DATA+AI SUMMIT 2022

PySpark in Apache Spark 3.3 and Beyond

Hyukjin Kwon
Software Engineer, Databricks

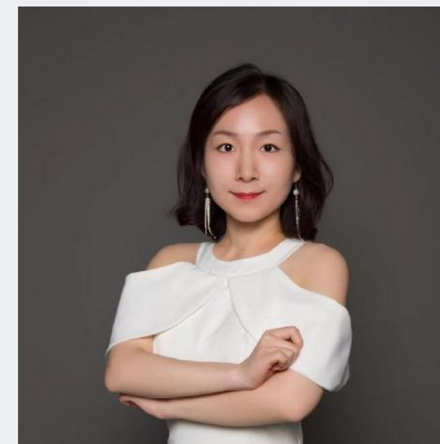
Xinrong Meng
Software Engineer, Databricks

ORGANIZED BY databricks

Who are you?

Hyukjin Kwon

- @HyukjinKwon in GitHub
- Tech lead, PySpark team @ Databricks
- Top 2 contributor in Apache Spark
- PySpark, SparkR, Spark SQL, etc.



Xinrong Meng

- @xinrong-databricks in GitHub
- PySpark team @ Databricks
- Major contributor in PySpark

Project Zen

- Be **Pythonic**
- Better and **easier use** of PySpark
- **Better interoperability** with other Python libraries

Pandas API on Spark

pandas provides data structures for in-memory analytics … using pandas to analyze datasets that are **larger than memory datasets somewhat tricky**. … it's worth considering *not using pandas*. **pandas isn't the right tool for all situations**. …

Pandas API on Spark

```
>>> from pandas import read_csv
>>> from pyspark.pandas import read_csv
>>> df = read_csv("data.csv")
```

Drop-in replacement

- Pandas API on Upcoming Apache Spark™ 3.2
- SPIP: Support pandas API layer on PySpark

What is this talk about?

What's new?

- Pandas API on Spark
  - Faster default index
  - Better API coverage
- New Functionalities
  - datetime.timedelta support
  - PyArrow batch interface
  - Python standard string formatter in sql
- Productivity
  - Better autocompletion
  - Python/Pandas UDF profiler
  - Error classification

What's next?

- Usability
  - Spark Connect project
  - Py4J improvement
  - Native NumPy support
  - Better docstrings
- Performance
  - Source-native index in Pandas API on Spark
  - Optimized createDataFrame with Arrow
- Feature parity
  - Observable API for Structured Streaming
  - Latest pandas API in Pandas API on Spark

What's new?

Pandas API on Spark

Faster default index

```
>>> import pandas as pd
>>> pd.DataFrame({"col": ["a", "b", "c"]})
  col
0   a
1   b
2   c
```

pandas' default (range) index

Sequence increasing one by one, challenging in distributed computation

Pandas API on Spark

Faster default index

Native impl. of distributed sequence index with **Catalyst Optimizer**

- Native implementation of distributed sequence index (SPARK-36338)
- Dedicated Spark plan with optimization rules enabled (SPARK-36559)

```
Sort ...
+- Aggregate ...
  +- ExistingRDD ...
```

Before

```
Sort ...
+- Aggregate ...
  +- Range ...
```

After

Catalyst Optimizer **optimizes/prunes** the default index **computation** bundled in RDD

Pandas API on Spark

Faster default index

Distributed sequence index **by default** (SPARK-37649)

