |
| Bob   | 32  |
```

Python

More Than Just a Simple Port

Sub-Tasks

1.	Port the initial implementation of Spark XM		
2.	XML: Implement FIleFormat Interface		
3.	XML: Update Spark Docs		
4.	XML: Add Python and sparkR binding including S		
5.	XML: Add SQL Expressions		RESOLVED
6.	XML: Add pyspark.sql.functions		RESOLVED
7.	XML: Spark connect support		RESOLVED
8.	XML: to_xml		RESOLVED
9.	XML: ArrayType and MapType support in from_xml		RESOLVED
16.	XML: Add DecimalType support in schema inference		RESOL
17.	XML: Add TimestampNTZType support		RESOLVED
18.	XML: Close InputStreamReader on read completion		RESOLVED
19.	XML: Fix XSD big integer conversion		RESOLVED
20.	XML: Use TypeCoercion.findTightestCommonType for compatibility check		RESOLVED
21.	XML: Validate XML element name on write		RESOLVED
22.	XML: Throw error on multiple XML data source		RESOLVED
23.	XML: Limit size of corrupt record		RESOLVED
	XML: Perf optimizations		RESOLVED

Databricks JDBC Dialect

Reading Databricks SQL out of the box

```python
spark.read.jdbc(
    "jdbc:databricks://…",
    "my_table",
    properties
).show()
```

```
+-------+-----+
| name  | age |
+-------+-----+
| Alice | 23  |
| Bob   | 32  |
```

Python

Delta Lake 4.0

The biggest Delta release yet

Deletion Vectors

Liquid clustering

Delta Kernel

Optimized Writes

Table features

Incremental checkpoints

Log compactions

Row IDs

Table cloning

MERGE improvements

CDF

Auto-compaction

VARIANT

Lightning fast semi-structured data

UniForm GA

Write once, read as all formats

Liquid GA

Easy migration from partitioned tables

Multi-cluster writes

Cross cloud, cross engines

Delta Connect

Spark Connect support

Collations

Flexible sort and comparison

Identity columns

Pain-free primary and foreign keys

Type widening

Data types expand with your data

UniForm GA

Towards full lakehouse format Interoperability

Delta Lake
With UniForm

Metadata

Metadata

Data

Parquet

Liquid clustering Usage Walkthrough

Create a new Delta table with liquid clustering

```
CREATE [EXTERNAL] TABLE tbl (id INT, name STRING) CLUSTER BY(id)
```

Change Liquid Clustering keys on existing clustered table:

```
ALTER TABLE tbl CLUSTER BY (name);
```

Clustering data in a Delta table with liquid clustering:

```
OPTIMIZE tbl;
```

What you don't need to worry about:

- Optimal file sizes
- Whether a column can be used as a clustering key
- Order of clustering keys

Public doc: https://docs.databricks.com/delta/clustering.html

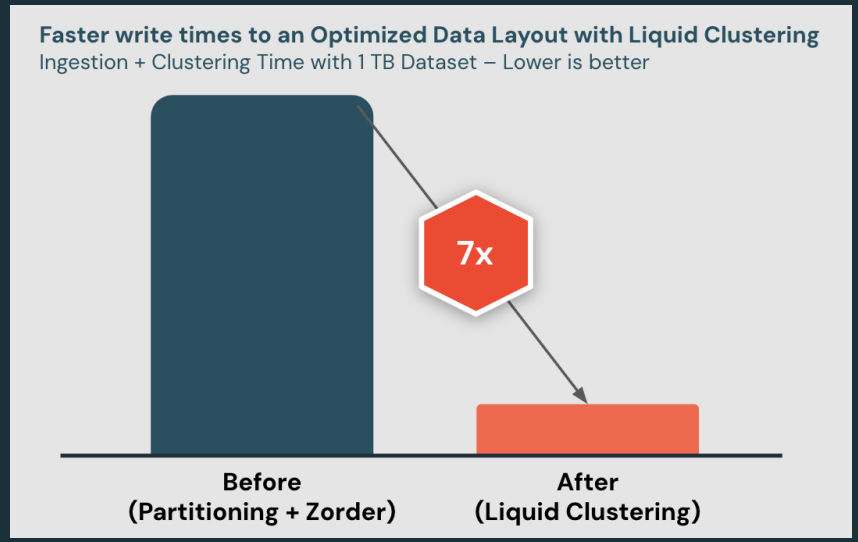
Liquid Clustering GA

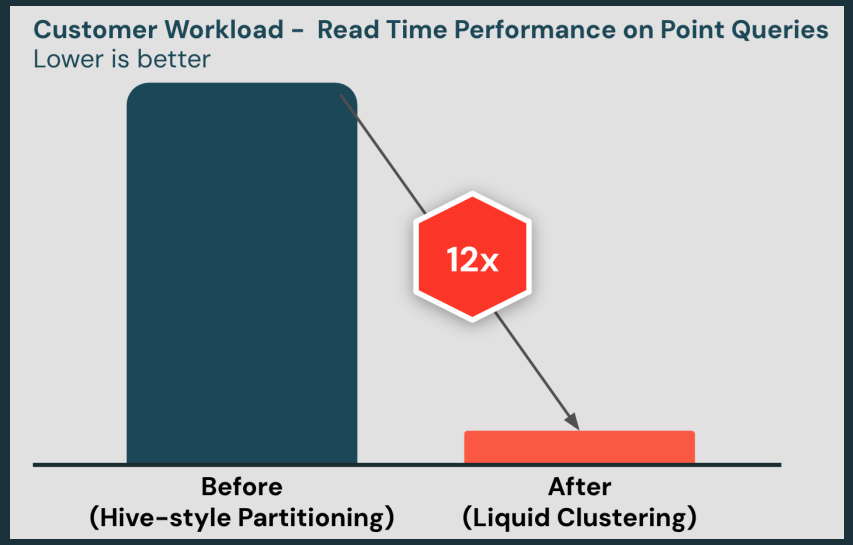
Easy to use

Up to 7x faster writes

Up to 12x faster reads

Highly flexible





Agenda

New Functionalities

Spark Connect, ANSI Mode, Arbitrary Stateful Processing V2, Collation Support, Variant Data Types, pandas 2.x Support

Extensions

Python Data Source APIs, XML/Databricks Connectors and DSV2 Extension, Delta 4.0

Custom Functions and Procedures

SQL UDFs, SQL Scripting, Python UDTF, Arrow optimized Python UDF, PySpark UDF Unified Profiler

Usability

Structured Logging Framework, Error Class Framework, Behavior Change Process

Python UDTF

Python User Defined Table Functions

This is a new kind of function that returns an *entire table* as output instead of a single scalar result value

- Once registered, they can appear in the FROM clause of a SQL query
- Or use the DataFrame API to call them

```python
from pyspark.sql.functions import udtf

@udtf(returnType="num: int, squared: int")
class SquareNumbers:
    def eval(self, start: int, end: int):
        for num in range(start, end + 1):
            yield (num, num * num)
```

Python

Python User Defined Table Functions

```
SELECT * FROM SquareNumbers(
  num => 1, squared => 3);

+-----+--------+
| num | squared|
+-----+--------+
|   1 |       1|
|   2 |       4|
|   3 |       9|
+-----+--------+
```

SQL

```
SquareNumbers(lit(1), lit(3)).show()

+-----+--------+
| num | squared|
+-----+--------+
|   1 |       1|
|   2 |       4|
|   3 |       9|
+-----+--------+
```

Python

SQL Lang

PySpark DataFrame APIs

Python UDTF

Polymorphic Analysis

Compute the output schema for each call depending on arguments, using `analyze`

```
class ReadFromConfigFile:
    @staticmethod
    def analyze(filename: AnalyzeArgument):
        with open(os.path.join(
            SparkFiles.getRootDirectory(),
            filename.value), "r") as f:
        # Compute the UDTF output schema
        # based on the contents of the file.
        return AnalyzeResult(
            from_file(f.read()))
    ...

ReadFromConfigFile(lit("config.txt")).show()

+------------+-------------+
| start_date | other_field |
+------------+-------------+
| 2024-04-02 |           1 |
+------------+-------------+
```

Python

Input Table Partitioning

Split input rows among class instances: `eval` runs once each row, then `terminate` runs last

```
class CountAndMax:
    def __init__(self):
        self._count = 0
        self._max = 0
    def eval(self, row: Row):
        self._count += 1
        self._max = max(self._max, row[0])
    def terminate(self):
        yield self._count, self._max

WITH t AS (SELECT id FROM RANGE(0, 100))
SELECT * FROM CountAndMax(
TABLE(t) PARTITION BY id / 10 ORDER BY id);

+-------+-----+
| count | max |
+-------+-----+
|    10 |   0 |
|    10 |   1 |
...
```

Python

Python UDTF

Variable Keyword Arguments

The `analyze` and `eval` methods may accept `*args` or `**kwargs`

```python
class VarArgs:
    @staticmethod
    def analyze(**kwargs: AnalyzeArgument):
        return AnalyzeResult(StructType(
            [StructField(key, arg.dataType)
            for key, arg in sorted(
                kwargs.items())]))
    def eval(self, **kwargs):
        yield tuple(value for _, value
            in sorted(kwargs.items()))

SELECT * FROM VarArgs(a => 10, b => 'x');

+----+-----+
| a  |   b |
+----+-----+
| 10 | "x" |
+----+-----+
```

Python

Custom Initialization

Create a subclass of `AnalyzeResult` and consume it in each subsequent `__init__`

```python
class SplitWords:
    @dataclass
    class MyAnalyzeResult(AnalyzeResult):
        numWords: int
        numArticles: int

    @staticmethod
    def analyze(text: str):
        words = text.split(" ")
        return MyAnalyzeResult(
            schema=StructType()
                .add("word", StringType())
                .add("total", IntegerType()),
            withSinglePartition=true,
            numWords=len(words)
            numArticles=len((
                word for word in words
                if word in ("a", "an", "the")))

    def __init__(self, r: MyAnalyzeResult):
    ...
```

Python

Arrow Optimized Python UDF

Enhancing Python UDFs with Apache Arrow

- Introduction to Arrow and Its Role in UDF Optimization:
  - Utilizes Apache Arrow
  - Supported since Spark 3.5 and **ON by default** since Spark 4.0
- Key Benefits
  - Enhances data serialization and deserialization speed
  - Provides standardized type coercion

Enabling Arrow Optimization

Local Activation in a UDF

- Activates Arrow optimization for a specific UDF, improving performance

