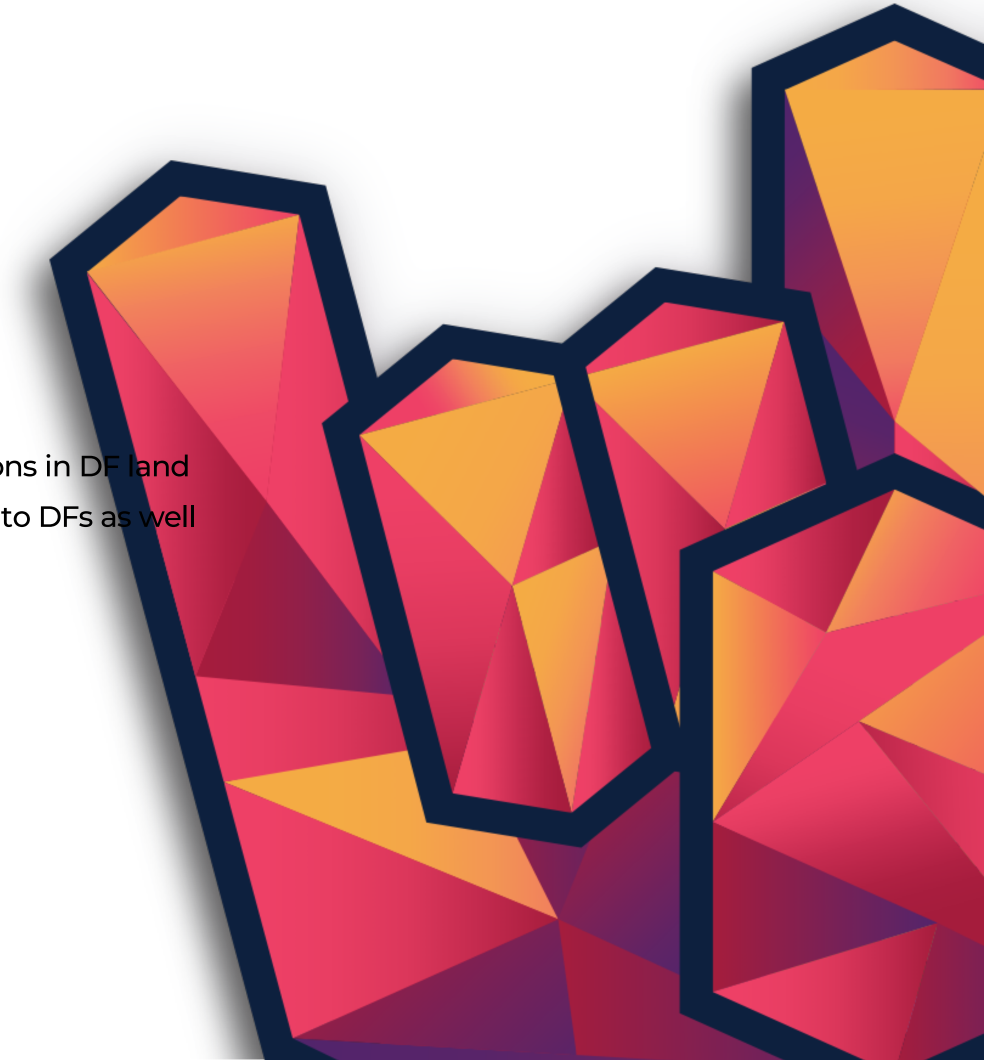
Optimizing RDD Transformations

Objective

Dedicated chapter on optimizing RDDs

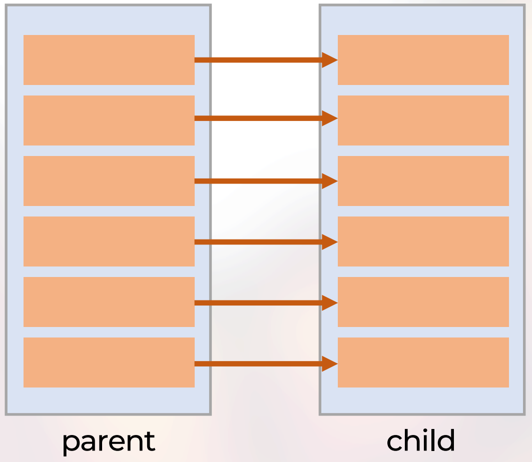
- you don't get the same control for the same operations in DF land
- I'll add comments in this chapter that are applicable to DFs as well



