# Spark SQL

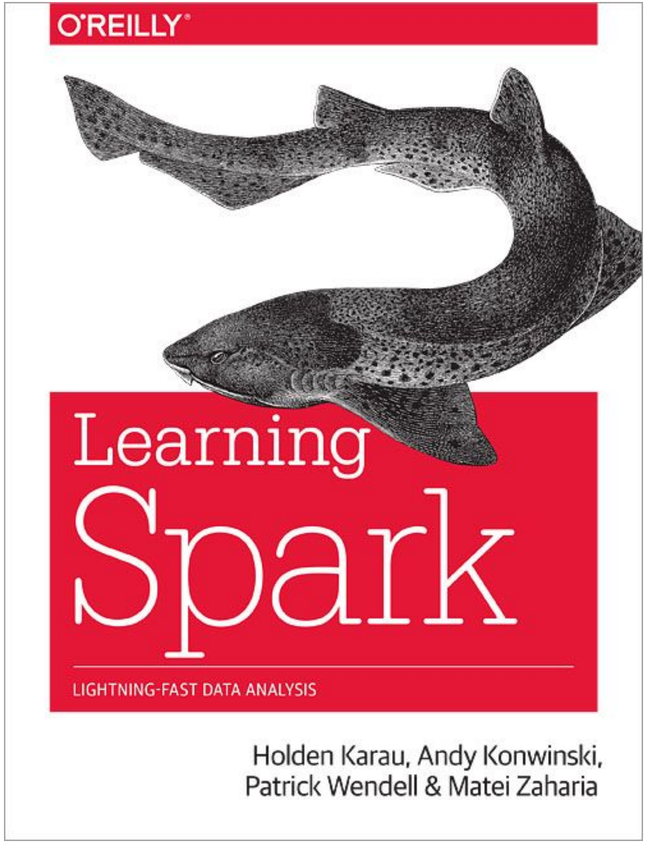
and a bit of partitioning

Anatolii Bardukov
YDATA 2022

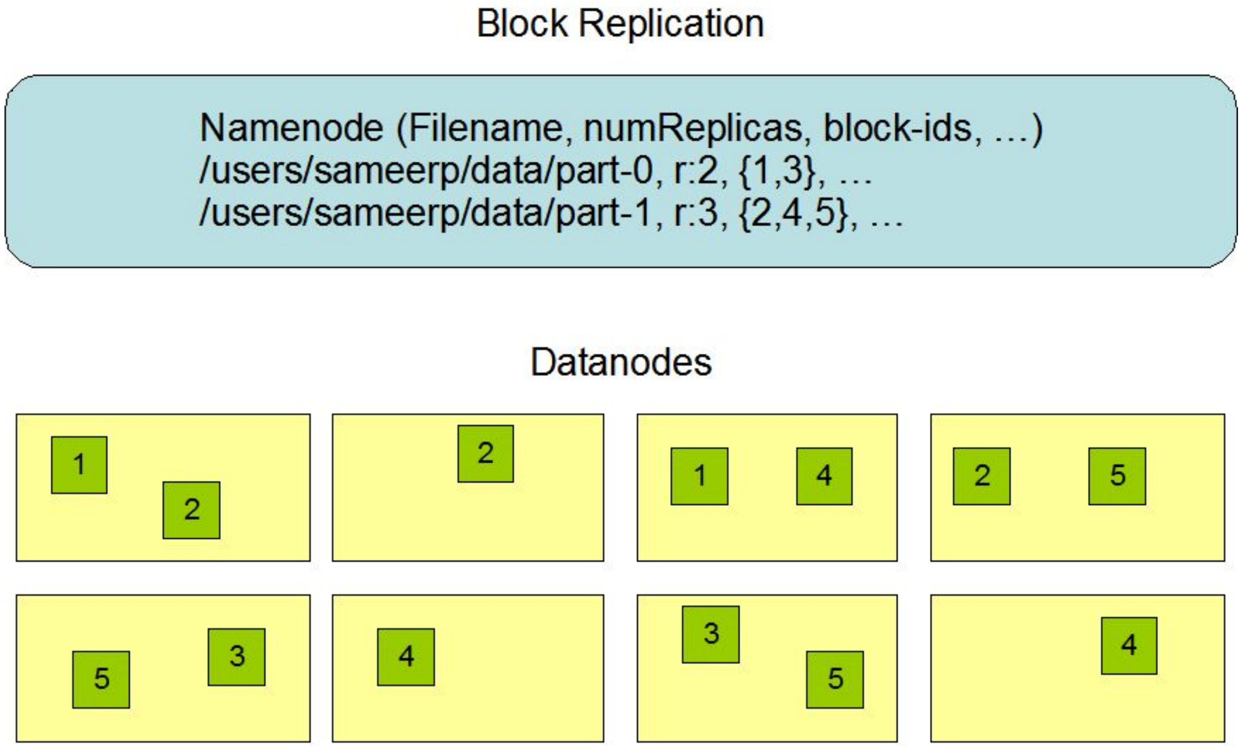
# Plan

1. RDD Joins
2. Partitions over RDD
3. SQL basics
4. SQL-on-Hadoop
5. Spark SQL
    a. simple selection
    b. functions
    c. joins
    d. aggregation
    e. UDF
    f. Window Function

# Book to read


O'REILLY®
Learning
Spark
LIGHTNING-FAST DATA ANALYSIS
Holden Karau, Andy Konwinski,
Patrick Wendell & Matei Zaharia

# Hadoop Partitions



Block Replication

Namenode (Filename, numReplicas, block-ids, …)
/users/sameerp/data/part-0, r:2, {1,3}, …
/users/sameerp/data/part-1, r:3, {2,4,5}, …

Datanodes

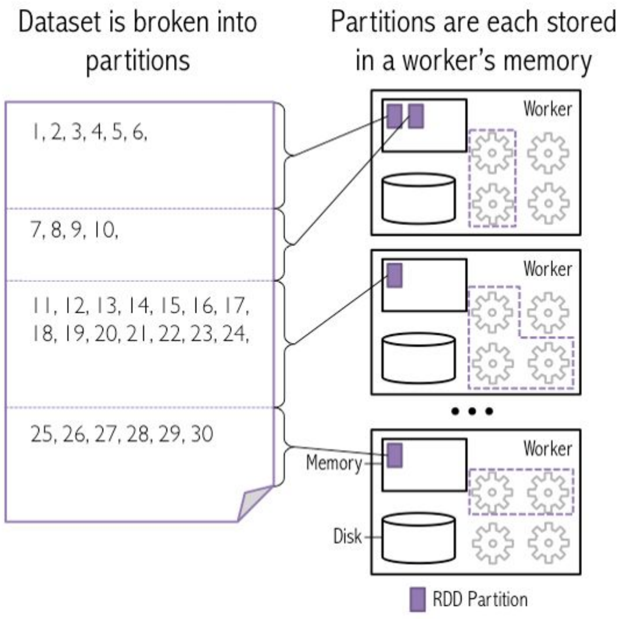
# RDD Partitions

Same idea

RDD is split into partitions

Possible to split RDD into partitions by key

Different partitions are consumed by different Spark workers

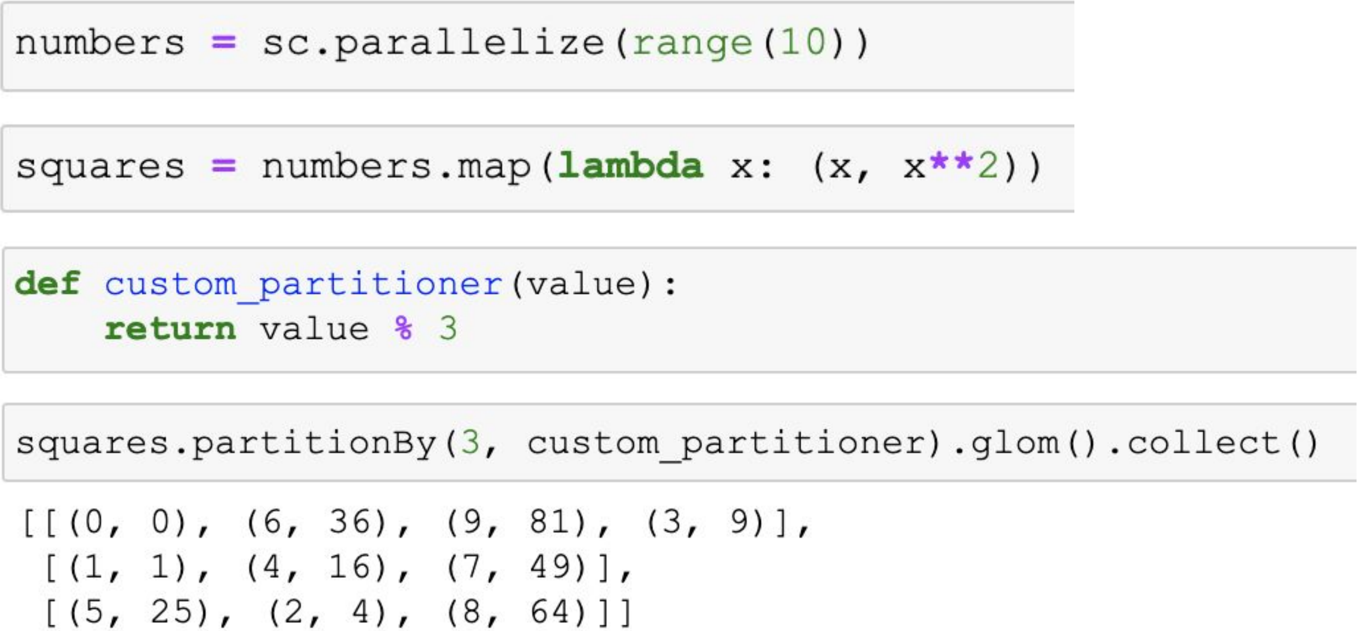


# RDD Partitions management

We can not tell exactly, where each partition goes in terms of nodes - inefficient

