MU5IN852

BDLE 2024

# Parallel data processing on Spark

Hubert Naacke
december 2024

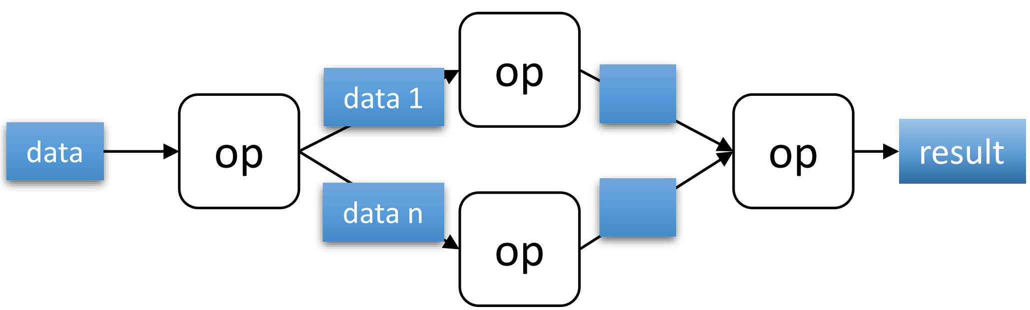
# Data processing

- Query processing over files
  - Select
  - Group by
  - Join
  - Order by (ordered data)

# Query processing on a parallel computing platform

- data can be distributed
- operations can execute in parallel

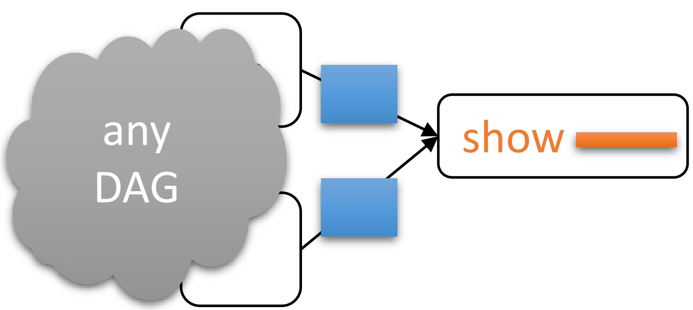
Notations :

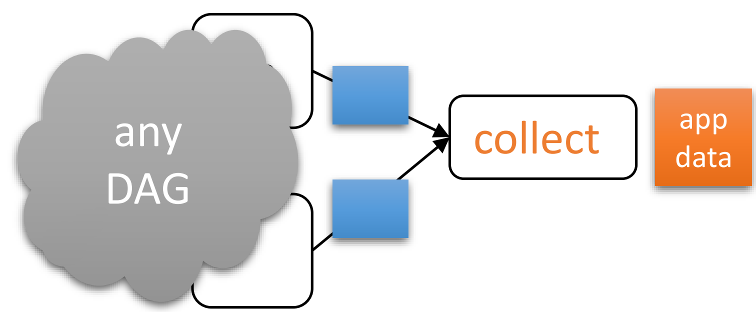



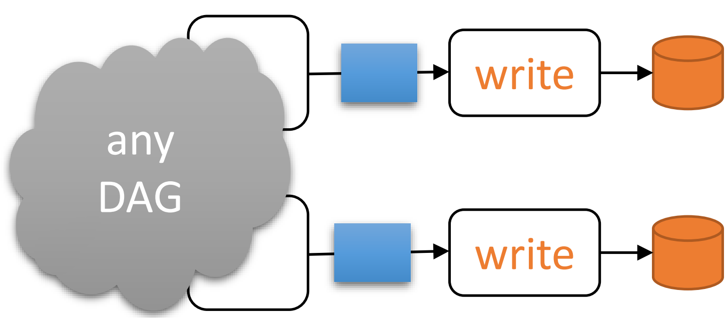

**DAG** (directed acyclic graph) of operations

# DAG execution

- Execute a DAG… what for ?
- DAG needs an « ending operation » to execute
  - serves an application that asks for results
    - show() displays a piece of result
    - collect() brings result into an application
  - saves result to a file
    - write(file)