Dependencies

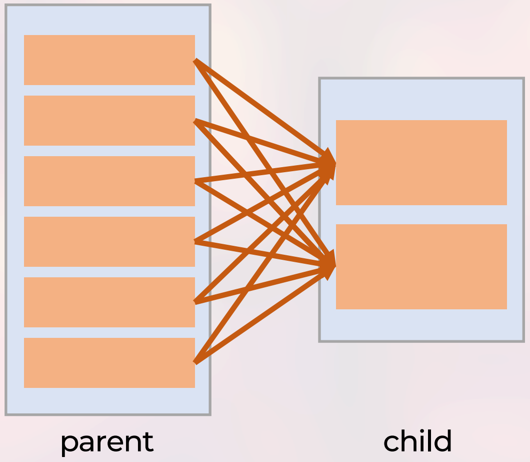
Narrow dependencies

- one input (parent) partition influences a single output (child) partition
- fast to compute
- examples: map, flatMap, filter, mapValues



Wide dependencies

- one input partition influences more than one output partitions
- involve a shuffle = data transfer between Spark executors
- are costly to compute
- examples: grouping, joining, sorting

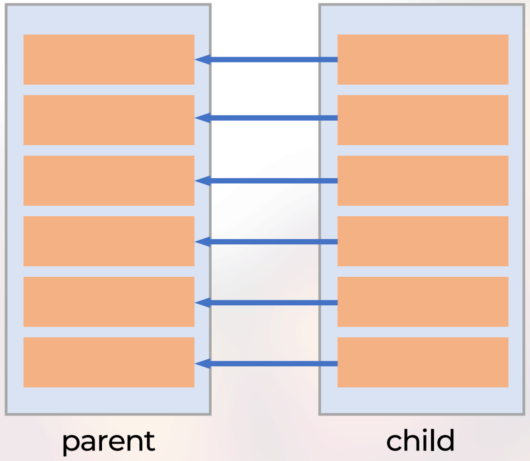


Dependencies

Expressed differently in terms of "depends on":

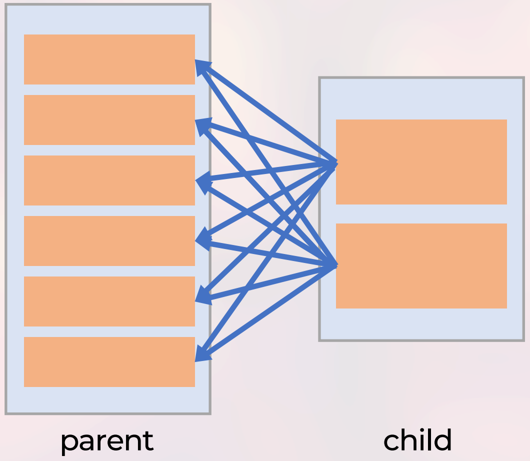
Narrow dependencies

- given a parent partition, a single child partition depends on it
- fast to compute
- examples: map, flatMap, filter, projections



Wide dependencies

- given a parent partition, more than one child partitions depend on it
- involve a shuffle = data transfer between Spark executors
- are costly to compute
- examples: grouping, joining, sorting



Perf Consequences

Narrow dependencies are fast

- no data transfer between executors
- can be executed in a single pass over data
- fault-tolerance: if an executor fails, recomputing a partition needs a single parent partition

Wide dependencies are bad for multiple reasons

- need data transfer between executors – slow!
- may need disk IO for shuffle files
- impose stage boundaries
- limit parallelism
- fault-tolerance: if an executor fails, recomputing a partition takes forever

RDD Implementations

RDD variations are different in two ways

- in the type of elements contained
- in the actual implementation of the RDD interface

RDDs have different APIs

- some operations are only available for RDDs of tuples

RDD transformations can be computed differently

- certain RDD implementations may hold additional information e.g. locality/ordering
- example: MappedRDD vs CoGroupedRDD

Spark rocks