databricks

# Deep Dive into the New Features of Apache Spark 3.2 and 3.3


Daniel Tenedorio dtenedor

Wenchen Fan cloud-fan

Xiao Li gatorsmile


Data + AI Summit 2022

# About Us

The Spark team at 

Daniel Tenedorio
(Github: dtenedor)

Wenchen Fan
(Github: cloud-fan)

Xiao Li
(Github: gatorsmile)

databricks

The Data and AI Company

# Lakehouse

One simple platform to unify all of your data, analytics, and AI workloads

## CUSTOMERS

**5000+**

Across the globe

## ORIGINAL CREATORS

APACHE Spark™ DELTA LAKE mlflow™

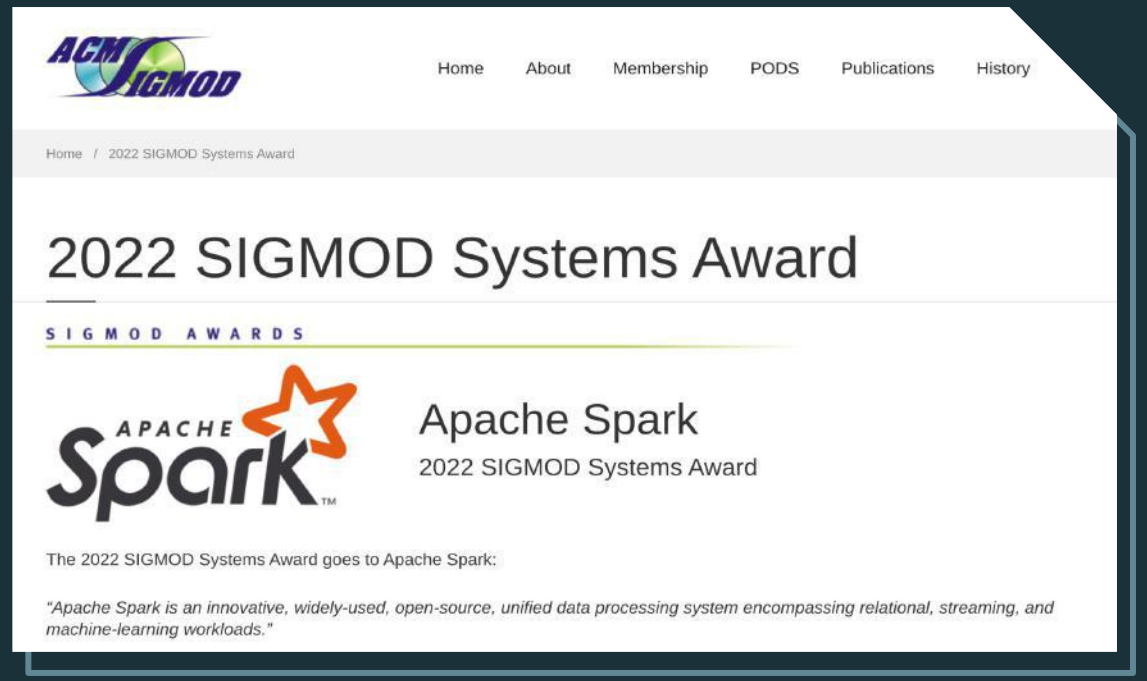
Award

Growth

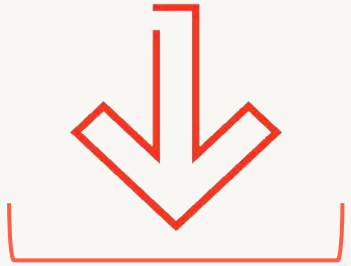
# Community Updates

# 2022 ACM SIGMOD Systems Award

"Recognize system with significant impact on the theory and practice of large scale data systems"

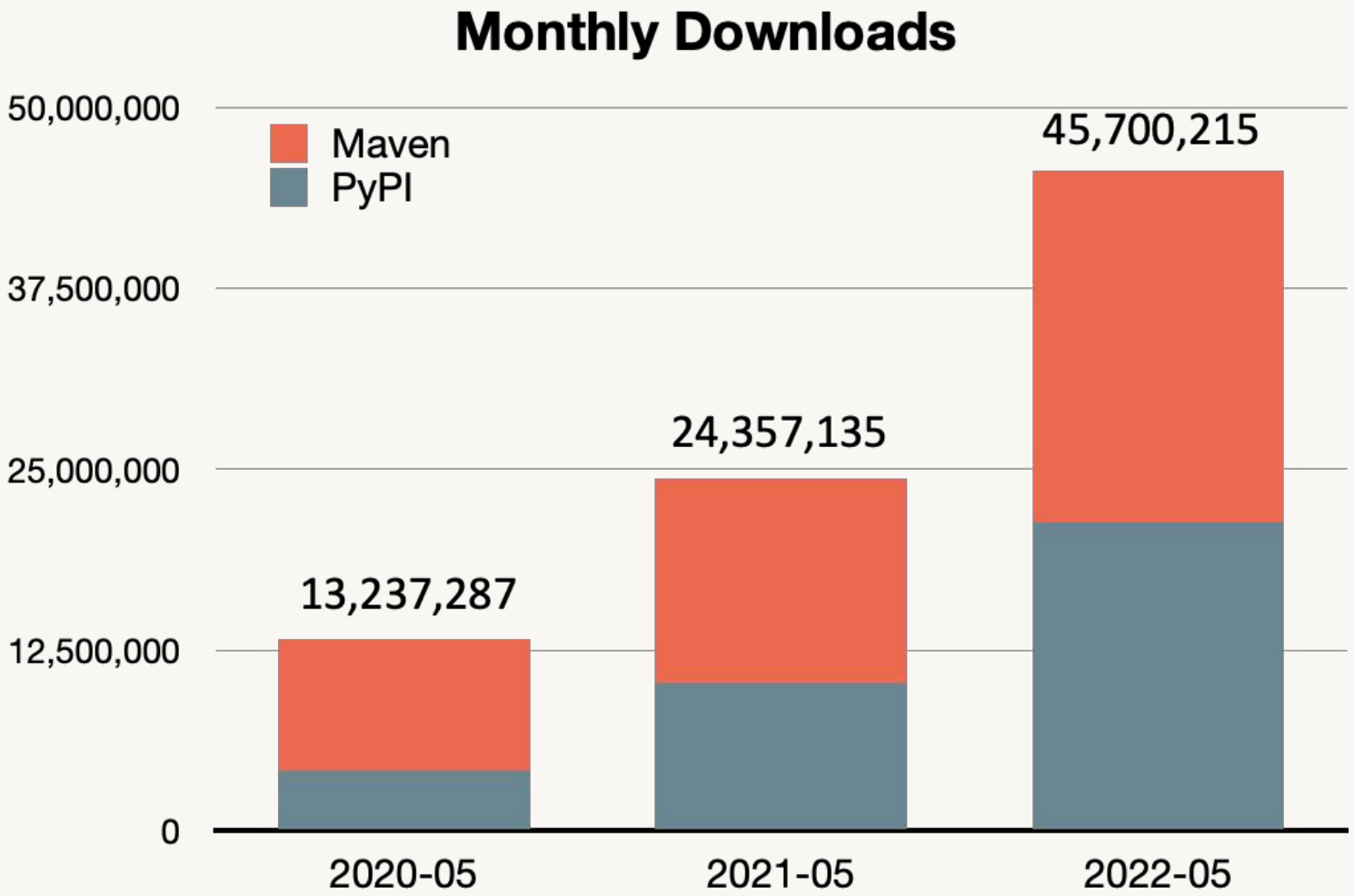



"Apache Spark is an innovative, widely-used, open-source, unified data processing system encompassing relational, streaming, and machine-learning workloads."



# 45 Millions

Monthly downloads
(maven and PyPI)



# Growth of PySpark

204

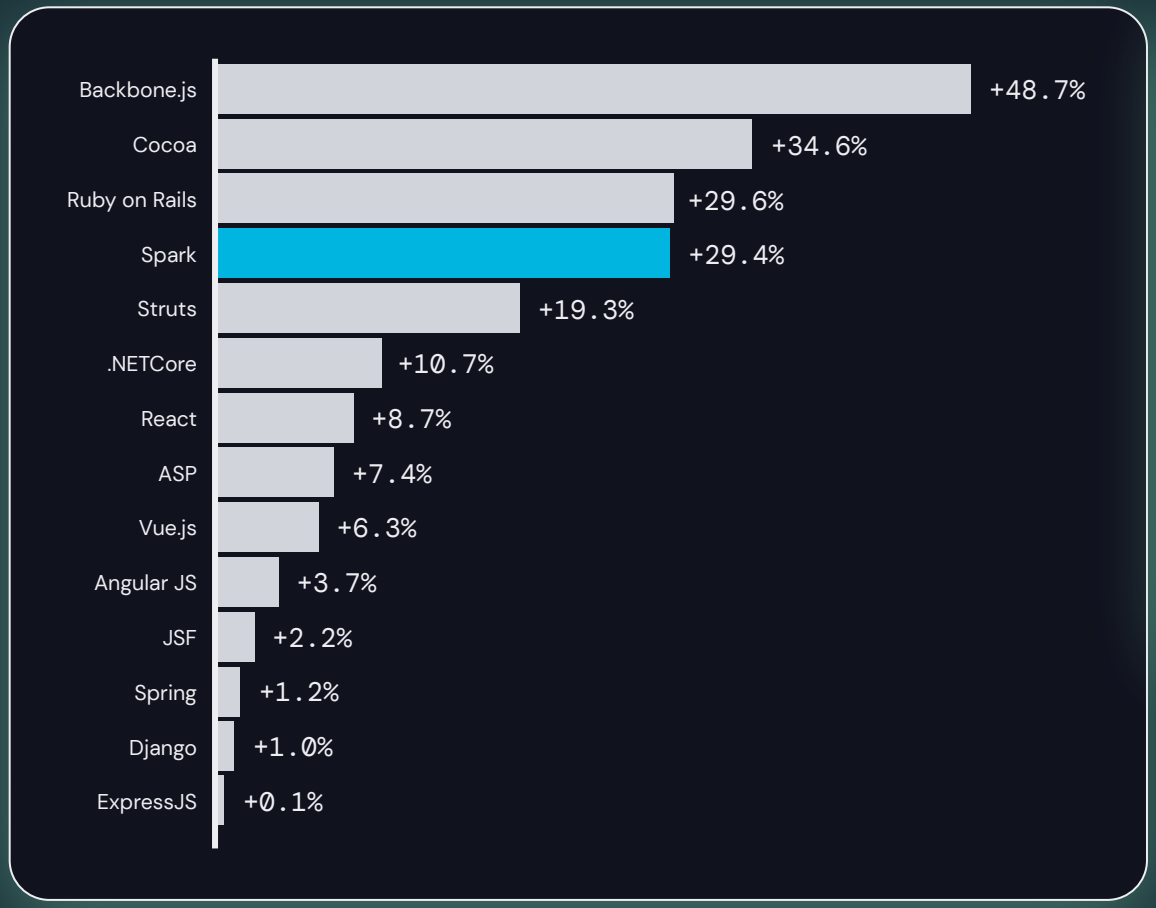
Number of Countries and Regions

(PyPI downloads in the last 12 months)

HackerRank 2020 Developer Survey

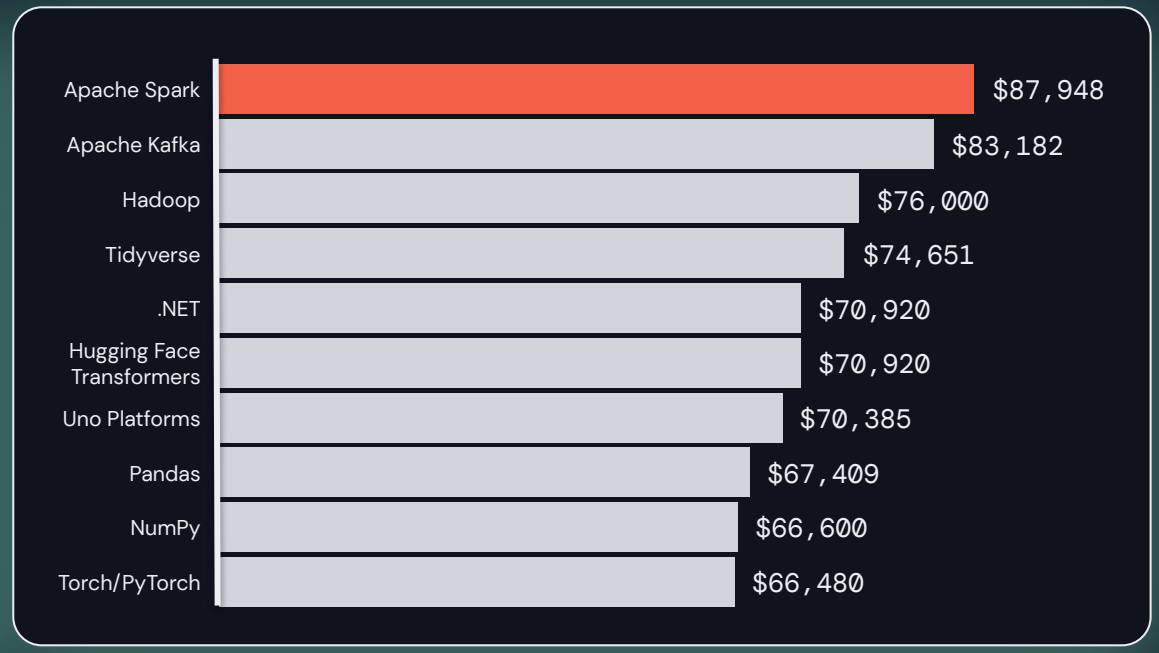
## Salary increase based on frameworks known

Over global average salary ($54,491 USD)




stackoverflow 2022 Developer Survey

## Top paying technologies

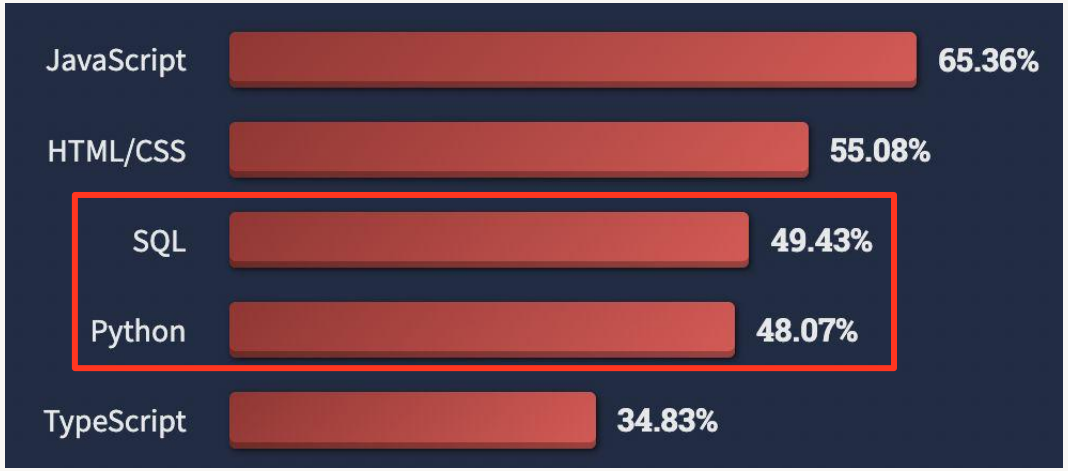




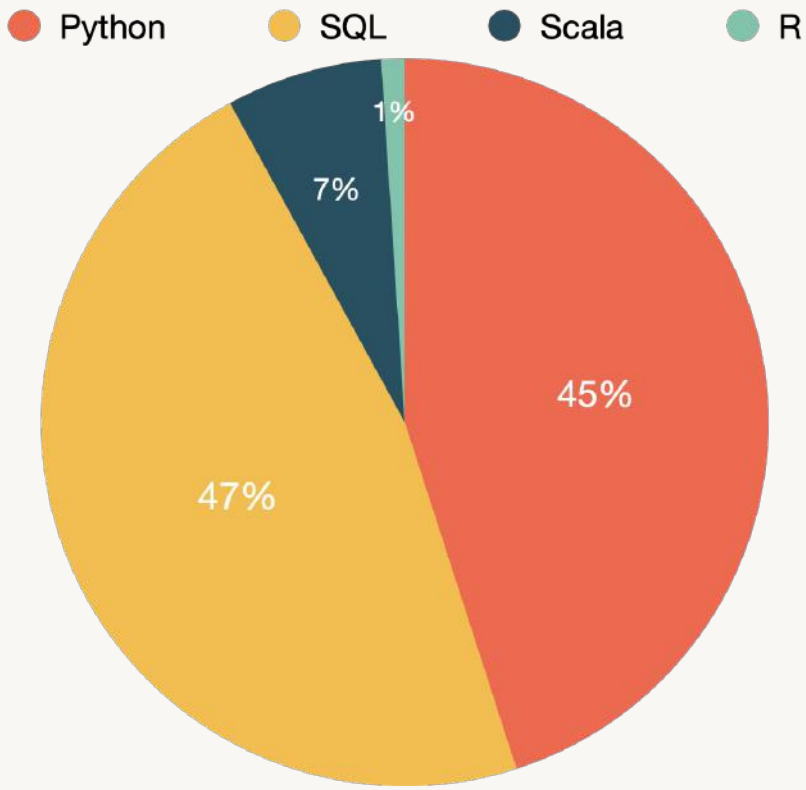
databricks

# SQL and Python

- SQL and Python are the **3rd** and **4th** most popular language in Stack Overflow Developer Survey 2022
- SQL and Python are the **top 2 languages** in Databricks' interactive workloads




Programming, language and scripting
in Stack Overflow Developer Survey 2022




Language Distribution in Databricks' Interactive Workloads

1,600+ JIRAs in Apache Spark 3.3

## Features

ANSI Enhancements

Trigger AvailableNow

Built-in Functions

Hidden File Metadata

## Performance

Runtime Filtering

Whole-stage codegen

QO/AQE Enhancements

Customized K8S Scheduler

## Python

Python/Pandas UDF Profiler

timedelta Support

pandas API Coverage

mapInArrow

## Usability

Error Message Improvements

SQL Configs in Spark UI

Python String Formatter

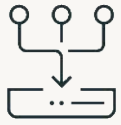
Inline Type Hints

## Extensions

Parquet Vectorized Reader

DSV2 Pushdown

DSV2 APIs



Hive Bucket Writing

## More

Java 17

Log4J 2

Apple Silicon

New Doc for SparkR

1,700+ JIRAs in Apache Spark 3.2

## ANSI SQL Compliance


ANSI Mode GA

Lateral Join

Implicit Type Cast

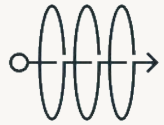
Interval Type

## Extensions

DSV2 API Enhancements

DSV2 Metrics

Parquet 1.12 (Column Index)



Complex Type Support in ORC

## Python

pandas APIs

Visualization and Plotting

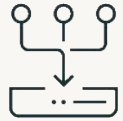
Pythonic Error Handling

Richer Input/Output

## Streaming

RocksDB State Store

Session Window



State Store APIs

Kafka 2.8

## Performance

Compiler Latency Reduction

Adaptive Optimization GA

Whole-stage codegen

AQE + DPP

## More

Explicit Error Classes

Push-based Shuffle

Java 17 Beta

Scala 2.13 Beta

pandas APIs

Visualization and Plotting

Pythonic Error Handling

Richer Input/Output

Inline Type Hints

Python String Formatter

Python/Pandas UDF Profiler

timedelta Support

mapInArrow

# Project Zen: Python Usability

# 2020: Project Zen

## OSS Spark Development Initiatives at Databricks

**Project Zen:** Greatly improve Python usability

- Better error reporting
- API ports from Koalas
- Improved performance
- Pythonic API design

**Adaptive Query Execution:** Cover most current optimizer decisions

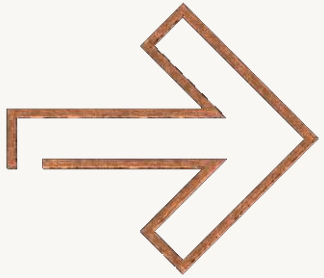
**ANSI SQL:** Run unmodified queries from major SQL engines