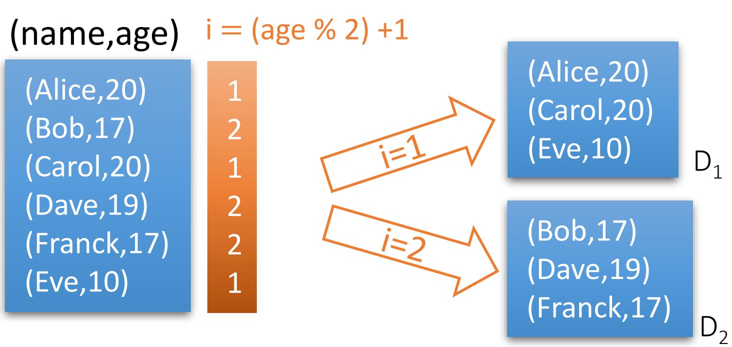


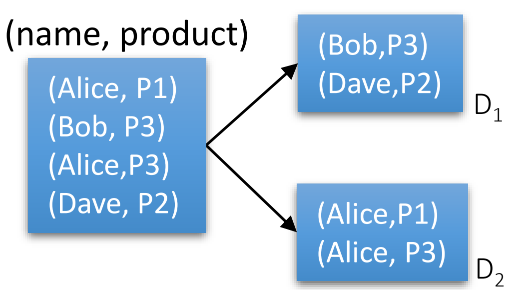



# Data partitionning

- Distribute a dataframe D into several partitions
  - Assigns a partition to each data element based on its attributes
  - Applies a partitionning function f over each element of D
  - $D_i = \{e \in D \mid f(e) = i\}$
- D.**repartition**( n, L)
  - n: number of partitions
  - L: list of attributes. All attributes by default
  - The partitionning function can be specified. Hash function by default



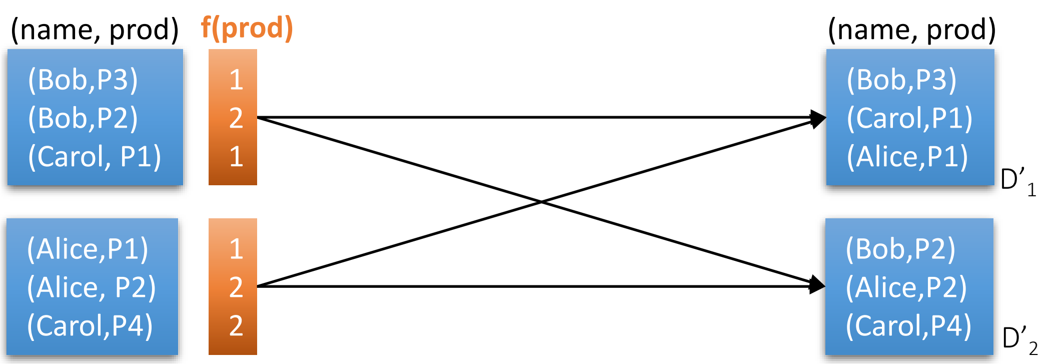

D.repartition(2, age)




D.repartition(2, name)

# Re-partitionning partitionned data

- Same syntax as initial partitionning
- D.**repartition**( n, L)
  - n: number of partitions
  - L: list of attributes
- Apply **f** for each tuple of Di to determine the target partition
  - Can be applied in parallel
- Transfer tuples to build the new partitions $D'_i$ .
  - This step is denoted *shuffle read*