But we can create partition function and amount of buckets

# Partition tuning example



```
numbers = sc.parallelize(range(10))
```

```
squares = numbers.map(lambda x: (x, x**2))
```

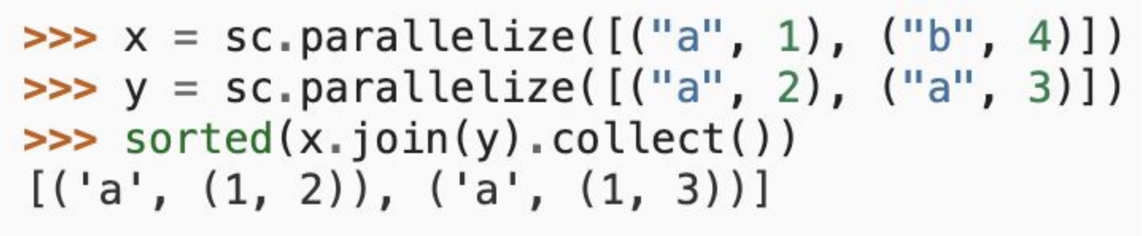
```
def custom_partitioner(value):
    return value % 3
```

```
squares.partitionBy(3, custom_partitioner).glom().collect()
```

```
[[(0, 0), (6, 36), (9, 81), (3, 9)],
 [(1, 1), (4, 16), (7, 49)],
 [(5, 25), (2, 4), (8, 64)]]
```

# RDD Joins

Transformation to join several different RDDs into single one using some key such that key remains same and all other rows are combined into the value



```
>>> x = sc.parallelize([("a", 1), ("b", 4)])
>>> y = sc.parallelize([("a", 2), ("a", 3)])
>>> sorted(x.join(y).collect())
[('a', (1, 2)), ('a', (1, 3))]
```

# Advanced Partition Tuning example

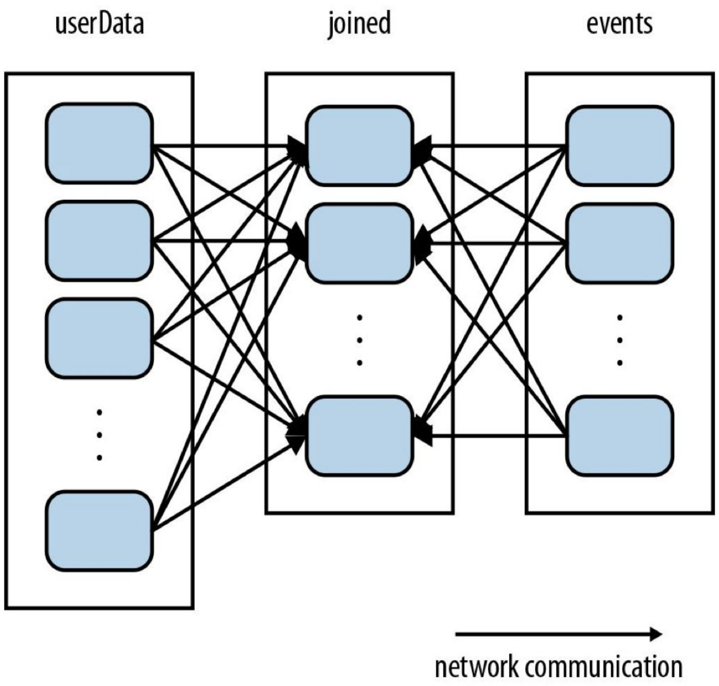


**Figure 4-4. Each join of userData and events without using partitionBy()**

# Advanced Partition Tuning example

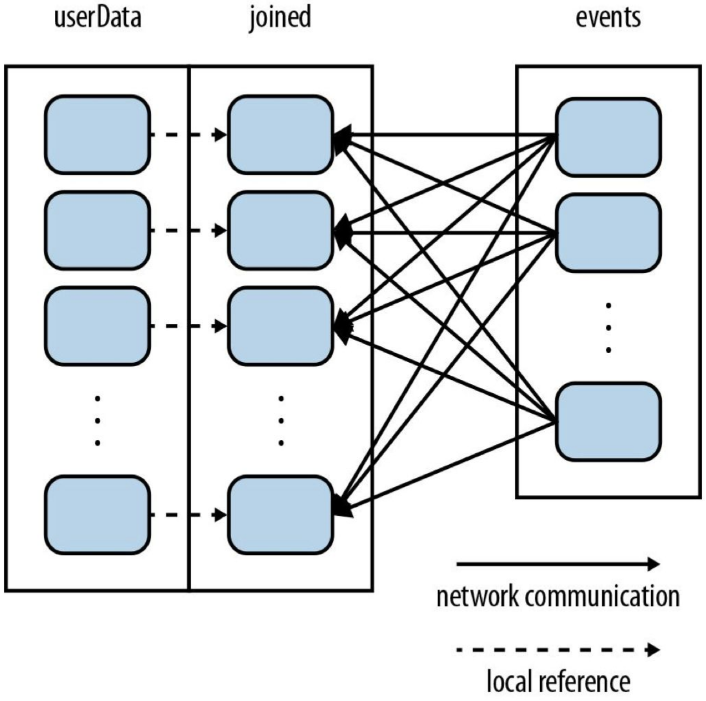
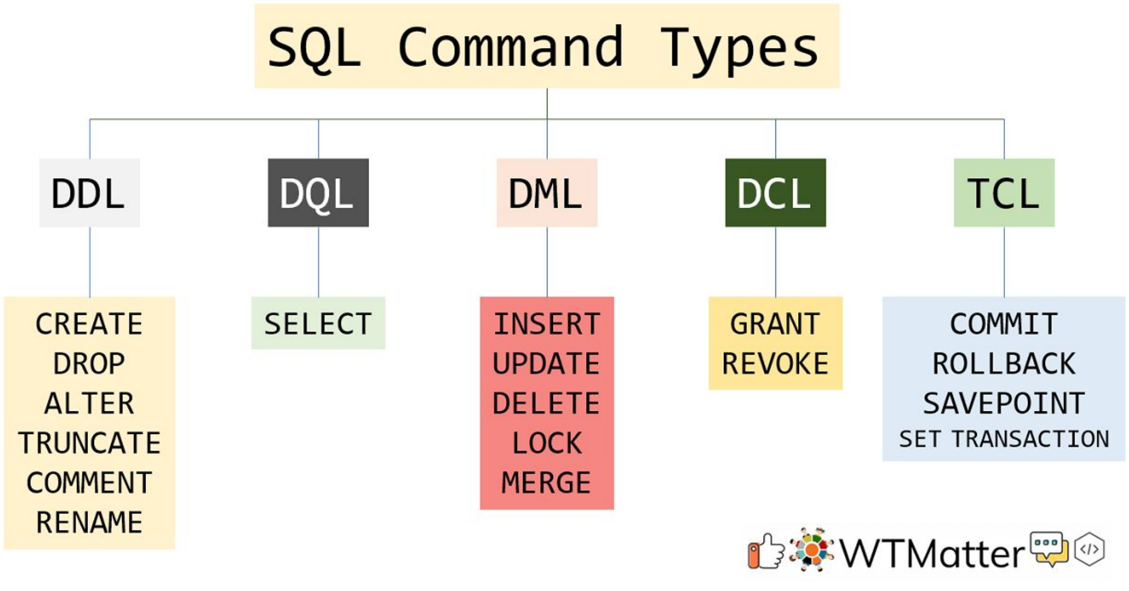