SPARK+AI SUMMIT

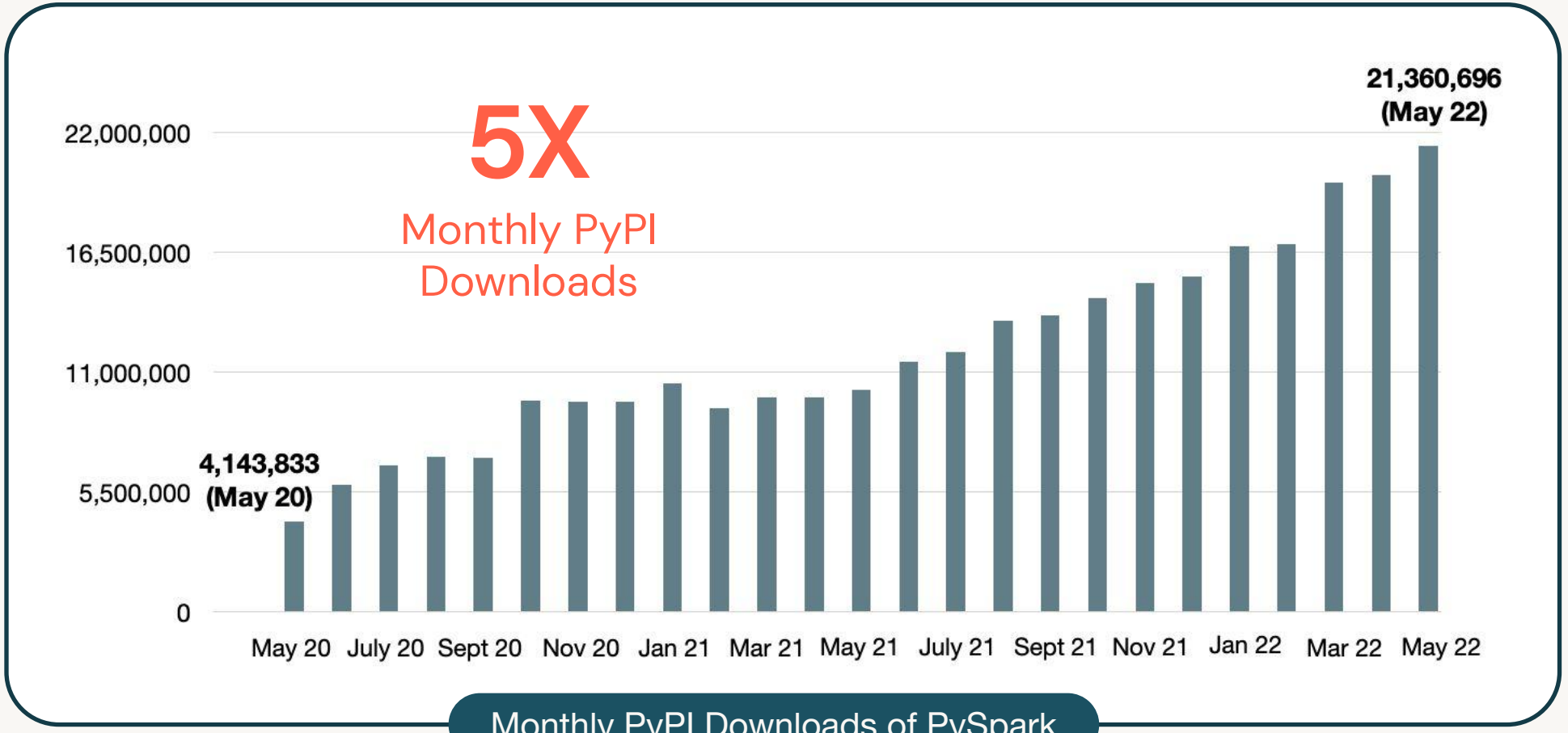
#Datateams #SparkAISummit

Spark + AI Summit 2020

# 2021: pandas APIs on Spark

# 2020–2022: PySpark Adoption

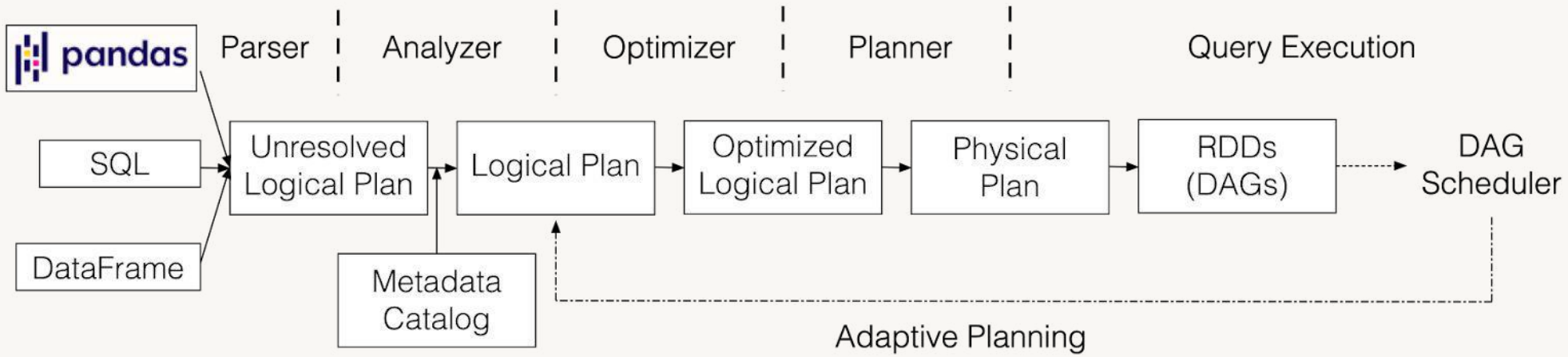



Monthly PyPI Downloads of PySpark

# pandas APIs on Spark

## Performance improvements

- Optimized single-machine performance
- Scalability beyond a single machine



# pandas APIs on Spark

## Performance improvements

New Performance Enhancements of Query Engine in Spark 3.2 & 3.3

Runtime Filtering

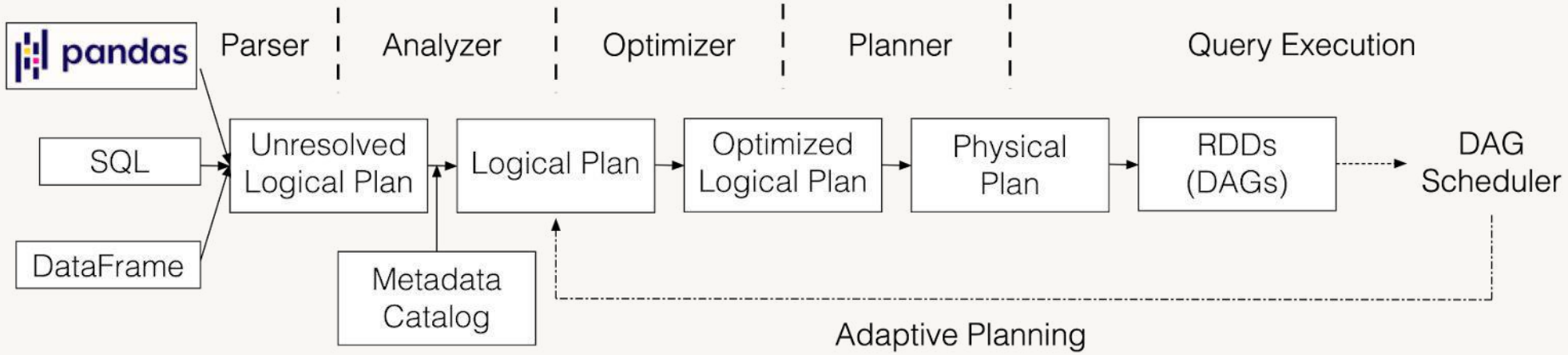
Whole-stage Codegen Improvement

QO/AQE Enhancements

Delta Lake 2.0

Parquet Vectorized Reader

...



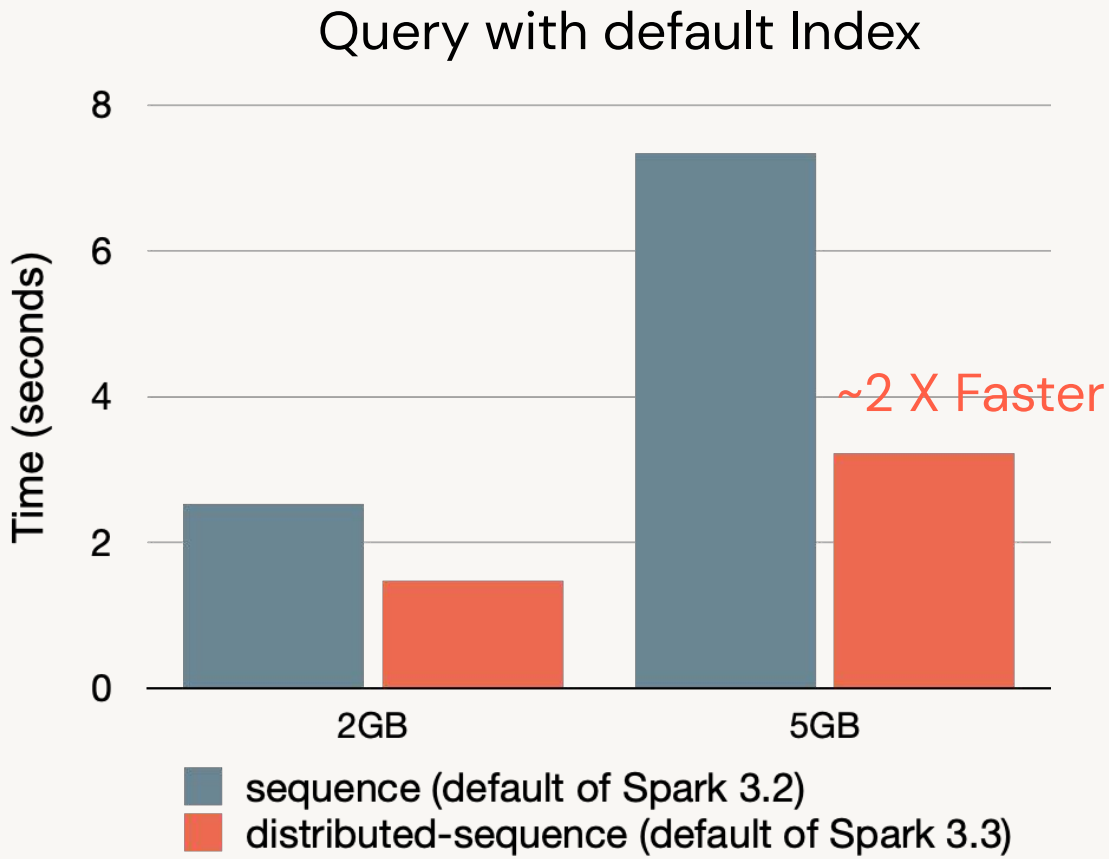
# pandas APIs on Spark

## Performance improvements

Distributed sequence index by default (SPARK-37649)

- Scale out better in distributed systems

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



Search the docs ...

Python Package Management
Spark SQL
**Pandas API on Spark**
Options and settings
From/to pandas and PySpark DataFrames
Transform and apply a function
Type Support in Pandas API on Spark
Type Hints in Pandas API on Spark
From/to other DBMSes
Best Practices
**Supported pandas API**
FAQ

# Supported pandas API

The following table shows the pandas APIs that implemented or non-implemented from pandas API on Spark. Some pandas API do not implement full parameters, so the third column shows missing parameters for each API.

- 'Y' in the second column means it's implemented including its whole parameter.
- 'N' means it's not implemented yet.
- 'P' means it's partially implemented with the missing of some parameters.

All API in the list below computes the data with distributed execution except the ones that require the local execution by design. For example, DataFrame.to_numpy() requires to collect the data to the driver side.

If there is non-implemented pandas API or parameter you want, you can create an Apache Spark JIRA to request or to contribute by your own.

The API list is updated based on the pandas 1.3 official API reference.

## DataFrame API

| API | Implemented | Missing parameters |
|---|---|---|
| `T()` | Y | |
| `abs()` | Y | |
| `add()` | P | `axis, level, fill_value` |

Better coverage in Spark 3.3 after the initial support in Spark 3.2

Compatibility with pandas 1.3

Full list of supported API [DOC]

# pandas APIs on Spark

## Better coverage in Spark 3.3 after the initial support in Spark 3.2

- ps.merge_asof (SPARK-36813)
- 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)

# pandas APIs on Spark

## ASOF JOIN: ps.merge_asof (SPARK-36813)

For each row in the left DataFrame:

- A “backward” search selects the last row in the right DataFrame whose ‘on’ key is less than or equal to the left‘s key. [Default]
- A “forward” search selects the first row in the right DataFrame whose 'on' key is greater than or equal to the left's key.
- A "nearest" search selects the row in the right DataFrame whose 'on‘ key is closest in absolute distance to the left's key.

Optionally match on equivalent keys with 'by' before searching with 'on'.

- Join records that have no exact match

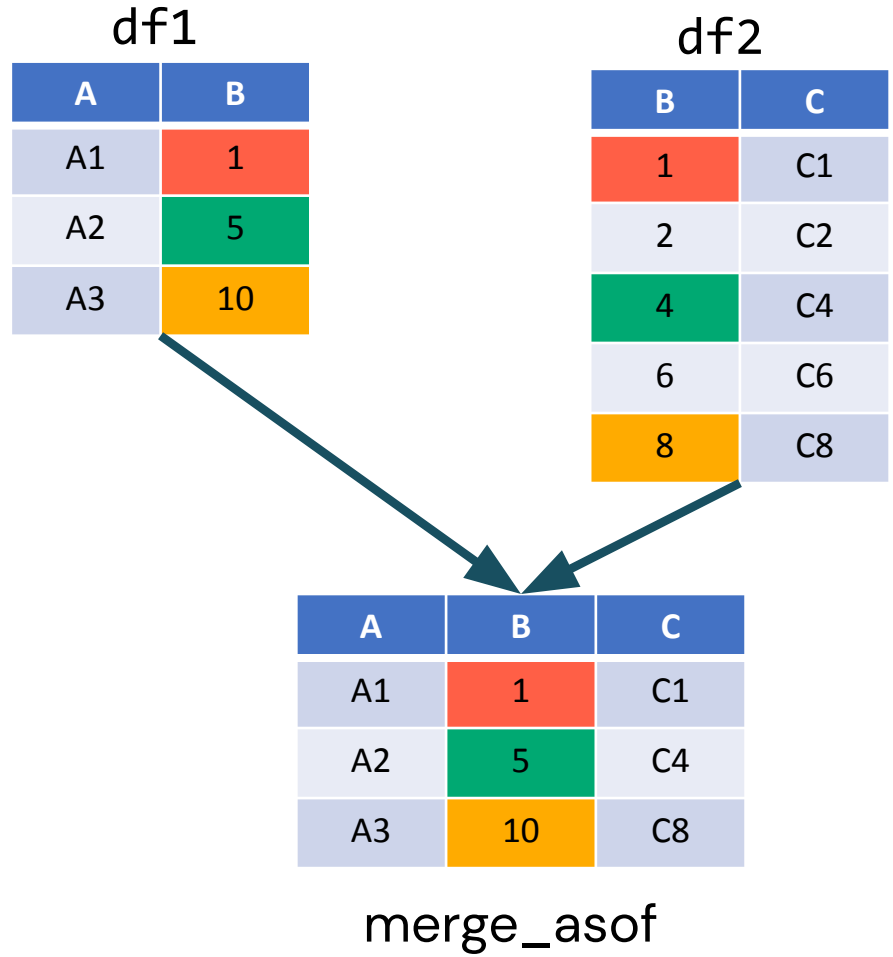
- An ordered left-join except that we match on nearest key rather than equal keys.

# pandas APIs on Spark

## ASOF JOIN: ps.merge_asof (SPARK-36813)

Join on column B

```
from pyspark import pandas as ps

df1 = ps.DataFrame(
    {"A": ['A1', 'A2', 'A3'],
     "B": [1, 5, 10]})
df2 = ps.DataFrame(
    {"B": [1, 2, 4, 6, 8],
     "C": ['C1', 'C2', 'C4', 'C6', 'C8']})

ps.merge_asof(df1, df2, on='B')
```

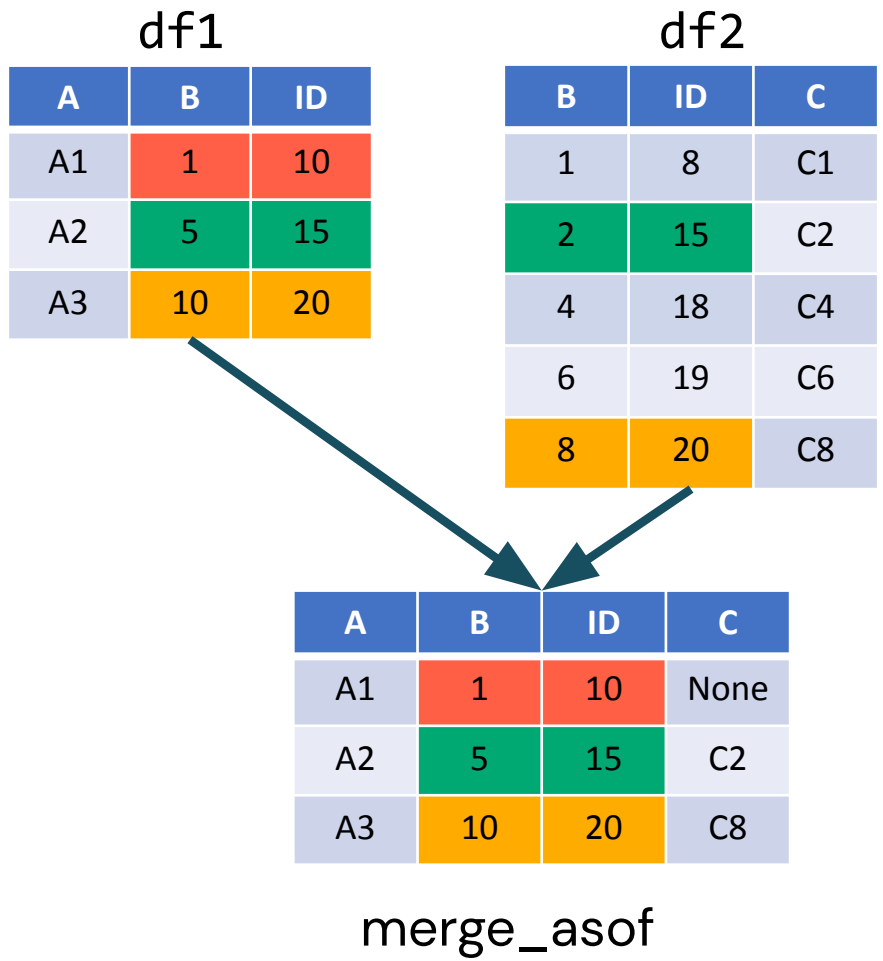


# pandas APIs on Spark

## ASOF JOIN: ps.merge_asof (SPARK-36813)

Join on column B by column ID

```
from pyspark import pandas as ps

df1 = ps.DataFrame(
    {"A": ['A1', 'A2', 'A3'],
     "B": [1, 5, 10],
     "ID": [10, 15, 20]})
df2 = ps.DataFrame(
    {"B": [1, 2, 4, 6, 8],
     "ID": [8, 15, 18, 19, 20],
     "C": ['C1', 'C2', 'C4', 'C6', 'C8']})

ps.merge_asof(df1, df2, on='B', by='ID')
```



# pandas APIs on Spark

## ASOF JOIN: ps.merge_asof (SPARK-36813)

Implementation via a scalar subquery

```
SELECT LEFT.*, __right__.*
FROM (
    SELECT
        LEFT.*,
        (
            SELECT min_by(struct(RIGHT.*), LEFT.t - RIGHT.t) AS __nearest_right__
            FROM RIGHT
            WHERE condition AND LEFT.t >= RIGHT.t AND RIGHT.t >= LEFT.t - tolerance
        ) AS __right__
    FROM LEFT
    )
WHERE __right__ IS NOT NULL
```

# Timedelta

## datetime.timedelta support

- A duration expressing the difference between two date, time, or datetime instances to microsecond resolution.
- Can be used in any place including Python/Pandas UDFs
- Internally mapped to **DayTimeIntervalType** in Spark SQL (SPARK-37275)
- Example:

```
ps.merge_asof(
    df1, df2, on='time', by='id', tolerance=datetime.timedelta(days=1))
```

# Timedelta

Join on column B by column ID in tolerance of `datetime.timedelta(days=1))`

df1

| date | id | quantity |
|---|---|---|
| 2020-1-1 | 1 | 100 |
| 2020-1-1 | 2 | 50 |
| 2020-1-2 | 1 | -50 |
| 2020-1-30 | 2 | 90 |

df2

| date | Id | price |
|---|---|---|
| 2019-12-31 | 1 | 100.0 |
| 2020-1-2 | 1 | 105.0 |
| 2020-1-2 | 2 | 195.0 |

```
from pyspark import pandas as ps; from datetime import date

df1 = ps.DataFrame(
   {"date": [date(2020, 1, 1), date(2020, 1, 1), date(2020, 1, 2), date(2020, 1, 2)],
    "id": [1, 2, 1, 2],
    "quantity": [100, 50, -50, 90]})
df2 = ps.DataFrame(
   {"date": [date(2019, 12, 31), date(2020, 1, 2), date(2020, 1, 2)],
    "id": [1, 1, 2],
    "price": [100.0, 105.0, 195.0]})

ps.merge_asof(df1, df2, on=date', by='id', tolerance=datetime.timedelta(days=1))
```

| date | id | quantity | price |
|---|---|---|---|
| 2020-1-1 | 1 | 100 | 100.0 |
| 2020-1-1 | 2 | 50 | NaN |
| 2020-1-2 | 1 | -50 | 105.0 |
| 2020-1-30 | 2 | 90 | NaN |

merge_asof

In Development

# pandas APIs on Spark

## Better coverage in the next releases

- ps.Series.autocorr
- ps.Series.duplicated
- ps.Series.interpolate
- ps.Series.resample
- ps.DataFrame.corrwith
- ps.DataFrame.interpolate
- ps.DataFrame.resample
- ps.groupby.GroupBy.ewm

# pandas APIs on Spark

## To pandas users:

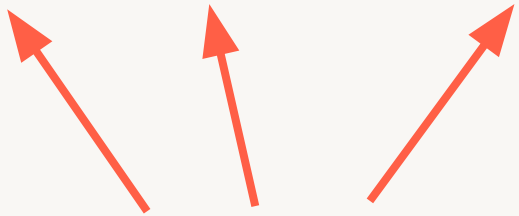
- A drop-in replacement for pandas
- Scale to fault-tolerant clusters of thousands of machines.
- Support streaming applications via foreachBatch APIs
- Integrate with the scalable Spark Mllib
- Integrate with the whole Spark ecosystem (e.g., Delta Lake, MLflow)
- Query data via SQL language

# PySpark

## Python standard string formatter in SQL [PEP 3101]

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

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



column id

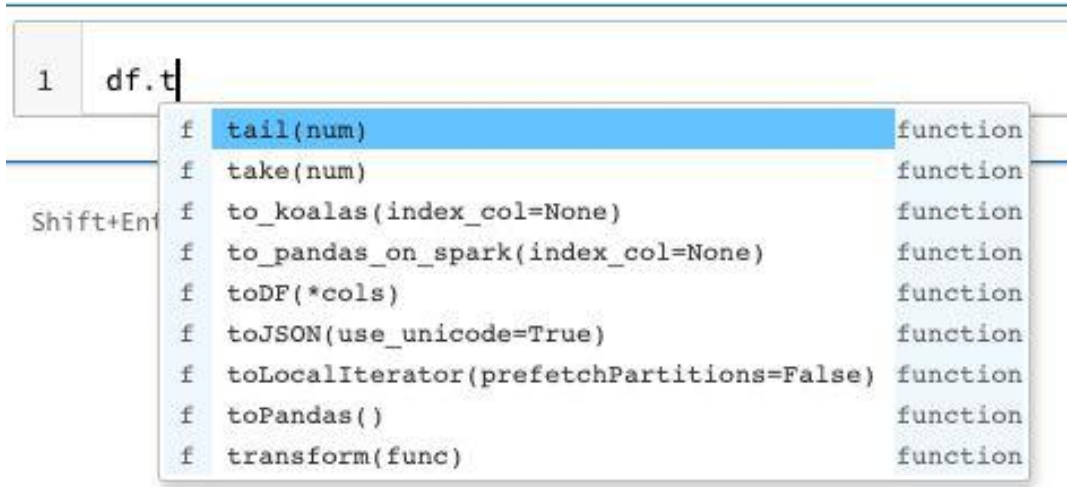
# PySpark

## Better autocompletion

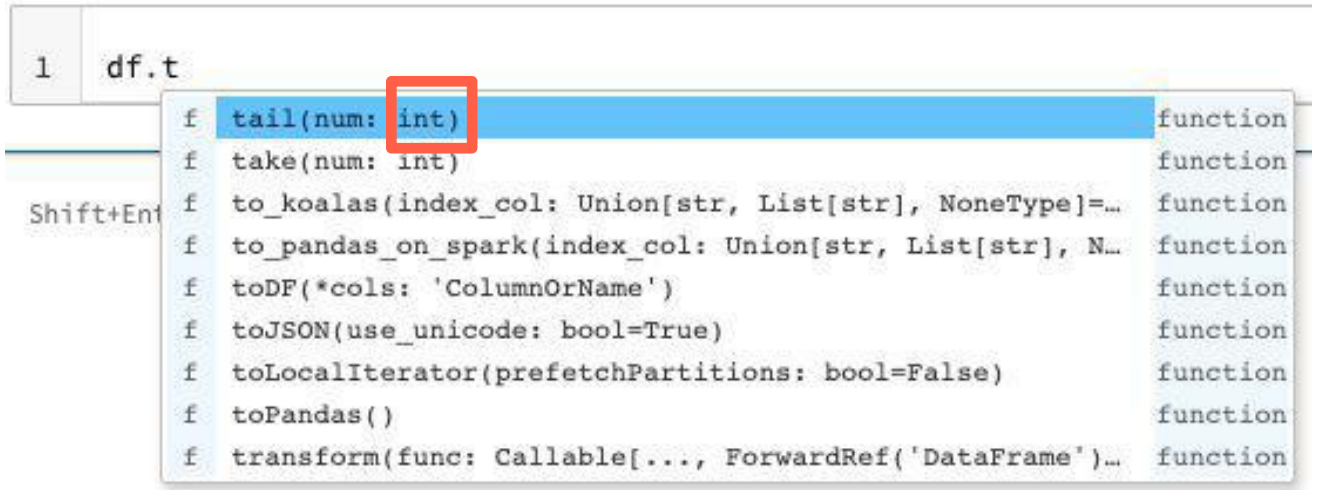
- Parameter types

In Databricks notebook,
<Tab> : Autocomplete

Spark 3.2




Spark 3.3



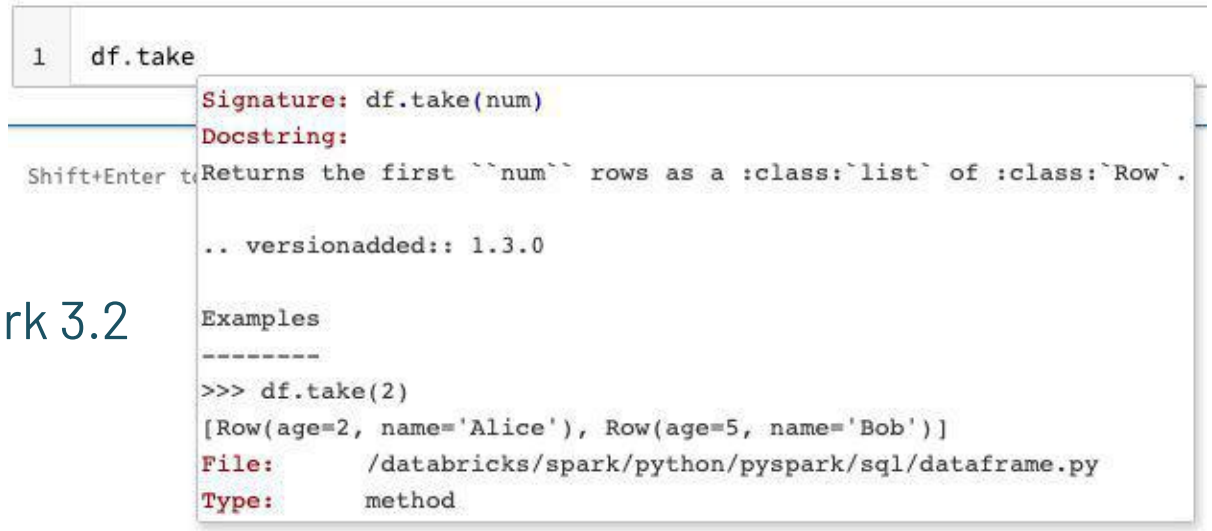
# PySpark

## Better documentation

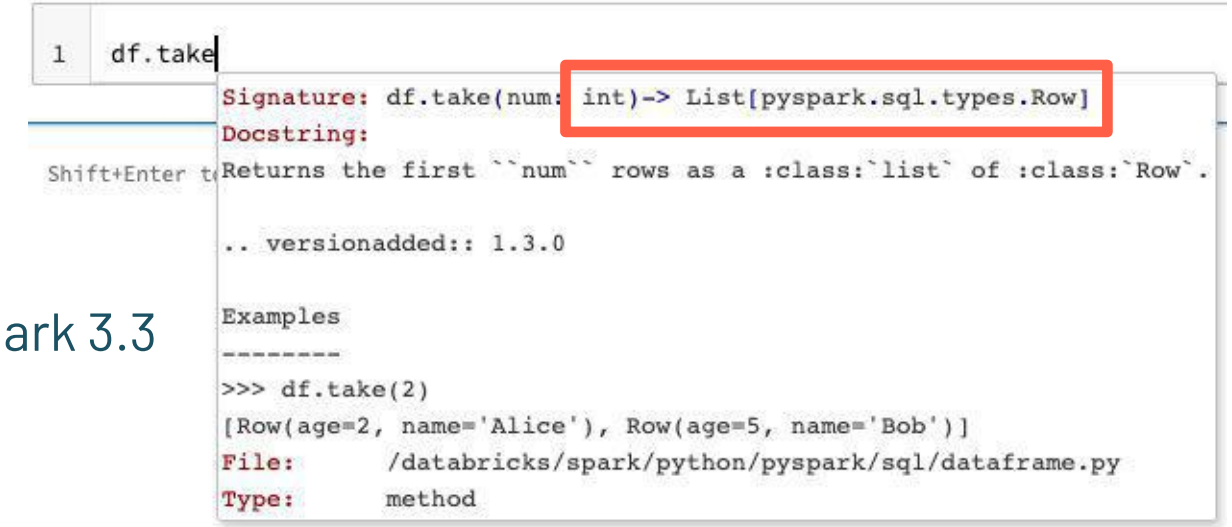
- Parameter types
- Return types

In Databricks notebook,
<Shift> + <Tab> : Docstring

Spark 3.2




Spark 3.3



# PySpark

## 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()
```

```
============================================================
Profile of UDF<id=12>
============================================================
         30 function calls in 10.009 seconds