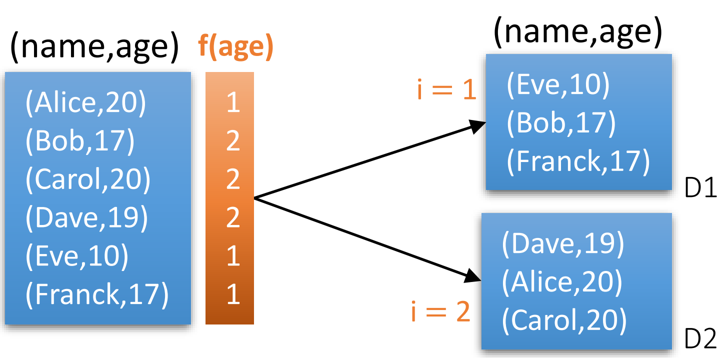



D is partitioned by name

D' = D.repartition(2, product)

# Partition by range

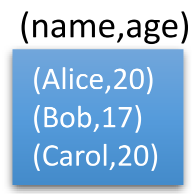
- D.**repartitionByRange**(n, a)
  - n: number of partitions
  - a: attribute
  - Dom(a) = $]a_0, a_1] \cup ]a_1, a_2] \cup \ldots \cup ]a_{n-1}, a_n]$
- Partitioning function
  - $f(a) = i$ iff $a_{i-1} < a \leq a_i$
- Exemple




**f(age)** = 1 iff $10 \leq$ age < 18
= 2 iff $18 \leq$ age < 25

D.repartitionByRange(2, age)

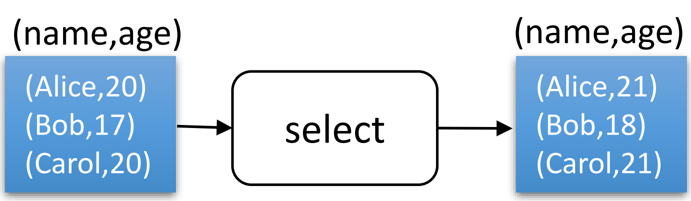
# SELECT over a single file




- Read a (semi) structured **data** file
  - May provide the file schema definition
    - D(id, name) expressed as schema = "string name, int age "
  - D = spark.read( data.csv , schema)

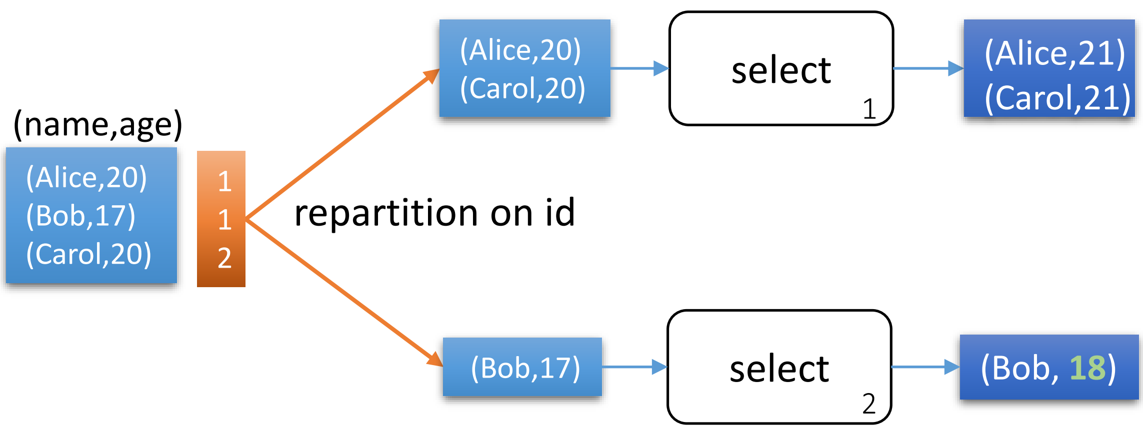
- Apply any function F for each element of D
  - F can be user-defined,
    - F may wrap a «black-box» function…

  - D.select(name, age+1)