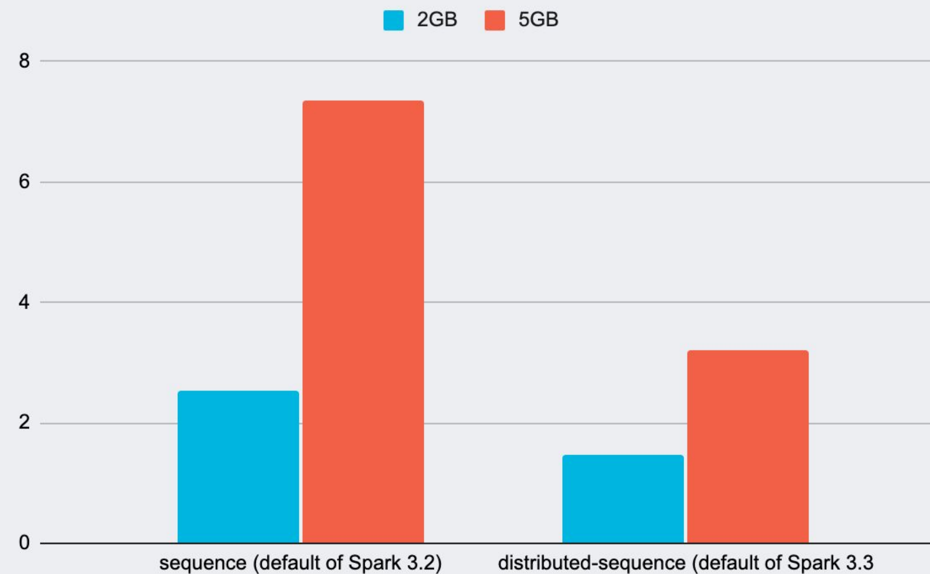
```
>>> import pyspark.pandas as ps
>>> ps.options.compute.default_index_type
'distributed-sequence'
```

'distributed-sequence' index by default

Scales out better and truly **distributed computation**

Pandas API on Spark

Faster default index



Scan 2GB/5GB with default Index (secs)

More than **2 times faster** (5 nodes cluster, i3.xlarge)

Pandas API on Spark

Better API coverage

Native implementation of **ps.merge_asof** (SPARK-36813)

```
>>> left = ps.DataFrame({"a": [1, 5, 10], "left_val": ["a", "b", "c"]})
>>> right = ps.DataFrame({"a": [1, 2, 3, 6, 7], "right_val": [1, 2, 3, 6, 7]})
>>> ps.merge_asof(left, right, on="a")
   a left_val  right_val
0   1        a          1
1   5        b          3
2  10        c          7
```

ps.merge_asof example in Pandas API on Spark

Pandas API on Spark

Better API coverage

```
AttachDistributedSequence ...
+- Project ...
   +- Join LeftOuter ...
      :- LogicalRDD ...
      +- Aggregate ...
         +- Join Inner ...
            :- Aggregate ...
            :  +- LogicalRDD ...
            +- LogicalRDD ...
```

Fully leverage all optimization such as **pruning**, **codegen** and **AQE** by Catalyst Optimizer

Pandas API on Spark

Better API coverage

New pandas-analog API

- DataFrame.combine_first (SPARK-36399)
- DataFrame.cov (SPARK-36396)
- TimedeltaIndex (SPARK-37525)
- MultiIndex.dtypes (SPARK-36930)
- ps.timedelta_range (SPARK-37673)
- ps.to_timedelta (SPARK-37701)
- Timedelta Series (SPARK-37525)
- ...

Full list of supported API is now available from Apache Spark 3.3

New Functionalities

datetime.timedelta support

```
>>> import datetime
>>> df = spark.createDataFrame([{'col': datetime.timedelta(minutes=10)}])
>>> row = df.select(df.col - datetime.timedelta(minutes=9, seconds=12)).first()
>>> row[0]
datetime.timedelta(seconds=48)
```

datetime.timedelta example in PySpark

Any datetime.timedelta can be used in any place including Python/Pandas UDFs

Internally mapped to **DayTimeIntervalType** in Spark SQL (SPARK-37275)

New Functionalities

datetime.timedelta support

```
>>> from datetime import timedelta; import pyspark.pandas as ps; import pandas as pd
>>> ps.from_pandas(
...     pd.Series([timedelta(minutes=1)],
...     index=pd.TimedeltaIndex([timedelta(days=1)])))
1 days    0 days 00:01:00
dtype: timedelta64[ns]
```

datetime.timedetla example in Pandas API on Spark

datetime.timedelta can also be used in Pandas API on Spark (SPARK-37525)

New Functionalities

PyArrow batch interface

```
>>> import pyarrow as pa; from typing import Iterator
>>> df = spark.createDataFrame([(1, "foo"), (2, None), (3, "bar"), (4, "bar")])
>>> def func(itr: Iterator[pa.RecordBatch]) -> Iterator[pa.RecordBatch]:
...     for batch in itr:
...         yield batch  # `batch` is pyarrow.RecordBatch.
...
>>> df.mapInArrow(func, df.schema)
```