```python
# An Arrow Python UDF
@udf(returnType='int', useArrow=True)
def arrow_slen(s):
    return len(s)
```

Python

Global Activation in a UDF

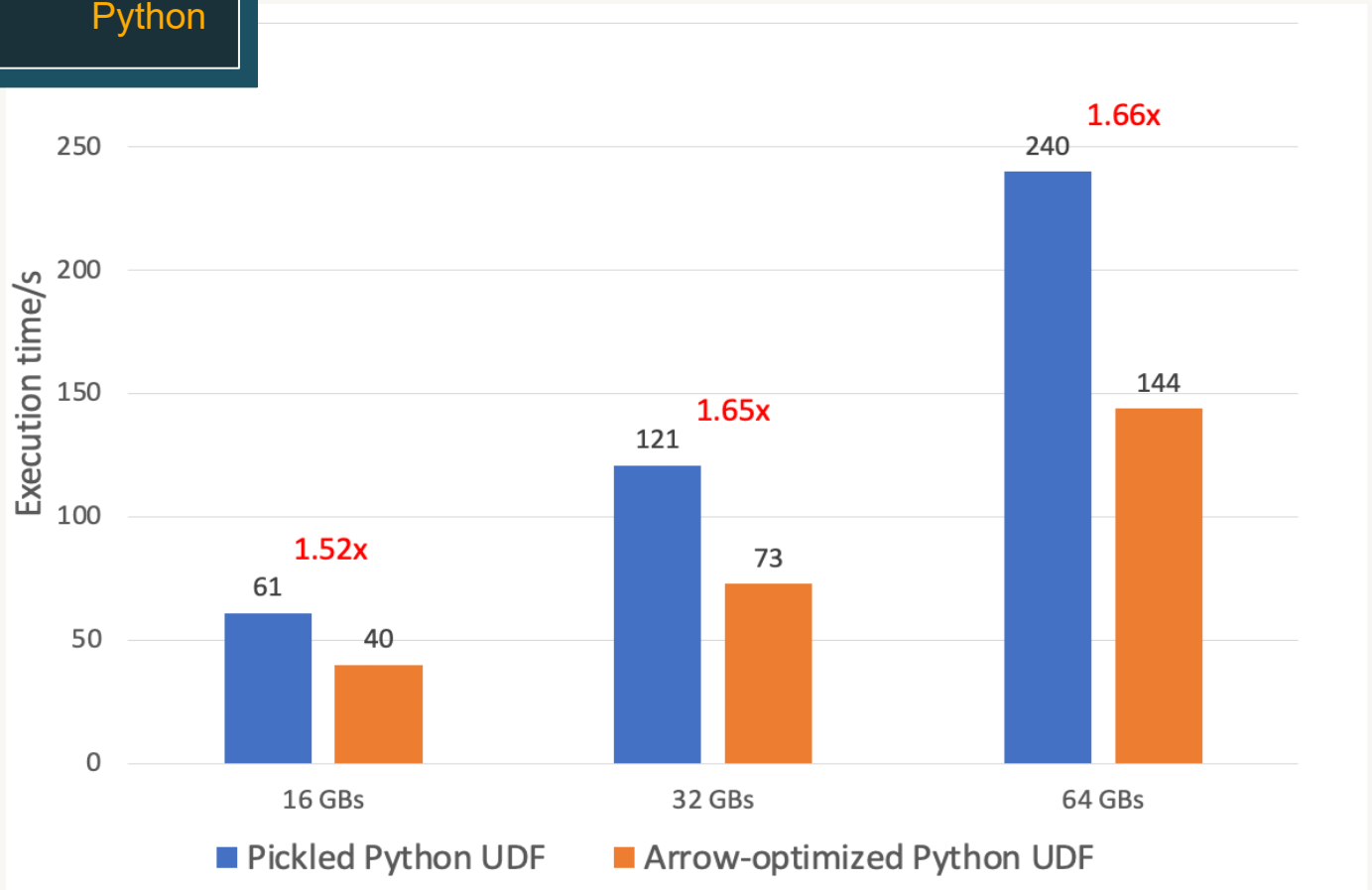
- Activates Arrow optimization for all Python UDFs in the Spark session

```python
spark.conf.set("spark.sql.execution.pythonUDF.arrow.enabled", True)

# An Arrow Python UDF
@udf(returnType='int')
def arrow_slen(s):
    return len(s)
```

Python

```python
@udf(returnType='int', useArrow=True)
def arrow_slen(s):
    return len(s)
```

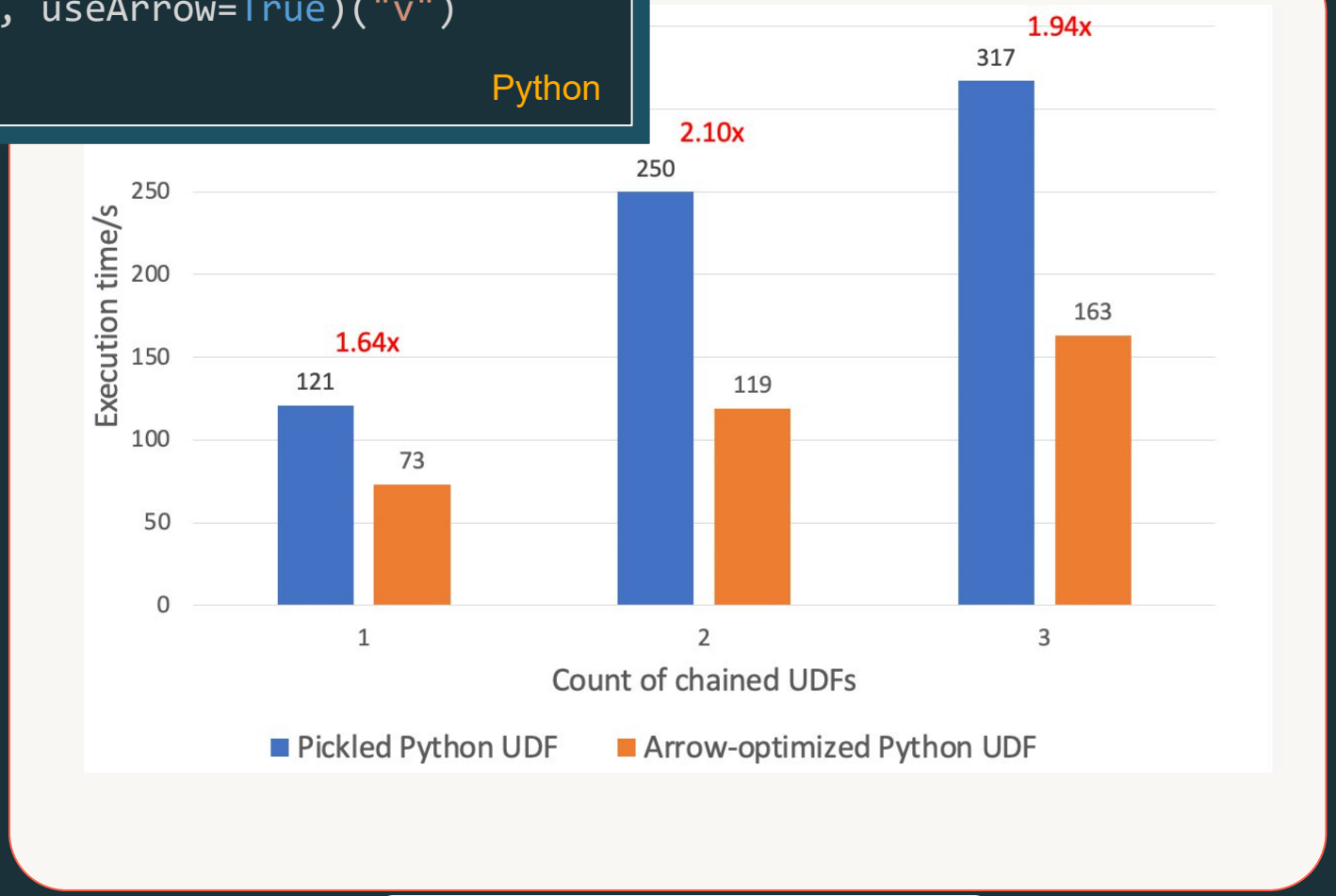
Python



Performance

```python
sdf.select(
    udf(lambda v: v + 1, DoubleType(), useArrow=True)("v"),
    udf(lambda v: v - 1, DoubleType(), useArrow=True)("v"),
    udf(lambda v: v * v, DoubleType(), useArrow=True)("v")
)
```

Python



Performance

Pickled Python UDF