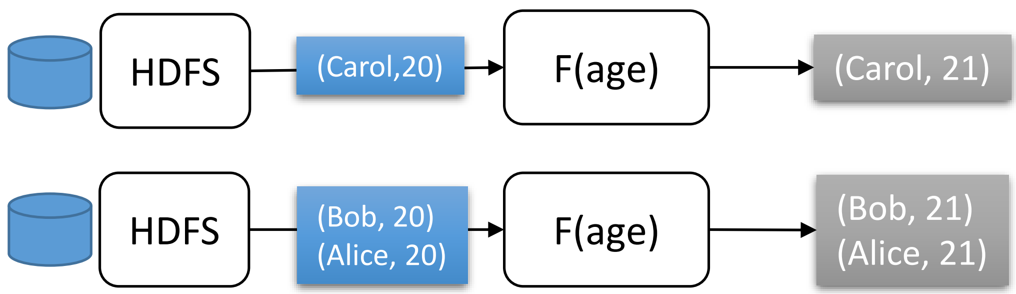
# Parallel SELECT over a single file

- Processing model : one core per partition
  - Degree of parallelism = min( #cores, #partitions)
- Example
  - D.**repartition**(2, id).select(name, age+1)



# Parallel Select over a distributed file (1/2)

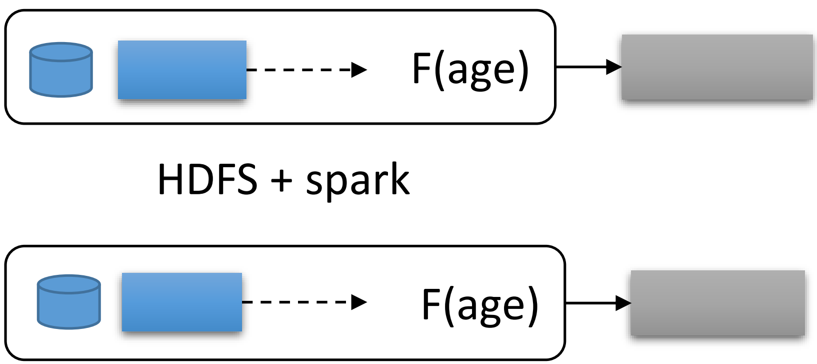
- Read a distributed file
  - D= spark.read( hdfs://data.csv )
- Apply any function F over each element of D
  - D.select(name, F(age) )




F(age) = age+1

# Parallel Select over a distributed file (2/2)

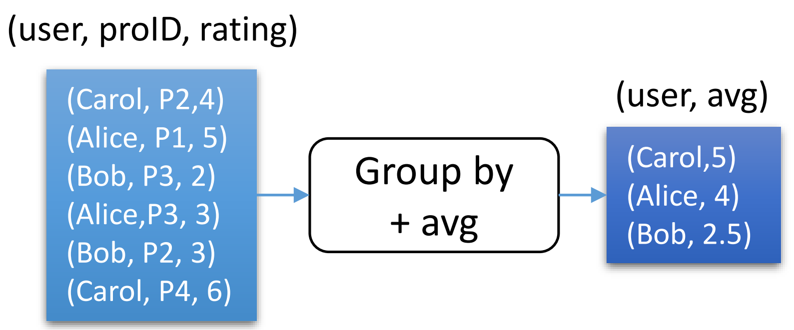
- Distributed storage *coupled* with compute nodes
  - Data locally stored