DataFrame.mapInArrow example

Batch processing with native **PyArrow's** **RecordBatch** in DataFrame (SPARK-37228)

New Functionalities

PyArrow batch interface

```
>>> import pyarrow as pa; import awkward as ak
>>> def func(itr: Iterator[pa.RecordBatch]) -> Iterator[pa.RecordBatch]:
...     for batch in iterator:
...         combinations = ak.combinations(ak.from_arrow(batch)["x"], 2, axis=1)
...         yield from ak.to_arrow_table(ak.Array({"c": combinations})).to_batches()
...
>>> df.mapInArrow(func, df.schema)
```

DataFrame.mapInArrow example with awkward Array

Nicely **vectorized** computation with **nested** data

New Functionalities

Python standard string formatter in sql

```
>>> mydf = spark.range(10)
>>> spark.sql("SELECT {tbl.id}, {tbl[id]}, {col} FROM {tbl}", tbl=mydf, col=mydf.id)
DataFrame[id: bigint, id: bigint, id: bigint]
```

String formatter example with SparkSession.sql API

Support of PEP 3101 **Advanced String Formatting**

DataFrame, **Column**, and Python built-in type supported (SPARK-37516)

Productivity

Better autocompletion

```python
def take(self, num: int) -> List[Row]:
...
```

Before: dataframe.pyi

```python
def take(self, num):
    """..."""
    return self.limit(num).collect()
```

Before: dataframe.py

```python
def take(self, num: int) -> List[Row]:
    """..."""
    return self.limit(num).collect()
```

After: dataframe.py

Inlined type hints from stub files (.pyi) in PySpark (SPARK-39370)

Productivity

Better autocompletion

```
1   df.take
Signature: df.take(num)
Docstring:
Returns the first ``num`` rows as a :class:`list` of :class:`R

.. versionadded:: 1.3.0

Examples
--------
>>> df.take(2)
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
```

Before

```
1   df.take
Signature: df.take(num: int)-> List[pyspark.sql.types.Row]
Docstring:
Returns the first ``num`` rows as a :class:`list` of :class:`R

.. versionadded:: 1.3.0

Examples
--------
>>> df.take(2)
[Row(age=2, name='Alice'), Row(age=5, name='Bob')]
```

After

Better autocompletion by showing **parameter types**

Productivity

Python/Pandas UDF Profiler

```
>>> from pyspark.sql.functions import udf; import time
>>> _ = spark.range(10).select(udf(lambda x: time.sleep(1))("id")).collect()
>>> sc.show_profiles()
...
  ncalls  tottime  percall  cumtime  percall filename:lineno(function)
     10   10.005    1.001   10.005    1.001 {built-in method time.sleep}
     10    0.000    0.000   10.005    1.001 <stdin>:1(<lambda>)
...
```

Python profiler support in all **Python and Pandas UDFs** (SPARK-37443)

Productivity

Error classification

```
1  {
2    "AMBIGUOUS_FIELD_NAME" : {
3      "message" : [ "Field name <fieldName> is ambiguous and has <n> matching fields in the struct." ],
4      "sqlState" : "42000"
5    },
6    "ARITHMETIC_OVERFLOW" : {
7      "message" : [ "<message>.<alternative> If necessary set <config> to \"false\" (except for ANSI interval type) to bypass this error." ],
8      "sqlState" : "22003"
9    },
10   "CANNOT_CAST_DATATYPE" : {
11     "message" : [ "Cannot cast <sourceType> to <targetType>." ],
12     "sqlState" : "22005"
13   },
```

error-classes.json

SPIP: Standardize Exception Messages in Spark in PySpark (SPARK-36953)

Productivity

Error classification

```
>>> try:
...     spark.sql("SELECT a")
... except AnalysisException as e:
...     # Exception has getSqlState and getErrorClass methods.
...     if e.getErrorClass() == "MISSING_COLUMN":
...         # Error handling
```

Error handling example

Standardized errors enables **the integration with other alerting system** such as Sentry

Even **translation**, **error mapping**, etc.

What’s next?