   Ordered by: internal time, cumulative time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
       10   10.009    1.001   10.009    1.001 {built-in method time.sleep}
       10    0.000    0.000   10.009    1.001 <command-1635211799533119>:2(<lambda>)
       10    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

ANSI Mode GA

Lateral Subqueries

Implicit Type Cast

Error Class

Interval Type

# ANSI SQL Compliance

# Lateral Subqueries

To cross-reference the preceding from_items in the same FROM clause. Such a construct is called a correlated or dependent join. A correlated join cannot be a RIGHT OUTER JOIN or a FULL OUTER JOIN.

```
SELECT *
FROM tab1,
  LATERAL (
    SELECT *
    FROM tab2
    WHERE tab1.id = tab2.id
  )
```

# Lateral Subqueries

Also, make your queries simpler and more efficient, like CTE

## Before

```
SELECT
    a1 + a2 + b3 + b4,
    (a1 + a2) * (b3 + b4),
    udf1(a1 + a2 + b3 + b4)
FROM tab1
```

## After

```
SELECT
    sumAB, sumA * sumB, udf1(sumAB)
FROM tab1,
LATERAL (SELECT
    (a1 + a2) as sumA),
LATERAL (SELECT
    (b3 + b4) as sumB),
LATERAL (SELECT
    (sumA + sumB) as sumAB)
```

# Interval Type

Represents intervals of time either on a scale of seconds or months

- day-time interval

```
SELECT INTERVAL -'200:13:50.3' HOUR TO SECOND;
SELECT CAST('11 23:4:0' AS INTERVAL DAY TO SECOND);
```

- year-month interval

```
SELECT INTERVAL '100-00' YEAR TO MONTH;
SELECT INTERVAL '-3600' MONTH;
```

- Examples

```
SELECT * FROM tab WHERE date1 - date2 > INTERVAL 2 DAYS
SELECT (now() - INTERVAL '3 hours 20 minutes' - INTERVAL '1 year')
```

# ANSI Mode

## Why it matters?

- Data quality and application sanity
- Reduce lock-in/improved portability
- Ease of integration with BI and external tools

spark.sql.ansi.enabled
(default: false)

Follows the SQL standard in how it deals with certain arithmetic operations and type conversions, similar to most databases and data warehouses.
Following this standard promotes better data quality, integrity, and portability.

| Operator | Description | Example | ANSI_MODE = true | ANSI_MODE = false |
|---|---|---|---|---|
| dividend / divisor | Returns dividend divided by divisor. | 1/0 | Runtime error | NULL |
| - expr | Returns the negated value of expr. | -(-128y) | Runtime error | -128y (Overflow) |
| expr1 - expr2 | Returns the subtraction of expr2 from expr1. | -128y - 1y | Runtime error | 127y (Overflow) |
| expr1 + expr2 | Returns the sum of expr1 and expr2. | 127y + 1y | Runtime error | -128y (Overflow) |
| dividend % divisor | Returns the remainder after dividend / divisor. | 1 % 0 | Runtime error | NULL |
| multiplier * multiplicand | Returns multiplier multiplied by multiplicand. | 100y * 100y | Runtime error | 16y (Overflow) |
| arrayExpr[index] | Returns the element of an arrayExpr at index. | Invalid array index | Runtime error | NULL |
| mapExpr[key] | Returns the value of mapExpr for key. | Invalid map key | Runtime error | NULL |
| divisor div dividend | Returns the integral part of the division of divisor by dividend. | 1 div 0 | Runtime error | NULL |

## What's changing?

- Operators

| Operator | Description | Example | ANSI_MODE = true | ANSI_MODE = false |
|---|---|---|---|---|
| abs(expr) | Returns the absolute value of the numeric value in expr. | abs(-128y) | Runtime error | -128y (Overflow) |
| element_at(mapExpr, key) | Returns the value of mapExpr for key. | Invalid map key | Runtime error | NULL |
| element_at(arrayExpr, index) | Returns the element of an arrayExpr at index. | Invalid array index | Runtime error | NULL |
| elt(index, expr1 [, ...] ) | Returns the nth expression. | Invalid index | Runtime error | NULL |
| make_date(y,m,d) | Creates a date from year, month, and day fields. | Invalid result date | Runtime error | NULL |
| make_timestamp(y,m,d,h,mi,s[,tz]) | Creates a timestamp from fields. | Invalid result timestamp | Runtime error | NULL |
| make_interval(y,m,w,d,h,mi,s) | Creates an interval from fields. | Invalid result interval | Runtime error | NULL |

## What's changing?

- Functions

| Operator | Description | Example | ANSI_MODE = true | ANSI_MODE = false |
|---|---|---|---|---|
| mod(dividend, divisor) | Returns the remainder after dividend / divisor. | mod(1, 0) | Runtime error | NULL |
| pmod(dividend, divisor) | Returns the positive remainder after dividend / divisor. | pmod(1, 0) | Runtime error | NULL |
| size(expr) | Returns the cardinality of expr. | size(NULL) | Runtime error | -1 |
| to_date(expr[,fmt]) | Returns expr cast to a date using an optional formatting. | Invalid expr or format string | Runtime error | NULL |
| to_timestamp(expr[,fmt]) | Returns expr cast to a timestamp using an optional formatting. | Invalid expr or format string | Runtime error | NULL |
| to_unix_timestamp(expr[,fmt]) | Returns the timestamp in expr as a UNIX timestamp. | Invalid expr or format string | Runtime error | NULL |
| unix_timestamp([expr[, fmt]]) | Returns the UNIX timestamp of current or specified time. | Invalid expr or format string | Runtime error | NULL |

# What's changing?

- Functions

| Source type | Target type | Example | ANSI_MODE = true | ANSI_MODE = false |
|---|---|---|---|---|
| Boolean | Timestamp | cast(TRUE AS TIMESTAMP) | Compile time error | 1970-01-01 00:00:00.000001 UTC |
| Date | Boolean | cast(DATE'2001-08-09' AS BOOLEAN) | Compile time error | NULL |
| Timestamp | Boolean | cast(TIMESTAMP'1970-01-01 00:00:00Z' AS BOOLEAN) | Compile time error | FALSE |
| Integral numeric | Binary | cast(15 AS BINARY) | Compile time error | binary representation |

## What's changing?

- Compile-time casting rules

| Source type | Target type | Condition | Example | ANSI_MODE = true | ANSI_MODE = false |
|---|---|---|---|---|---|
| String | Non-string | Invalid input | cast('a' AS INTEGER) | Runtime error | NULL |
| Array, Struct, Map | Array, Struct, Map | Invalid input | cast(ARRAY('1',’2','3') AS ARRAY<DATE>) | Runtime error | NULL |
| Numeric | Numeric | Overflow | cast(12345 AS BYTE) | Runtime error | NULL |
| Numeric | Integral numeric | Truncation | cast(5.1 AS INTEGER) | Runtime error | 5 |

*Use try_cast() for null on error behavior

## What’s changing?

- Runtime casting

# What's changing

## Casting Rules – implicit type coercion

- Non ANSI mode: a set of inconsistent rules
- ANSI mode: determined by type precedence and least common type resolution (SQL data type casting rules) for
  - Type promotion safely expands a type to a wider type.
  - Implicit downcasting narrows a type. The opposite of promotion.
  - Implicit crosscasting transforms a type into a type of another type family.

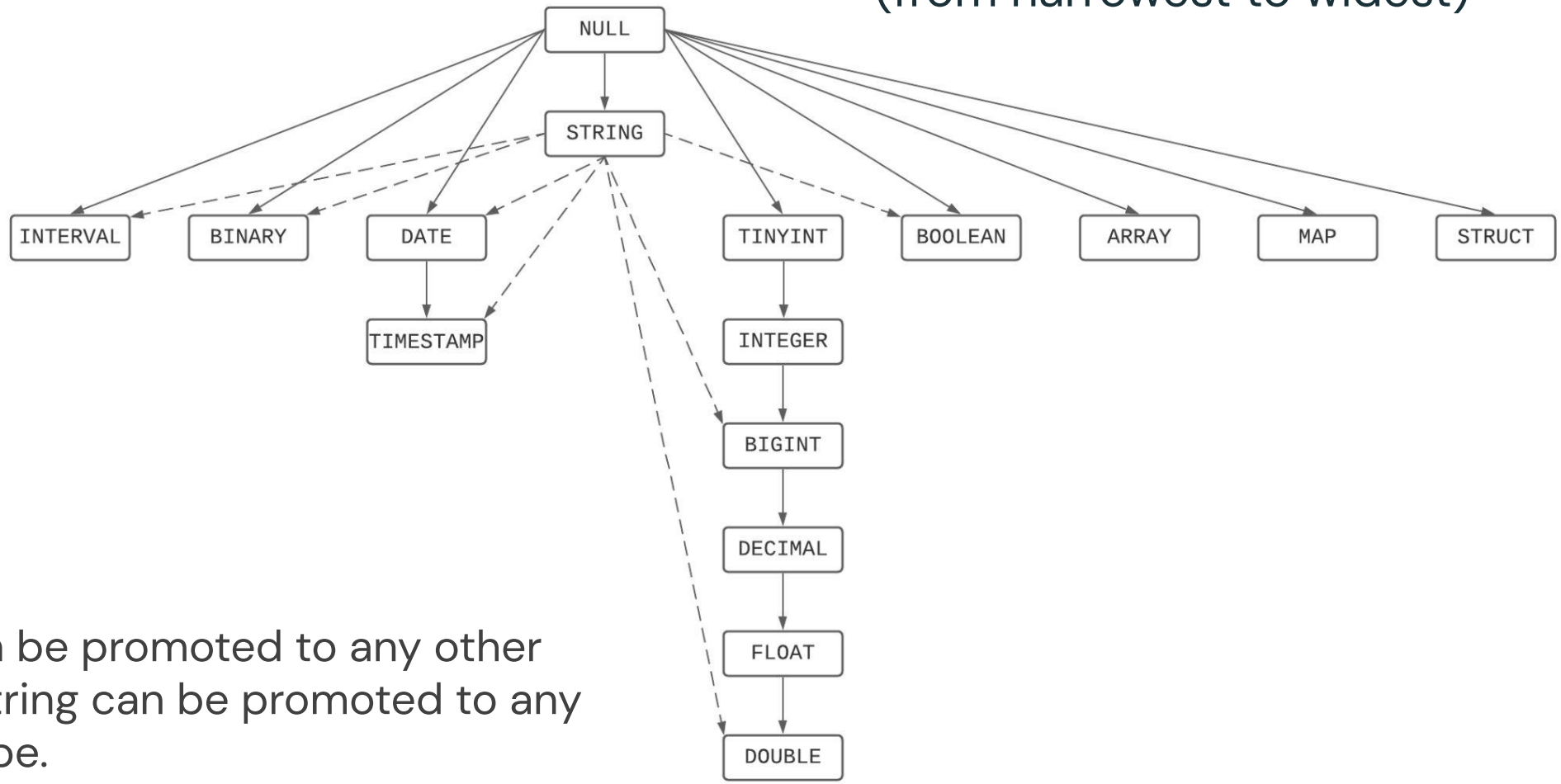
    e.g., string to any simple type

# What's changing

## Casting Rules – implicit type coercion

Type precedence graph

(from narrowest to widest)




Note: NULL can be promoted to any other type, while a String can be promoted to any simple data type.

# What's changing

## Casting Rules – implicit type coercion

```
SELECT 5 = '5.1';
```

- **Non ANSI mode**: true
- **ANSI mode**: throw SparkNumberFormatException: [CAST_INVALID_INPUT] The **value '5.1' of the type "STRING" cannot be cast to "BIGINT"** because it is malformed. Correct the value as per the syntax, or change its target type. Use `try_cast` to tolerate malformed input and return NULL instead. If necessary set "spark.sql.ansi.enabled" to "false" to bypass this error.

  == SQL(line 1, position 8) ==
  select 5 = '5.1'
  ^^^^^^^^^^

# ANSI Mode

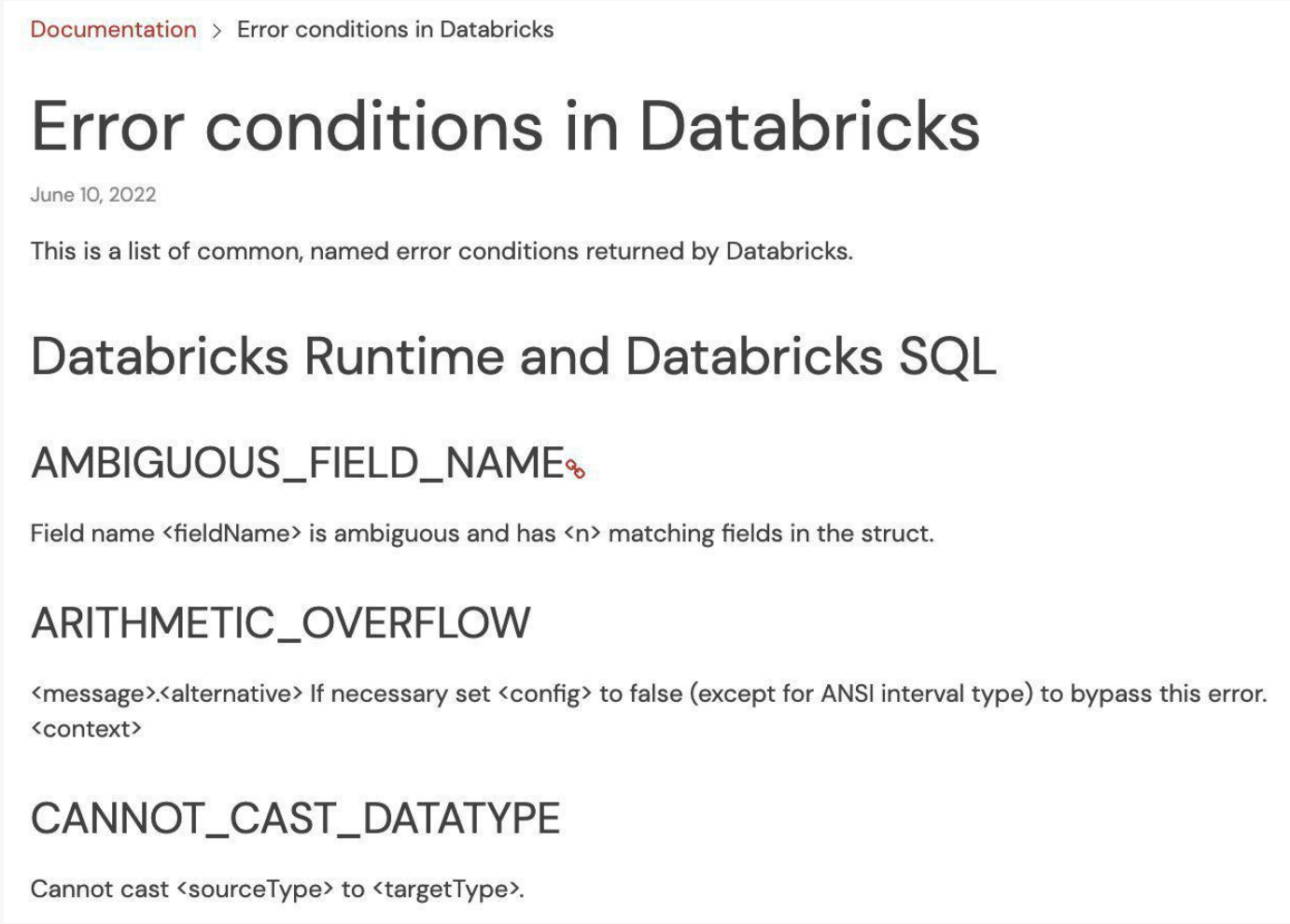
In Development

# Error Framework

## Actionable, stable, documented

A structured protocol and API to pass error structures and retrieve formatted output. Instead of a Java exception dump.

- Better error context exposed to users in common runtime cases
- Error Message Docs
  - Example: DIVIDE_BY_ZERO



Documentation > Error conditions in Databricks

# Error conditions in Databricks

June 10, 2022

This is a list of common, named error conditions returned by Databricks.

## Databricks Runtime and Databricks SQL

### AMBIGUOUS_FIELD_NAME

Field name <fieldName> is ambiguous and has <n> matching fields in the struct.

### ARITHMETIC_OVERFLOW

<message>.<alternative> If necessary set <config> to false (except for ANSI interval type) to bypass this error.
<context>

### CANNOT_CAST_DATATYPE

Cannot cast <sourceType> to <targetType>.

In Development

Documentation > Error conditions in Databricks > DIVIDE_BY_ZERO error class

# DIVIDE_BY_ZERO error class

June 10, 2022

Division by zero. To return NULL instead, use `try_divide`. If necessary set <ansiConfig> to fals
interval type) to bypass this error.

## Parameters

- **ansiConfig**: The name of the configuration to change the behavior.

## Explanation

Databricks raises this error whenever it attempts to divide an INTERVAL, or numeric by `0`. The
provided with this error isolates the object and the expression within which the error occurrec
operators such as mod, which can cause this error include those that are performing division
complex formulas.

What happened?

In Development

# Mitigation

The mitigation of the error depends on the cause :

- **Is the expression causing the error correct?**

  If the expression is incorrect fix it so the `0` value cannot occur, and retry the query.

- **Is the data correct?**

  If the input data should be able to result in the `0` values being passed, you may need to e
  source or clean it prior to passing the data to the function as an argument.

  Data cleaning can mean to exclude the offending rows, to convert the `0` values into `NULL`
  to convert the data to another acceptable value using if(expr = 0, alt, expr).

If the expression and the data are correct, and you want to tolerate the division by zero, you
an alternative, change the argument to nullif(expr, 0). This will cause the expression to return
error. If you prefer you can ue nvl(try_divide(expr1, expr2), alt) to turn the resulting `NULL` into
such as neutral elements for addition `0` or multiplication `1`.

As a solution of last resort, when the expression or dataflow cannot be changed, you can dis
by setting the provided `ansiconfig` to `false`. Note that this setting is consequential beyon
condition.

How can I fix it?

# Examples

In Development

SQL

```
-- A DIVIDE_BY_ZERO in a embedded in view. The context information isolates the faiing functio
> CREATE OR REPLACE TEMPORARY VIEW v(c1) AS SELECT 1/val FROM VALUES(1), (0) AS T(val)
> SELECT c1 FROM v;
  [DIVIDE_BY_ZERO] Division by zero. To return NULL instead, use `try_divide`. If necessary
  == SQL of VIEW v(line 1, position 7) ==
  SELECT 1/val FROM VALUES(1), (0) AS T(val)
         ^^^^^

-- Tolerating division by zero by turning the result to NULL using try_divide.
> CREATE OR REPLACE TEMPORARY VIEW v(c1) AS SELECT try_divide(1, val) FROM VALUES(1), (
> SELECT c1 FROM v;
  1
  NULL

-- Tolerating division by zero by turning the result to NULL using nullif
> CREATE OR REPLACE TEMPORARY VIEW v(c1) AS SELECT 1 / nullif(val, 0) FROM VALUES(1),
> SELECT c1 FROM v;
  1
  NULL

-- Filtering out offensive rows
> CREATE OR REPLACE TEMPORARY VIEW v(c1) AS SELECT 1/val FROM VALUES(1), (0) AS T(val)
> SELECT c1 FROM v;
  1
```

More examples!