Applies to any distributed input : file or cached result of a pipeline

# Centralised GROUP BY key

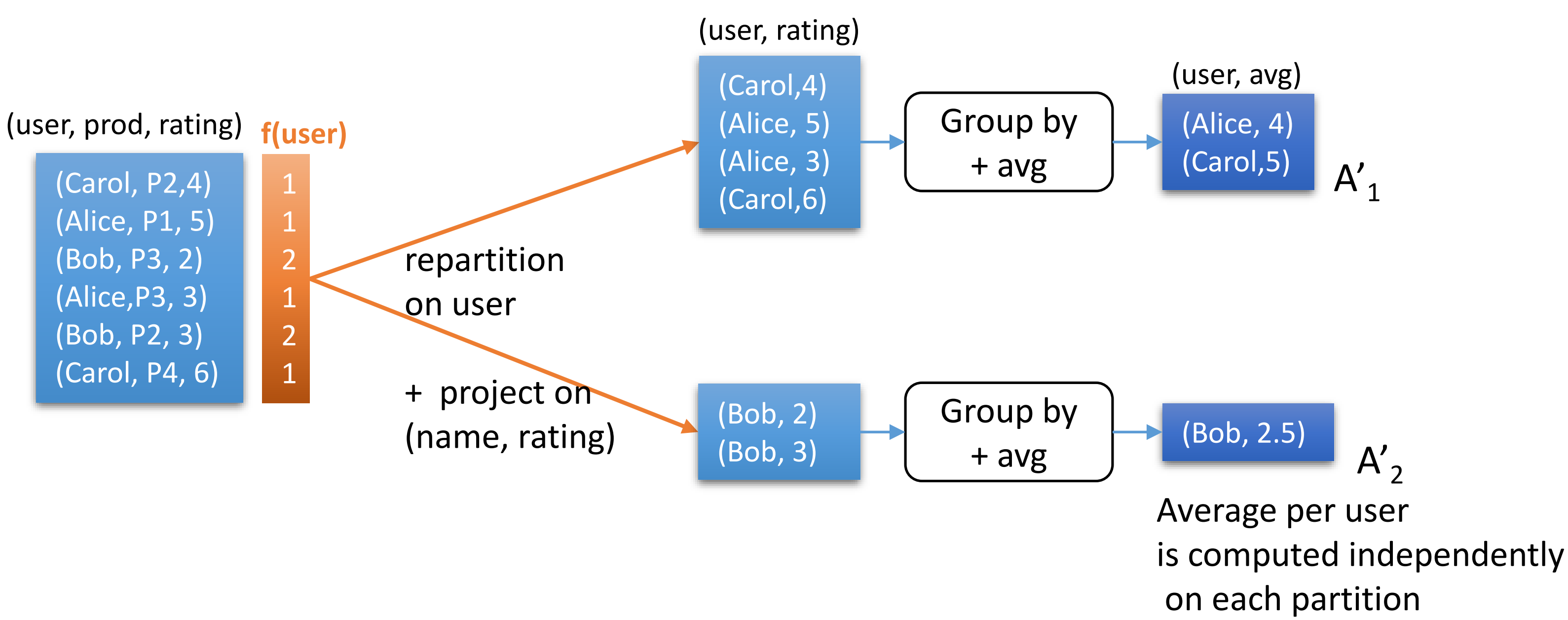
- Data: ratings(user, prodID, rating)
  - ratings = spark.read( ratings.csv )
    - ratings is small enough to fit in one partition
- Compute average rating per user
  - A = ratings.**groupBy**(user).**avg**(rating)