**Figure 4-5. Each join of userData and events using partitionBy()**

# SQL

# SQL

*Structured Query Language* - **declarative** domain-specific language usually used in (relational) DBMS, takes advantage of **relations** within the **structured** data



# SQL ANSI standard

Consists of 10 parts, latest release SQL:2016, SQL 2019 (adds part 15, MDarrays)

https://www.iso.org/standard/63555.html - first part

Usually, DBMS implement SQL features before they go to standard:

Oracle: PL/SQL

Postgres: PL/pgSQL

# SQL syntax difference example

https://www.w3schools.com/sql/sql_top.asp

SELECT **TOP** n from … - *MS Access*

SELECT * from … **LIMIT** N - *MySQL*

SELECT * from … WHERE **ROWNUM** <= n - *Oracle*

# DDL

Data **Definition** Language

Subset of SQL responsible for definition of data structures

Commands: CREATE/ALTER/DROP database, table, index, view

Exists in Spark SQL

# DDL

```
CREATE TABLE [table name] ( [column definitions] ) [table parameters]
```

```
CREATE TABLE employees (
    id            INTEGER       PRIMARY KEY,
    first_name    VARCHAR(50)   not null,
    last_name     VARCHAR(75)   not null,
    fname         VARCHAR(50)   not null,
    dateofbirth   DATE          not null
  );
```

# DQL

Data **Query** Language
(possible alternatives: Data Retrieval Language/Data Selection Language)

Describes details of SELECT statement syntax, used to retrieve data from database

Usually treated as a part of DML

Exists in Spark SQL

# DQL

```
SELECT [ hints , ... ] [ ALL | DISTINCT ] { named_expression [ , ... ] }
    FROM { from_item [ , ...] }
    [ WHERE boolean_expression ]
    [ GROUP BY expression [ , ...] ]
    [ HAVING boolean_expression ]
```

```
[ WITH with query [ , ... ] ]
select statement [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select statement, ... ]
    [ ORDER BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [ , ...] } ]
    [ SORT BY { expression [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [ , ...] } ]
    [ CLUSTER BY { expression [ , ...] } ]
    [ DISTRIBUTE BY { expression [, ...] } ]
    [ WINDOW { named window [ , WINDOW named_window, ... ] } ]
    [ LIMIT { ALL | expression } ]
```

```
SELECT name, age FROM person ORDER BY age DESC NULLS FIRST;
```

# DML

Data **Manipulation** Language

Describes set of commands responsible for updates of the data inside database,

i.e.:

INSERT / UPDATE / DELETE

Partially exists in Spark SQL

# DML

```
INSERT INTO [ TABLE ] table_identifier [ partition_spec ]
    { VALUES ( { value | NULL } [ , ... ] ) [ , ( ... ) ] | query }
```

```
INSERT INTO students VALUES ('Amy Smith', '123 Park Ave, San Jose', 111111);
```

# DCL

Data **Control** Language

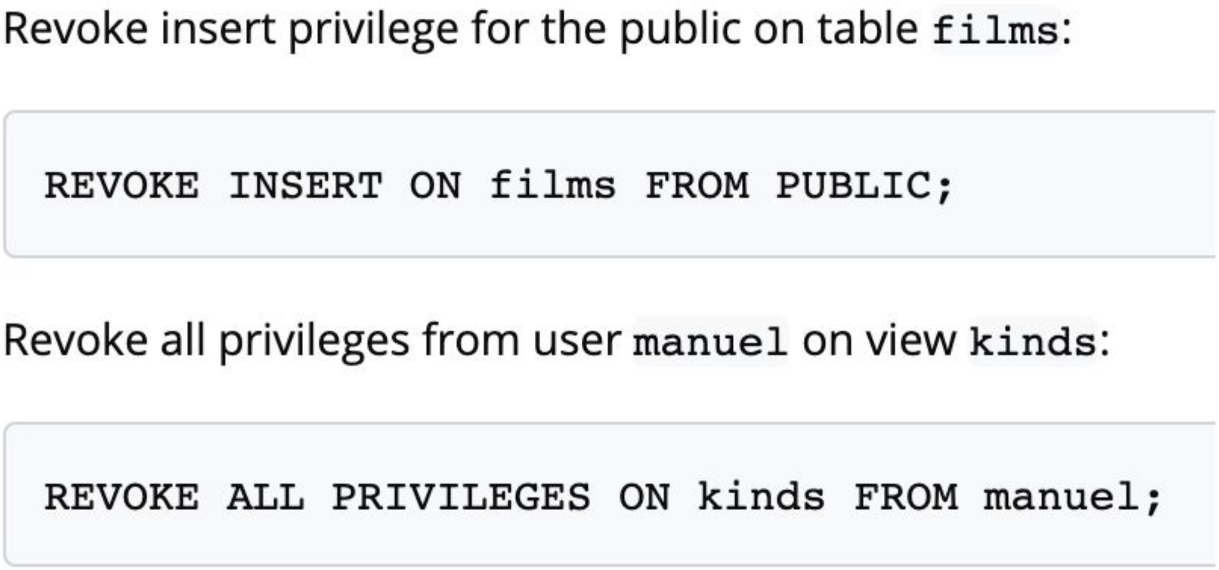
Used to control access to data and manipulate permissions:

GRANT / DENY / REVOKE

Does **not** exists in Spark SQL

# DCL

Postgres Example



Revoke insert privilege for the public on table `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from user `manuel` on view `kinds`:

```
REVOKE ALL PRIVILEGES ON kinds FROM manuel;
```

# TCL

**Transaction Control** Language

Used to manage transactions in the DB:

COMMIT / ROLLBACK / SAVEPOINT

# SQL on Hadoop

Shark (SQL on Spark)

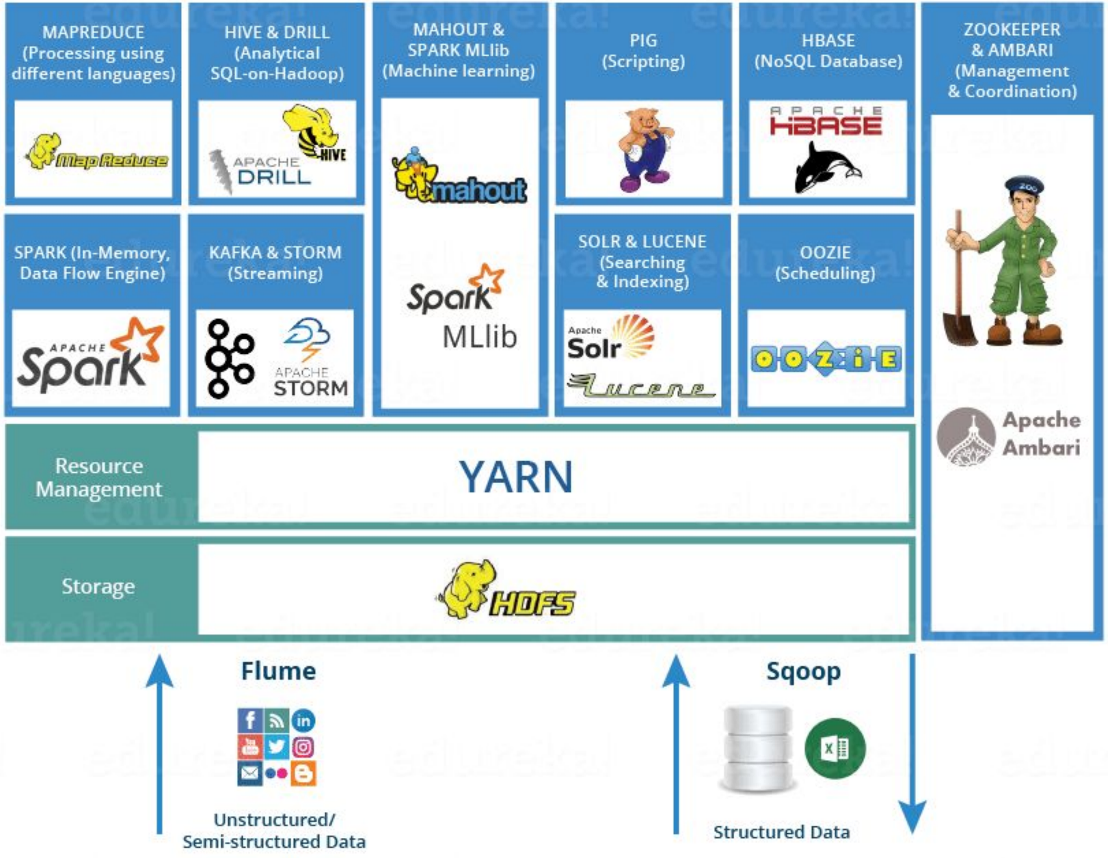
Spark SQL

Hive

Impala

Presto

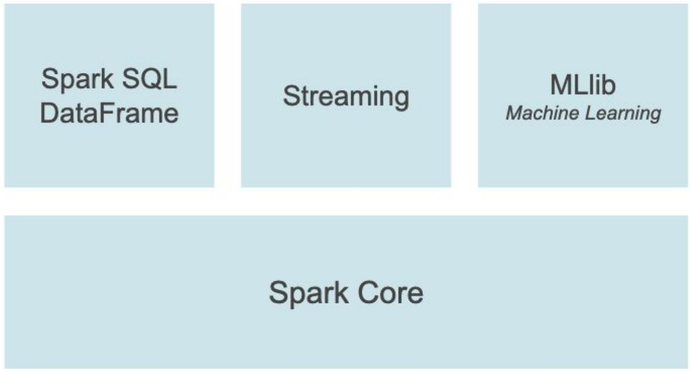
...