Java 17
Beta

Scala 2.13
Beta

Log4J 2

Apple Silicon

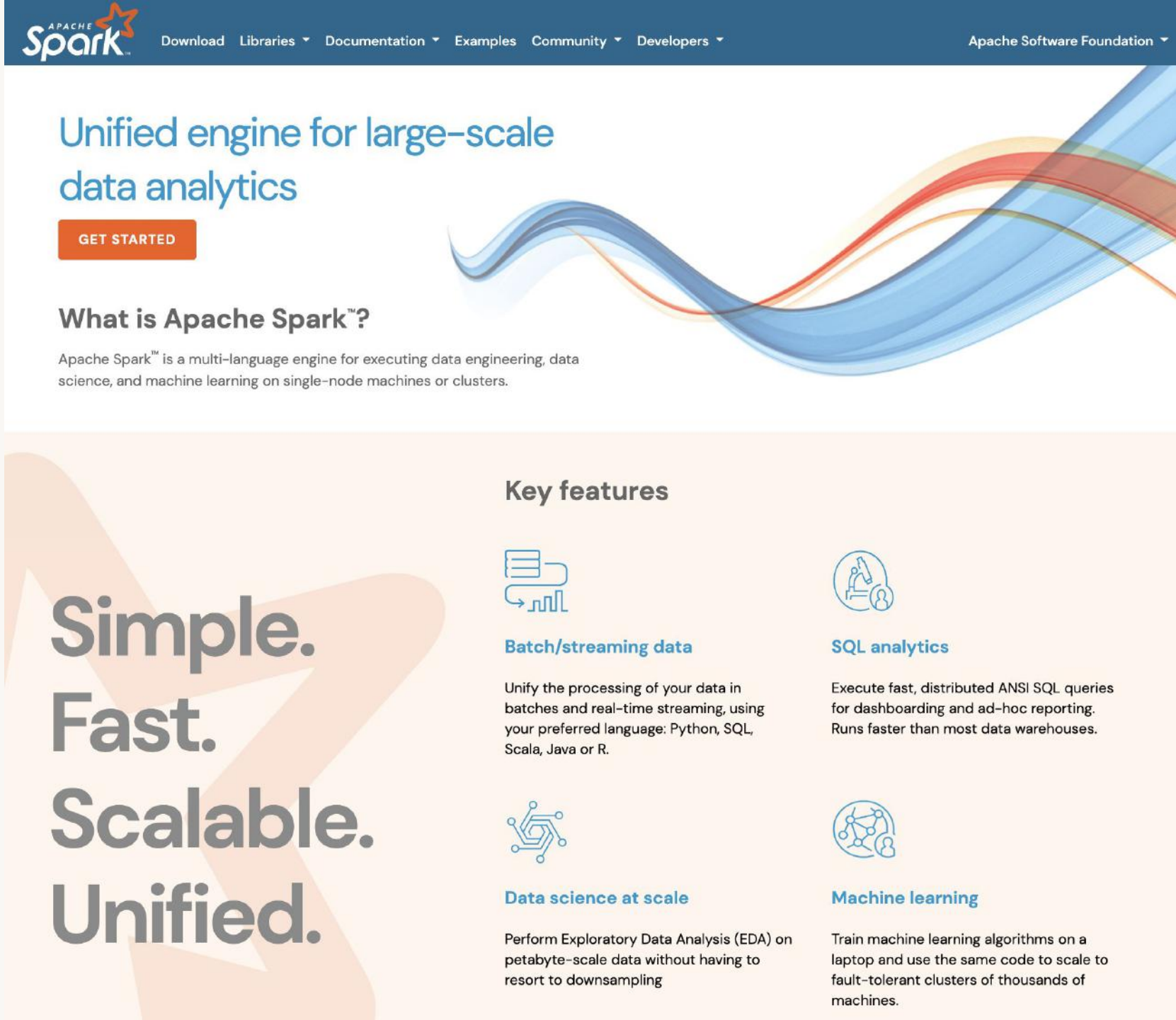
New Website for
Spark

Docker Image

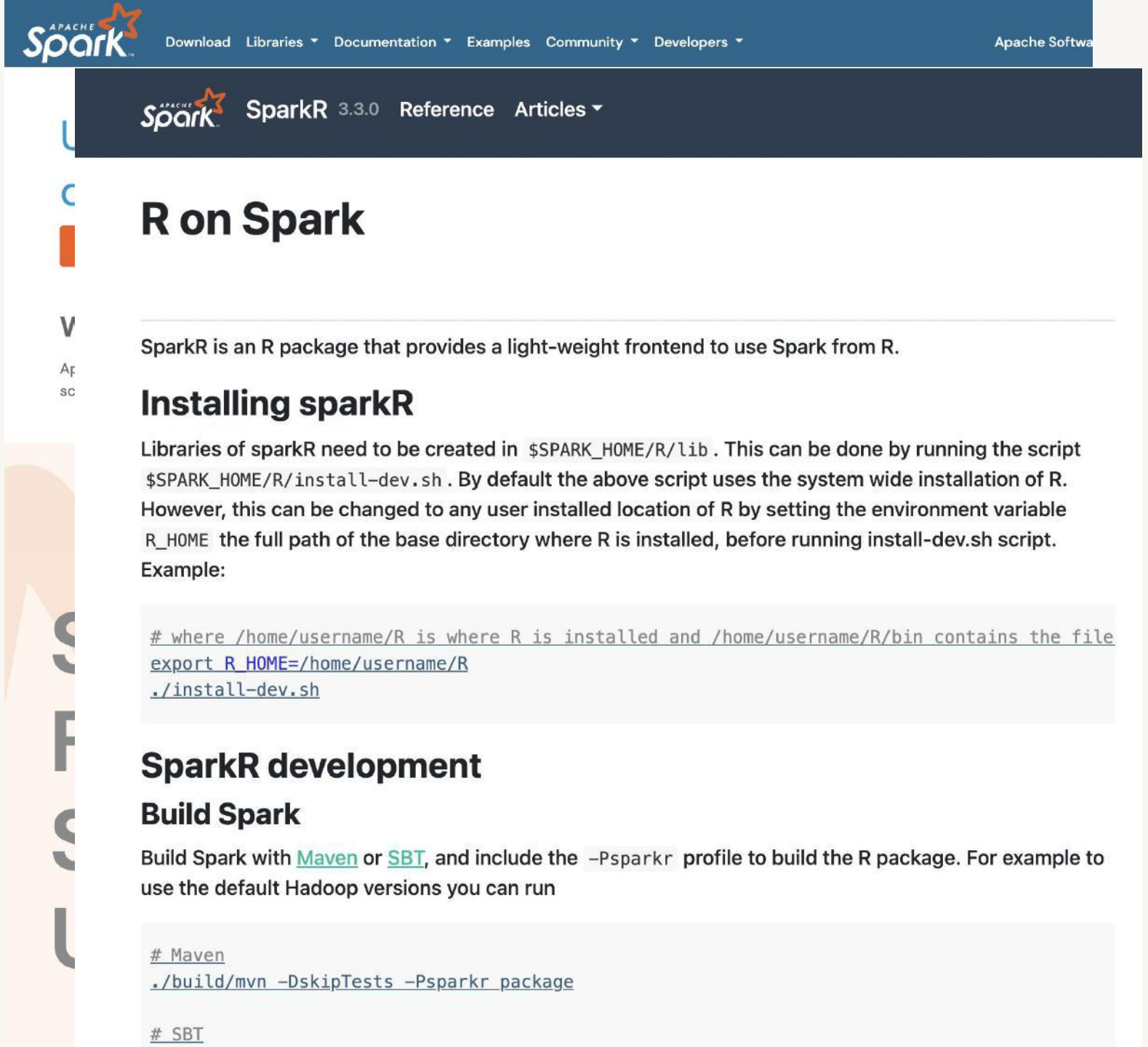
New Doc for
SparkR

# Documentation and Environments

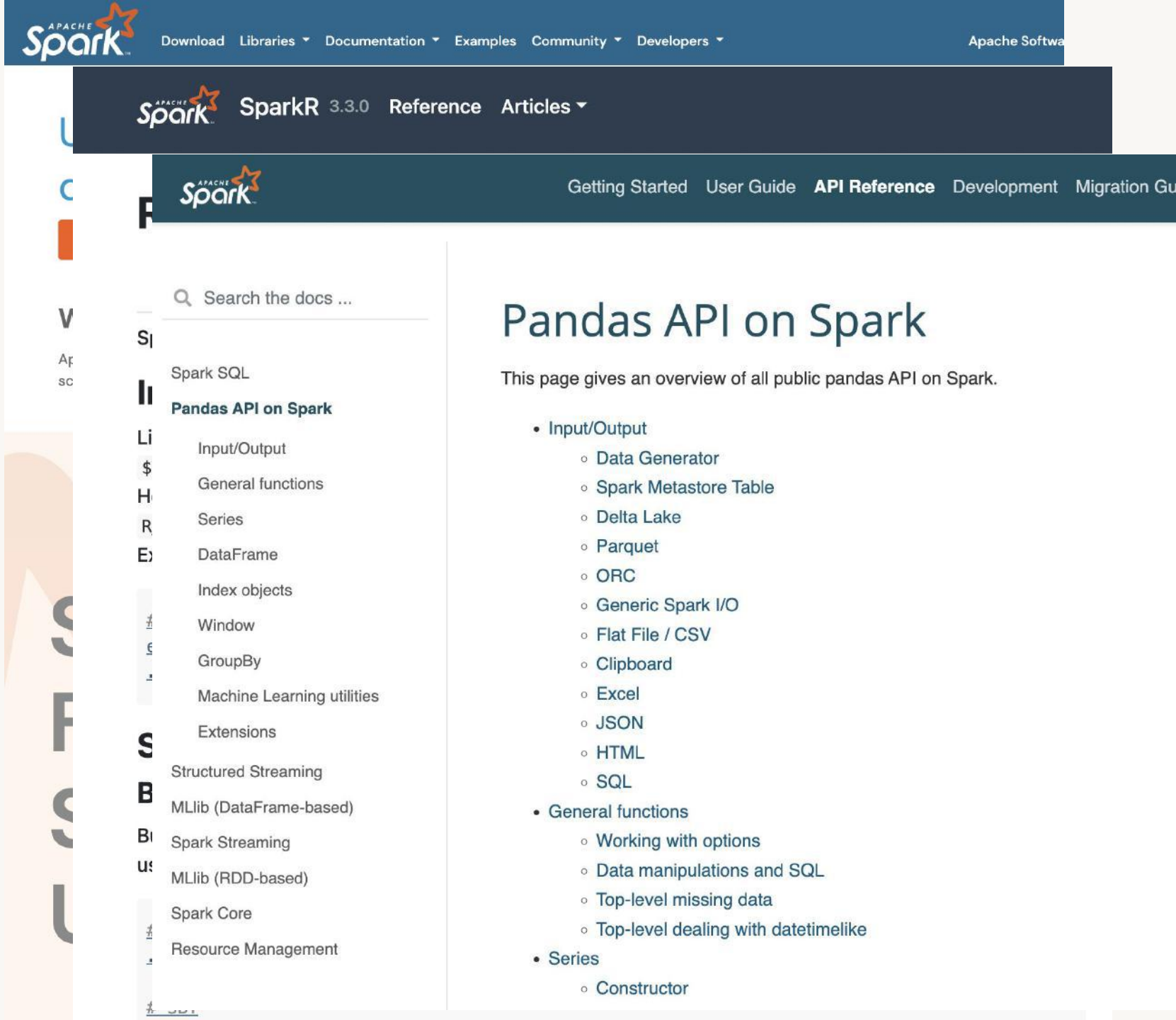
- New homepage



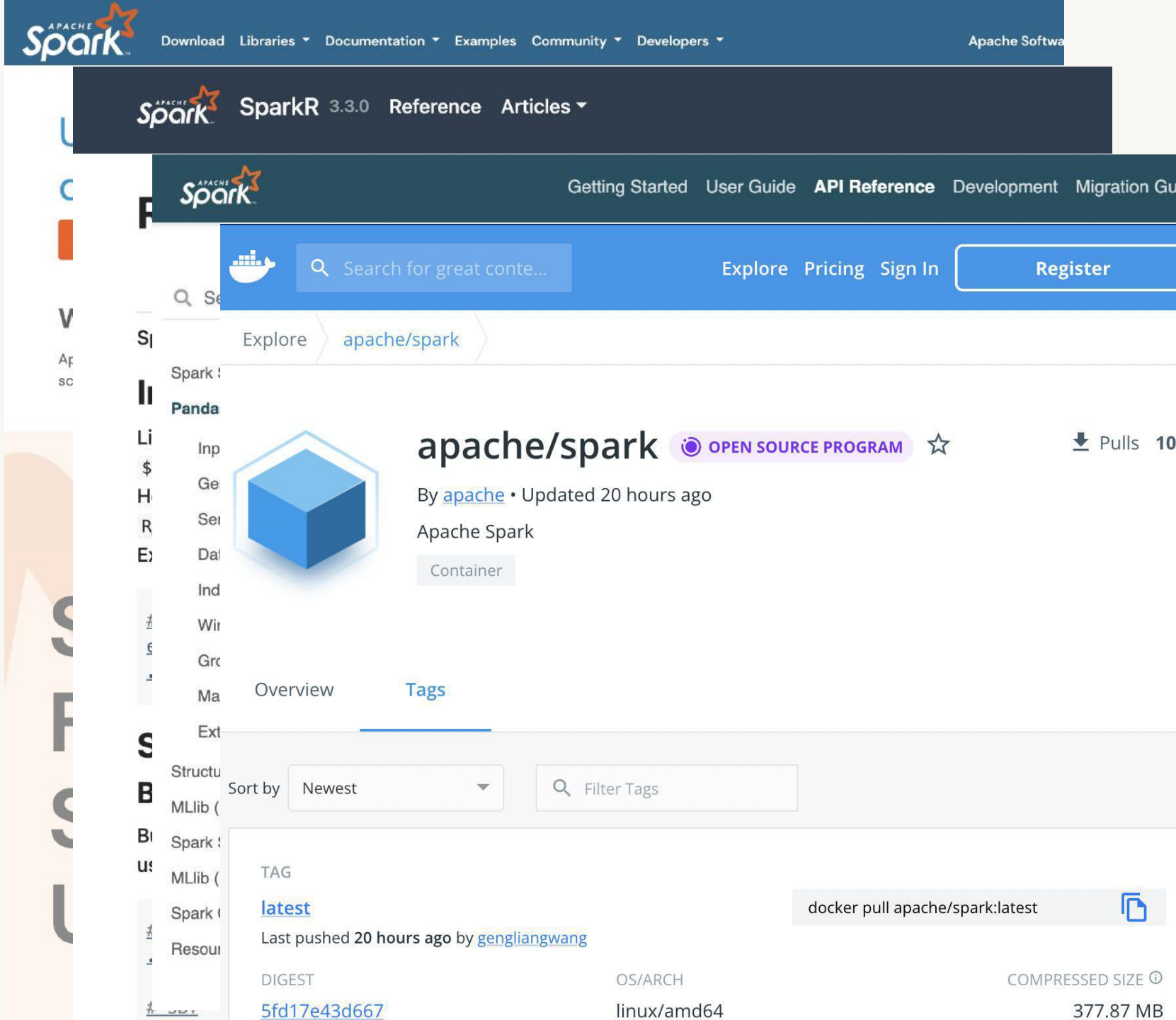
- New homepage
- New API reference doc for SparkR



- New homepage
- New API reference doc for SparkR
- New API reference doc for pandas APIs



- New homepage
- New API reference doc for SparkR
- New API reference doc for pandas APIs
- New official docker image available in docker hub



# Docker

## Productivity improvements


```
docker pull apache/spark
docker run -it apache/spark /opt/spark/bin/spark-shell
```

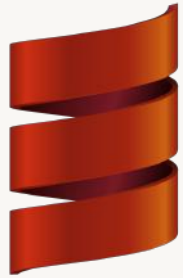
```
Spark context available as 'sc' (master = local[*], app id = local-1656135775713).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 3.3.0
      /_/

Using Scala version 2.12.15 (OpenJDK 64-Bit Server VM, Java 11.0.15)
Type in expressions to have them evaluated.
Type :help for more information.

scala> val df = spark.read.json("logs.json")
```

```
docker run -it apache/spark /opt/spark/bin/spark-sql
```

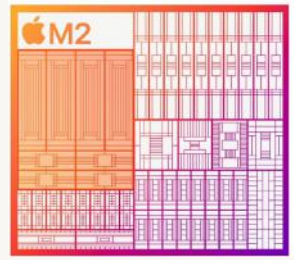
```
Spark master: local[*], Application Id: local-1656082203388
spark-sql> SELECT 1+1;
2
```

Log4J2
Upgrade

Java 17
Beta



Scala 2.13
Beta




Apple
Silicon

Runtime Filtering

Adaptive Optimization GA

Whole-stage codegen

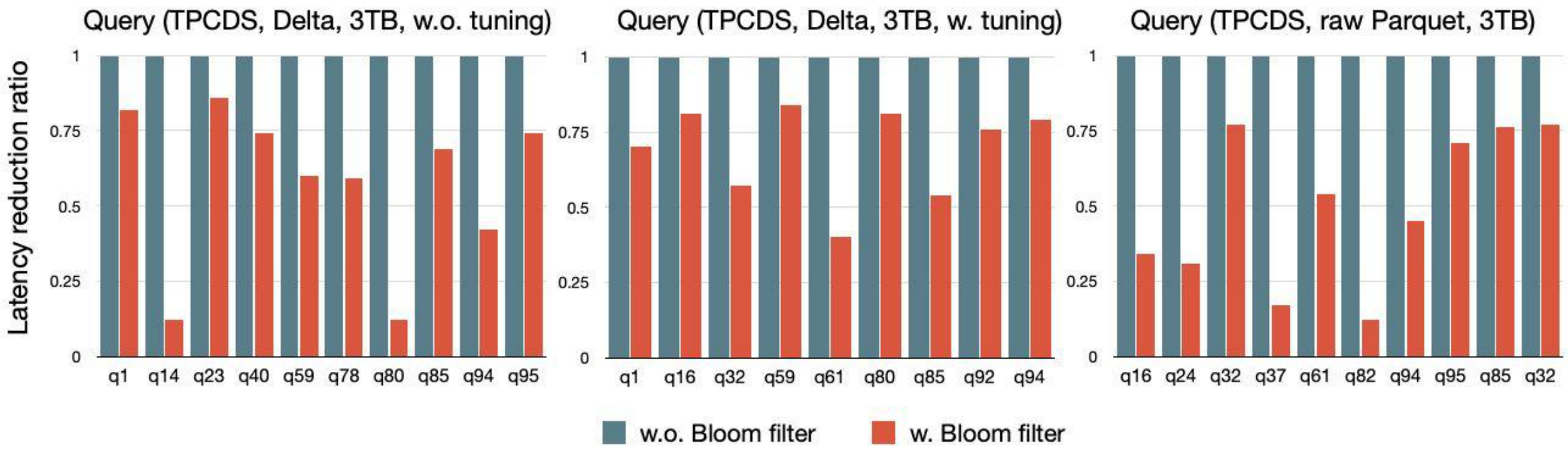
Compiler Latency Reduction

# Performance

# Runtime Filtering

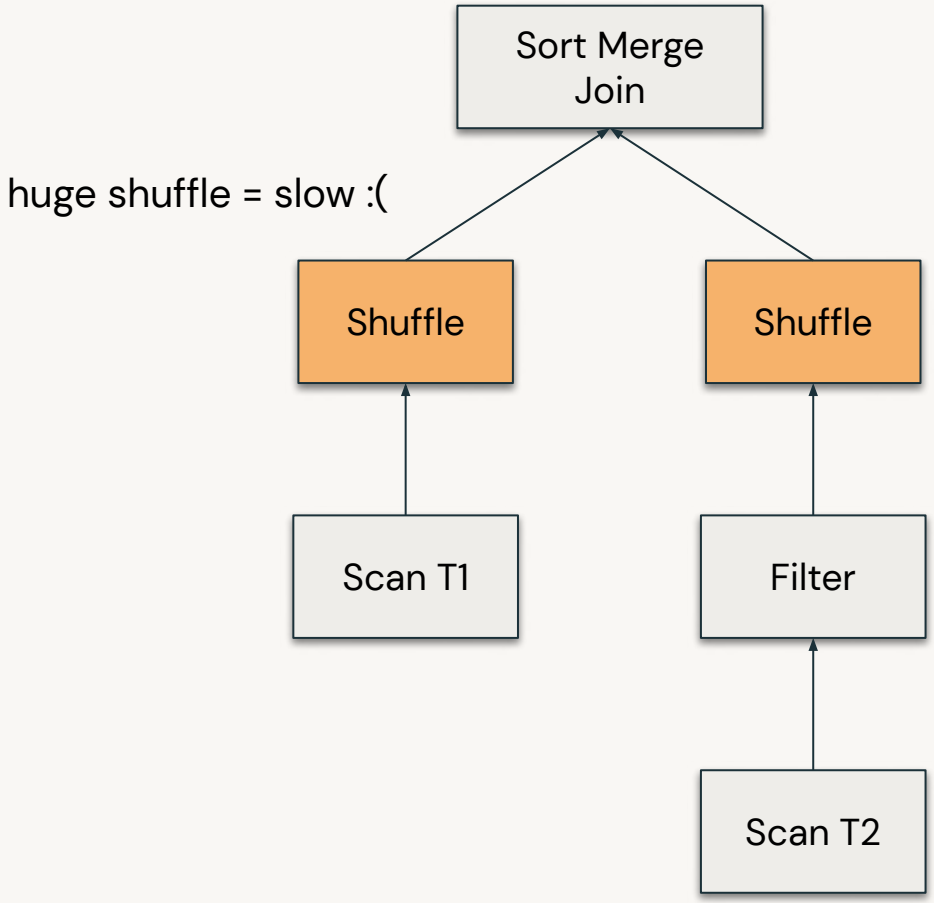
## Bloom filter joins

- Inject and push down Bloom filters for filtering data early on and reducing the intermediate data sizes for shuffle and computation.



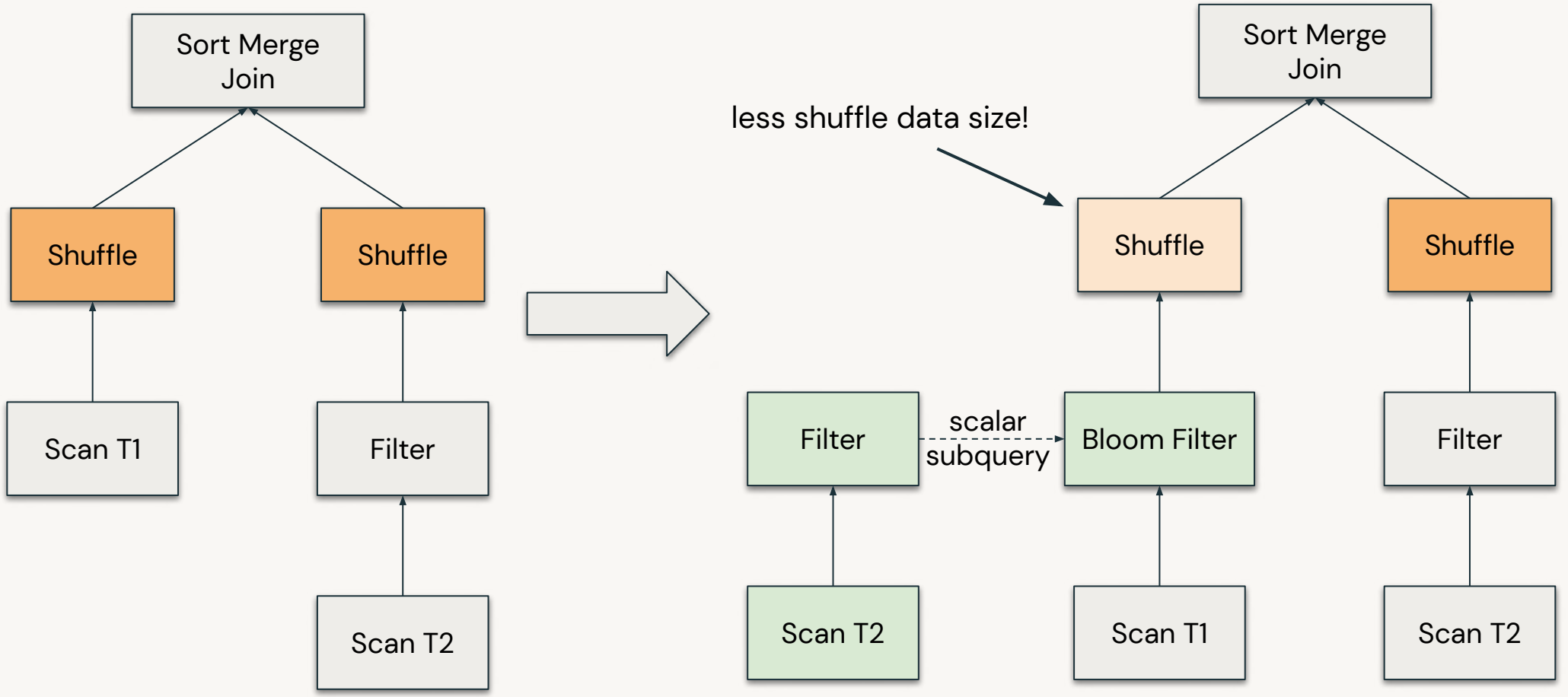
# Runtime Filtering

## Bloom filter joins



# Runtime Filtering

## Bloom filter joins



# Runtime Filtering

## Bloom filter joins

How is this different from Dynamic Partition Pruning?