```
>>> df.select(udf(lambda x: x, 'string')('value').alias('date_in_string')).show()
+--------------------------------------------------------------------+
|                                                      date_in_string |
+--------------------------------------------------------------------+
|java.util.GregorianCalendar[time=?,areFieldsSet=false,areAllFieldsSet..|
|java.util.GregorianCalendar[time=?,areFieldsSet=false,areAllFieldsSet..|
+--------------------------------------------------------------------+
```

Python

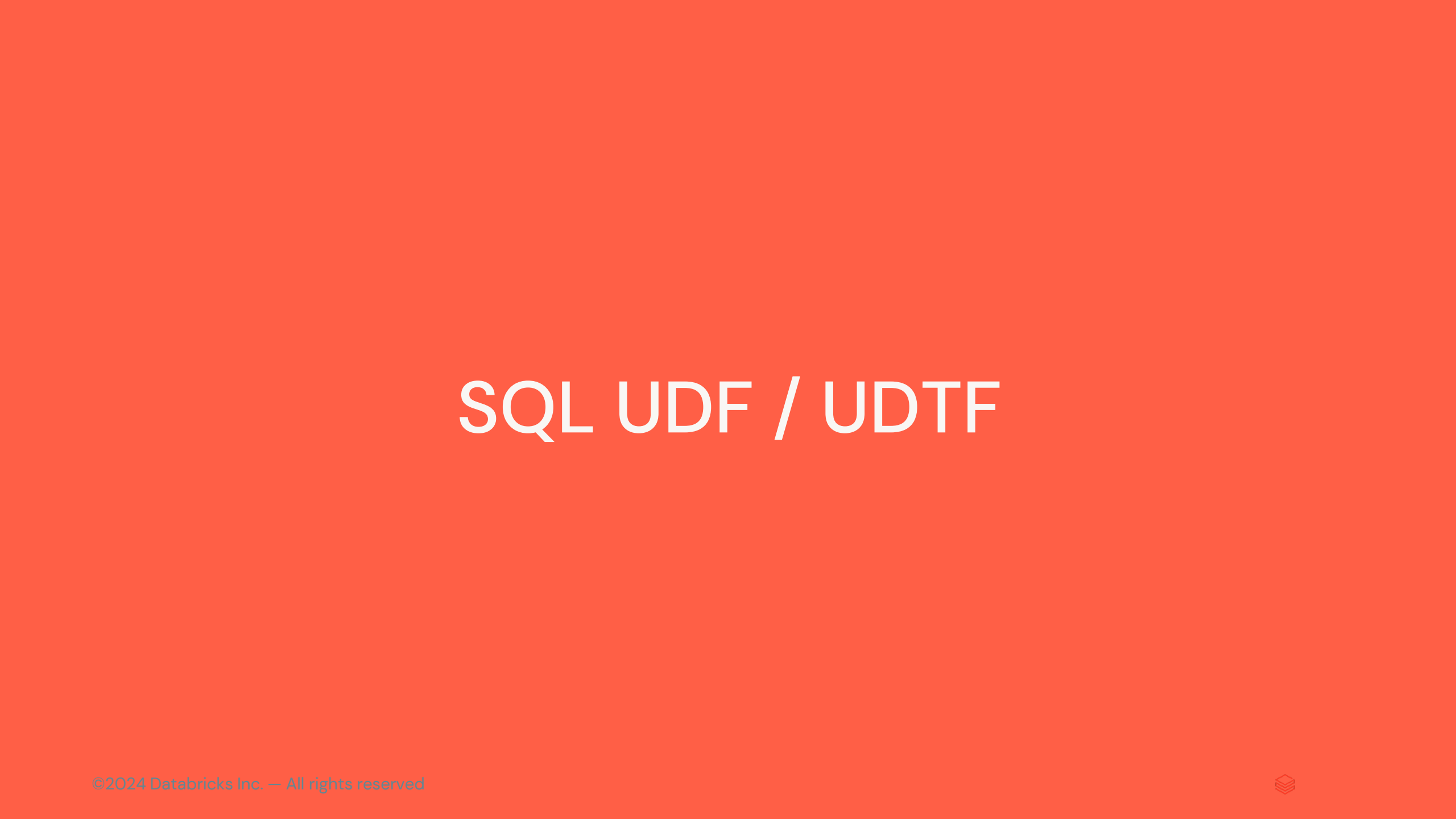
Arrow-optimized Python UDF

```
>>> df.select(udf(lambda x: x, 'string')('value').alias('date_in_string')).show()
+--------------+
|date_in_string|
+--------------+
|    1970-01-01|
|    1970-01-02|
+--------------+
```

Comparing Pickled and Arrow-optimized Python UDFs on type coercion [Link]

Python

SQL UDF / UDTF



Easily extend SQL function library

- SQL User Defined Scalar Functions
  - Persisted SQL Expressions
- SQL User Defined Table Functions
  - Persisted Parameterized Views
- Support named parameter invocation and defaulting
- Table functions with lateral correlation

SQL User Defined Scalar Functions

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

SQL User Defined Scalar Functions

Persists complex expression patterns

```sql
> CREATE FUNCTION roll_dice(
  num_dice INT DEFAULT 1 COMMENT 'number of dice to roll (Default: 1)’,
  num_sides INT DEFAULT 6 COMMENT 'number of sides per die (Default: 6)'
) COMMENT 'Roll a number of n-sided dice’
  RETURN aggregate(
    sequence(1, roll_dice.num_dice, 1),
    0,
    (acc, x) -> (rand() * roll_dice.num_sides) :: INT,
    acc -> acc + roll_dice.num_dice
  );

> SELECT roll_dice();
3
-- Roll 3 6-sided dice
> SELECT roll_dice(3);
15
-- Roll 3 10-sided dice
> SELECT roll_dice(3, 10)
21
```

SQL

SQL User Defined Table Functions

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

SQL User Defined Table Functions

Persist complex parameterized queries

```sql
CREATE FUNCTION weekdays(start DATE,end DATE)
RETURNS TABLE(day_of_week STRING, day DATE)
RETURN SELECT
  to_char(day, 'E’),
  day
FROM
(
  SELECT sequence(weekdays.start, weekdays.end)
) AS t(days),
LATERAL(explode(days)) AS dates(day)
WHERE
  extract(DAYOFWEEK_ISO FROM day) BETWEEN 1 AND 5;
```

SQL

SQL User Defined Table Functions

Persist complex parameterized queries

```
> SELECT
  day_of_week,
  day
FROM
  weekdays(DATE '2024-01-01',
    DATE '2024-01-14');

Mon     2022-01-01
…
Fri     2022-01-05
Mon     2022-01-08
```

SQL

```
> -- Return weekdays for date ranges originating from a
LATERAL correlation
> SELECT
  weekdays.*
FROM
VALUES
  (DATE '2020-01-01'),
  (DATE '2021-01-01') AS starts(start),
  LATERAL weekdays(start, start + INTERVAL '7' DAYS);

Wed     2020-01-01
Thu     2020-01-02
Fri     2020-01-03
…
```

SQL

Named parameter invocation

Self documenting and safer SQL UDF invocation

```
> DESCRIBE FUNCTION roll_dice;
Function: default.roll_dice
Type: SCALAR
Input: num_dice INT
       num_sides INT
Returns: INT

> -- Roll 1 10-sided dice - skip dice count
> SELECT roll_dice(num_sides => 10)
7

> -- Roll 3 10-sided dice - reversed order
> SELECT roll_dice(num_sides => 10, num_dice => 3)
21
```

SQL

Stored Procedure

External Stored Procedures

[VOTE] SPIP: Stored Procedures API for Catalogs spark-dev ×

L. C. Hsieh via spark.apache.org
to Spark

Hi all,

I'd like to start a vote for SPIP: Stored Procedures API for Catalogs.

Please also refer to:

- Discussion thread:
https://lists.apache.org/thread/7r04pz544c9qs3gc8q2nyj3fpzfnv8oo
- JIRA ticket: https://issues.apache.org/jira/browse/SPARK-44167
- SPIP doc: https://docs.google.com/document/d/1rDcggNl9YNcBECsfgPcoOecHXYZOu29QYFrloo2lPBg/

External Iceberg Stored Procedure


```
// position args only
> CALL iceberg.system.rollback_to_timestamp(
    array('ns1', 'ns2', 'tbl_name'),
    NOW() - INTERVAL '2' HOURS);
  ----------------------------------------
  | previous_snapshot_id | current_snapshot_id |
  ----------------------------------------
  |        8798794823489 |      34141242342351 |
  ----------------------------------------
```

SQL Scripting

It's SQL, but with control flow!

- 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

SQL Scripting

```SQL
BEGIN
  DECLARE c INT = 10;
  WHILE c > 0 DO
    INSERT INTO t VALUES (c);
    SET VAR c = c - 1;
  END WHILE;
END
```


SQL Scripting

```sql
-- 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;
```

SQL

PySpark UDF Unified Profiling

Overview of Unified Profiling

- Key Components: Performance and memory profiling
- Benefits: Tracks function calls, execution time, memory usage
- Replacement for Legacy Profiling
  - Drawbacks of Legacy Profiling
  - Advantages of New Unified Profiling
    - Session-based, works with Spark Connect, runtime toggling

Overview of Unified Profiling

- How to Enable:
  - Performance Profiler: spark.conf.set("spark.sql.pyspark.udf.profiler", "perf")
  - Memory Profiler: spark.conf.set("spark.sql.pyspark.udf.profiler", "memory")
- API Features: "show", "dump", and "clear" commands
  - Show results:
    - Performance: spark.profile.show(type="perf")
    - Performance: spark.profile.show(type="memory")
  - Dump results: spark.profile.dump("/your_path/...")
  - Clear results: spark.profile.clear()

PySpark Performance Profiler

```python
from pyspark.sql.functions import pandas_udf

df = spark.range(10)
@pandas_udf("long")
def add1(x):
  return x + 1

added = df.select(add1("id"))

spark.conf.set("spark.sql.pyspark.udf.profiler", "perf")
added.show()
```

Python

PySpark Performance Profiler

2 minutes ago (<1s) Cell 2

```
spark.profile.show(type="perf")
```

```
============================================================
Profile of UDF<id=50>
============================================================
         1108 function calls (1088 primitive calls) in 0.003 seconds

   Ordered by: internal time, cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        4    0.000    0.000    0.002    0.000 series.py:368(__init__)
        4    0.000    0.000    0.000    0.000 {built-in method _operator.add}
        4    0.000    0.000    0.003    0.001 base.py:1339(_arith_method)
        4    0.000    0.000    0.002    0.000 series.py:3075(_construct_result)
        4    0.000    0.000    0.000    0.000 cast.py:1598(maybe_cast_to_integer_arra
  240/236    0.000    0.000    0.000    0.000 {built-in method builtins.isinstance}
       12    0.000    0.000    0.000    0.000 series.py:671(name)
        4    0.000    0.000    0.001    0.000 construction.py:494(sanitize_array)
        4    0.000    0.000    0.001    0.000 construction.py:714(_try_cast)
        4    0.000    0.000    0.000    0.000 blocks.py:2385(new_block)
        8    0.000    0.000    0.000    0.000 _ufunc_config.py:32(seterr)
        4    0.000    0.000    0.000    0.000 generic.py:5931(__finalize__)
        8    0.000    0.000    0.000    0.000 construction.py:396(extract_array)
```

2 minutes ago (<1s) Cell 3

```
added.explain()
```

```
== Physical Plan ==
*(2) Project [pythonUDF0#59L AS add1(id)#51L]
+- ArrowEvalPython [add1(id#48L)#50L], [pythonUDF0#59L], 200
   +- *(1) ColumnarToRow
      +- PhotonResultStage
         +- PhotonRange Range (0, 10, step=1, splits=4)
```

PySpark Memory Profiler