MAPREDUCE (Processing using different languages)
MapReduce
HIVE & DRILL (Analytical SQL-on-Hadoop)
APACHE DRILL
HIVE
MAHOUT & SPARK MLlib (Machine learning)
mahout
Spark MLlib
PIG (Scripting)
HBASE (NoSQL Database)
APACHE HBASE
ZOOKEEPER & AMBARI (Management & Coordination)
ZOO
Apache Ambari
SPARK (In-Memory, Data Flow Engine)
APACHE Spark
KAFKA & STORM (Streaming)
APACHE STORM
SOLR & LUCENE (Searching & Indexing)
Apache Solr
Lucene
OOZIE (Scheduling)
OOZIE
Resource Management
YARN
Storage
HDFS
Flume
Unstructured/ Semi-structured Data
Sqoop
Structured Data

# Spark SQL

Spark SQL is a Spark module for structured data processing

It provides a programming abstraction called DataFrames and can also act as a distributed SQL query engine



# Spark SQL architecture

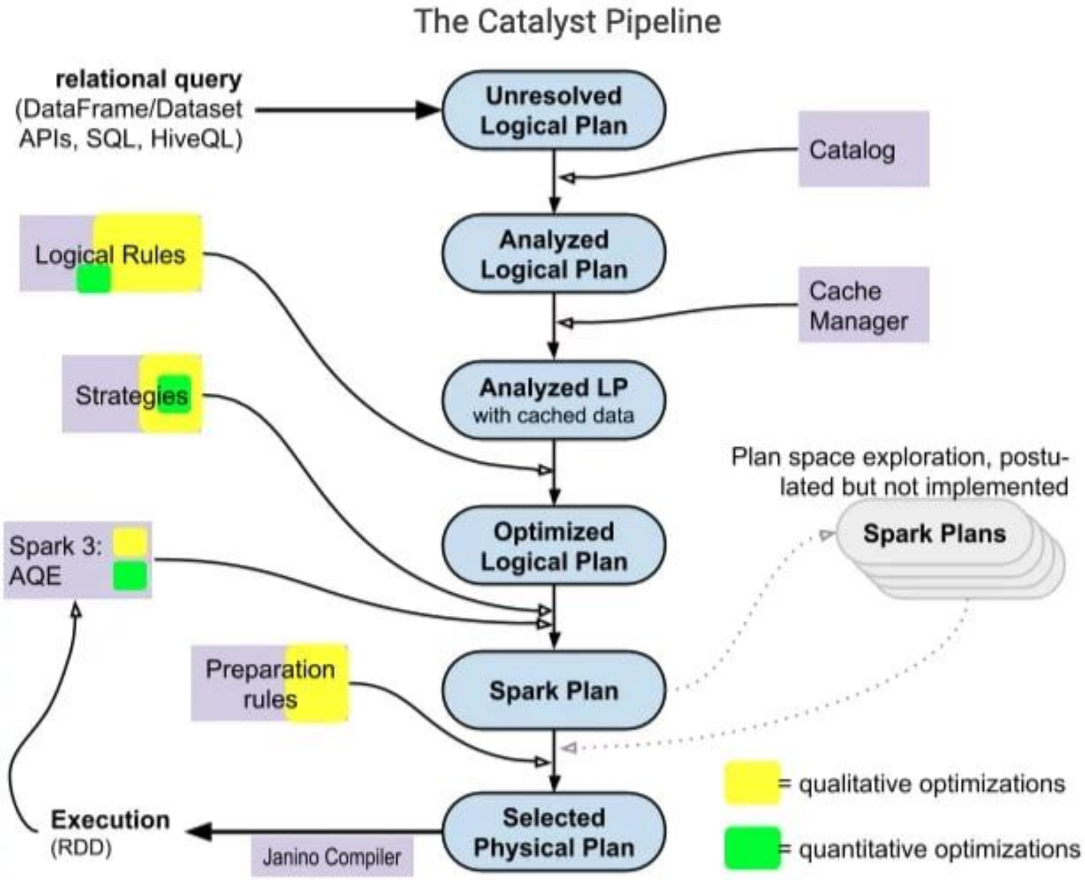
DataFrame API / SQL query goes to planner

Optimizations performed

Logical plan becomes Physical

Plan is materialized to RDD

Adaptive Query Execution optimizes the query while run

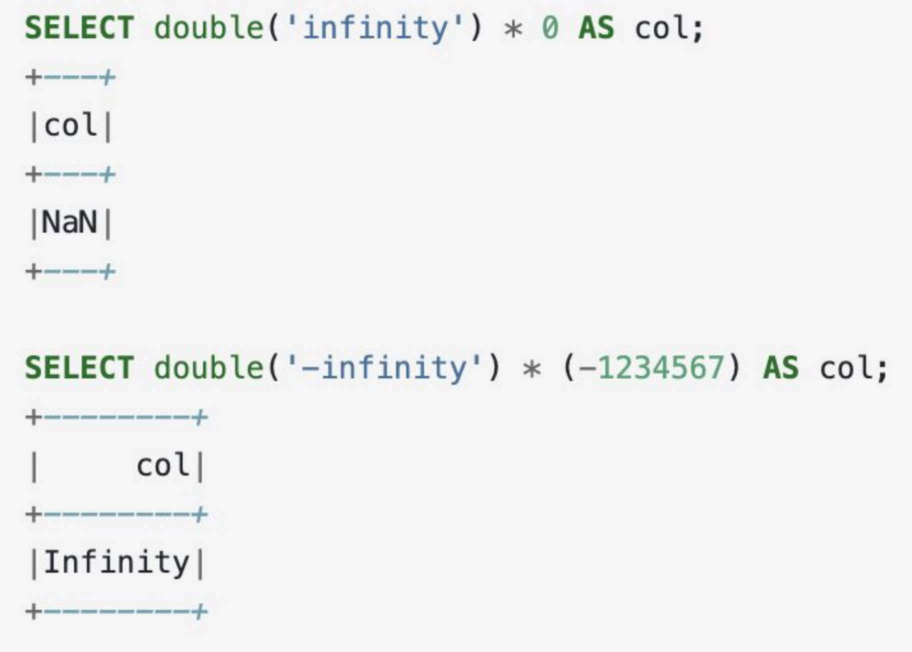


# Spark SQL types

Before we continue with Spark SQL syntax and commands, data types to work with: https://spark.apache.org/docs/latest/sql-ref-datatypes.html

- Numeric types
- String type
- Binary type
- Boolean type
- Datetime type
- Complex types

# Spark SQL types



```
SELECT double('infinity') * 0 AS col;
+---+
|col|
+---+
|NaN|
+---+

SELECT double('-infinity') * (-1234567) AS col;
+--------+
|     col|
+--------+
|Infinity|
+--------+
```

# Spark SQL syntax

https://spark.apache.org/docs/latest/sql-ref-syntax.html

- Data Definition Statements
- Data Manipulation Statements
- Data Retrieval (Queries)
- Auxiliary Statements

# Spark SQL SELECT

Probably, most useful statement, used to query data from tables:

```
SELECT 10;

SELECT 10 as column_name;

SELECT “value”;

SELECT col_name from table_name;
```