- Works for non-partitioned columns as join keys
- Works for large build side (bloom filter is data sampling)
- Only benefits if we can push down bloom filter through shuffles
- Cannot reduce table scan IO

# Runtime Filtering

## Bloom filter joins

Example: SELECT * FROM R JOIN S ON R.r_sk = S.s_sk
(Note: R could be a table subquery)

```
...
+ SortMergeJoin
|     Scan R
+ Scan S
```

# Runtime Filtering

## Bloom filter joins

Example: SELECT * FROM R JOIN S ON R.r_sk = S.s_sk
(Note: R could be a table subquery)

```
...
+ BroadcastHashJoin
|    + Filter: may_contain(bf, hash(r_sk))
|      |     + Subquery subquery#3
|      |          +Aggregate [bf_agg] (8192, hash(s_sk)) AS bf
|      |               + SubqueryBroadcast [s_sk]
|      |                    + BroadcastExchange [s_sk, ...] #1
|      |                         + Filter [...]
|      |                          + Scan S
|     Scan R
+ ReusedExchange [s_sk, ...] #1
```

# Runtime Filtering

## Bloom filter joins

Another example: TPC-DS Q32

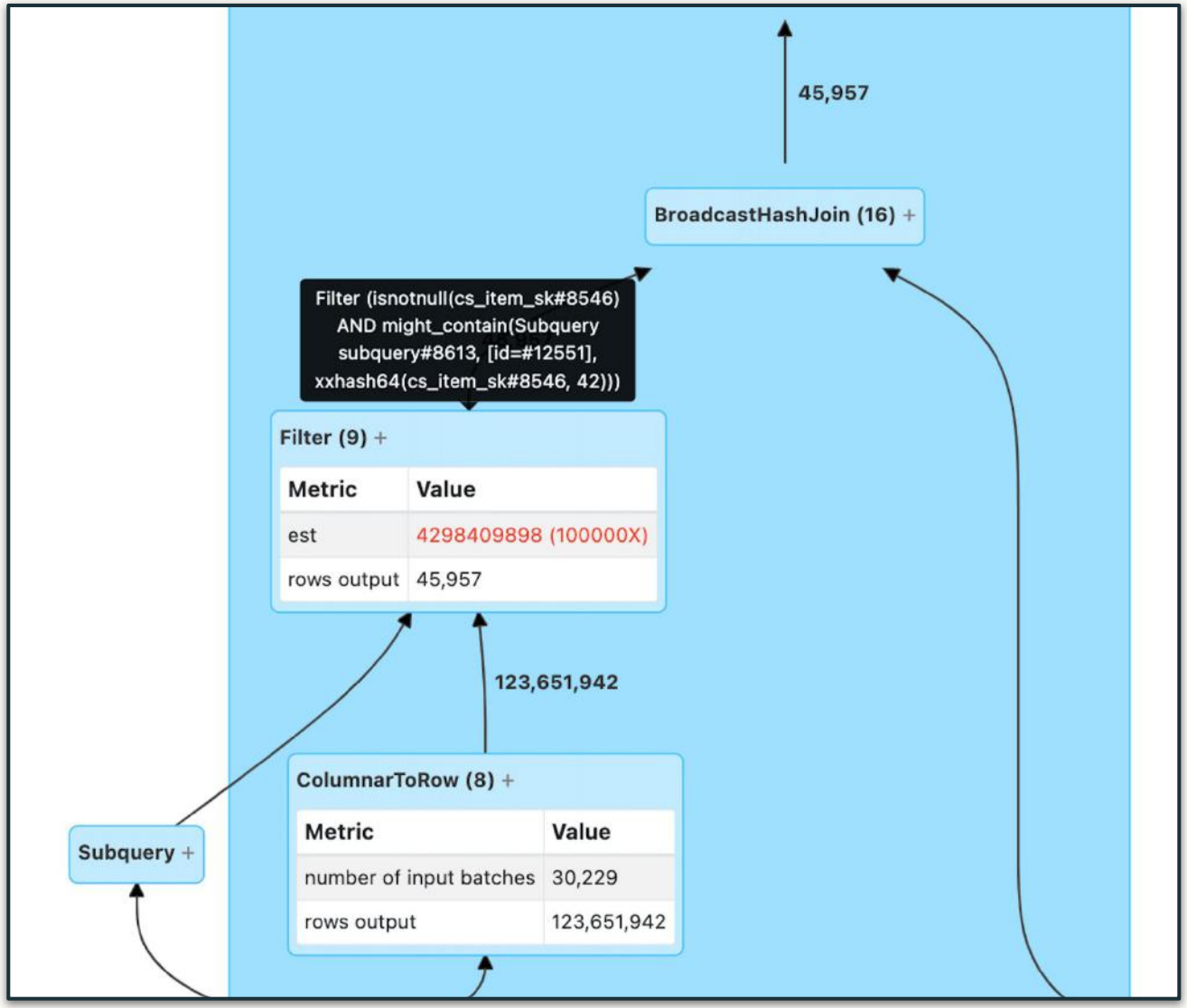
```
select sum(cs_ext_discount_amt) as `excess discount amount`
from
   catalog_sales, item, date_dim
where
  i_manufact_id = 977
  and i_item_sk = cs_item_sk
  and d_date between cast ('2000-01-27' as date) and (cast('2000-01-27' as date) + interval '90' day)
  and d_date_sk = cs_sold_date_sk
  and cs_ext_discount_amt > (
         select 1.3 * avg(cs_ext_discount_amt)
         from catalog_sales, date_dim
         where cs_item_sk = i_item_sk
          and d_date between cast ('2000-01-27' as date) and (cast('2000-01-27' as date) + interval '90' day)
          and d_date_sk = cs_sold_date_sk)
limit 100
```

# Runtime Filtering

## Bloom filter joins

Another example: TPC-DS Q32

- In the Spark UI, we see the new bloom filter.
- It reduced the rows generated by the preceding ColumnarToRow operator from 123.6M down to 46K!



# Adaptive Query Execution

## Enabled by default since Spark 3.2

Adaptive query execution recomputes plans at shuffle boundaries to:

- Switch sort merge joins to hash joins (**SPARK-35282**)
- Enable broadcasting for hash joins (**SPARK-35264**)

# Adaptive Query Execution

## Enabled by default since Spark 3.2

Other adaptive query execution features:

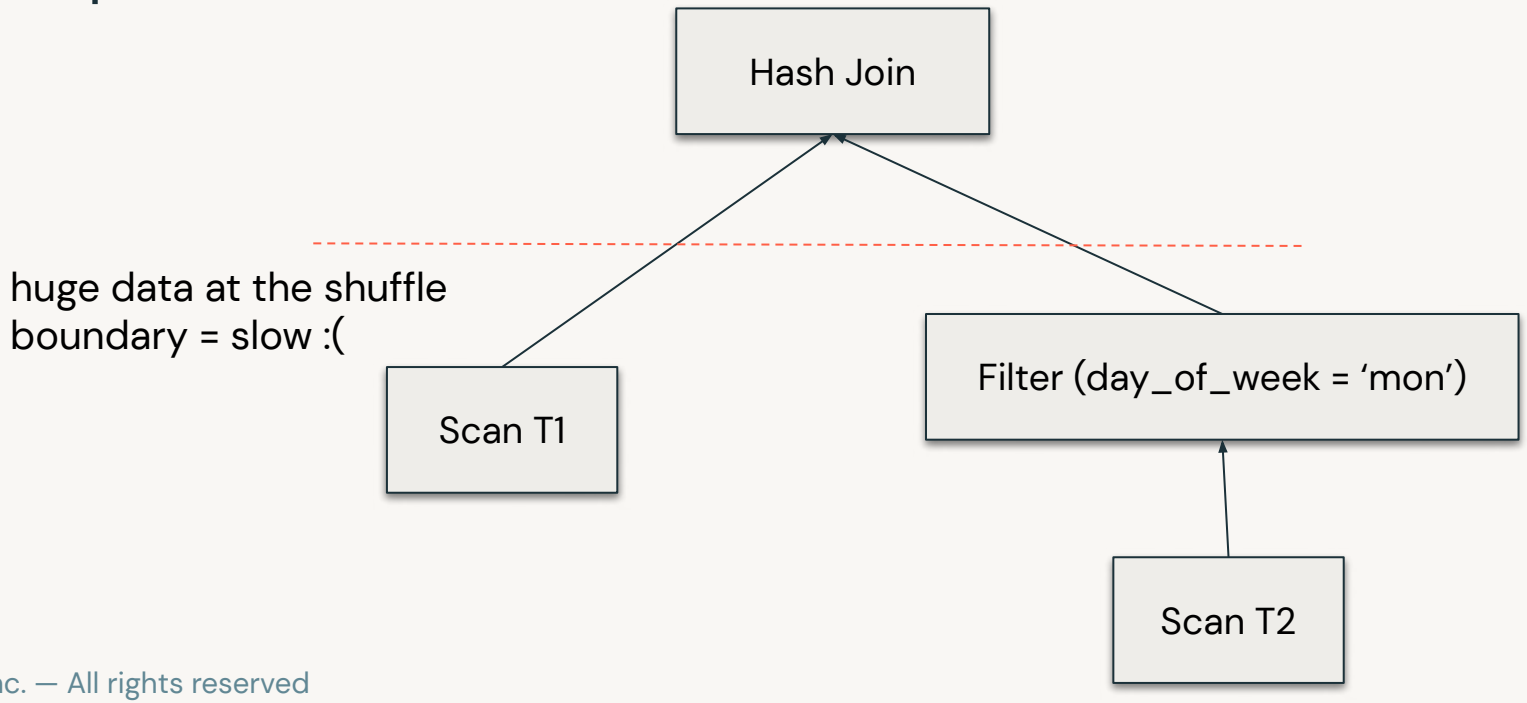
- Dynamic partition pruning (**`SPARK-34168`**)
- Eliminate limits during adaptive query execution (**`SPARK-36424`**)
- Optimize away one-row plans (**`SPARK-38162`**)

# Adaptive Query Execution

## Enabled by default since Spark 3.2

- Dynamic partition pruning (**`SPARK-34168`**)

Problem: scanning huge tables can materialize a lot of **shuffle data** and slow down queries!

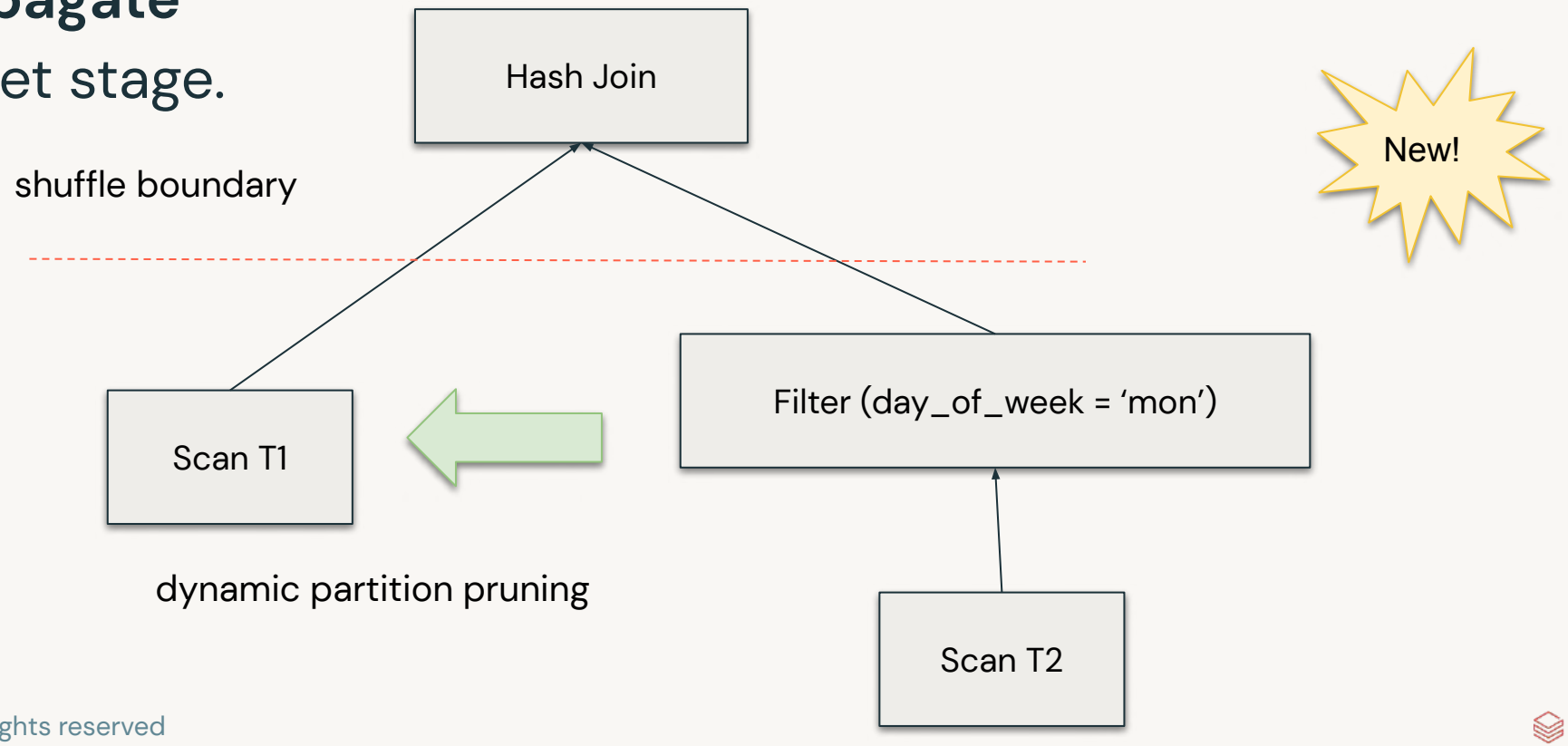


# Adaptive Query Execution

## Enabled by default since Spark 3.2

- Dynamic partition pruning (**`SPARK-34168`**)

Solution: **propagate filters** to target stage.

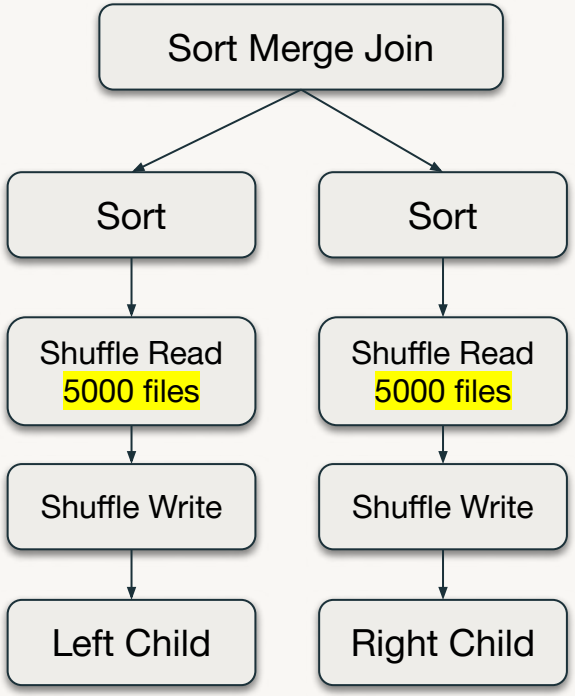


# Adaptive Query Execution

## Enabled by default since Spark 3.2

Problem: **many small file partitions** at the shuffle boundary slow down query runtimes due to excessive per-file processing overhead!

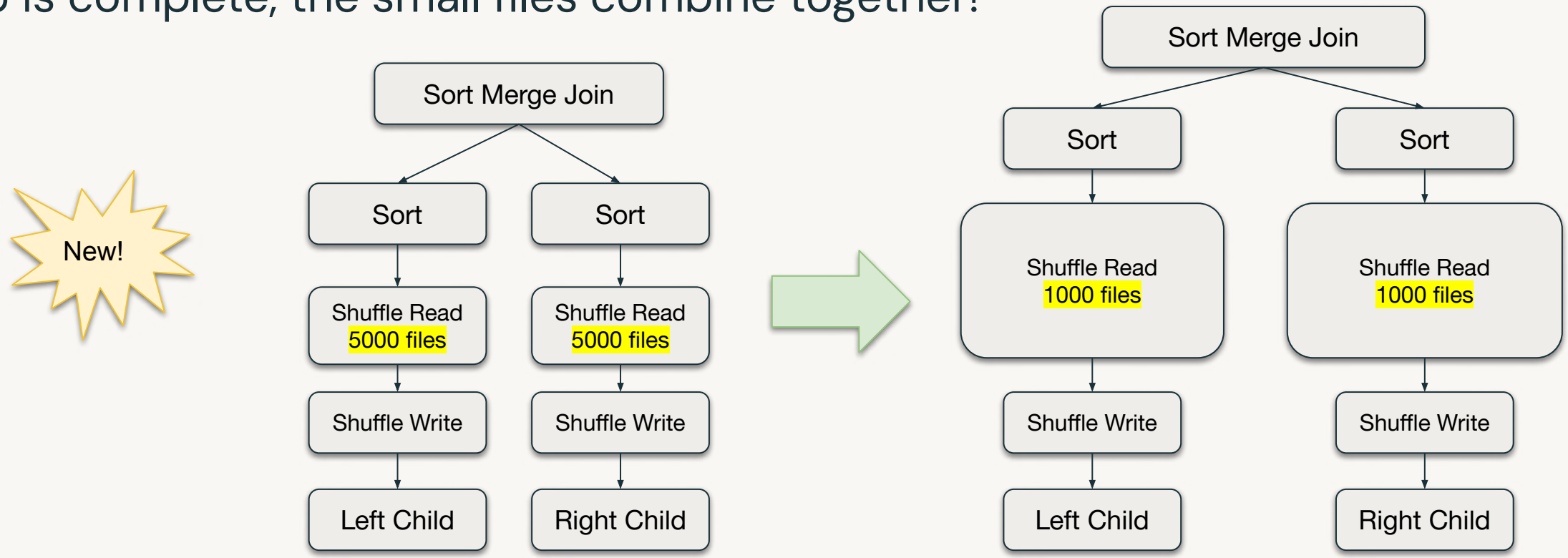
huge number of file partitions at the shuffle boundary = slow :(



# Adaptive Query Execution

## Enabled by default since Spark 3.2

Solution: Dynamically **merge partitions** together. After the shuffle write step is complete, the small files combine together!

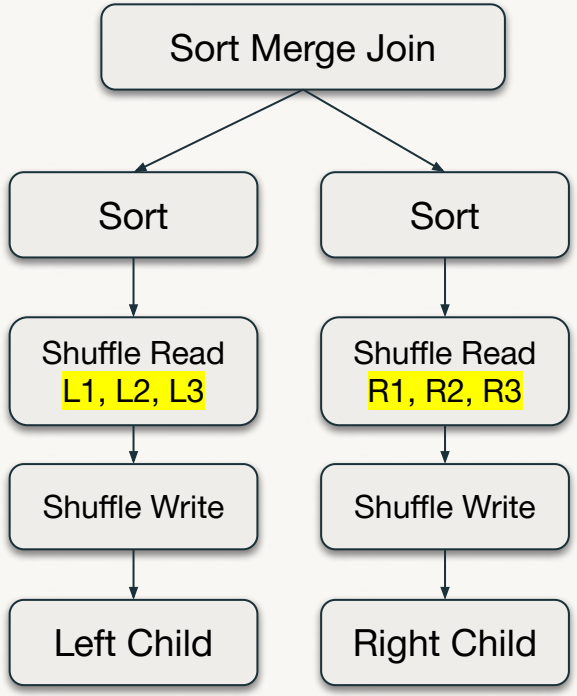


# Adaptive Query Execution

## Enabled by default since Spark 3.2

Problem: **skewed joins** slow down query runtimes by introducing "stragglers" which prevent stages from finishing quickly!

huge number of rows with the same join key in the same worker = slow :(

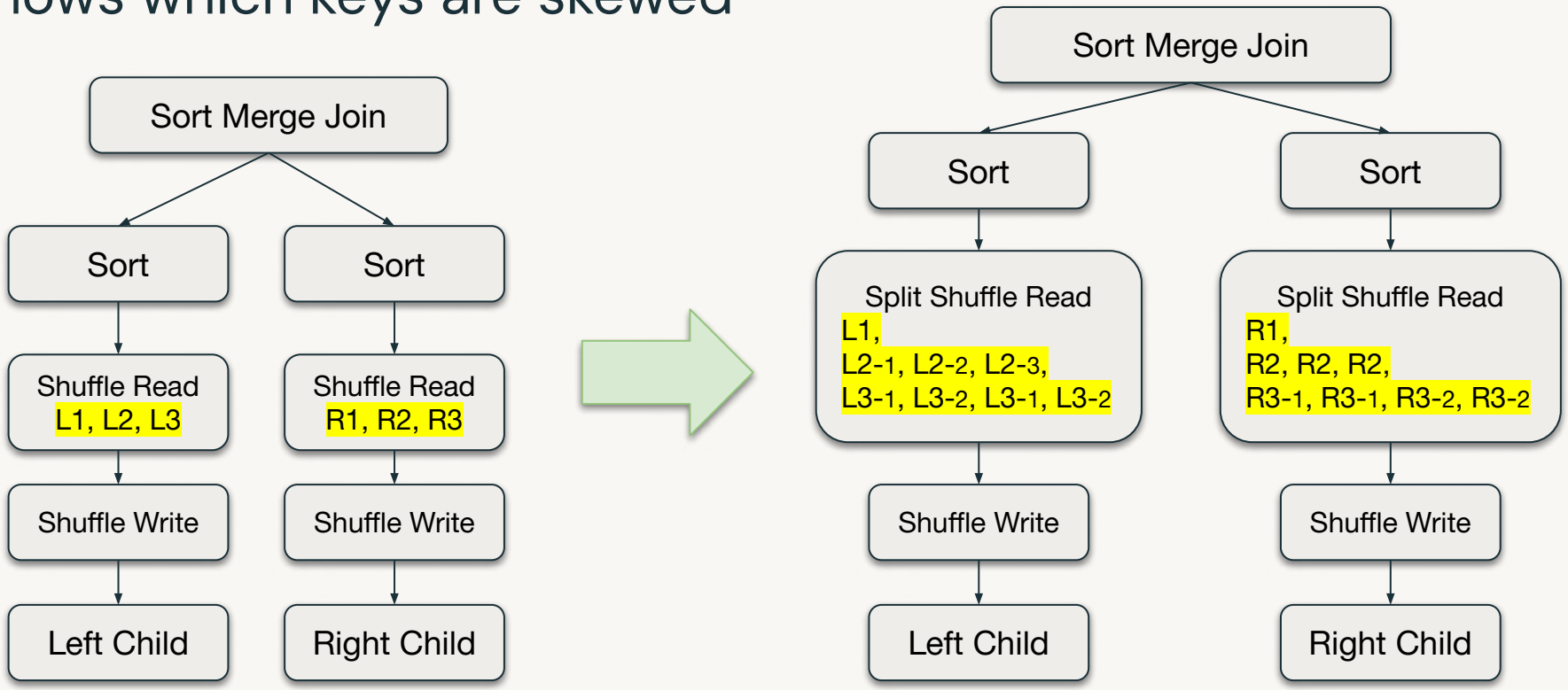


# Adaptive Query Execution

## Enabled by default since Spark 3.2

Solution: Handling **skewed joins** dynamically after the shuffle write step is complete. Spark knows which keys are skewed and performs a split shuffle read to compensate!

New!