```python
from pyspark.sql.functions import pandas_udf

df = spark.range(10)
@pandas_udf("long")
def add1(x):
  return x + 1

added = df.select(add1("id"))

spark.conf.set("spark.sql.pyspark.udf.profiler", "memory")
added.show()
```

Python

PySpark Memory Profiler

2 minutes ago (<1s)

```
spark.profile.show(type="memory")
```

```
============================================================
Profile of UDF<id=4>
============================================================
Filename: /root/.ipykernel/1259/command-1653643849461535-3519972363

Line #    Mem usage    Increment  Occurrences   Line Contents
=============================================================
     4    606.5 MiB    606.5 MiB           4   @pandas_udf("long")
     5                                         def add1(x):
     6    606.5 MiB      0.0 MiB           4     return x + 1
```

Just now (<1s)

```
added.explain()
```

```
== Physical Plan ==
*(2) Project [pythonUDF0#14L AS add1(id)#5L]
+- ArrowEvalPython [add1(id#2L)#4L], [pythonUDF0#14L], 200
   +- *(1) ColumnarToRow
      +- PhotonResultStage
         +- PhotonRange Range (0, 10, step=1, splits=4)
```

Agenda

New Functionalities

Spark Connect, ANSI Mode, Arbitrary Stateful Processing V2, Collation Support, Variant Data Types, pandas 2.x Support

Extensions

Python Data Source APIs, XML/Databricks Connectors and DSV2 Extension, Delta 4.0

Custom Functions and Procedures

SQL UDFs, SQL Scripting, Python UDTF, Arrow optimized Python UDF, PySpark UDF Unified Profiler

Usability

Structured Logging Framework, Error Class Framework, Behavior Change Process

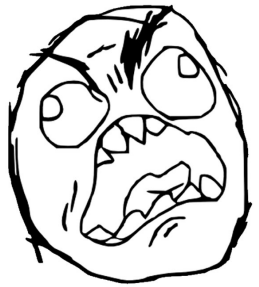
Structured Logging Framework

What are we going to build to improve this?

- Transition to Structured Logging in Apache Spark
- Introducing Spark System Log Directories

Analyzing Spark logs is challenging due to their unstructured nature

```
24/05/24 11:53:49 INFO TaskSetManager: Finished task 3.0 in stage 1992.0 (TID 10959) in 83 ms on 10.10.107.91 (executor driver) (10/10)
24/05/24 11:53:49 INFO TaskSchedulerImpl: Removed TaskSet 1992.0, whose tasks have all completed, from pool
24/05/24 11:53:49 INFO DAGScheduler: ResultStage 1992 (collect at <console>:1) finished in 84 ms
24/05/24 11:53:49 INFO DAGScheduler: Job 1992 is finished. Cancelling potential speculative or zombie tasks for this job
24/05/24 11:53:49 INFO TaskSchedulerImpl: Canceling stage 1992
24/05/24 11:53:49 INFO TaskSchedulerImpl: Killing all running tasks in stage 1992: Stage finished
24/05/24 11:53:49 INFO DAGScheduler: Job 1992 finished: collect at <console>:1, took 0.084681 s
24/05/24 11:53:49 INFO BlockManagerInfo: Removed broadcast_2984_piece0 on 10.10.107.91:49365 in memory (size: 39.1 KiB, free: 434.4 MiB)
24/05/24 11:53:49 INFO InMemoryFileIndex: It took 7 ms to list leaf files for 1 paths.
24/05/24 11:53:49 INFO SparkContext: Starting job: parquet at <console>:1
24/05/24 11:53:49 INFO DAGScheduler: Got job 1993 (parquet at <console>:1) with 1 output partitions
24/05/24 11:53:49 INFO DAGScheduler: Final stage: ResultStage 1993 (parquet at <console>:1)
24/05/24 11:53:49 INFO DAGScheduler: Parents of final stage: List()
24/05/24 11:53:49 INFO DAGScheduler: Missing parents: List()
24/05/24 11:53:49 INFO DAGScheduler: Submitting ResultStage 1993 (MapPartitionsRDD[5981] at parquet at <console>:1), which has no missing parents
24/05/24 11:53:49 INFO MemoryStore: Block broadcast_2989 stored as values in memory (estimated size 114.5 KiB, free 434.0 MiB)
24/05/24 11:53:49 INFO MemoryStore: Block broadcast_2989_piece0 stored as bytes in memory (estimated size 41.3 KiB, free 434.0 MiB)
24/05/24 11:53:49 INFO BlockManagerInfo: Added broadcast_2989_piece0 in memory on 10.10.107.91:49365 (size: 41.3 KiB, free: 434.3 MiB)
24/05/24 11:53:49 INFO SparkContext: Created broadcast 2989 from broadcast at DAGScheduler.scala:1641
24/05/24 11:53:49 INFO DAGScheduler: Submitting 1 missing tasks from ResultStage 1993 (MapPartitionsRDD[5981] at parquet at <console>:1) (first 15 tasks are for partitions Vector(0))
24/05/24 11:53:49 INFO TaskSchedulerImpl: Adding task set 1993.0 with 1 tasks resource profile 0
24/05/24 11:53:49 INFO TaskSetManager: Starting task 0.0 in stage 1993.0 (TID 10966) (10.10.107.91,executor driver, partition 0, PROCESS_LOCAL, 7936 bytes)
24/05/24 11:53:49 INFO Executor: Running task 0.0 in stage 1993.0 (TID 10966)
24/05/24 11:53:49 INFO Executor: Finished task 0.0 in stage 1993.0 (TID 10966). 1959 bytes result sent to driver
24/05/24 11:53:49 INFO TaskSetManager: Finished task 0.0 in stage 1993.0 (TID 10966) in 6 ms on 10.10.107.91 (executor driver) (1/1)
24/05/24 11:53:49 INFO TaskSchedulerImpl: Removed TaskSet 1993.0, whose tasks have all completed, from pool
24/05/24 11:53:49 INFO DAGScheduler: ResultStage 1993 (parquet at <console>:1) finished in 11 ms
24/05/24 11:53:49 INFO DAGScheduler: Job 1993 is finished. Cancelling potential speculative or zombie tasks for this job
24/05/24 11:53:49 INFO TaskSchedulerImpl: Canceling stage 1993
24/05/24 11:53:49 INFO TaskSchedulerImpl: Killing all running tasks in stage 1993: Stage finished
24/05/24 11:53:49 INFO DAGScheduler: Job 1993 finished: parquet at <console>:1, took 0.011631 s
24/05/24 11:53:49 INFO FileSourceStrategy: Pushed Filters:
24/05/24 11:53:49 INFO FileSourceStrategy: Post-Scan Filters:
24/05/24 11:53:49 INFO MemoryStore: Block broadcast_2990 stored as values in memory (estimated size 215.6 KiB, free 433.8 MiB)
24/05/24 11:53:49 INFO MemoryStore: Block broadcast_2990_piece0 stored as bytes in memory (estimated size 39.1 KiB, free 433.7 MiB)
24/05/24 11:53:49 INFO BlockManagerInfo: Added broadcast_2990_piece0 in memory on 10.10.107.91:49365 (size: 39.1 KiB, free: 434.3 MiB)
24/05/24 11:53:49 INFO SparkContext: Created broadcast 2990 from collect at <console>:1
24/05/24 11:53:49 INFO FileSourceScanExec: Planning scan with bin packing, max size: 4595313 bytes, open cost is considered as scanning 4194304 bytes.
24/05/24 11:53:49 INFO SparkContext: Starting job: collect at <console>:1
24/05/24 11:53:49 INFO DAGScheduler: Got job 1994 (collect at <console>:1) with 10 output partitions
```



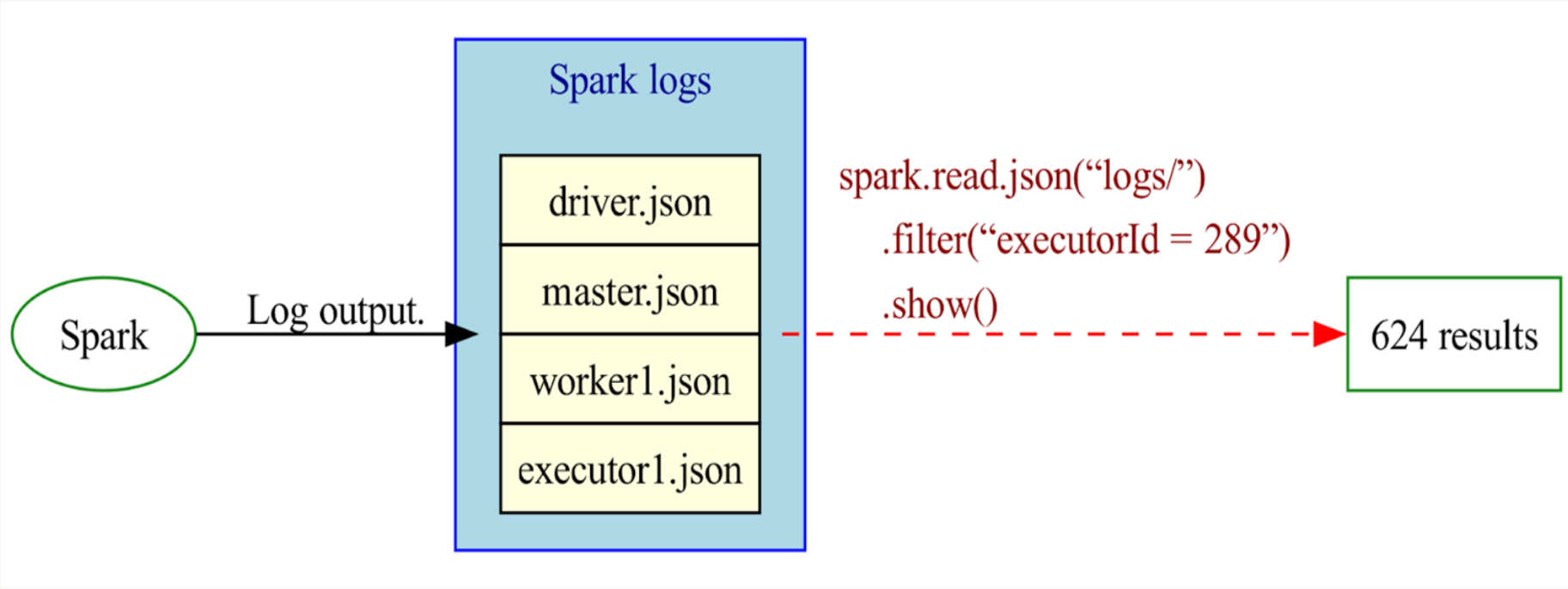
Structured Spark Logging

Starting from Spark 4.0, the default log format is JSON lines, making it easier to parse and analyze.