Usability

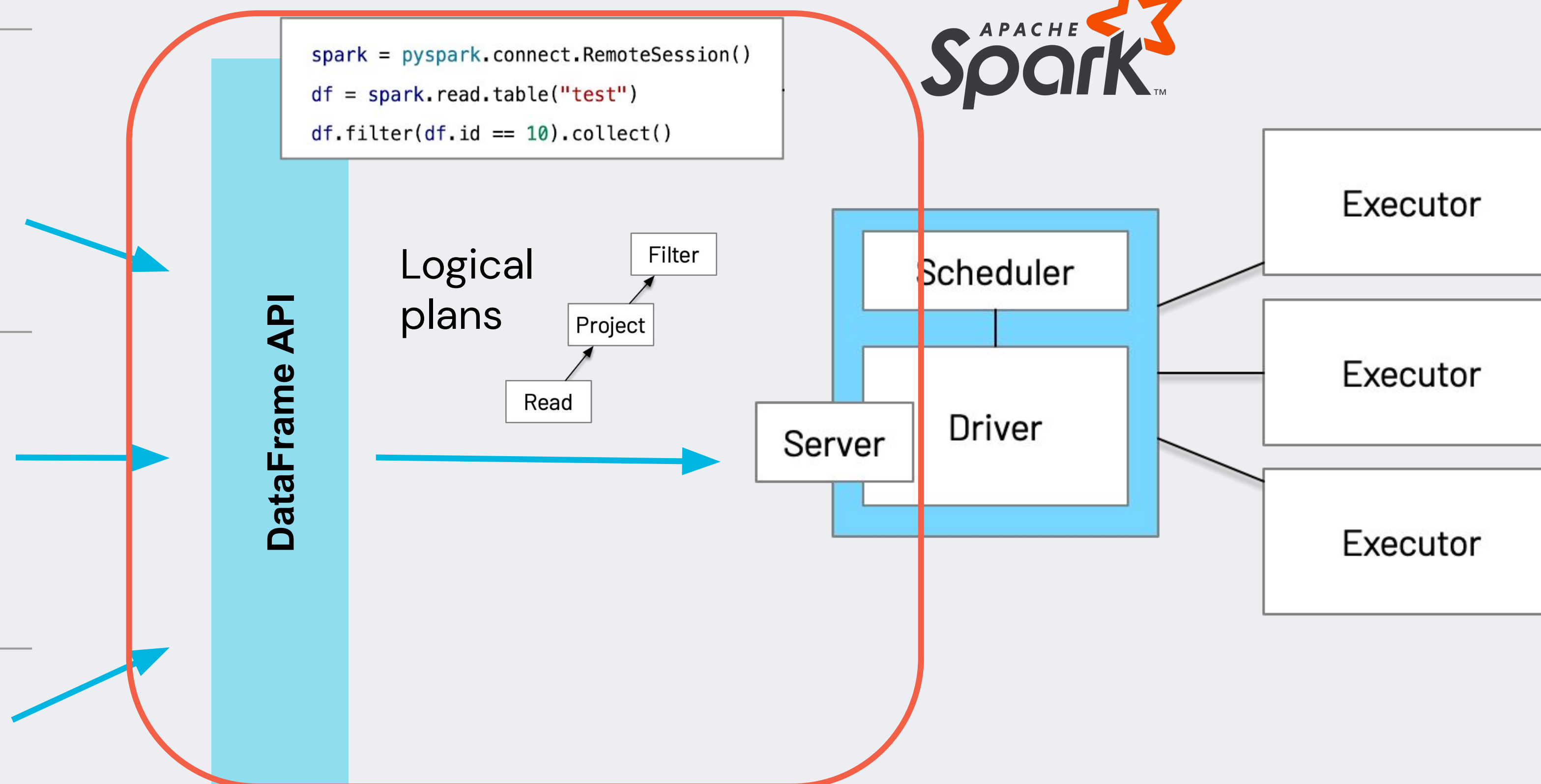
Spark Connect project

Applications

Modern data application

IDEs / Notebooks

Programming Languages / SDKs



Usability

Spark Connect project

What does it mean to end-users? It opens a lot of possibilities

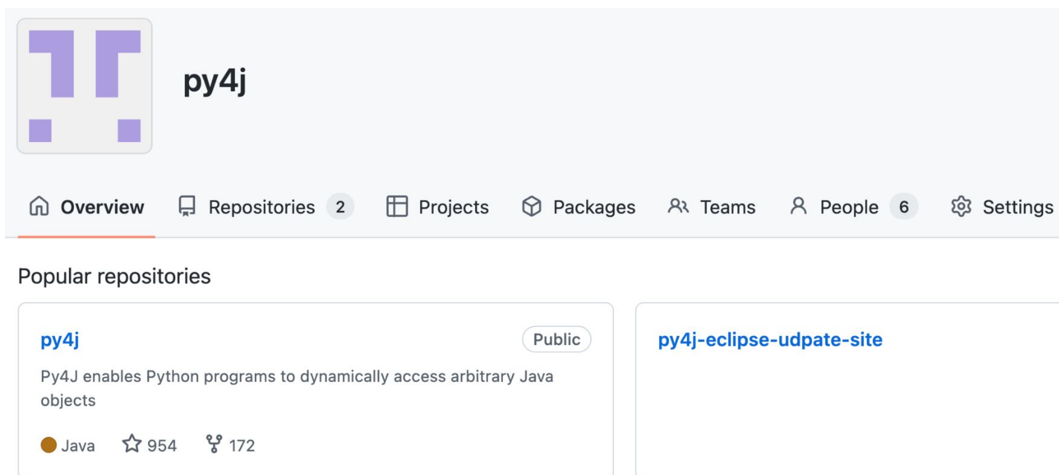
- Dispatch Driver to the cluster, Spark as a service
  - Enables IDE, notebooks and interactive shell with distributing the computation from Driver to a cluster
- Support other languages
  - Provides GRPC protocol for the remote session
  - C#, Go, …
- Security
  - Decoupling of the client inside the Spark runtime environment.
- Upgradability
  - Seamless upgrades between Spark versions

See also SPIP: Spark Connect - A client and server interface for Spark (SPARK-39375)

Usability

Py4J improvement

Brand new Py4J organization



Brand new Py4J maintainers

	Github	Organization
Barthelemy Dagenais	@bartdag	@Power Go
Hyukjin Kwon	@HyukjinKwon	@Databricks
Haejoon Lee	@itholic	@Databricks
Josh Rosen	@JoshRosen	@Databricks
Takuya Ueshin	@ueshin	@Databricks
Xinrong Meng	@xinrong-databricks	@Databricks
Yuchen Huo	@yuchenhuo	@Databirkcs

Usability

Py4J improvement

Short term plan

- PySpark issues
  - Resource leak issue (py4j/py4j#471)
  - Notebook interruption issue (py4j/py4j#440, SPARK-37004)
- **Python 3.10** support
  - py4j/py4j#477
- Support Py4J community
  - py4j/py4j#482, py4j/py4j#487, …