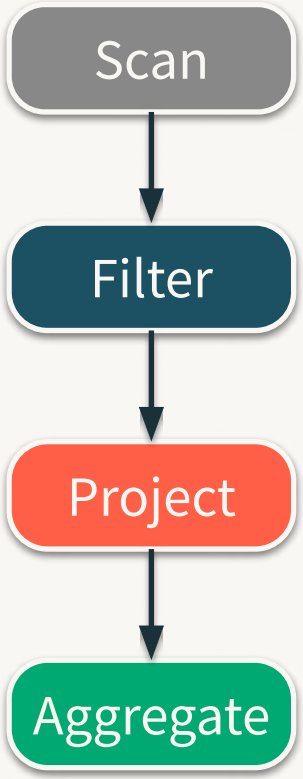
# Adaptive Query Execution

Blog post:
https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html

# Whole-Stage Code Generation

- Inspired by Thomas Neumann's paper

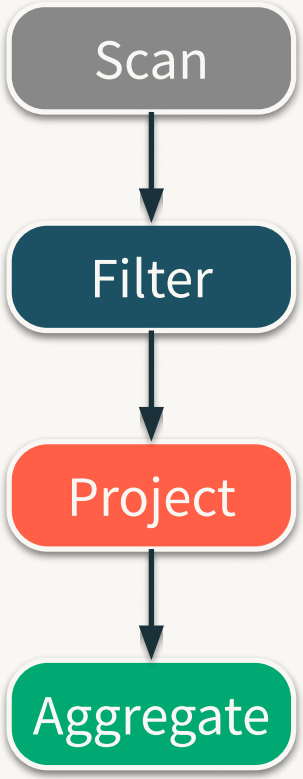
- Fuses a string of operators (oftentimes the entire stage) into one codegen operator that runs the generated code.




```
long count = 0;
for (item in sales) {
  if (price < 100) {
    count += 1;
  }
}
```

# Whole-Stage Code Generation

- Creates a general-purpose execution engine just like Volcano model but without Volcano's performance downsides:
  - No virtual function calls
  - Data in CPU registers
  - Loop unrolling & SIMD



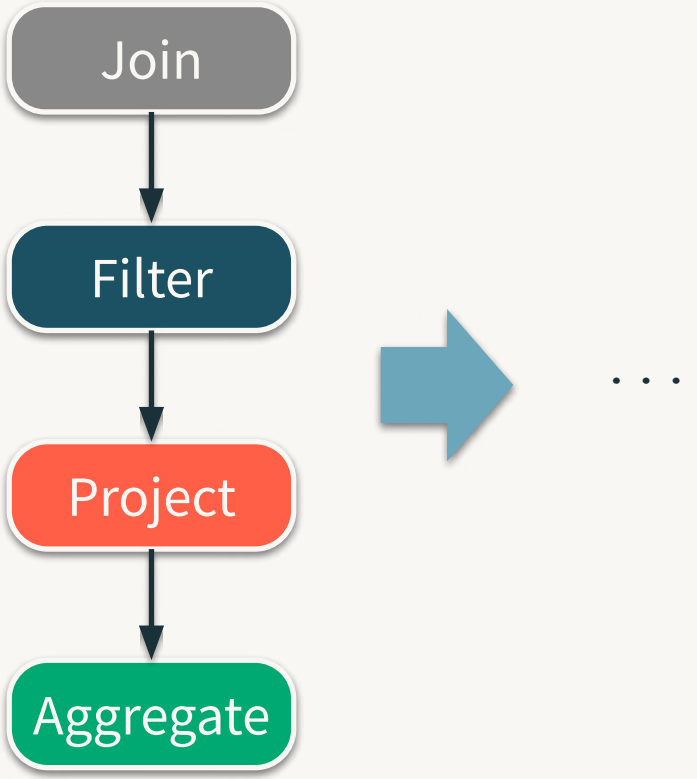

```
long count = 0;
for (item in sales) {
  if (price < 100) {
    count += 1;
  }
}
```

# Whole-Stage Code Generation

## More coverage

New!

- Sort-based aggregates without grouping (**SPARK-37564**)
- Full outer sort merge join (**SPARK-35352**)
- Full outer hash join (**SPARK-32567**)
- Existence sort merge join (**SPARK-37316**)



# Whole-Stage Code Generation

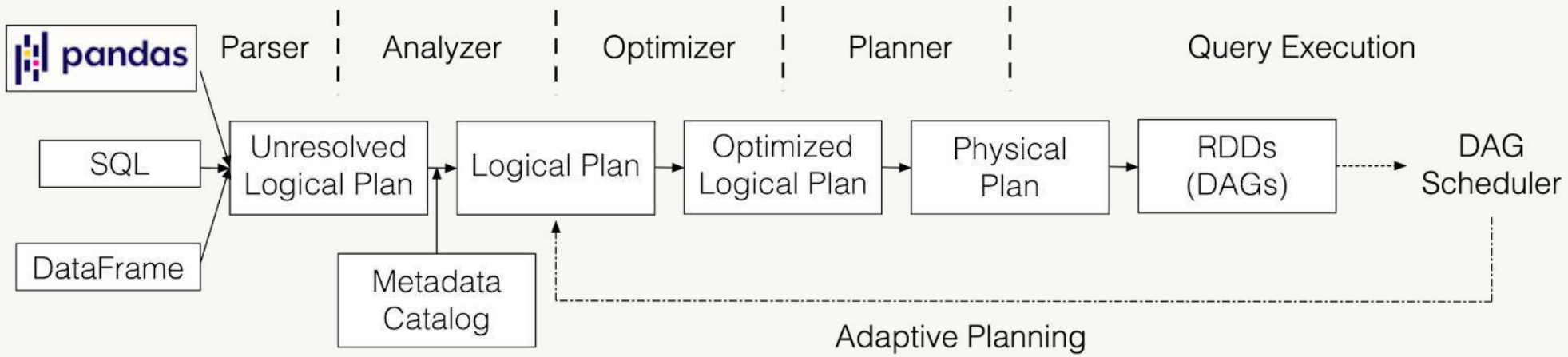
## More coverage

```
override def doGenCode(ctx: CodegenContext, ev: ExprCode): ExprCode = {
  if (bloomFilter == null) {
    ev.copy(isNull = TrueLiteral, value = JavaCode.defaultLiteral(dataType))
  } else {
    val bf = ctx.addReferenceObj(
      objName = "bloomFilter", bloomFilter, classOf[BloomFilter].getName)
    val valueEval = valueExpression.genCode(ctx)
    ev.copy(code = code"""
    ${valueEval.code}
    boolean ${ev.isNull} = ${valueEval.isNull};
    ${CodeGenerator.javaType(dataType)} ${ev.value} = ${CodeGenerator.defaultValue(dataType)};
    if (!${ev.isNull}) {
      ${ev.value} = $bf.mightContainLong((Long)${valueEval.value});
    }""")
  }
}
```

# Compiler Latency Reduction

## Small queries now run faster

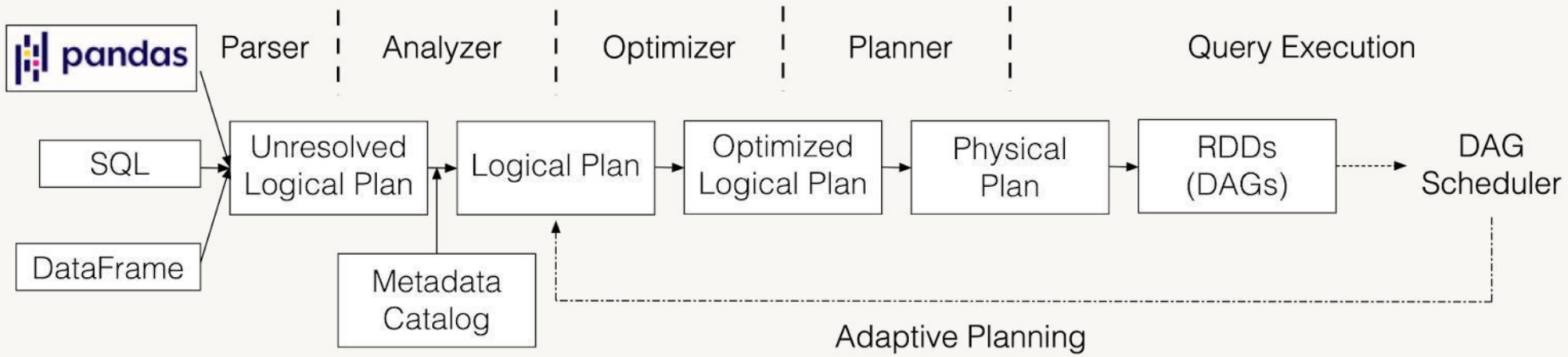
- Query plans are immutable.
- The query planning sequence runs many rules until reaching "fixed point."



# Compiler Latency Reduction

## Small queries now run faster

- Each rule transforms the query plan for one purpose.
- It traverses the query plan and makes a copy to "change" the plan.



# Compiler Latency Reduction

## Make small queries faster

- Prune subtrees during query plan traversal (**SPARK-35042**)
  - cheaper tree traversal
- Speed up copying plan nodes (**SPARK-34989**)
  - cheaper plan copies

```
/**
 * Remove no-op operators from the query plan that do not make any modifications.
 */
object RemoveNoopOperators extends Rule[LogicalPlan] {

  def apply(plan: LogicalPlan): LogicalPlan = plan.transformUpWithPruning(
    _.containsAnyPattern(PROJECT, WINDOW), ruleId) {
```

# Compiler Latency Reduction

## Make small queries faster

- Combine type coercion rules (**SPARK-35103**)
  - fewer tree traversals

```
abstract class TypeCoercionBase {
  /**
   * A collection of [[Rule]] that can be used to coerce differing types that participate in
   * operations into compatible ones.
   */
  def typeCoercionRules: List[Rule[LogicalPlan]]
```

```
object TypeCoercion extends TypeCoercionBase {

  override def typeCoercionRules: List[Rule[LogicalPlan]] =
    WidenSetOperationTypes ::
    new CombinedTypeCoercionRule(
      InConversion ::
      PromoteStrings ::
      DecimalPrecision ::
```

# Compiler Latency Reduction

## Make small queries faster

- Speed up copying plan nodes (**SPARK-34989**)
  - cheaper plan copies


```
trait UnaryLike[T <: TreeNode[T]] { self: TreeNode[T] =>
```

```
override final def mapChildren(f: T => T): T = {
  val newChild = f(child)
  if (newChild fastEquals child) {
    this.asInstanceOf[T]
  } else {
    CurrentOrigin.withOrigin(origin) {
      val res = withNewChildInternal(newChild)
      res.copyTagsFrom(this.asInstanceOf[T])
      res
    }
  }
}
```

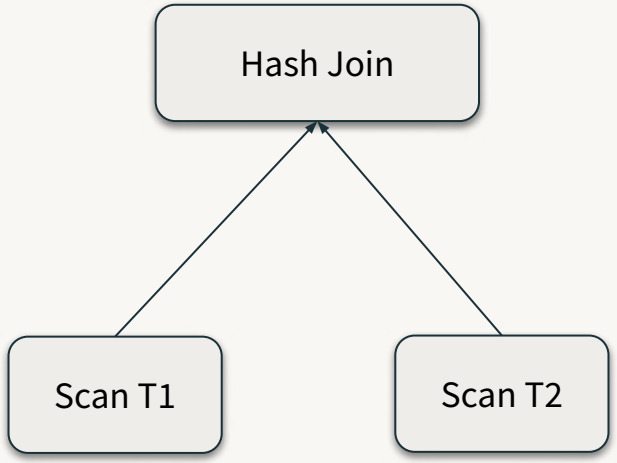
TPC-DS query speedups!

| Query | Speedup |
|---|---|
| q4 | 29% |
| q9 | 81% |
| q14a | 31% |
| q14b | 28% |
| q22 | 33% |
| q33 | 29% |
| q34 | 25% |
| q39 | 27% |

# Compiler Latency Reduction

## Make small queries faster

- Prune subtrees during query plan traversal (**SPARK-35042**)
  - cheaper tree traversal
- Speed up copying plan nodes (**SPARK-34989**)
  - cheaper plan copies
- Combine type coercion rules (**SPARK-35103**)
  - fewer tree traversals
- Many more!



Trigger AvailableNow

Built-in Functions

Hidden File Metadata

RocksDB State Store

Session Window

SQL Configs in Spark UI

# Features

# Built-in functions

## 43 new functions are added in Spark 3.2 and 3.3

# Built-in functions

## 43 new functions are added in Spark 3.2 and 3.3

```
> SELECT to_number('454', '999');
 454
> SELECT to_number('454.00', '000.00');
 454.00
> SELECT to_number('12,454', '99,999');
 12454
> SELECT to_number('$78.12', '$99.99');
 78.12
> SELECT to_number('12,454.8-', '99,999.9S');
 -12454.8
```

# Built-in functions

## 43 new functions are added in Spark 3.2 and 3.3

```
> SELECT contains('Spark SQL', 'Spark');
 true
> SELECT contains('Spark SQL', 'SPARK');
 false
> SELECT contains('Spark SQL', null);
 NULL
> SELECT contains(x'537061726b2053514c', x'537061726b');
 true
```

# Built-in functions

## 43 new functions are added in Spark 3.2 and 3.3

```
> SELECT hex(aes_encrypt('Spark', '0000111122223333'));
 83F16B2AA704794132802D248E6BFD4E380078182D1544813898AC97E709B28A94
> SELECT hex(aes_encrypt('Spark SQL', '0000111122223333', 'GCM'));
 6E7CA17BBB468D3084B5744BCA729FB7B2B7BCB8E4472847D02670489D95FA97DBBA7D3210
> SELECT base64(aes_encrypt('Spark SQL', '1234567890abcdef', 'ECB', 'PKCS'));
 3lmwu+Mw0H3fi5NDvcu9lg==
```

# Built-in functions

## 43 new functions are added in Spark 3.2 and 3.3

```
> SELECT map_contains_key(map(1, 'a', 2, 'b'), 1);
 true
> SELECT map_contains_key(map(1, 'a', 2, 'b'), 3);
 false
```

# Built-in functions

## 43 new functions are added in Spark 3.2 and 3.3

```
scala> sql("SELECT histogram_numeric(col, 5) FROM
VALUES (0), (1), (2), (10) AS tab(col);").show()
+----------------------------+
|histogram_numeric(col, 5)|
+----------------------------+
|      [{0, 1.0}, {1, 1....|
+----------------------------+
```

# Hidden file metadata column

## Make it easier to access file metadata

The hidden file metadata column is a special column named **`_metadata`** that can only be accessed by name, but not **`SELECT *`**.

| Name | Type | Description | Example |
|---|---|---|---|
| _metadata.file_path | String | The absolute file path of the input file. | file:/tmp/spark-7f600b30-b3ec-43a8-8cd2-686491654f9b/f0.csv |
| _metadata.file_name | String | The name of the input file along with the extension. | f0.csv |
| _metadata.file_size | Long | The length of the input file, in bytes. | 628 |
| _metadata.file_modification_time | Timestamp | The modification timestamp of the file. | 2021-12-20 20:05:21 |

# Hidden file metadata column

## Make it easier to access file metadata

The hidden file metadata column is a special column named **`_metadata`** that can only be accessed by name, but not **`SELECT *`**.

```
Q24X9L472X:/tmp$ ls -l *.csv
-rw-r--r--  1 Daniel.Tenedorio  wheel  8 Jun 28 16:32 file.csv
-rw-r--r--  1 Daniel.Tenedorio  wheel  8 Jun 28 16:32 file2.csv
Q24X9L472X:/tmp$ cat *.csv
1, 2, 3
4, 5, 6
```

Let's look at a brief example of how we can go about using this feature.

# Hidden file metadata column

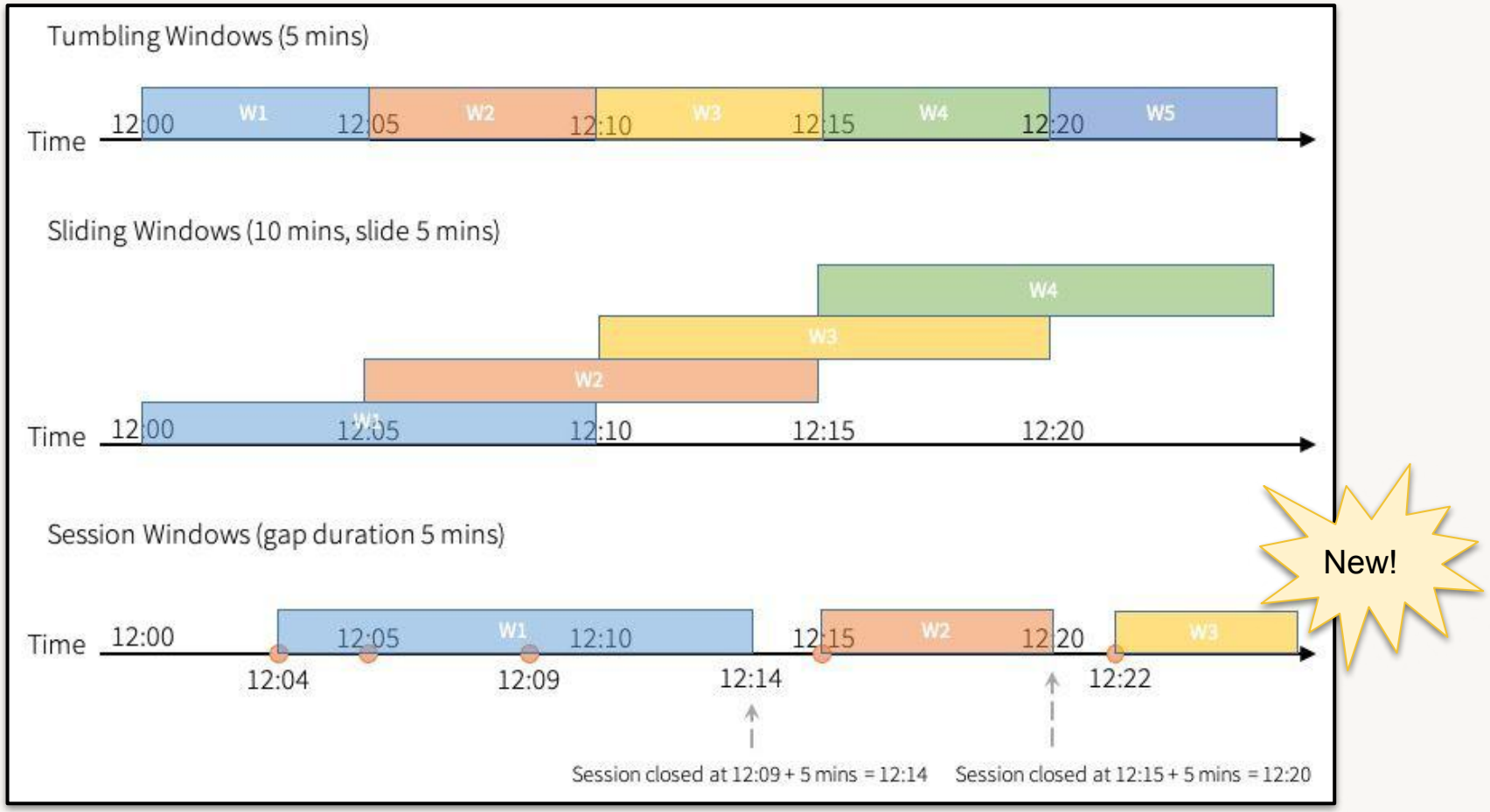
## Make it easier to access file metadata

The hidden file metadata column is a special column named **`_metadata`** that can only be accessed by name, but not **`SELECT *`**.

```
scala> spark.read.format("csv").load("file:/tmp/*.csv").select(
     |   "_metadata.file_path",
     |   "_metadata.file_size",
     |   "_metadata.file_modification_time").show()
+--------------------+---------+--------------------+
|           file_path|file_size|file_modification_time|
+--------------------+---------+--------------------+
| file:/tmp/file.csv|        8|   2022-06-28 16:32:46|
|file:/tmp/file2.csv|        8|   2022-06-28 16:32:55|
+--------------------+---------+--------------------+
```

# Session Window

## Define sessions based on activity within structured streaming



# Session Window

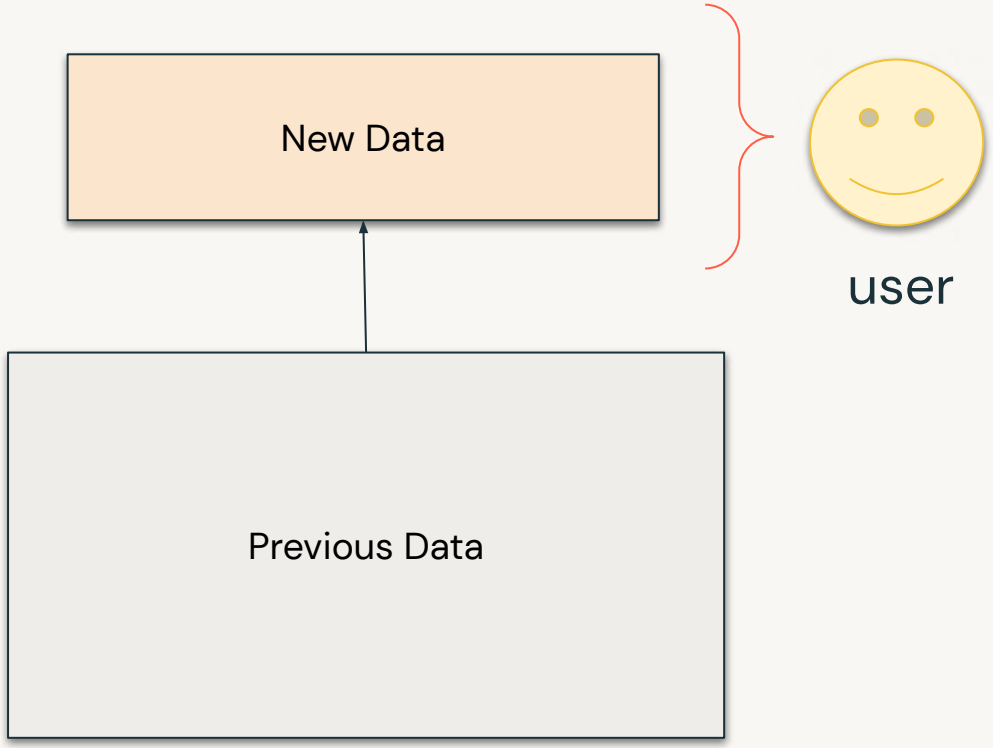
## Define sessions based on activity within structured streaming

```
import spark.implicits._

val events = ... // streaming DataFrame of schema { timestamp: Timestamp, userId: String }

// Group the data by session window and userId, and compute the count of each group
val sessionizedCounts = events
    .withWatermark("timestamp", "10 minutes")
    .groupBy(
        session_window($"timestamp", "5 minutes"),
        $"userId")
    .count()
```

# New Structured Streaming Trigger: `AvailableNow`

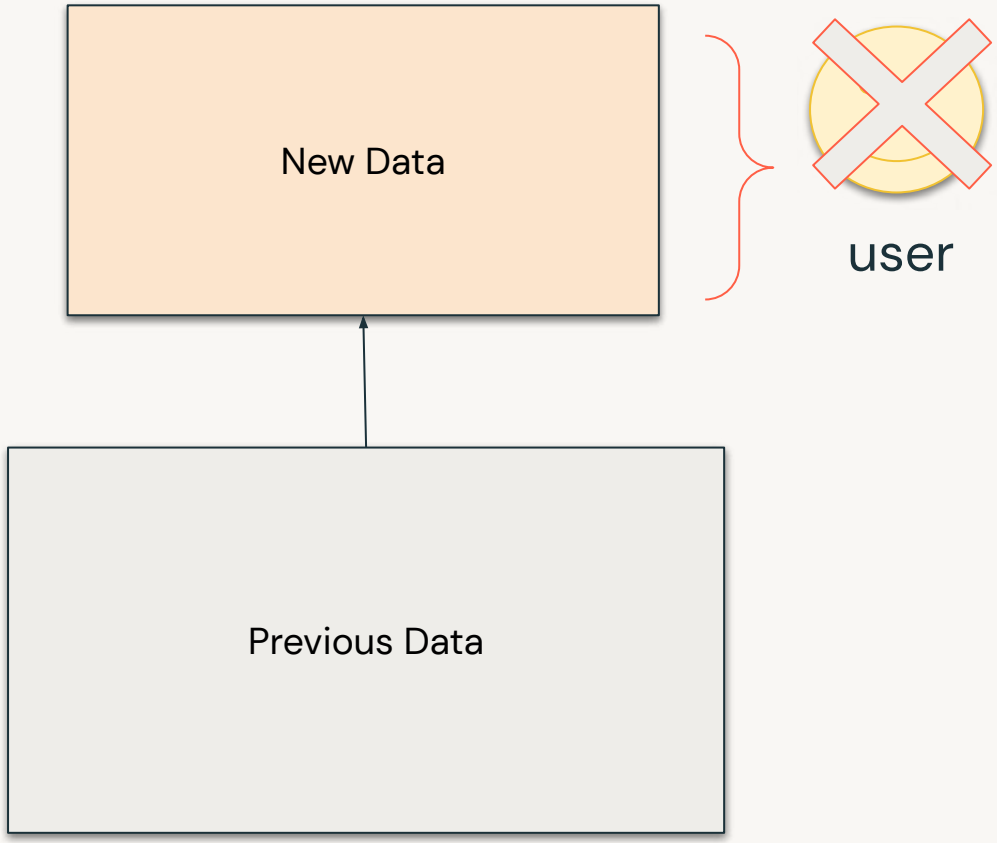
Background: the existing *one-time micro-batch trigger* works when:

- You want to update your dashboard daily, where latency can be large.
- You want to process the input data incrementally, without processing the full data set every time.
- You want to save resources by avoiding keeping a long running streaming application online.