```json
{
  "ts": "2023-03-12T12:02:46.661-0700",
  "level": "ERROR",
  "msg": "Fail to know the executor 289 is alive or not",
  "context": {
    "executor_id": "289"
  },
  "exception": {
    "class": "org.apache.spark.SparkException",
    "msg": "Exception thrown in awaitResult",
    "stackTrace": "..."
  },
  "source": "BlockManagerMasterEndpoint"
}
```

Json

Use Spark to Analyze Spark Logs



System Log Directories

```python
logs = spark.read.json("/var/spark/logs.json")

# To get all the errors on host 100.116.29.4
errors_host_logs = logs.filter(
    (col("context.host") == "100.116.29.4") & (col("level") == "ERROR"))

# To get all the exceptions from Spark
spark_exceptions_logs = logs.filter(
    col("exception.class").startswith("org.apache.spark"))
```

Python

System Log Directories

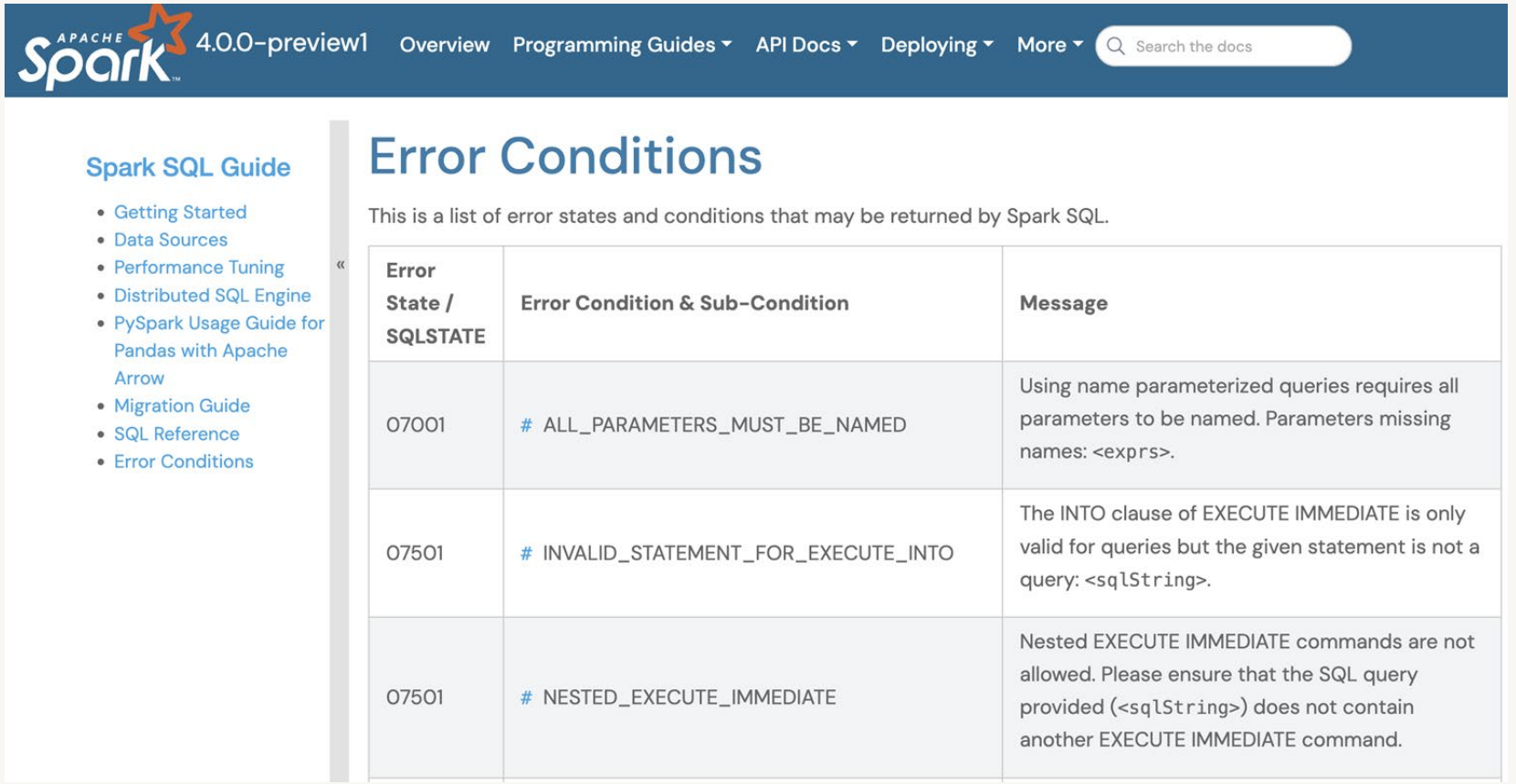
```python
logs = spark.read.json("/var/spark/logs.json")


# To get all the executor loss logs
executor_lost_logs = logs.filter(
    col("msg").contains("Lost executor"))
# To get all the distributed logs about executor 289
executor_289_logs = logs.filter(
    col("context.executor_id") == 289)
```

Python

Error Conditions and Messages

Error Conditions

800+ top-frequency error conditions from the server.



Apache Spark 4.0.0-preview1 Overview Programming Guides API Docs Deploying More Search the docs

Spark SQL Guide

- Getting Started
- Data Sources
- Performance Tuning
- Distributed SQL Engine
- PySpark Usage Guide for Pandas with Apache Arrow
- Migration Guide
- SQL Reference
- Error Conditions

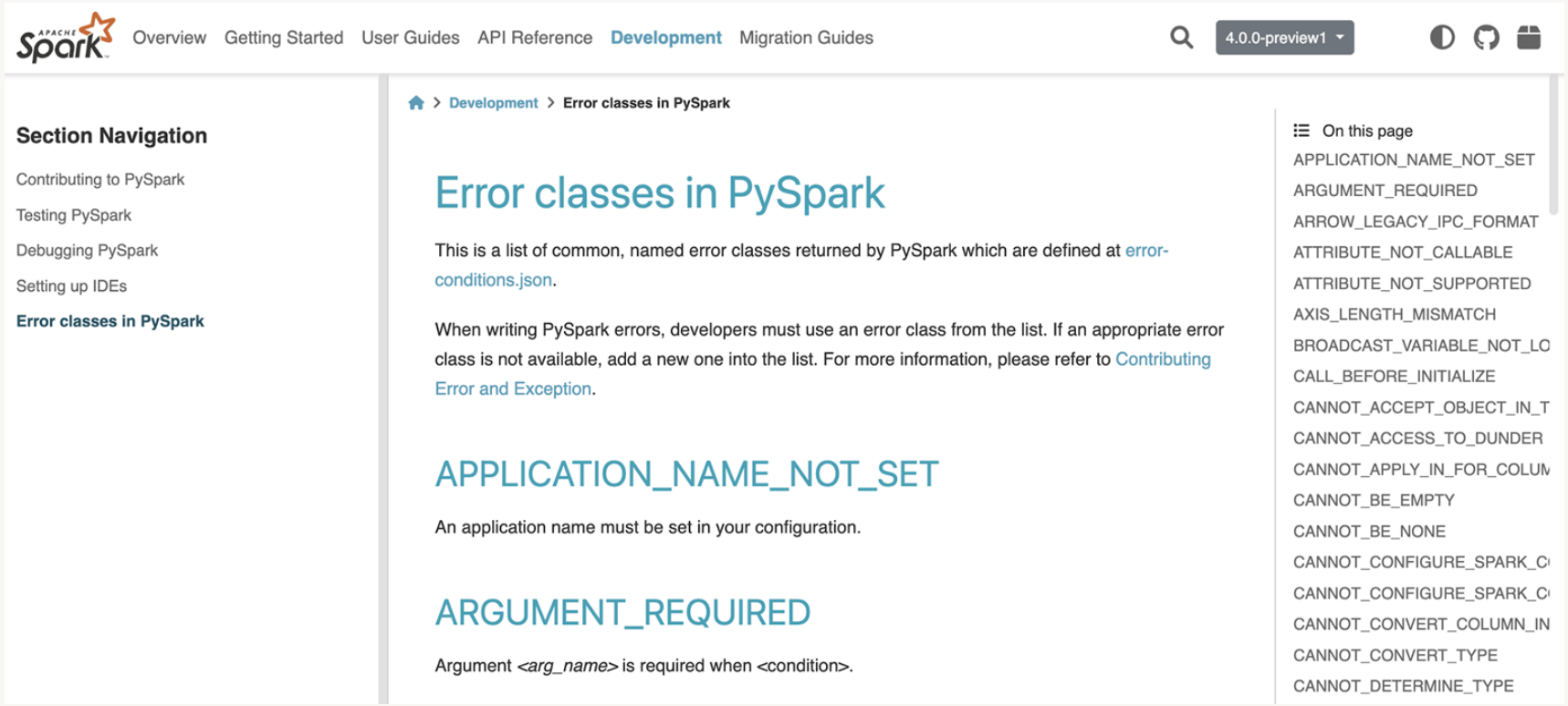
Error Conditions

This is a list of error states and conditions that may be returned by Spark SQL.

Error State / SQLSTATE	Error Condition & Sub-Condition	Message
07001	# ALL_PARAMETERS_MUST_BE_NAMED	Using name parameterized queries requires all parameters to be named. Parameters missing names: <exprs>.
07501	# INVALID_STATEMENT_FOR_EXECUTE_INTO	The INTO clause of EXECUTE IMMEDIATE is only valid for queries but the given statement is not a query: <sqlString>.
07501	# NESTED_EXECUTE_IMMEDIATE	Nested EXECUTE IMMEDIATE commands are not allowed. Please ensure that the SQL query provided (<sqlString>) does not contain another EXECUTE IMMEDIATE command.

Error Conditions in PySpark

All 200+ error conditions issued by PySpark client

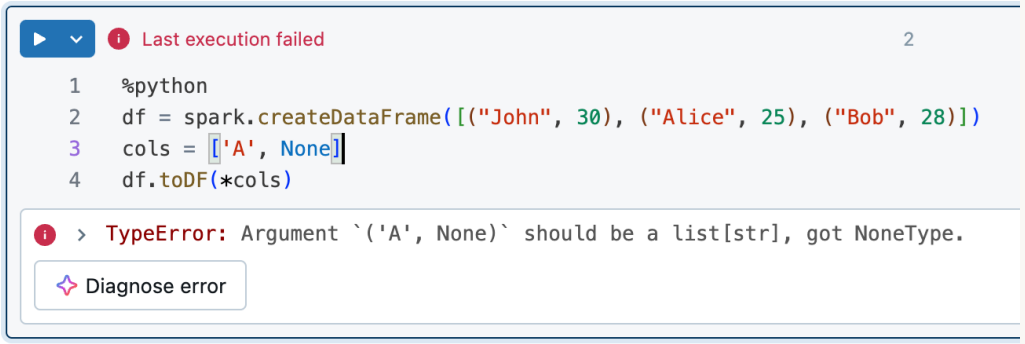


Quality

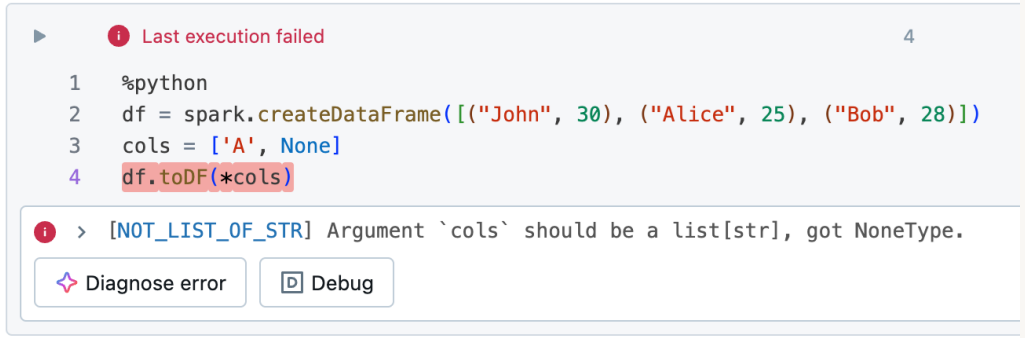
- Clear and specific error classes
- Improved documentation
- Consistency and standardization
- Enhanced debugging and maintenance

PySpark Errors

Before



After



Spark 3.5