Centralised computation
of group by + avg

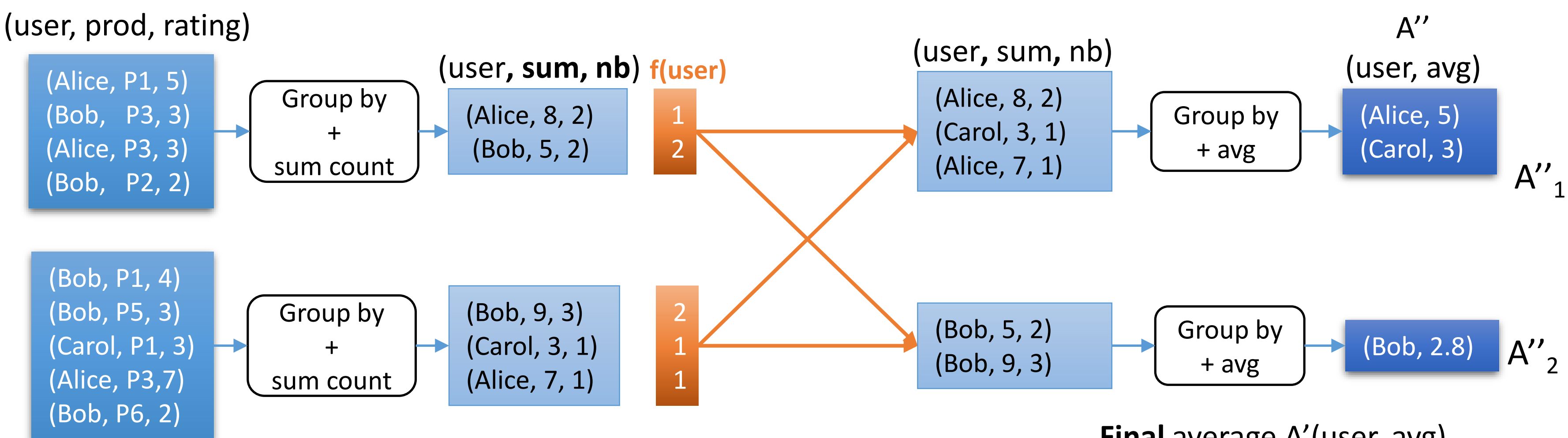
# Parallel GROUP BY key over 1 file

- Data: ratings(user, prodID, rating)
  - ratings = spark.read( ratings.csv )
- Compute average rating per user
  - A' = ratings.repartition(2, user).groupBy(user).avg(rating)



# Group by key over distributed data

- Ratings(user, prod, rating)
  - ratings = spark.read( hdfs://ratings.csv )
    - ratings is large, thus stored in many partitions
- Compute average rating per user
  - A'' = ratings.groupBy(user).avg(rating)