- Automated release process
  - py4j/py4j#466, py4j/py4j#463, …

Long term plan

- Related issues for **NumPy support** in PySpark
  - py4j/py4#163
- **JDK 11** and **17** compliance
  - py4j/py4j#485
- **Py4J 1.0** release
- Security features
  - Prevent arbitrary accesses to JVM

Usability

```
>>> spark.createDataFrame(np.array([1, 2])).collect()
[Row(value=1), Row(value=2)]
>>> spark.createDataFrame(np.array([[1, 2], [3, 4]])).collect()
[Row(_1=1, _2=2), Row(_1=3, _2=4)]
```

DataFrame creation, one/two-dimensional NumPy arrays

```
>>> from pyspark.sql.functions import lit
>>> lit(np.int64(1))
Column < '1' >
```

NumPy instances as parameters

Usability

Better docstrings

DataFrame.union(*other*) [source]

Return a new DataFrame containing union of rows in this and another DataFrame.

This is equivalent to *UNION ALL* in SQL. To do a SQL-style set union (that does deduplication of elements), use this function followed by distinct().

Also as standard in SQL, this function resolves columns by position (not by name).

New in version 2.0.

DataFrame.union docs w/o examples

Examples

```
>>> df.columns
['age', 'name']
```

```
>>> df.columns
NameError: name 'df' is not defined
```

No self-contained examples

Problem? No example or parameter documentation

Can't run when I just copy and paste the example

Usability

Better docstrings

Examples

Constructing DataFrame from a dictionary.