```
1  from pyspark.sql.functions import col
2  from pyspark.sql.types import LongType
3
4  spark.conf.set("spark.sql.ansi.enabled", True)
5
6  df = spark.range(1, 20).withColumn("id_divided", col("id") / 0).write.format("json") \
7    .save("/gatorsmile/test10.json")
```

▸ (1) Spark Jobs

```
> org.apache.spark.SparkException: Job aborted due to stage failure: Task 1 in stage 3.0 failed 4 times, m
t recent failure: Lost task 1.3 in stage 3.0 (TID 40) (10.68.151.99 executor 0): org.apache.spark.SparkArithmetic
Exception: [DIVIDE_BY_ZERO] Division by zero. Use `try_divide` to tolerate divisor being 0 and return NULL inst…
```

Spark 4.0

```
1  from pyspark.sql.functions import col
2  from pyspark.sql.types import LongType
3
4  spark.conf.set("spark.sql.ansi.enabled", True)
5
6  df = spark.range(1, 20).withColumn("id_divided", col("id") / 0).write.format("json") \
7    .save("/gatorsmile/test9.json")
```

▸ (1) Spark Jobs

```
> [DIVIDE_BY_ZERO] Division by zero. Use `try_divide` to tolerate divisor being 0 and return NULL instead. I
f necessary set "spark.sql.ansi.enabled" to "false" to bypass this error. SQLSTATE: 22012
```

Behavior Changes

Overview of Apache Spark Versioning Policy

- Semantic Versioning Structure: [MAJOR].[FEATURE].[MAINTENANCE]
  - MAJOR: Long-term API stability
  - FEATURE: New features and improvements
  - MAINTENANCE: Frequent, urgent patches
- API Compatibility Commitments: Maintain compatibility across feature releases, reducing the need for users to refactor code.

Categories of Impactful Behavior Changes

- **Query Results Impact**: Changes affecting the accuracy and outcome of data queries.
- **Schema and Configuration Changes**: Adjustments to the database or application schema and Spark configuration settings.
- **API Modifications**: Alterations to the public and developer APIs across multiple programming languages.
- **Error Handling Adjustments**: Modifications in how errors are classified and handled within the system.
- **Deployment and Management Revisions**: Changes in the methods and tools used for deploying and managing Spark environments.

Best Practices for API Changes

- General Approach:
  - Avoid API changes whenever possible.
  - Prefer deprecating features over direct modifications to ensure smoother transitions.
  - Implement legacy flags to allow users to opt into previous behaviors temporarily, easing the transition to new versions.
  - Communicate changes clearly through deprecation warnings and documentation updates.