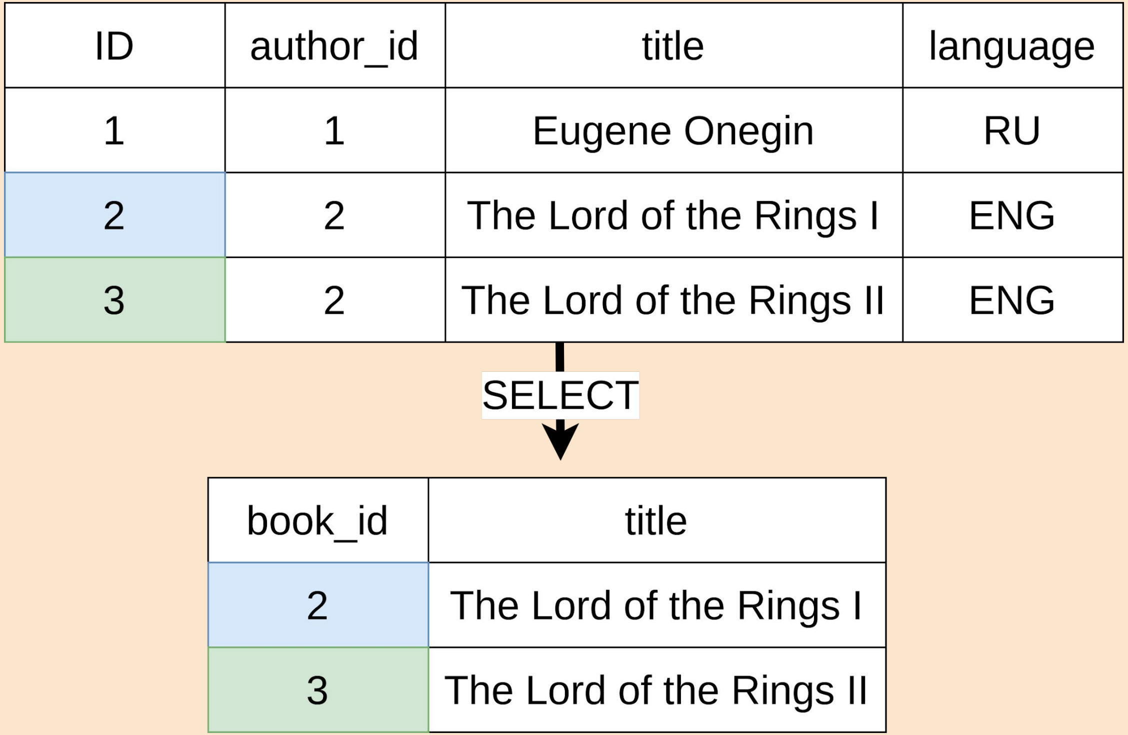
Has a lot of additional params

# Spark SQL WHERE

Filtering condition (boolean expression) for data coming from FROM section of SELECT statement

```
SELECT
  col_name
FROM
  table_name
WHERE
  col_name > 10
```

# SQL small task (table name: books)



| ID | author_id | title | language |
|---|---|---|---|
| 1 | 1 | Eugene Onegin | RU |
| 2 | 2 | The Lord of the Rings I | ENG |
| 3 | 2 | The Lord of the Rings II | ENG |

SELECT

| book_id | title |
|---|---|
| 2 | The Lord of the Rings I |
| 3 | The Lord of the Rings II |

# Answer

```
SELECT id AS "book_id", title

FROM books

WHERE title like '%Lord%’
```

# Spark SQL Functions

We can use different functions in select statement

Spark defines a set of built-in function and supports user-defined functions (UDFs)

Those functions would be applied to data rows

Different types:

- simple, i.e. *lower* for string data
- specific for JSON/Datetime/Map/Array, i.e. *map_concat*
- aggregation/window, i.e. *min, max, mean* for numeric values

# Spark SQL Functions

Analytical functions are applied to the group of rows:

SELECT count(*) FROM table_name

would return count of rows from table

# Spark SQL GROUP BY

Aggregation function (like *count* we had previously) can be used for group of rows.

GROUP BY clause is used to create such groups, where aggregation functions would be applied

For example, we can calculate amount of books written by the author:

SELECT author, count(*) FROM books **GROUP BY author**

# Spark SQL HAVING

Used to filter the results produced by GROUP BY based on the specified condition

Just like WHERE condition, but after GROUP BY is applied

How can we select only authors, who have written more than 100 books?

```
SELECT author, count(*) as books_cnt
FROM books

…
```

# Spark SQL HAVING

Used to filter the results produced by GROUP BY based on the specified condition

Just like WHERE condition, but after GROUP BY is applied

How can we select only authors, who have written more than 100 books:

```
SELECT author, count(*) as books_cnt
FROM books
GROUP BY author
HAVING books_cnt > 100
```

# Spark SQL naming and aliasing

In previous example we used naming of the column:

SELECT 10 **as col_name**;

We can also use aliases for tables:

SELECT **t**.* from table **t**;

# Spark SQL ORDER BY

Used to return the result rows in a sorted manner in the specified order

Guarantees a total order in the output

For example, we can order books by the year there were written:

```
SELECT name, year FROM books ORDER BY year DESC
```

# Spark SQL SORT BY

Used to return the result rows sorted within each partition in the specified order

When there is more than one partition SORT BY may return result that is partially ordered

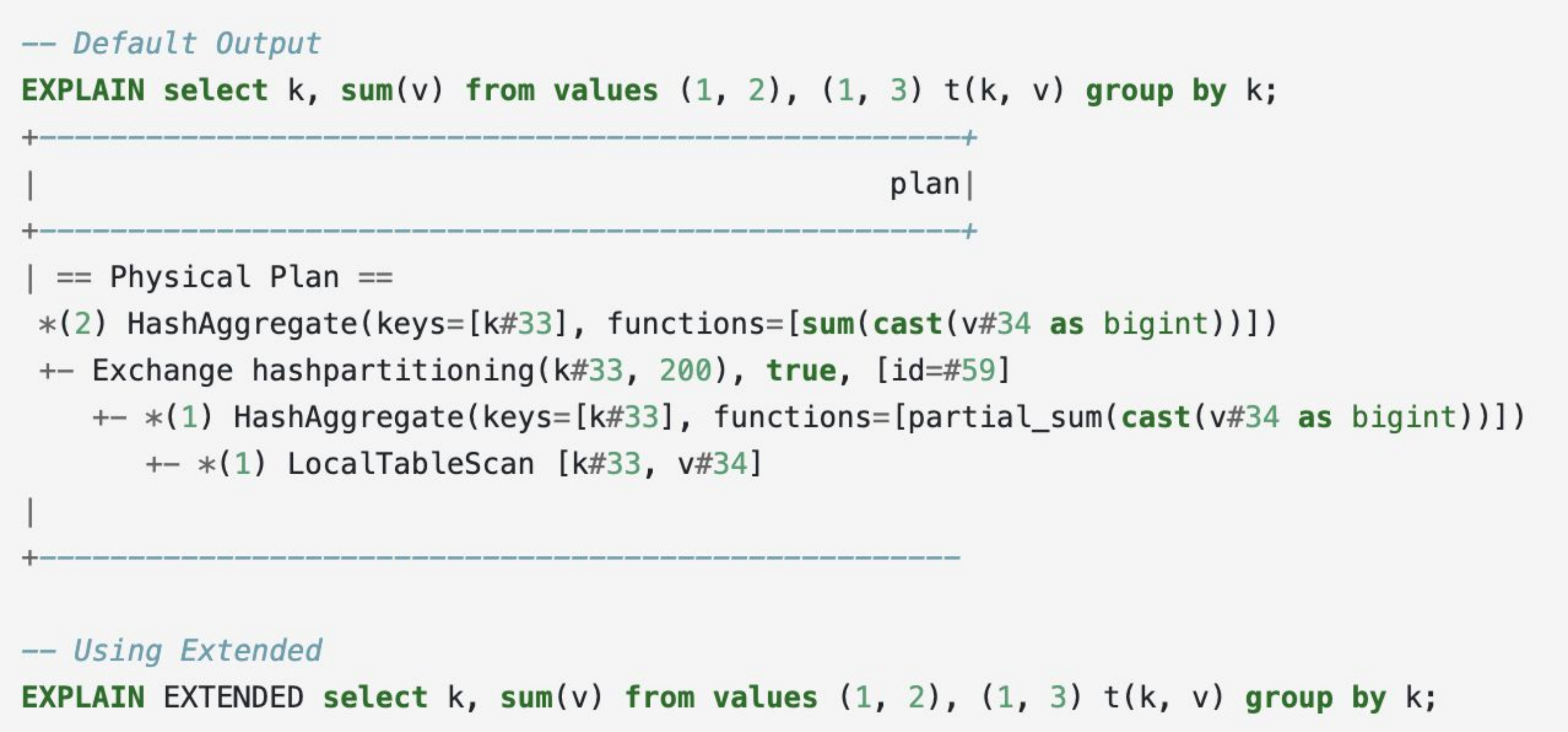
Different than ORDER BY clause which guarantees a total order of the output

# Spark SQL EXPLAIN

Statement is used to provide logical/physical plans for an input statement

By default, this clause provides information about a physical plan only, but can be EXTENDED