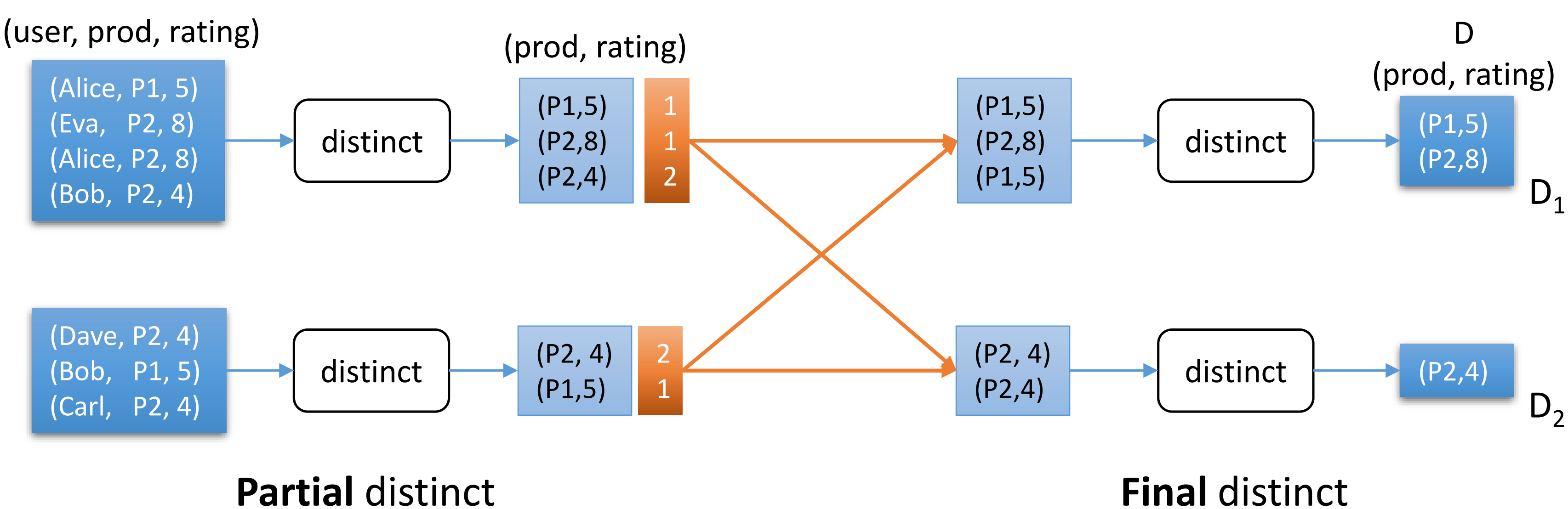



**Partial** average (user, sum, nb) is computed independently on each partition

**Final** average A'(user, avg) is computed independently on each partition

# Evaluate distinct() operator over distributed data

- Rate(user, prod, rating)
  - Rate = spark.read( hdfs://ratings.csv )
- Compute the set of distinct (product,rating) tuples
  - D = R.select(product, rating).**distinct**()



# JOIN operator

- Outline
  - Join algorithms
    - Parallel hash join
    - Nested-loop broadcast join
  - Joining skewed data

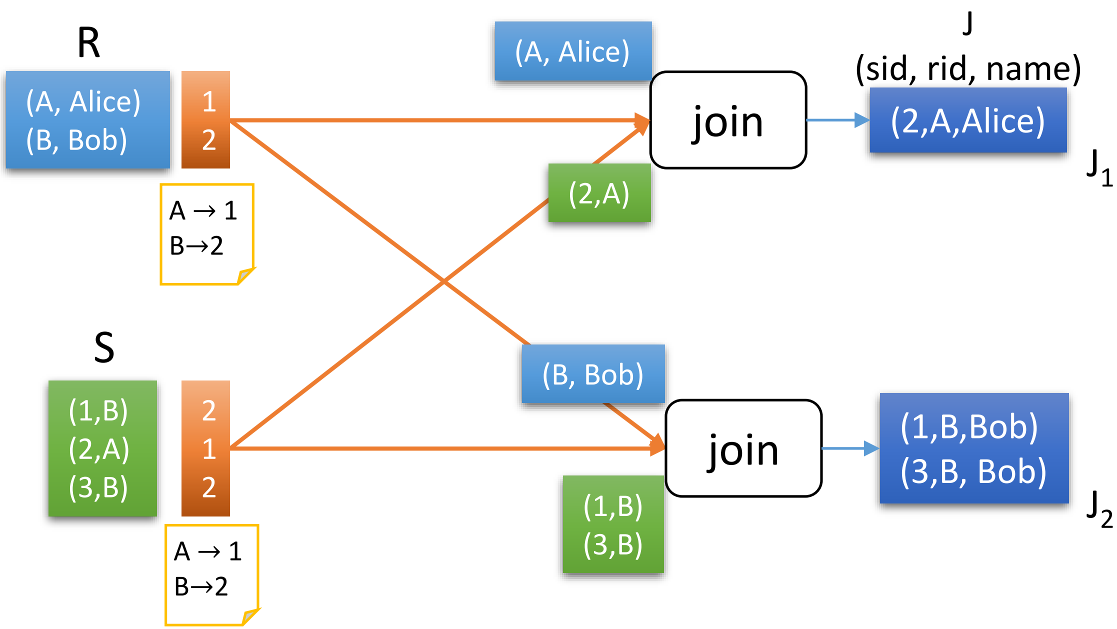
- Running example

Data : **R** (rID, name)
**S** (sID, rID) S.rID references R.rID

Query: Join R with S to get (sID, rID, name) tuples
R.join(S, rID)