Best Practices for API Changes

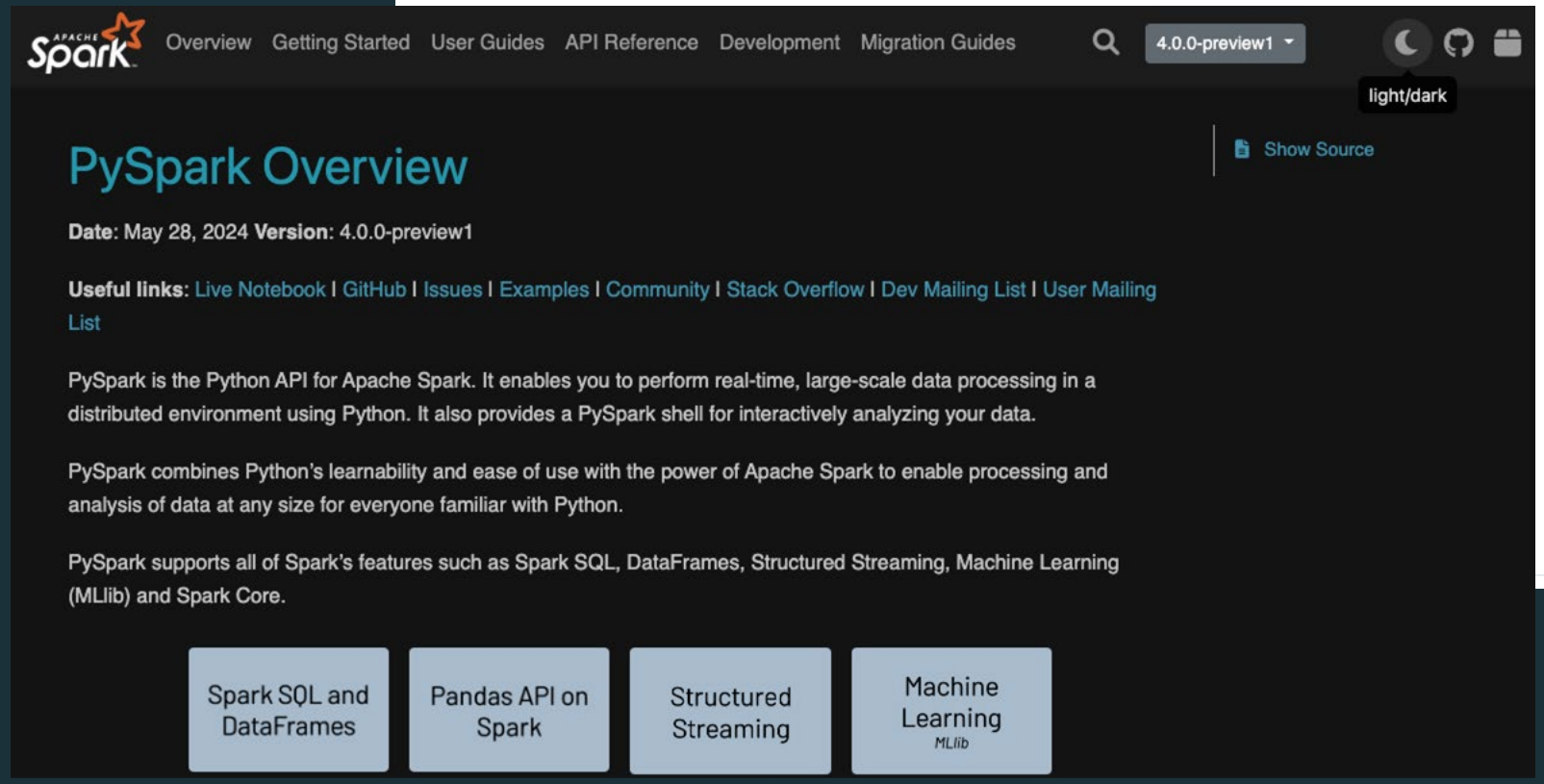
- User-facing documentation
  - **Migration Guide Updates**: Regularly update the migration guide with detailed information on changes affecting user operations.
  - **Legacy Configs:** Include information on legacy configurations that might help users transition smoothly between versions.
- Error Messages:
  - C**larity and Actionability**: Ensure that all error messages are clear and direct, informing the user precisely what went wrong.
  - **Workarounds**: Wherever possible, provide actionable advice within the error message, including configuration changes that can revert to previous behaviors or other immediate solutions.

Best Practices for API Changes

- PR Descriptions
  - **Detail**: Provide comprehensive explanations of the changes, highlighting the modifications and their implications.
  - **Transparency**: Explain clearly how the new behavior differs from the old, and the reasons for these changes.

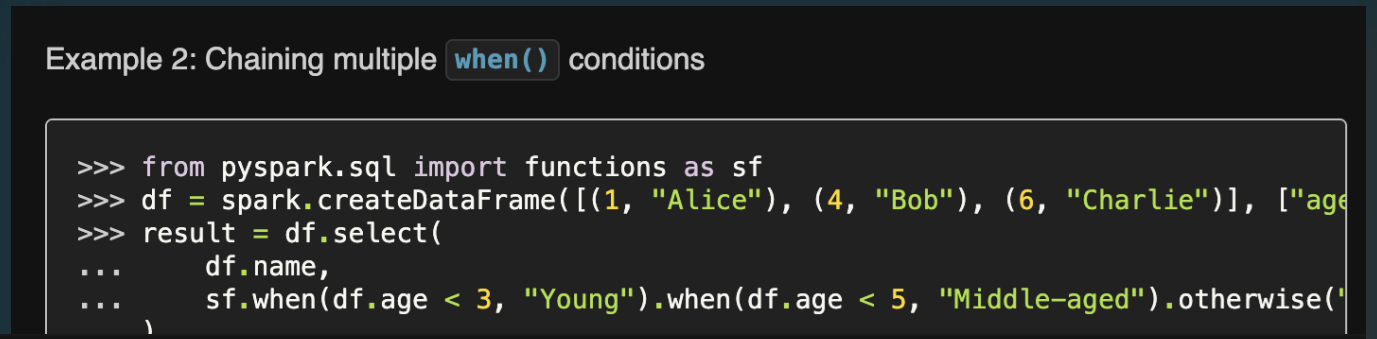
Documentation

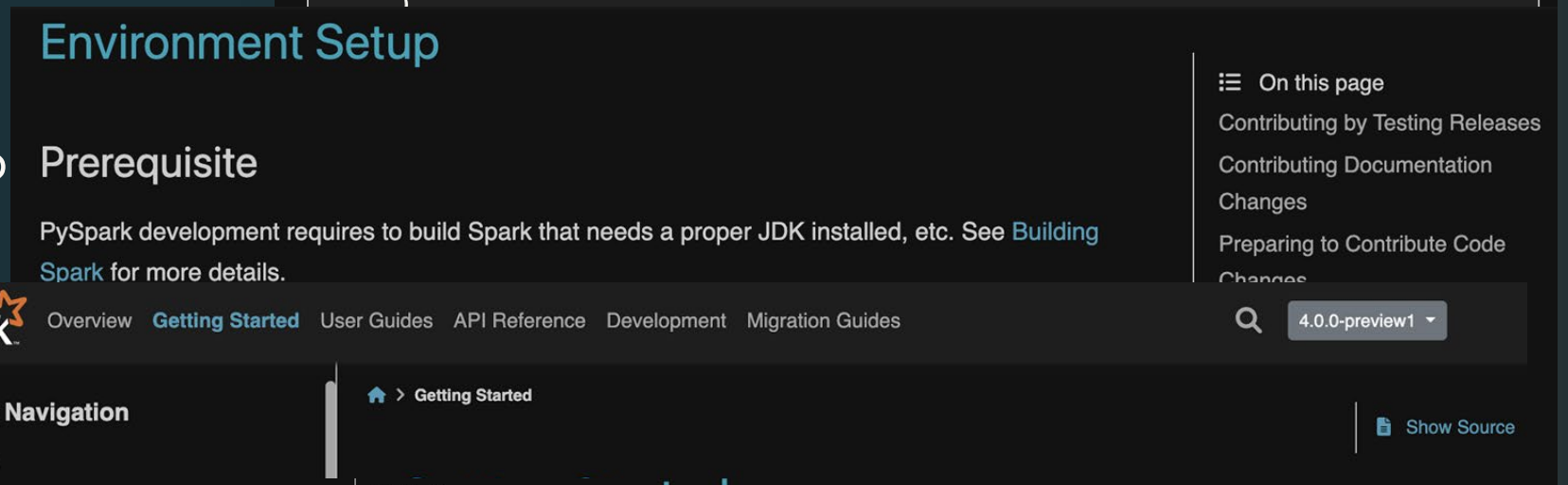
PySpark Doc – Dark Mode

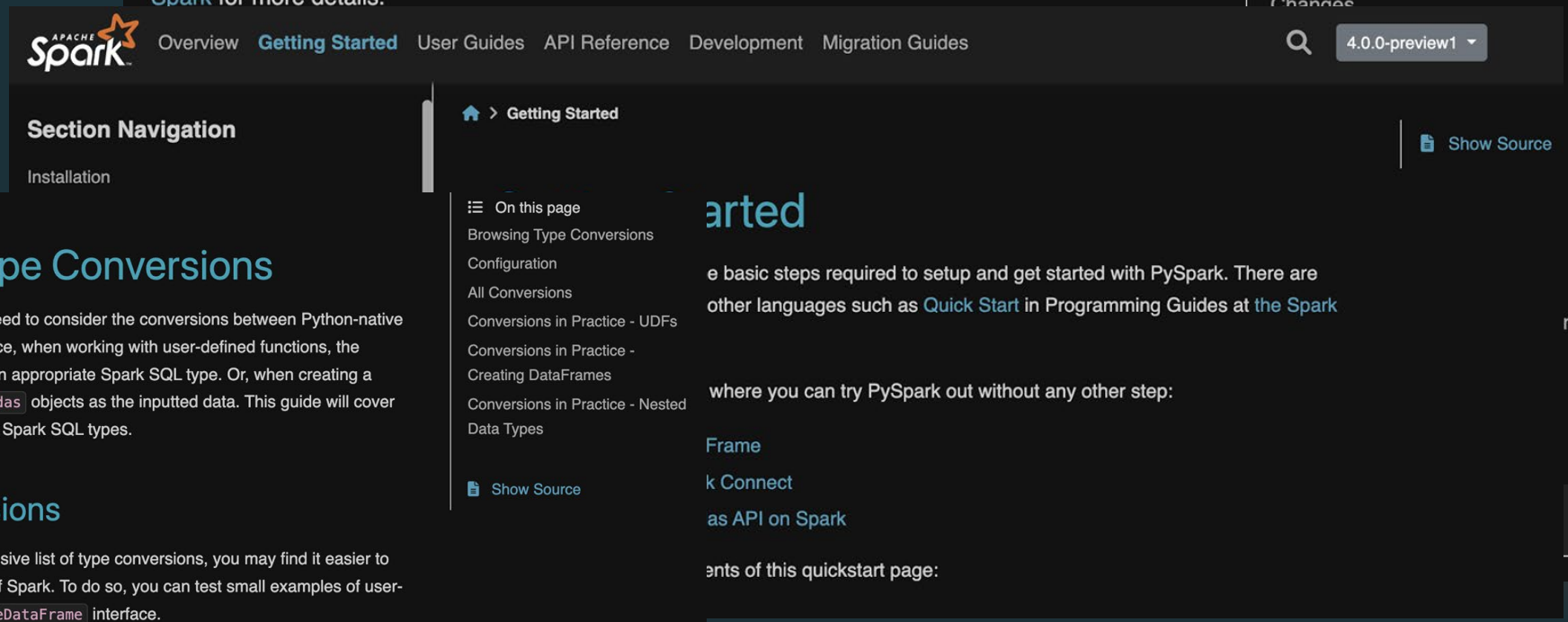


PySpark Doc

- SPARK-44728
  - More examples
  - Environment Setup
  - Quickstart
  - Type System







Versionless Spark Programming Guide

[Proposal] Versionless Spark Programming Guide

- Motivation: Allow for real-time updates and rapid content iteration without release schedule constraints.
- Small changes pose less SEO deranking risk than major updates.
- Transition 9 existing programming guides to Spark website repo.

Quick Start

RDDs, Accumulators, Broadcasts Vars

SQL, DataFrames, and Datasets

Structured Streaming

Spark Streaming (DStreams)

MLlib (Machine Learning)

GraphX (Graph Processing)

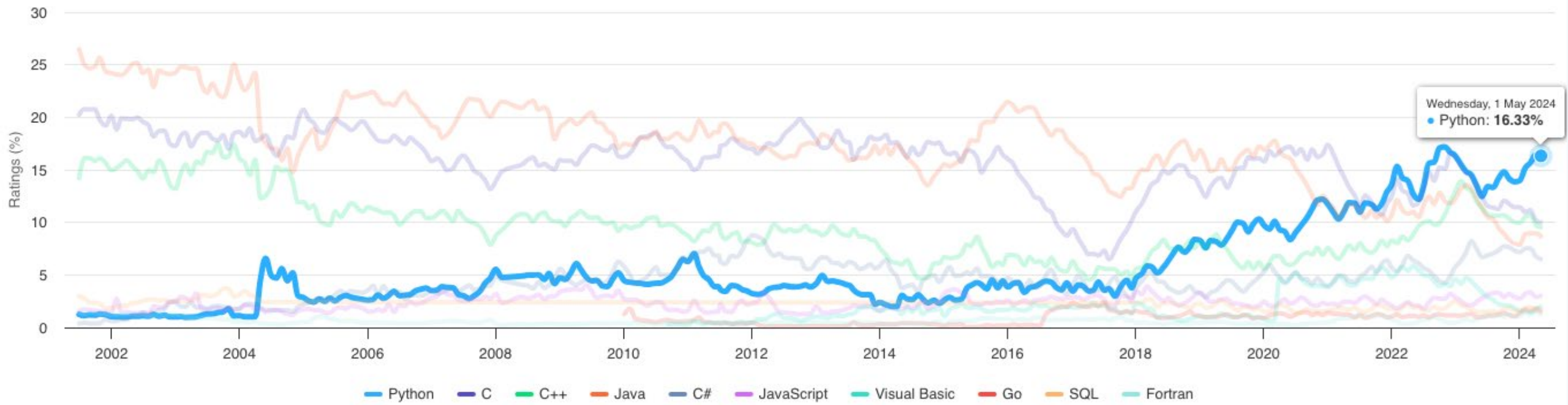
SparkR (R on Spark)

PySpark (Python on Spark)

Python: The Number One Choice

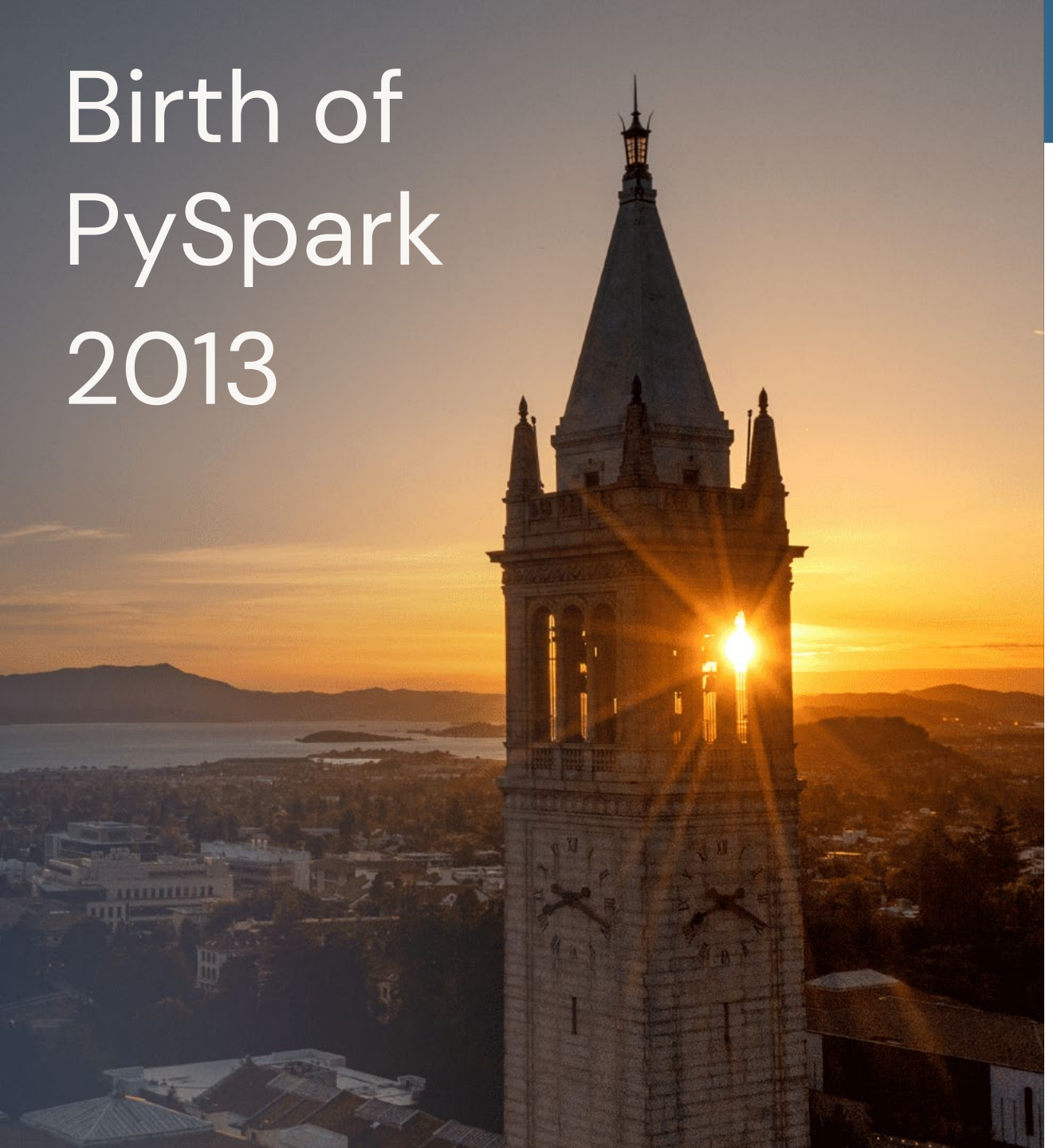
TIOBE Programming Community Index

Source: www.tiobe.com



May 2024	May 2023	Change	Programming Language	Ratings	Change
1	1		Python	16.33%	+2.88%
2	2		C	9.98%	-3.37%
3	4	↑	C++	9.53%	-2.43%
4	3	↓	Java	8.69%	-3.53%
5	5		C#	6.49%	-0.94%

Birth of PySpark 2013



APACHE Spark™

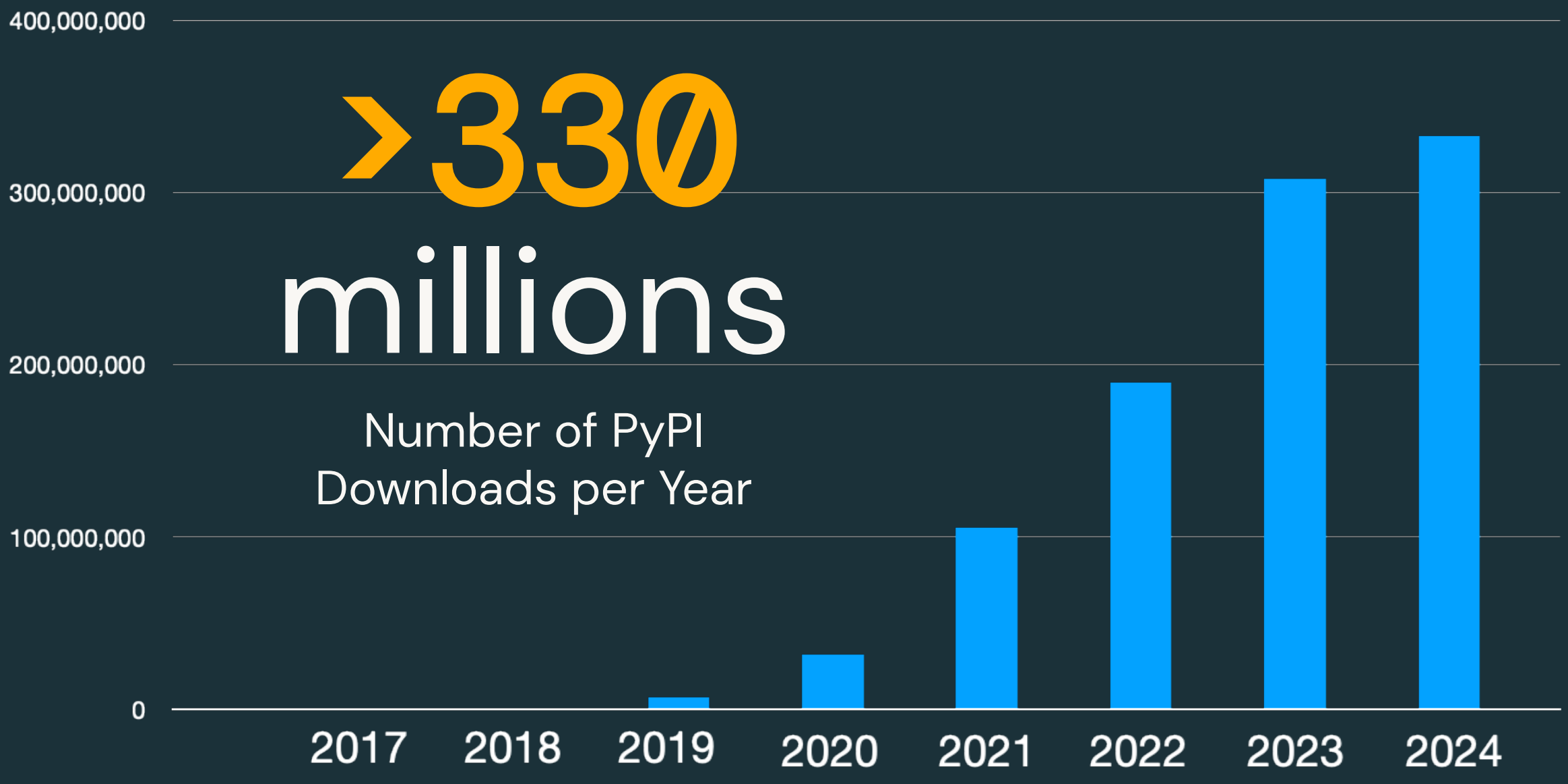
Spark Release 0.7.0

The Spark team is proud to release version 0.7.0, a new major release that brings several new features. Most notable are a Python API for Spark and an alpha of Spark Streaming. (Details on Spark Streaming can also be found in this technical report.) The release also adds numerous other improvements across the board. Overall, this is our biggest release to date, with 31 contributors, of which 20 were external to Berkeley.