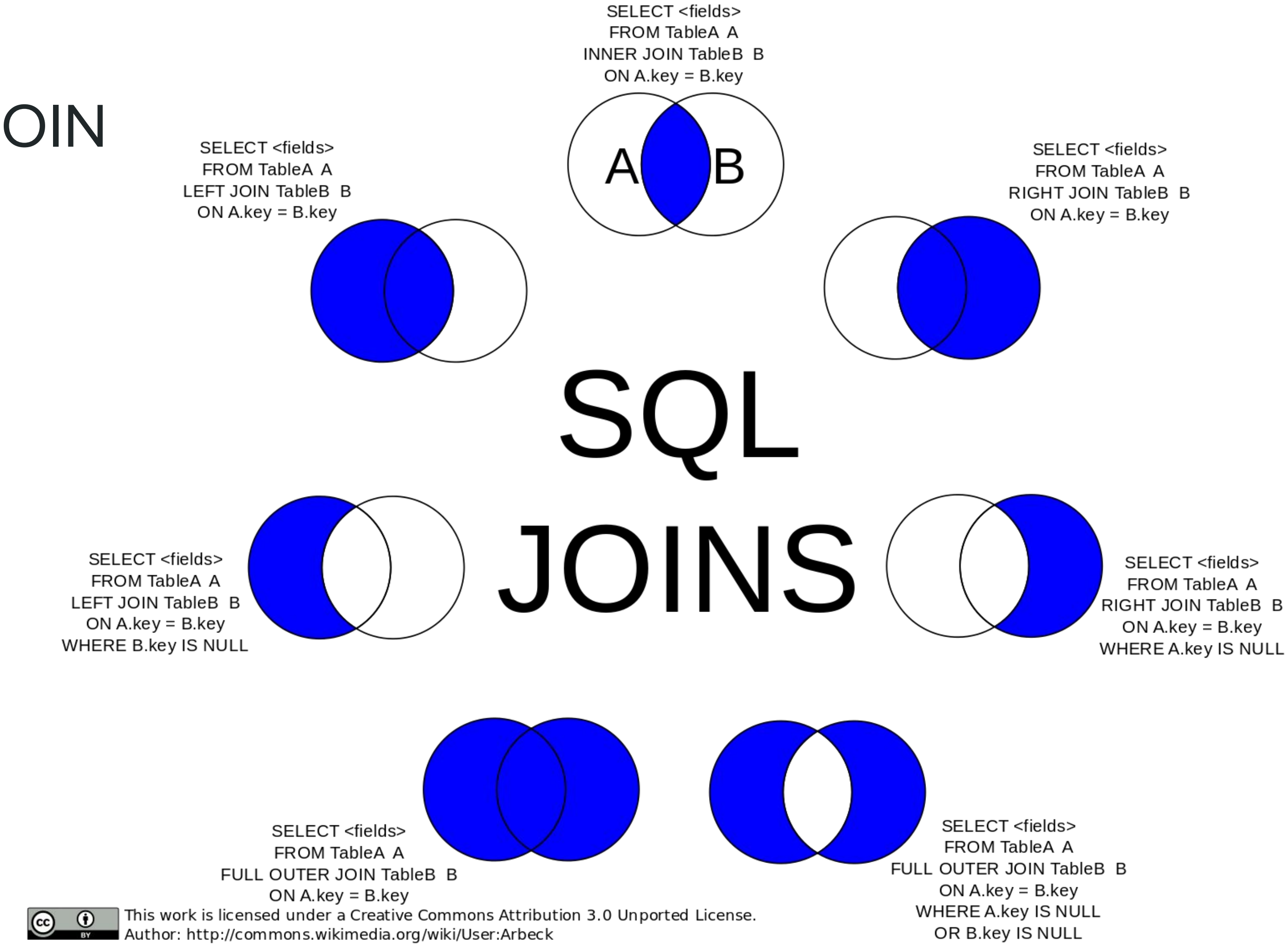
# Spark SQL EXPLAIN



```
-- Default Output
EXPLAIN select k, sum(v) from values (1, 2), (1, 3) t(k, v) group by k;
+----------------------------------------------------+
|                                                plan|
+----------------------------------------------------+
| == Physical Plan ==
 *(2) HashAggregate(keys=[k#33], functions=[sum(cast(v#34 as bigint))])
 +- Exchange hashpartitioning(k#33, 200), true, [id=#59]
    +- *(1) HashAggregate(keys=[k#33], functions=[partial_sum(cast(v#34 as bigint))])
       +- *(1) LocalTableScan [k#33, v#34]
|
+---------------------------------------------------


-- Using Extended
EXPLAIN EXTENDED select k, sum(v) from values (1, 2), (1, 3) t(k, v) group by k;
```

# Spark SQL JOIN



```
SELECT <fields>
FROM TableA  A
INNER JOIN TableB  B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA  A
RIGHT JOIN TableB  B
ON A.key = B.key
```

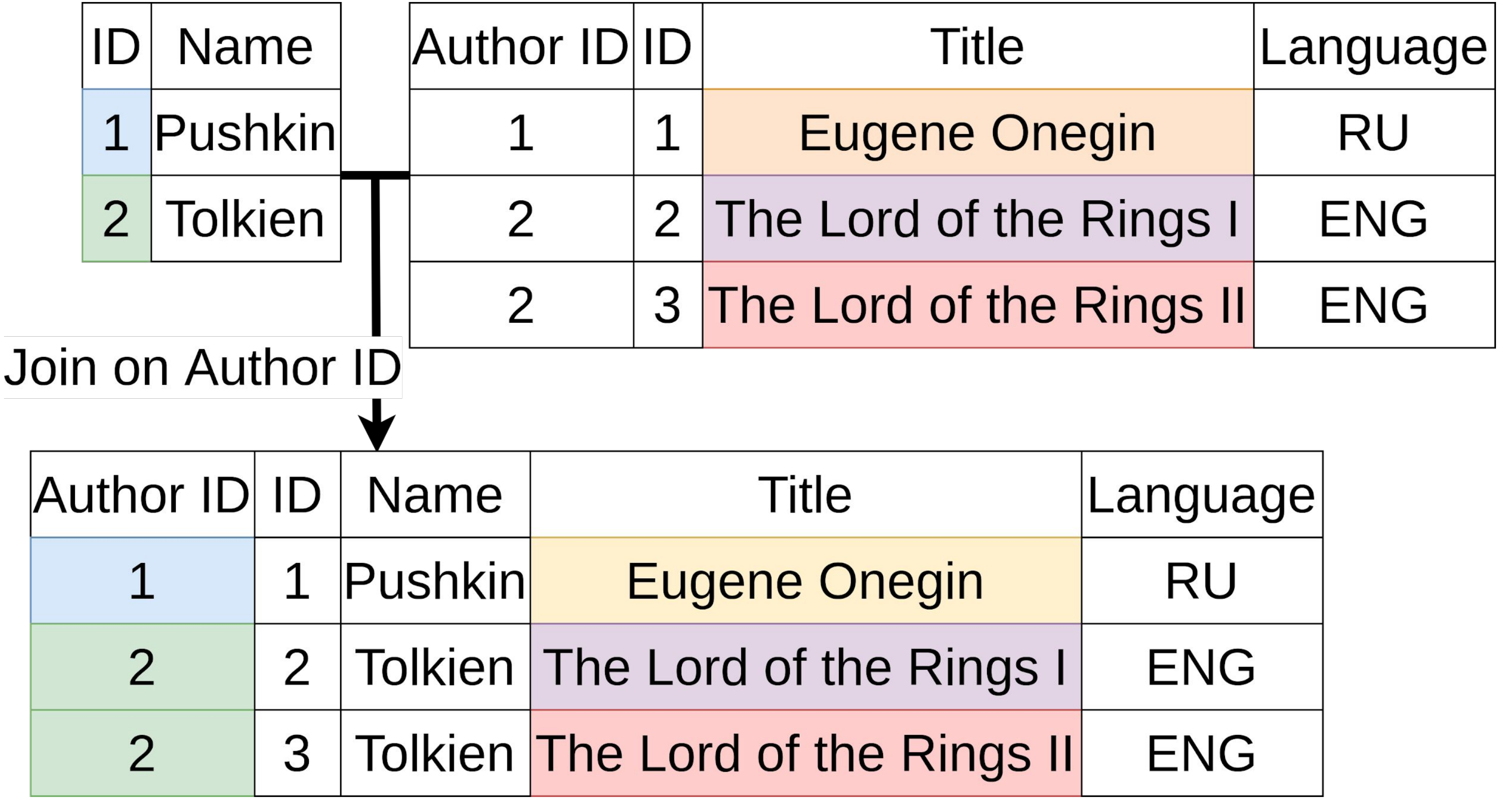
```
SELECT <fields>
FROM TableA  A
LEFT JOIN TableB  B
ON A.key = B.key
WHERE B.key IS NULL
```

```
SELECT <fields>
FROM TableA  A
RIGHT JOIN TableB  B
ON A.key = B.key
WHERE A.key IS NULL
```

```
SELECT <fields>
FROM TableA  A
FULL OUTER JOIN TableB  B
ON A.key = B.key
```

```
SELECT <fields>
FROM TableA  A
FULL OUTER JOIN TableB  B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key IS NULL
```