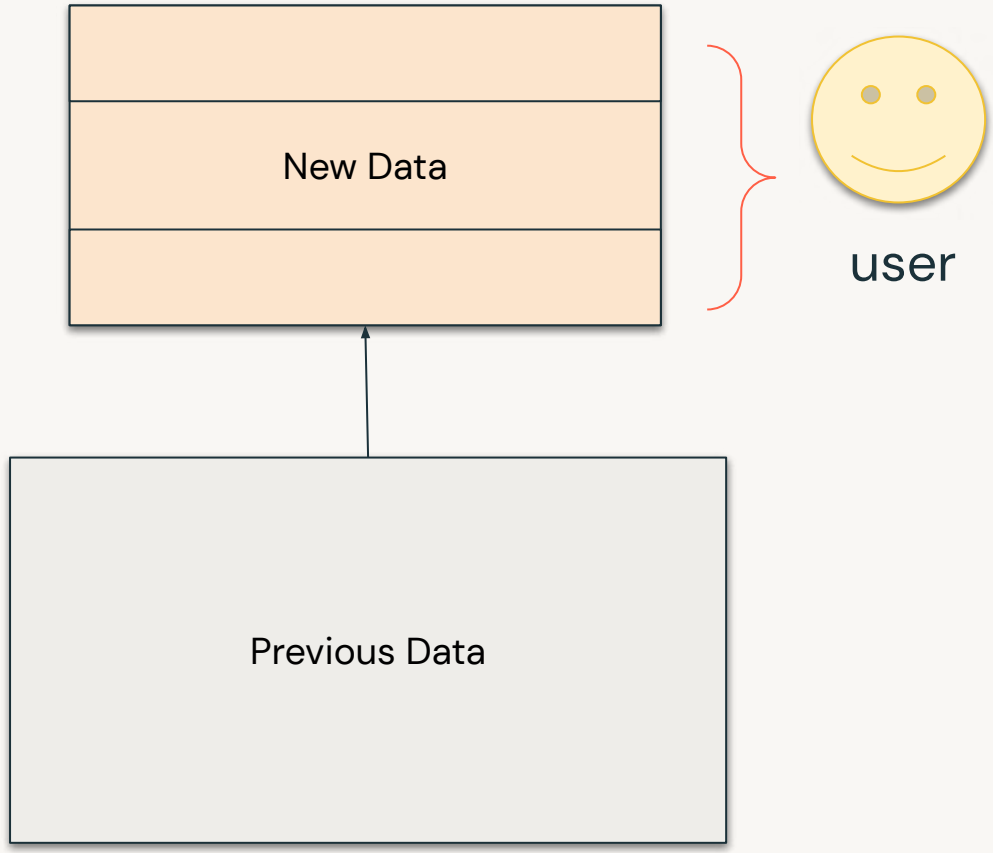
# New Structured Streaming Trigger: `AvailableNow`

- How it works: the existing *one-time micro-batch trigger* starts a streaming application from checkpoint, processes all the new data until now, then stops.
- Problem: this can OOM if the new data until now is large!



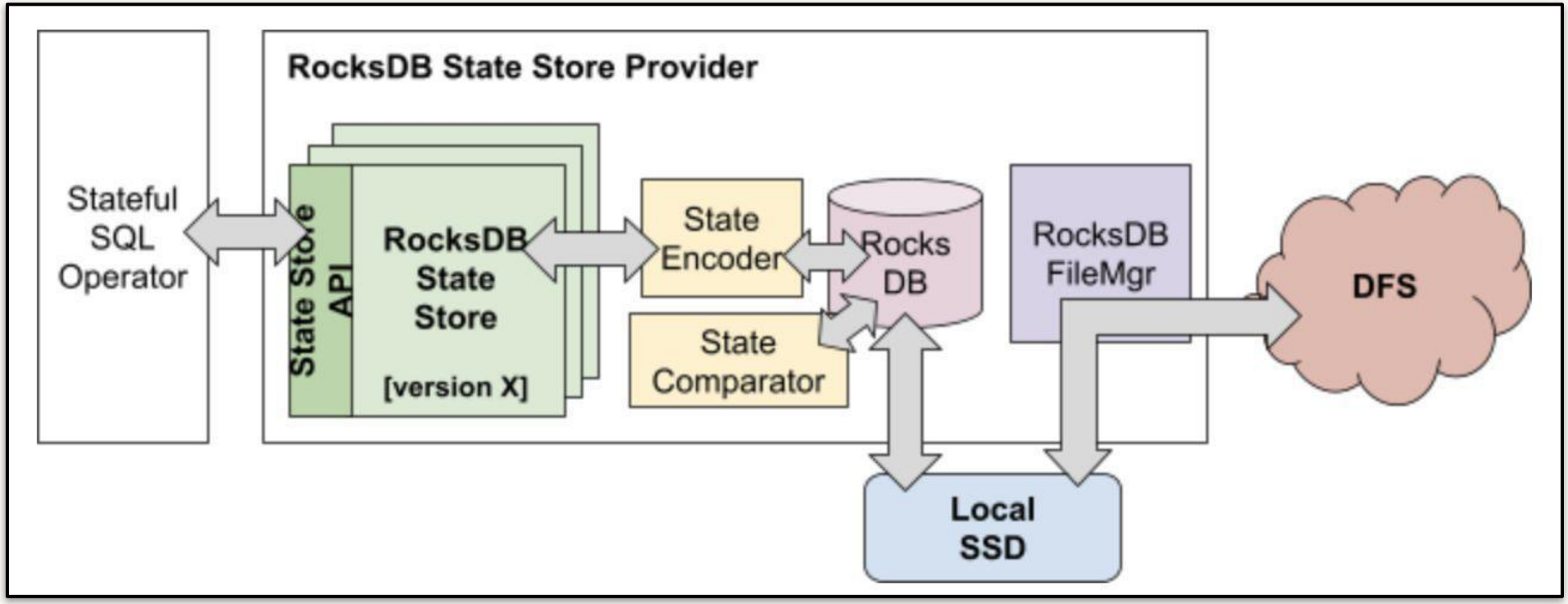
# New Structured Streaming Trigger: `AvailableNow`

- Solution: the new trigger **`AvailableNow`** fixes this problem by running multiple micro-batches to process new data instead.



# RocksDB State Store

## More powerful and efficient state store

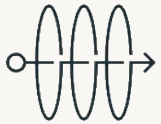



- Now supports storing unlimited per-operator state!
- Allows retrieving state via range scan instead of full scan.

DSV2 API Enhancements

DSV2 Metrics

Parquet 1.12 (Column Index)



Complex Type Support in ORC

Parquet Vectorized Reader

DSV2 Pushdown

Hive Bucket Writing

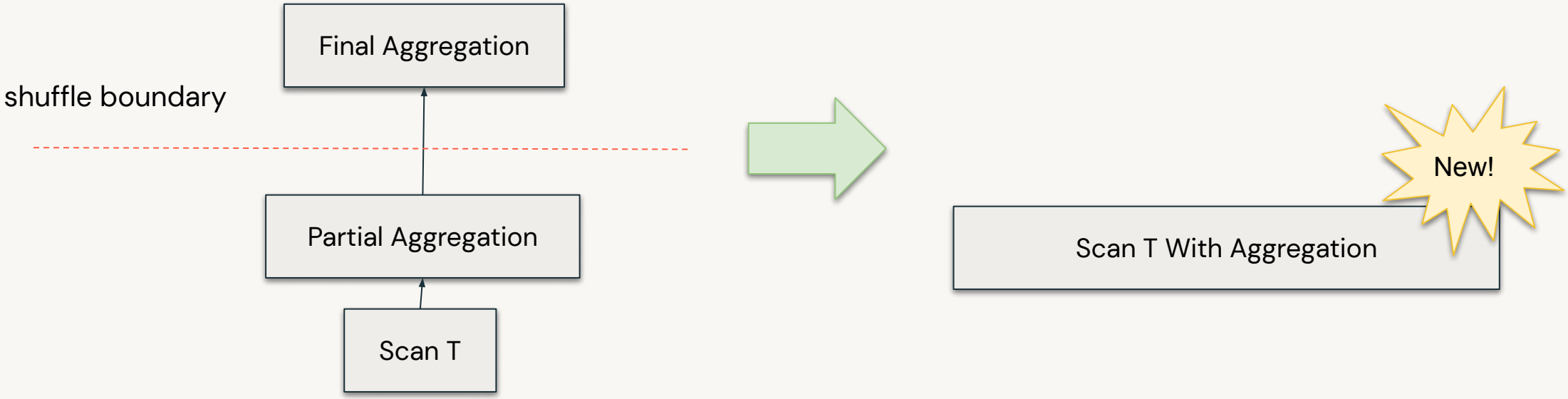
DELTA LAKE

# Extensions

# Data Source V2: New APIs

## Make your data sources more powerful

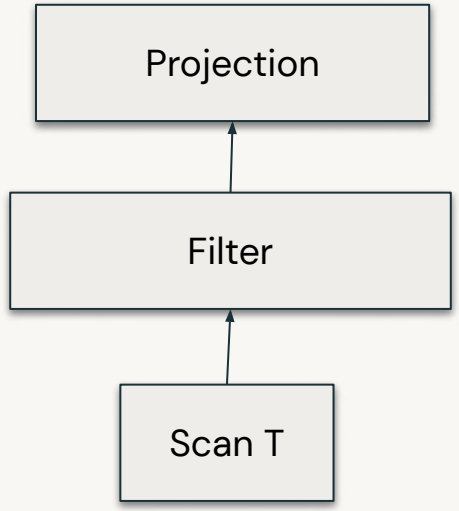
- Aggregate pushdown APIs (**SPARK-34952**)
- FunctionCatalog API (**SPARK-27658**)
- Required sorting and clustering for writes (**SPARK-23889**)



# Data Source V2: New APIs

## Make your data sources more powerful

- Read and write metrics (**SPARK-34338, SPARK-36030**)
- Dynamic filtering (**SPARK-35779**)
- Storage Partitioned Join API and framework (**SPARK-37375**)
- Many more…




Metrics:

- 78234963358 bytes read
- 87563 files scanned
- 456 files skipped

# Data Source V2: New APIs

## Make your data sources more powerful

- APIs for group-based row-level operations (**SPARK-38625**)

> A mix-in interface for `Table` row-level operations support. Data sources can implement this interface to indicate they support rewriting data for DELETE, UPDATE, MERGE operations.
>
> Since: 3.3.0

```
public interface SupportsRowLevelOperations extends Table {
```

> Returns a `RowLevelOperationBuilder` to build a `RowLevelOperation`. Spark will call this method while planning DELETE, UPDATE and MERGE operations that require rewriting data.
>
> Params: info – the row-level operation info such as command (e.g. DELETE) and options
>
> Returns: the row-level operation builder

```
  RowLevelOperationBuilder newRowLevelOperationBuilder(RowLevelOperationInfo info);
}
```

# Data Source V2: New APIs

## Make your data sources more powerful

- APIs for group-based row-level operations (**SPARK-38625**)

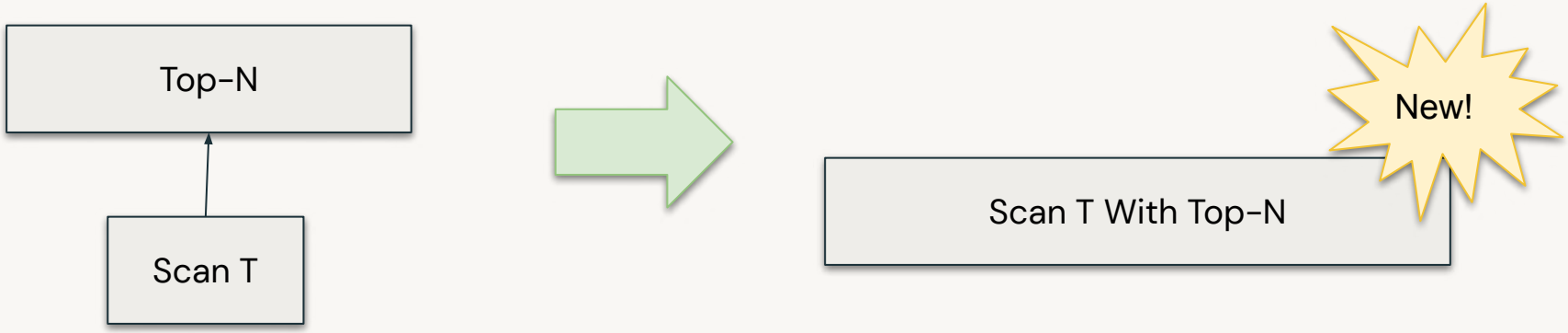
> Returns a `ScanBuilder` to configure a `Scan` for this row-level operation.
>
> Data sources fall into two categories: those that can handle a delta of rows and those that need to replace groups (e.g. partitions, files). Data sources that handle deltas allow Spark to quickly discard unchanged rows and have no requirements for input scans. Data sources that replace groups of rows can discard deleted rows but need to keep unchanged rows to be passed back into the source. This means that scans for such data sources must produce all rows in a group if any are returned. Some data sources will avoid pushing filters into files (file granularity), while others will avoid pruning files within a partition (partition granularity).
>
> For example, if a data source can only replace partitions, all rows from a partition must be returned by the scan, even if a filter can narrow the set of changes to a single file in the partition. Similarly, a data source that can swap individual files must produce all rows from files where at least one record must be changed, not just rows that must be changed.

```
ScanBuilder newScanBuilder(CaseInsensitiveStringMap options);
```

# Data Source V2: Better Pushdown APIs

## Make your data sources more powerful

- DataSource V2 API: Top-N + Sample + LIMIT pushdown (**SPARK-37483** + **SPARK-37038** + **SPARK-37020**)
- Partial + complete aggregate pushdown (**SPARK-37839** + **SPARK-37644**)



# Data Source V2: Better Pushdown APIs

## Make your data sources more powerful

- A new framework to represent Catalyst expressions (**SPARK-37960**)
- General aggregate functions in DataSource V2 API (**SPARK-37789**)

The general implementation of `AggregateFunc`, which contains the upper-cased function name, the `isDistinct` flag and all the inputs. Note that Spark cannot push down partial aggregate with this function to the source, but can only push down the entire aggregate.

The currently supported SQL aggregate functions:

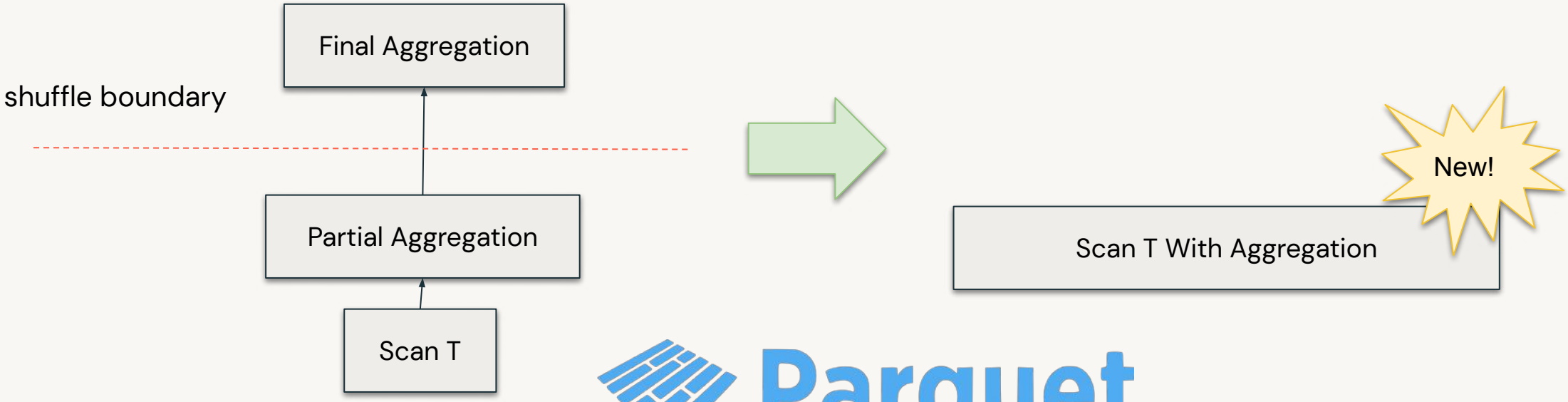
1. `VAR_POP(input1)`
   Since 3.3.0
2. `VAR_SAMP(input1)`
   Since 3.3.0
3. `STDDEV_POP(input1)`
   Since 3.3.0
4. `STDDEV_SAMP(input1)`
   Since 3.3.0
5. `COVAR_POP(input1, input2)`
   Since 3.3.0
6. `COVAR_SAMP(input1, input2)`
   Since 3.3.0
7. `CORR(input1, input2)`
   Since 3.3.0

Since: 3.3.0

```
@Evolving
public final class GeneralAggregateFunc implements AggregateFunc {
  private final String name;
  private final boolean isDistinct;
  private final Expression[] children;

  public String name() { return name; }
  public boolean isDistinct() { return isDistinct; }
```

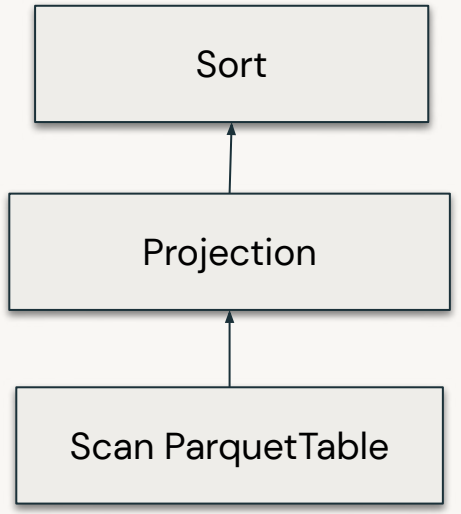
# Built-in File Source Improvements

- Column indexes in Parquet vectorized reader (**SPARK-34289**)
- Aggregate (**MIN/MAX/COUNT**) push down for Parquet (**SPARK-36645**)



# Built-in File Source Improvements

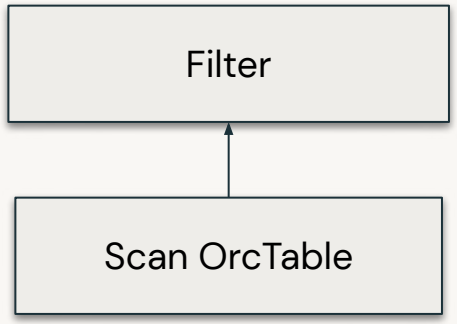
- Column indexes in Parquet vectorized reader (**SPARK-34289**)
- Aggregate (**MIN/MAX/COUNT**) push down for Parquet (**SPARK-36645**)
- Complex types for Parquet vectorized reader (**SPARK-34863**)


```
Schema:
STRUCT<
  fieldA INT,
  fieldB ARRAY<STRING>,
  ...>
```

# Built-in File Source Improvements

- Apache ORC forced positional evolution (**SPARK-32864**)
- Nested columns in ORC vectorized reader (**SPARK-34862**)
- Aggregate push down for ORC (**SPARK-34960**)
- Many more!


```
Schema:
STRUCT<
  fieldA INT,
  fieldB ARRAY<STRING>,
  ...>

 with aggregation:
  SUM(fieldA)
```

# Delta Lake is the foundation of the Lakehouse

The open source table storage layer that brings the best of data lakes and data warehouses

# Delta Lake 2.0 – All of Delta is now open

In Development

ACID Transactions

Scalable Metadata

Time Travel

Open Source

OPTIMIZE

OPTIMIZE ZORDER

Change data feed

Generated column support w/ partitioning

Clones

Coming Soon!

Unified Batch/Streaming

Schema Evolution /Enforcement

Audit History

DML Operations

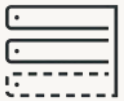
Table Restore

S3 Multi-cluster writes

Data Skipping via Column Stats

Identity Columns

Iceberg to Delta converter



Compaction

MERGE Enhancements

Stream Enhancements

Simplified Logstore

Generated Columns

Multi-part checkpoint writes

Column Mapping

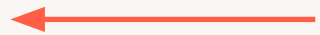
Subqueries in deletes and updates

Fast metadata only deletes

# Delta Features

## Generated columns [DELTA-880]

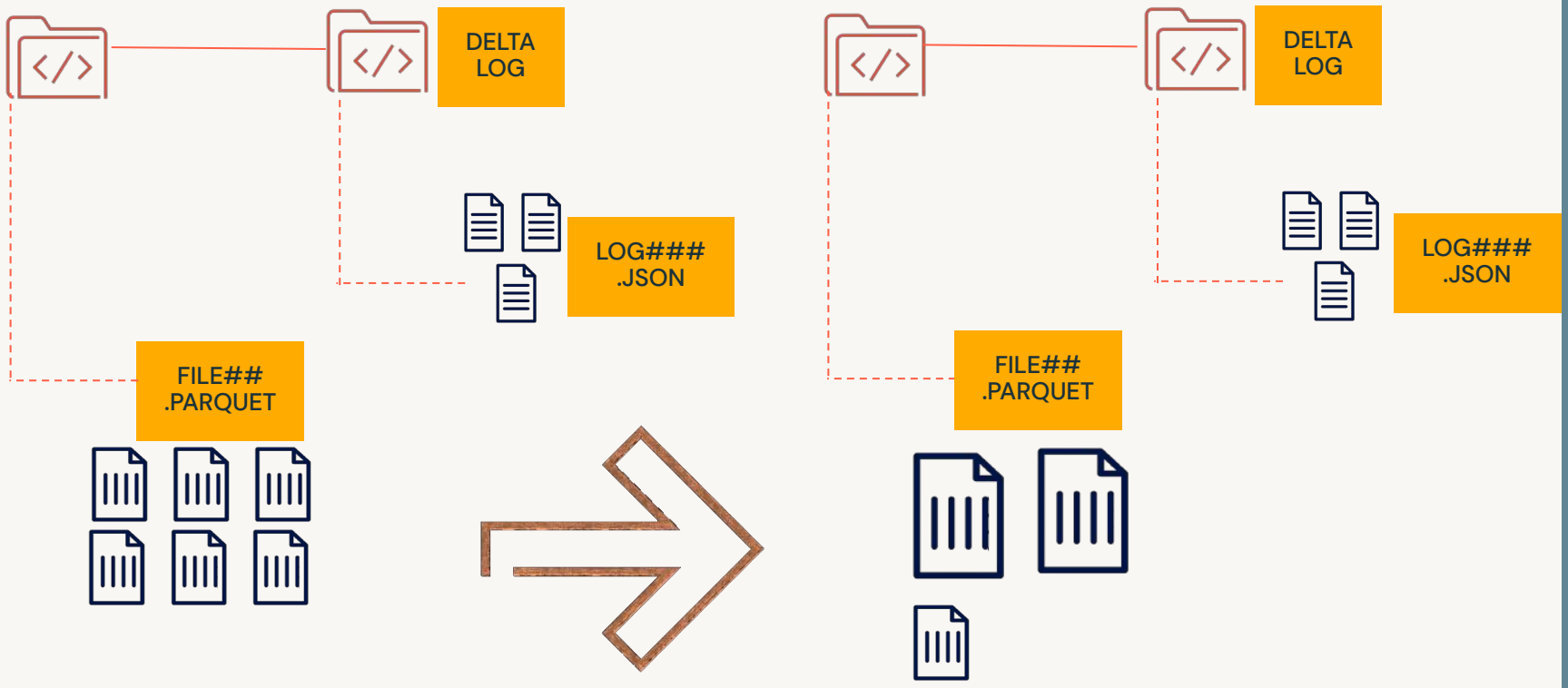
```
CREATE TABLE messages (
        id bigint, type string, data string
      , eventTime timestamp,
      , eventDate date GENERATED ALWAYS AS ( CAST(eventTime AS DATE) )
         COMMENT "derived from eventTime"
      )
USING delta
PARTITIONED BY (eventDate)


SELECT *
FROM messages
WHERE eventTime >= date'2022-06-01'
```



Partition Filter inferred from filter on original column

# Delta Features

## Optimizing data layout with compaction [DELTA-927]




Rearrange the data and/or metadata to speed up queries and/or reduce the metadata size.