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = ps.DataFrame(data=d, columns=['col1', 'col2'])
>>> df
   col1  col2
0     1     3
1     2     4
```

pyspark.pandas.DataFrame docs

```
>>> d = {'col1': [1, 2], 'col2': [3, 4]}
>>> df = ps.DataFrame(
...     data=d, columns=['col1', 'col2'])
>>> df
  col1  col2
0     1     3
1     2     4
```

Running the example

Should have documentation that have self-contained examples with detailed explanations for each (SPARK-32082)

Performance

Source-native index in Pandas API on Spark

```
>>> ps.read_parquet(
...     "path").spark.explain()
== Physical Plan ==
AttachDistributedSequence ...
+- FileScan parquet ...
```

Read with default index

```
>>> ps.read_parquet("path",
...     index_col="id").spark.explain()
== Physical Plan ==
+- FileScan parquet ...
```

Read with user-specified index

Problem? **Default index** computation is potentially **expensive**

Modern sources such as DBMS, Parquet and ORC have **its own index within the storage**, that Pandas API on Spark can leverage **without default index computation**.

Performance

Optimized createDataFrame with Arrow

```
>>> import pandas as pd
>>> spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", True)
>>> spark.createDataFrame(pd.DataFrame({'a': [1, 2, 3, 4]})).explain()
```

Create a DataFrame from a pandas DataFrame with Arrow

```
== Physical Plan ==
  *(1) Scan ExistingRDD arrow[a#0L]
```

Before

```
== Physical Plan ==
 LocalTableScan [a#0L]
```

After

Local table scan that performs **the computation fast (~1.7x)** in driver side for **small data**.

Feature parity

Observable API for Structured Streaming

```
class StreamingQueryListener(ABC):
    @abstractmethod
    def onQueryStarted(self, event: "QueryStartedEvent") -> None:
    @abstractmethod
    def onQueryProgress(self, event: "QueryProgressEvent") -> None:
    @abstractmethod
    def onQueryTerminated(self, event: "QueryTerminatedEvent") -> None:
```

pyspark.sql.streaming.StreamingQueryListener class

Streaming query listener will be available in PySpark

Feature parity

Observable API for Structured Streaming

```
>>> spark.streams.addListener(MyListener())
>>>
>>> my_observed_csv = my_csv.observe(
...     "metric",
...     count(lit(1)).alias("cnt"),  # num of processed rows
...     count(col("_corrupt_record")).alias("malformed"))  # num of malformed rows
```

CSV malformed records as metric to the streaming query listener

DataFrame.observe & listener will enable **the integration with other systems** like Sentry

See also How to Monitor Streaming Queries in PySpark

Feature parity

Latest pandas API

Pandas API on Spark follows the latest pandas releases (SPARK-38819)

- Each release of PySpark in Apache Spark will have one matched version
- Development branch follows the latest API, fixes and improvements
  - Apache Spark 3.3.0 is matched with pandas 1.3.0

Roadmap is in progress, see also
issues.apache.org/jira/SPARK-38819#comment-17543450

Key takeaways

PySpark in Apache Spark 3.3

- Pandas API on Spark: **faster default index** with **better API coverage**
    - 2 ~ 3 times faster default index
    - Full list of supported API
- datetime.timedelta support
    - Everywhere in PySpark
- **PyArrow batch interaction** within PySpark DataFrame
    - View PySpark DataFrame as PyArrow RecordBatches
- **Python and Pandas UDF profiler** for your function
    - 'spark.python.profile' configuration and SparkContext.show_profiles

PySpark in future Apache Spark

- **Spark connect** project
  - Interactive shell with the driver dispatched to the cluster, Spark as a service
- Native **NumPy support**
  - Everywhere in PySpark
- **Further optimization on default index** in Pandas API on Spark
  - Remove default index creation by leveraging sources' index
- **Observable API** for Structured Streaming
  - Alterting, integration with other external alerting systems
- **Latest pandas API support on PySpark**

DATA+AI SUMMIT 2022

Thank you

Hyukjin Kwon

Software Engineer, Databricks

Xinrong Meng

Software Engineer, Databricks