# Spark SQL JOIN



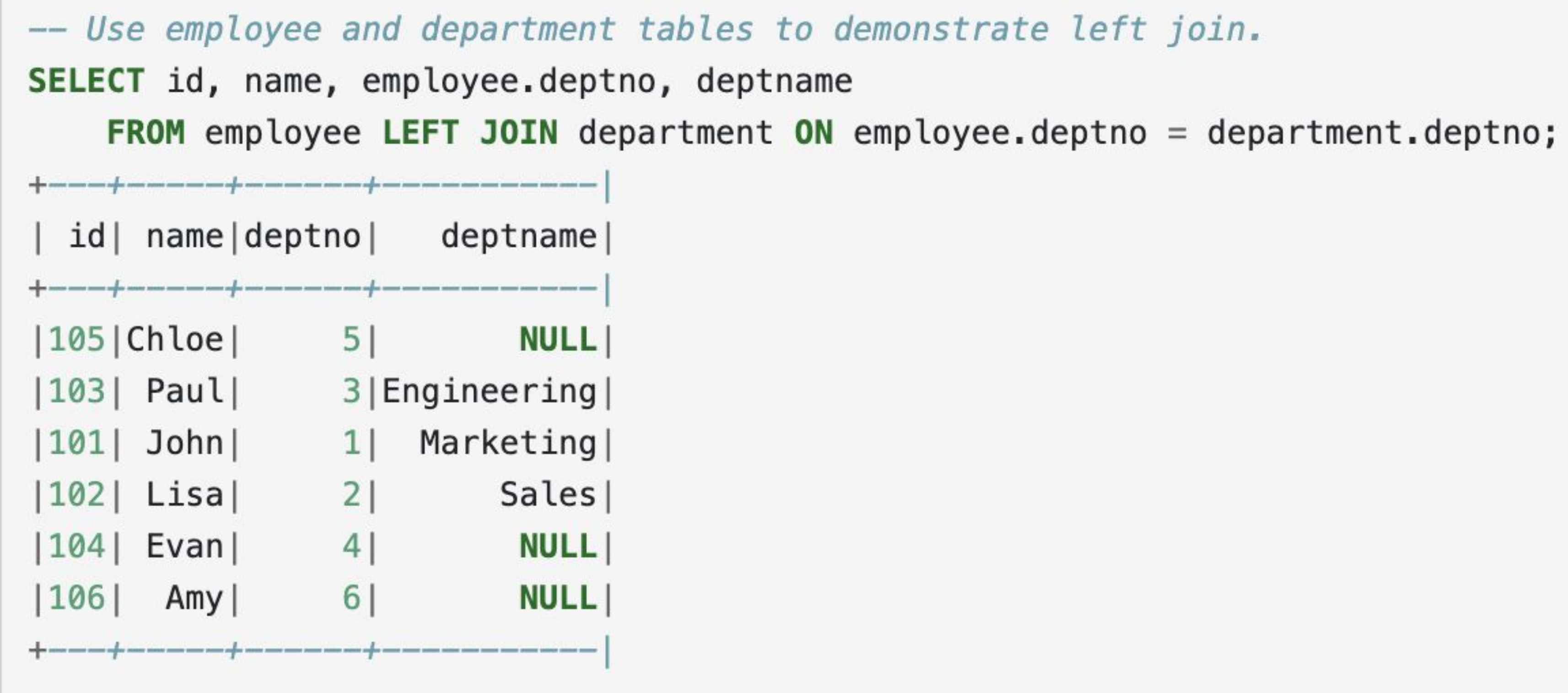
| ID | Name |
|---|---|
| 1 | Pushkin |
| 2 | Tolkien |

| Author ID | ID | Title | Language |
|---|---|---|---|
| 1 | 1 | Eugene Onegin | RU |
| 2 | 2 | The Lord of the Rings I | ENG |
| 2 | 3 | The Lord of the Rings II | ENG |

Join on Author ID

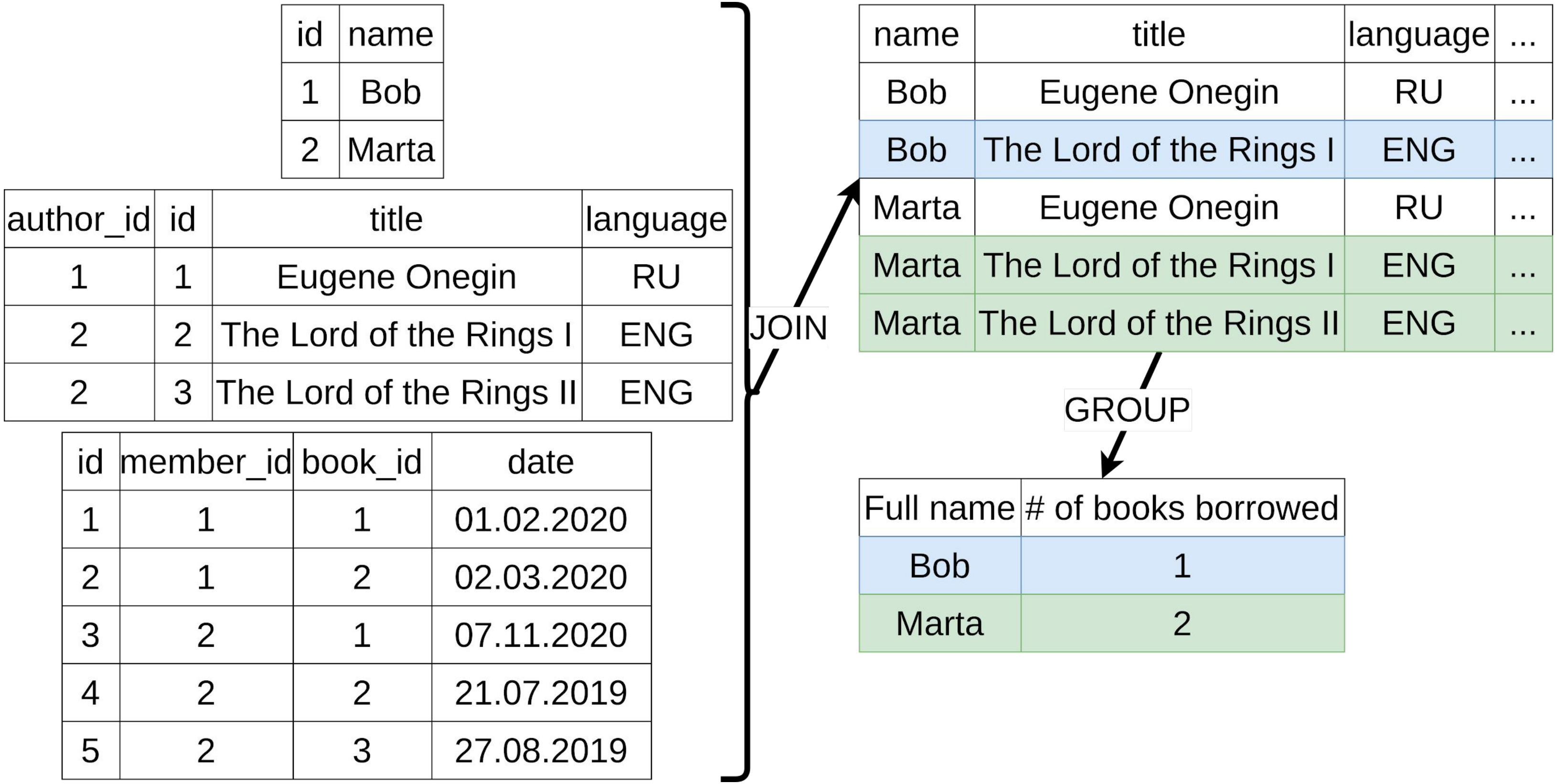
| Author ID | ID | Name | Title | Language |
|---|---|---|---|---|
| 1 | 1 | Pushkin | Eugene Onegin | RU |
| 2 | 2 | Tolkien | The Lord of the Rings I | ENG |
| 2 | 3 | Tolkien | The Lord of the Rings II | ENG |