You can download Spark 0.7.0 as either a source package (4 MB tar.gz) or prebuilt package (60 MB tar.gz).

Python API

Spark 0.7 adds a Python API called PySpark that makes it possible to use Spark from Python, both in standalone programs and in interactive Python shells. It uses the standard CPython runtime, so your programs can call into native libraries like NumPy and SciPy. Like the Scala and Java APIs, PySpark will automatically ship functions from your main program, along with the variables they depend on, to the cluster. PySpark supports most Spark features, including RDDs, accumulators, broadcast variables, and HDFS input and output.




210
Countries
and Regions
PyPI downloads of
PySpark
in the last 12 months

PySpark's View of the World

DataFrame APIs

SQL

pandas APIs

PySpark

Catalyst Optimizer

Adaptive Execution

Spark Ecosystem (connectors, 3rd party libraries)

Key Focus of PySpark

Functionality Parity

- Complete feature availability
- Python native APIs

Ease of Use

- No JVM Knowledge Required
- Pythonic APIs

Performance Parity

- Matched Performance with Scala
- Optimized query compiler and engine

Ecosystem Integration

- Integrated with Python ecosystem
- Spark ecosystem growth

Key Enhancements in PySpark 3.5 and 4.0

Functionality Parity

- 180 new built-in functions in Spark 3.5+
- Python native data source APIs
- Stateful streaming processing V2
- Python UDTFs

Ease of Use

- Spark Connect
- Error Framework
- Enriched Documentation
- Unified UDF profiling

Performance

- Spark Connect
- Arrow-optimized Python UDF
- Variant type for semi-structured processing
- Advanced optimizer and adaptive execution

Ecosystem Integration

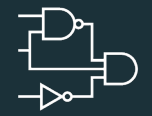
- Compatible with pandas 2 (pandas API)
- Delta Lake 4.0
- [ML] Distributed training with TorchDistributor
- Arrow integration: toArrow API

Python UDF Memory/CPU Profiler

Pandas 2 Compatibility

pandas API on Spark Coverage

NumPy Inputs



UDF-level Dependency Control [WIP]

applyInArrow

180+ New Python Functions

Spark Connect

mapInArrow

Deepspeed Distributor

PySpark

Visualization/ Plotting

Richer Connectors

Pythonic Error Handling

[WIP] DF / UDF Debuggability

Arrow-optimized Python UDFs

DF.toArrow()

Python Arbitrary Stateful Processing

Type Annotations

Python UDTF

PySpark Testing API

Thank you for your contributions!