# Delta Features

## Data skipping [DELTA-923]

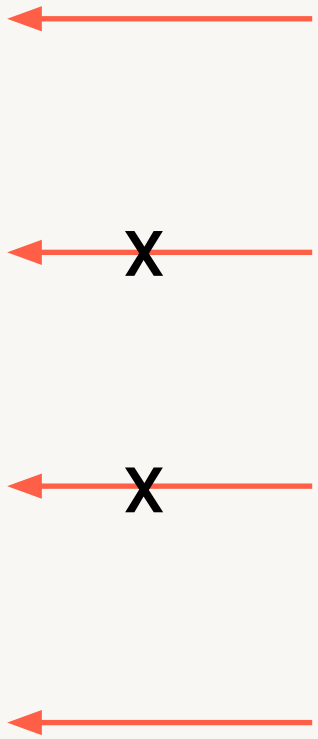
SELECT ROW WHERE NUM = 70

| | | |
|---|---|---|
| 1 | 60 | MARY |
| 2 | 80 | TOM |
| 3 | 60 | LISA |

| | | |
|---|---|---|
| 1 | 50 | JAY |
| 2 | 60 | MIKE |
| 3 | 55 | SUE |

| | | |
|---|---|---|
| 1 | 10 | ANDY |
| 2 | 40 | TONY |
| 3 | 20 | ROSE |

| | | |
|---|---|---|
| 1 | 30 | HARRY |
| 2 | 100 | RON |
| 3 | 70 | JEN |



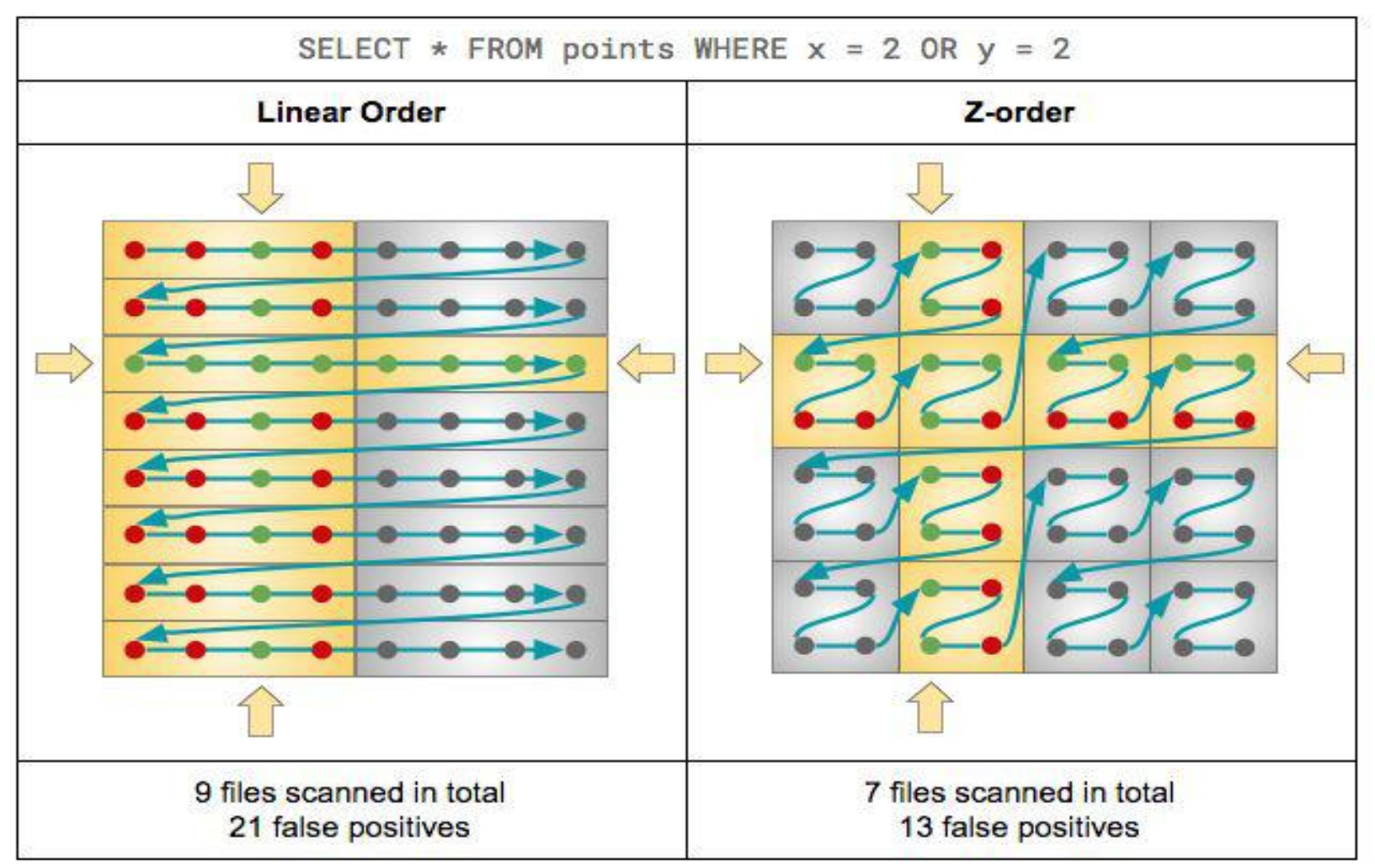

| FILE | MIN | MAX |
|---|---|---|
| 1 | 60 | 80 |
| 2 | 50 | 60 |
| 3 | 10 | 40 |
| 4 | 30 | 100 |

Track file-level stats like **min & max** / leverage them to avoid scanning irrelevant files

In Development

# Delta Features

## Optimizing data layout – Z-Ordering [DELTA-1134]

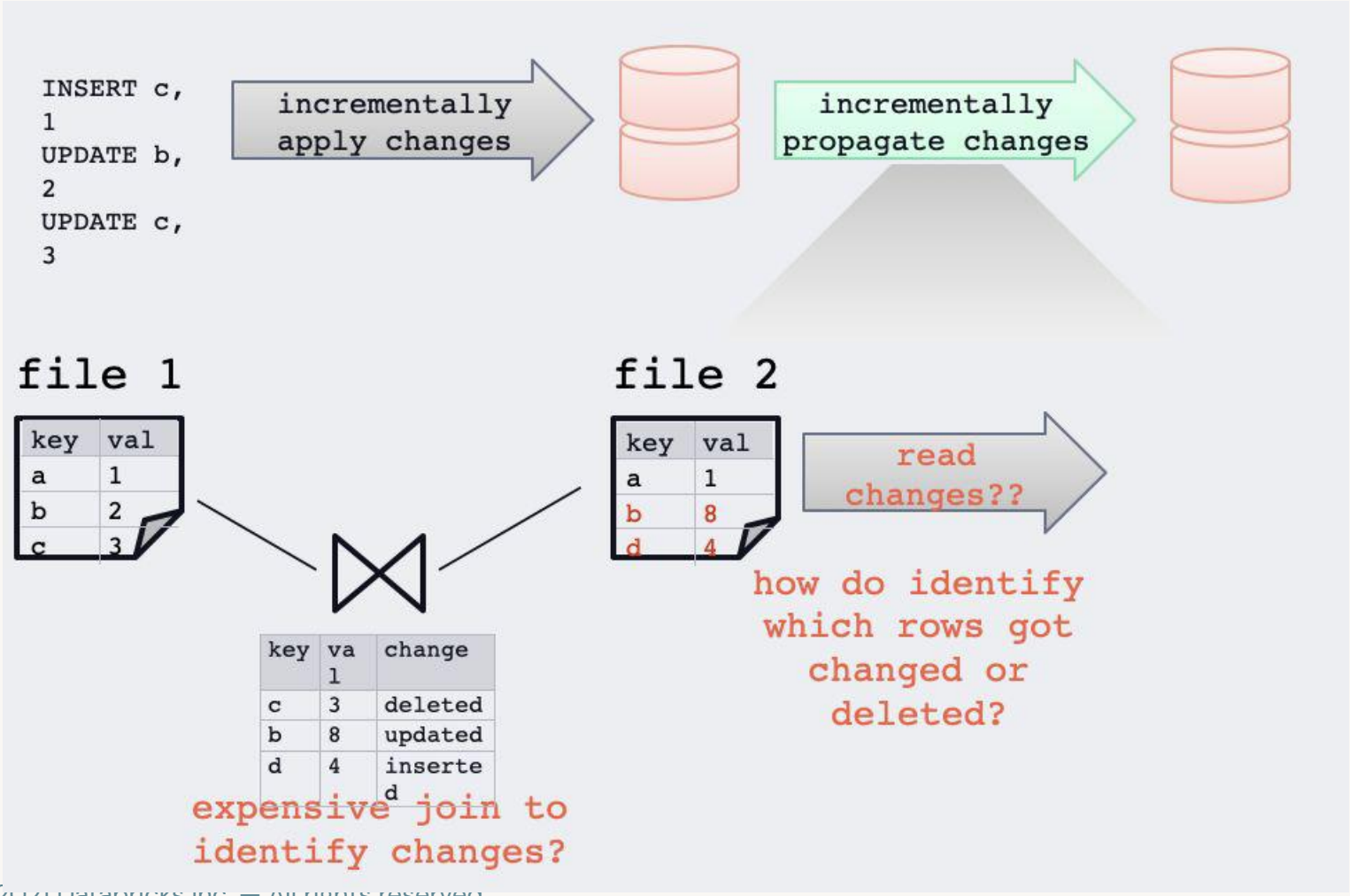



Data clustering via multi-column locality-preserving space-filling curves with offline sorting

In Development

# Delta Features

## Change Data Feed!




Problem: Reading just the changed rows is inefficient without more information.

Joining between two versions can work if there are unique keys, but highly inefficient.

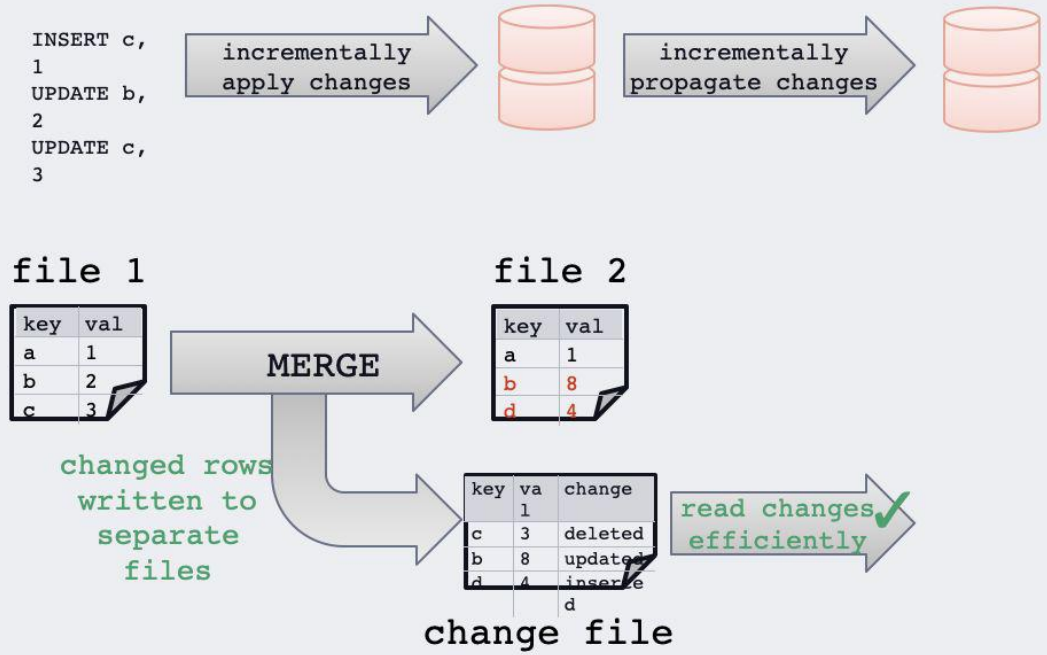
In Development

# Delta Features

## Change Data Feed!

Store the row-level changes in a separate set of files

- Merge/update/delete will produce the additional change files
- Reading change data is efficient as they are separate files, no filtering needed
- Reading normal data unaffected and still efficient

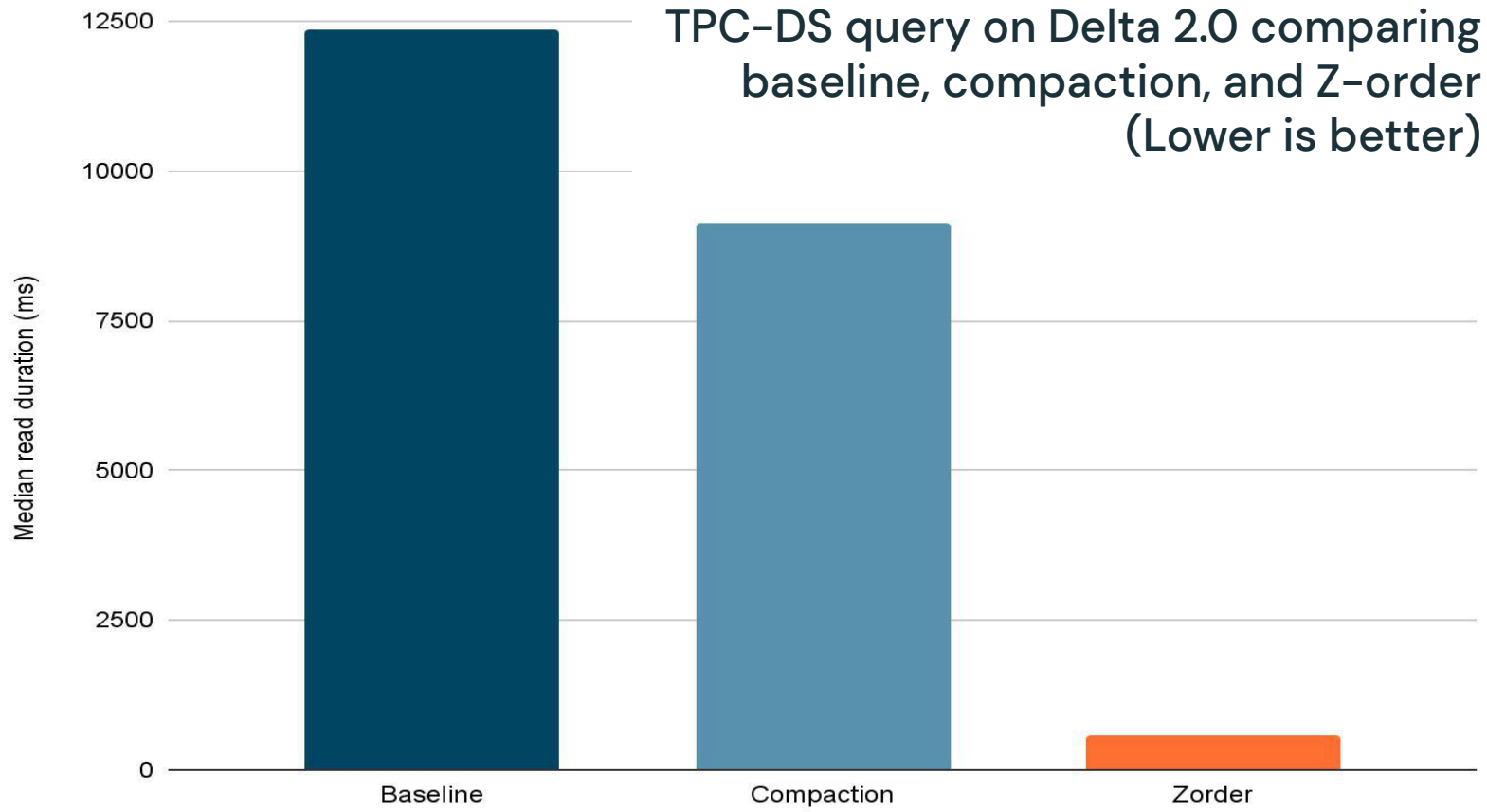



Solution: Store row-level changes generated by update, delete, and merge operations.

In Development

# Delta Features

## Z-order + optimize [DELTA-1134]




Data clustering via multi-column locality-preserving space-filling curves with offline sorting

# Major Spark Improvements in 3.2 & 3.3

**Python**

pandas APIs

Pythonic Error Handling

Python/Pandas UDF Profiler

**SQL**


ANSI Mode

Lateral Join

Interval Type

**Performance**

Adaptive Execution

Compiler Latency Reduction

Bloom Filter

**Usability**

Hidden File Metadata

Inline Type Hints

Visualization and Plotting

**Streaming**

RocksDB State Store

Trigger `AvailableNow`

Session Window

**Misc.**

Log4J 2

Java 17 Beta

Push-based Shuffle

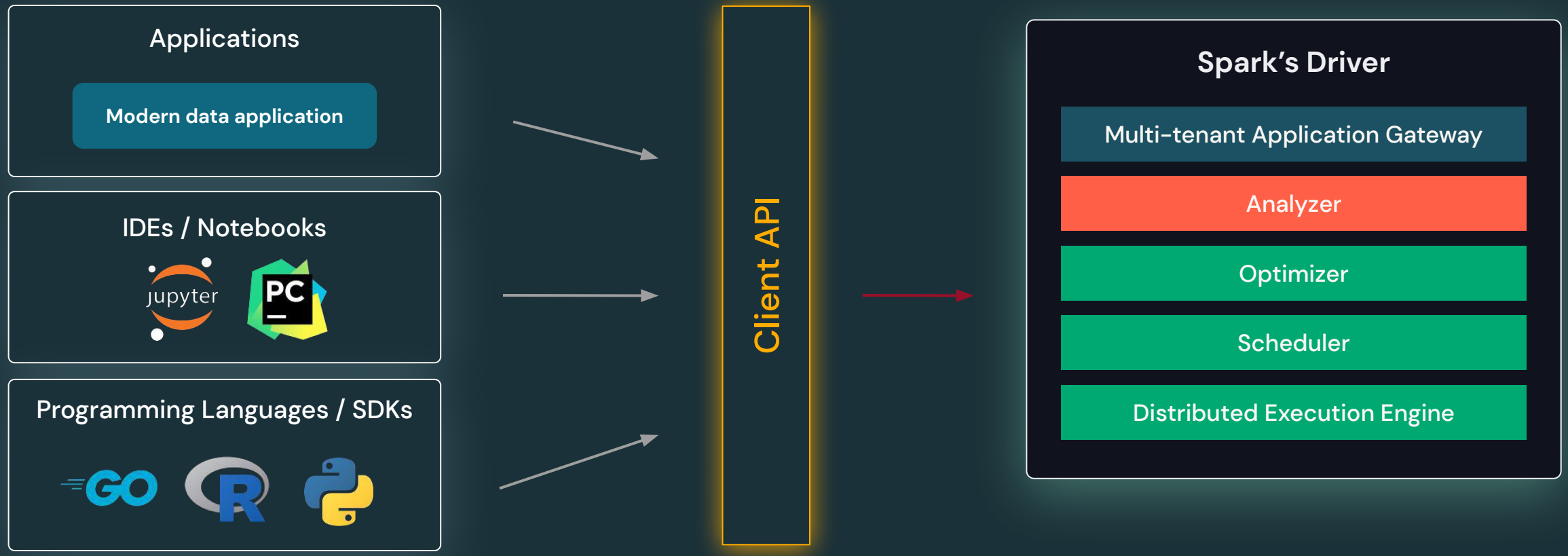
databricks

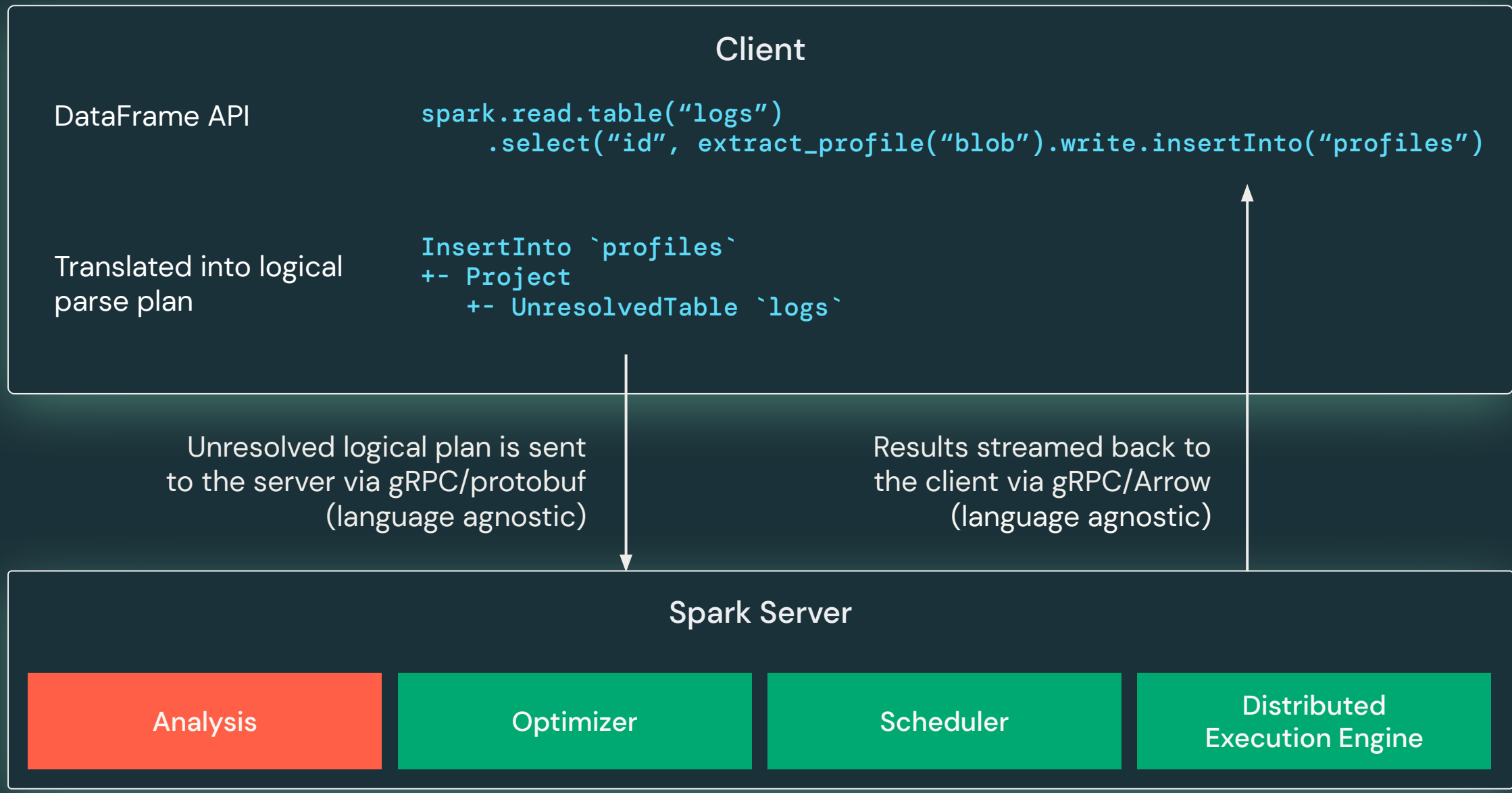
Next Frontier:

# Spark Connect

# Spark Connect

## Thin client, with full power of Apache Spark





# Spark Connect

Power of Apache Spark, everywhere, with a thin client

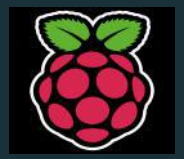
Interactive Development (notebooks / IDEs)

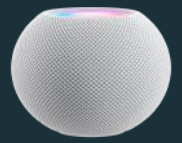


Application Servers / Web Services



Edge Devices






# Spark Connect

## Overcoming multi-tenant operational issues

| | |
|---|---|
| OOM Isolation | OOM takes place locally in thin client, without killing Spark driver |
| Dependency conflicts | Dependencies are contained for each application in their client |
| Upgradability | Servers can be upgraded independent of clients, for performance and security patches |
| Debuggability | Step-through debugging in the client, just like other applications |
| Observability | Logging/metrics in thin client, just like other applications |


Thank you for your contributions!

databricks

Daniel Tenedorio (daniel.tenedorio @ databricks)
Wenchen  Fan (wenchen @ databricks)
Xiao Li (lixiao @ databricks)