# Spark SQL Join



```
-- Use employee and department tables to demonstrate left join.
SELECT id, name, employee.deptno, deptname
    FROM employee LEFT JOIN department ON employee.deptno = department.deptno;
+---+-----+------+-----------|
| id| name|deptno|   deptname|
+---+-----+------+-----------|
|105|Chloe|     5|       NULL|
|103| Paul|     3|Engineering|
|101| John|     1|  Marketing|
|102| Lisa|     2|      Sales|
|104| Evan|     4|       NULL|
|106|  Amy|     6|       NULL|
+---+-----+------+-----------|
```

# Harder question



| id | name |
|---|---|
| 1 | Bob |
| 2 | Marta |

| author_id | id | title | language |
|---|---|---|---|
| 1 | 1 | Eugene Onegin | RU |
| 2 | 2 | The Lord of the Rings I | ENG |
| 2 | 3 | The Lord of the Rings II | ENG |

| id | member_id | book_id | date |
|---|---|---|---|
| 1 | 1 | 1 | 01.02.2020 |
| 2 | 1 | 2 | 02.03.2020 |
| 3 | 2 | 1 | 07.11.2020 |
| 4 | 2 | 2 | 21.07.2019 |
| 5 | 2 | 3 | 27.08.2019 |

JOIN

| name | title | language | ... |
|---|---|---|---|
| Bob | Eugene Onegin | RU | ... |
| Bob | The Lord of the Rings I | ENG | ... |
| Marta | Eugene Onegin | RU | ... |
| Marta | The Lord of the Rings I | ENG | ... |
| Marta | The Lord of the Rings II | ENG | ... |

GROUP

| Full name | # of books borrowed |
|---|---|
| Bob | 1 |
| Marta | 2 |

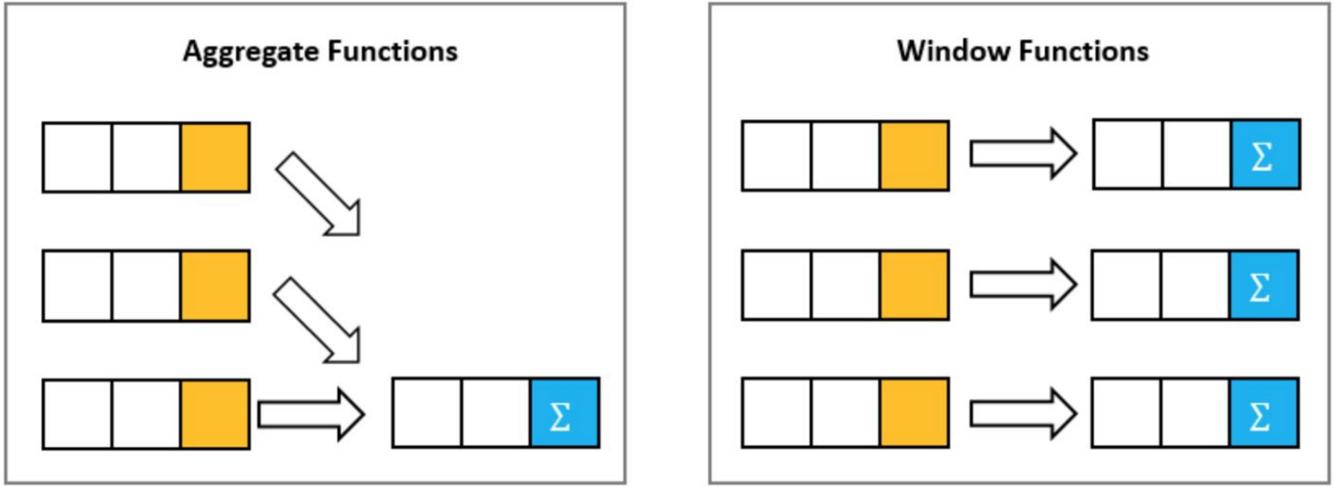
# Answer

```
SELECT
   m.name AS "Full Name",
   count(*) AS "# of books borrowed"
FROM borrowings br
JOIN books b ON br.book_id = b.id
JOIN members m ON br.member_id = m.id
WHERE b.language = ’EN’
GROUP BY m.name;
```

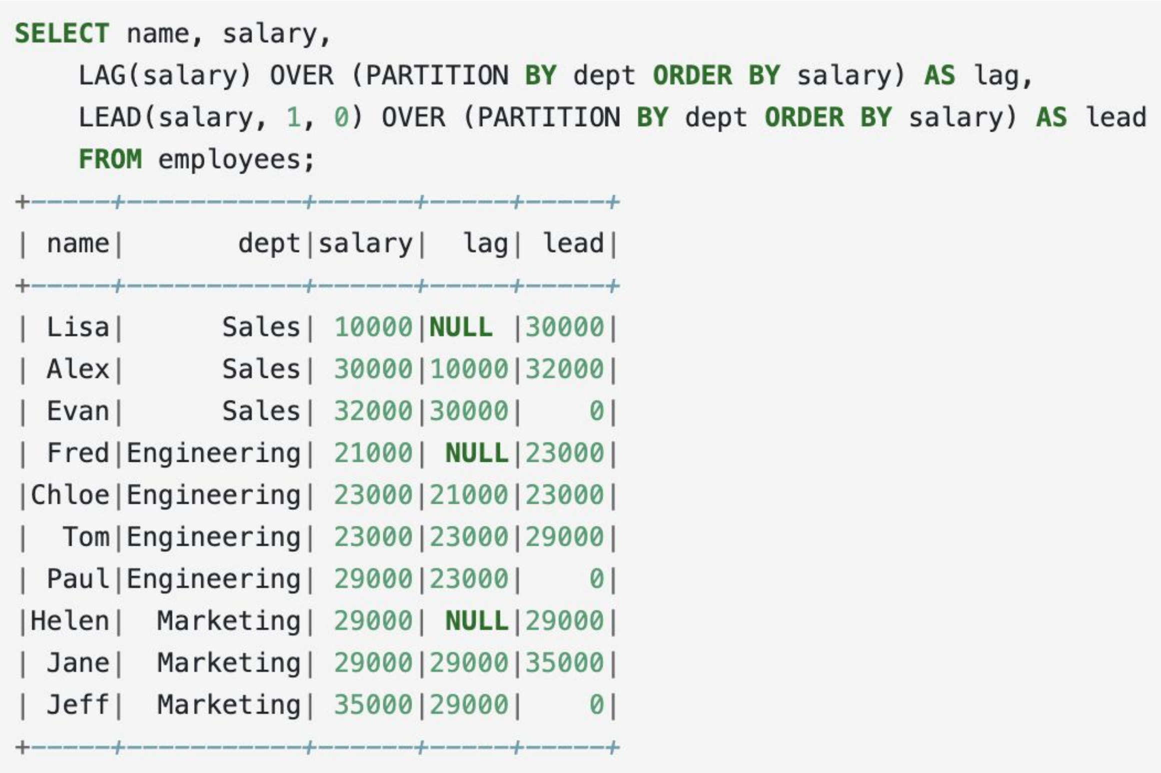
# Spark SQL Window Function

Operate on a group of rows, referred to as a window, and calculate a return value for each row based on the group of rows

Window functions are useful for processing tasks such as calculating a moving average, computing a cumulative statistic, or accessing the value of rows given the relative position of the current row.



# Spark SQL Window Function



```
SELECT name, salary,
    LAG(salary) OVER (PARTITION BY dept ORDER BY salary) AS lag,
    LEAD(salary, 1, 0) OVER (PARTITION BY dept ORDER BY salary) AS lead
    FROM employees;
+-----+-----------+------+-----+-----+
| name|       dept|salary|  lag| lead|
+-----+-----------+------+-----+-----+
| Lisa|      Sales| 10000|NULL |30000|
| Alex|      Sales| 30000|10000|32000|
| Evan|      Sales| 32000|30000|    0|
| Fred|Engineering| 21000| NULL|23000|
|Chloe|Engineering| 23000|21000|23000|
|  Tom|Engineering| 23000|23000|29000|
| Paul|Engineering| 29000|23000|    0|
|Helen|  Marketing| 29000| NULL|29000|
| Jane|  Marketing| 29000|29000|35000|
| Jeff|  Marketing| 35000|29000|    0|
+-----+-----------+------+-----+-----+
```

# Spark SQL Window Function

Examples of functions, that can be used within window frame:

Ranking Functions

Syntax: RANK | DENSE_RANK | PERCENT_RANK | NTILE | ROW_NUMBER

Analytic Functions

Syntax: CUME_DIST | LAG | LEAD

Aggregate Functions

Syntax: MAX | MIN | COUNT | SUM | AVG | ...

# Window Function question

How to get top N (by salary) employees from each department?

table: Employee(name: string, salary: int, department: string)

# Window Function question

How to get top N (by salary) employees from each department?
table: Employee(name: string, salary: int, department: string)

```
SELECT * FROM (

    SELECT e.*,

    ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) rn

    FROM Employee e

)

WHERE rn <= N
```