# Parallel hash Join

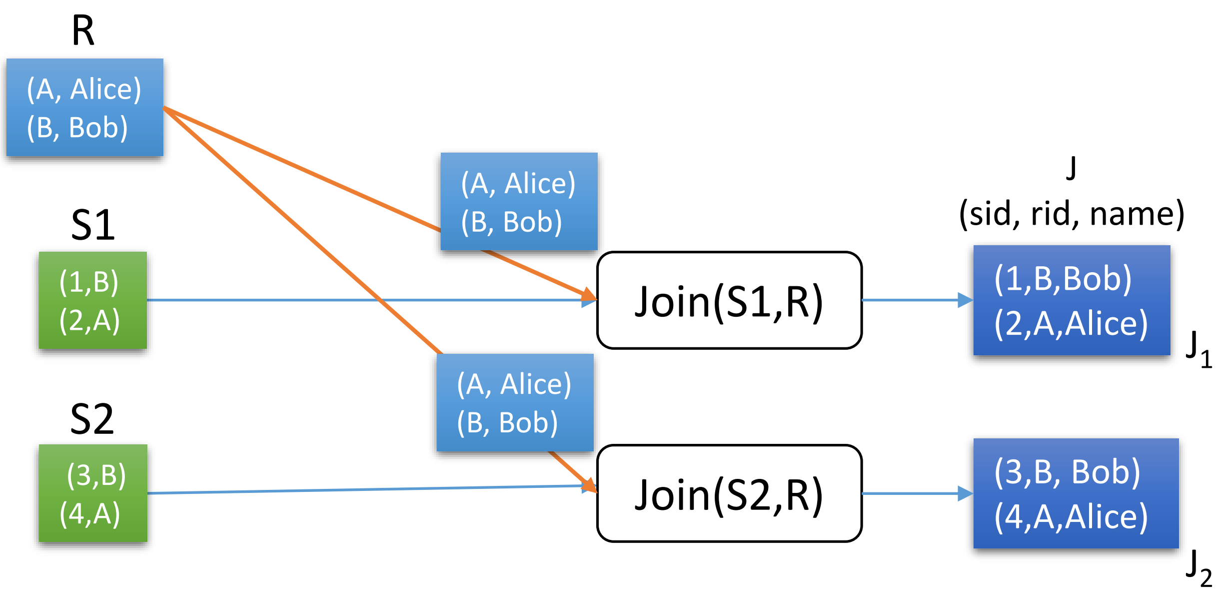
- Join R(rid, name) with S(sid, rid) to get J(sid, rid, name)
  - J = R.join(S, rid)
- Distribute the 2 relations R and S on the join key
  - Using the **same** partitioning function




Also works if R and S are initially distributed

# Nested loop broadcast join

- Initial state
  - S is **large** thus stored on many partitions
  - S is distributed on any key (not on rid in general)
- Join R and S
  - Replicate R data to every compute unit
  - J = S.join(broadcast(R), rid)




Also works if S is skewed on rid

# Skew aware parallel join

- Context
  - Two large relations R, S
  - S is skewed on the join key
    - S has many tuples with the same value of the join key
- Join R with S using nested loop broadcast join ?
  - No because is would cause too much data transfer
- Start a parallel hash join
  - Distribute R and S
- Observe that S is skewed
  - Isolate Si : the part of S that is skewed
- Distribute Si (**not** on the join key!)
- Join Si with Ri using a nested loop broadcast join

# Process other operators in parallel?

- Subtract
- Order by *attributes*
- Limit *n*
- Select f() over (partition by … )
- any combinations of these operators …

- Optimized execution ? Save data transfer :
  - Do not repartition already «adequately » partitioned data
  - Do not broadcast the same relation multiple times
  - …

# Extensible user defined operations

- mapPartition(f)
  - function f applied on each partition
- repartition
  - repartition function
- agregation

# Conclusion

- Builtin parallel (and distributed) processing
  - For every SQL operator
    - select, project, join, group by, order by, over, …
- Ease of use
  - Logical data model
    - Hides data distribution and parallel processing
  - SQL compliant: process any query (+ UDF) in parallel
- Query optimisation
  - 2 parallel join algorithms implemented
  - Skew aware
  - Dynamic query planning
- Control and monitoring
  - Tuning : degree of parallelism, …
  - Execution report: shuffle size, task durations