PROCESSING BIG DATA

Introduction to Processing Big Data

- **Moving Computation to Data**
- Parallel Operations
- Distributed Aggregations
- Accumulating and Broadcasting
- Working with Records of Pairs
- Multi-Dataset Operations
- From Concepts to Frameworks

Moving Computation to Data

- Some typical properties of Big Data:
  - Data is split into multiple partitions residing in distant disks
  - The dataset is a huge collection of records
  - No orderings should be assumed a-priori
- Moving the data around is intolerably expensive

Input Split:
R1, 1
R1, 2
...
R1, n_1

Input Split:
R2, 1
R2, 2
...
R2, n_2

Input Split:
R3, 1
R3, 2
...
R3, n_3

Input Split:
R4, 1
R4, 2
...
R4, n_4

Distributed Collection

The entire collection is split into partions, possibly residing on multiple machines

Moving Computation to Data

- The key is moving computation to data, that is, performing local computations in parallel and reducing the amount of communication when data needs to be transferred across nodes
- The MapReduce programming model makes complex computations on distributed collections possible in a parallel and distributed fashion with a minimal amount of communication
  - Communication effectiveness (reducing the amount of data moving between nodes) will be key to implementing such operations on Big Data

analyticscenter

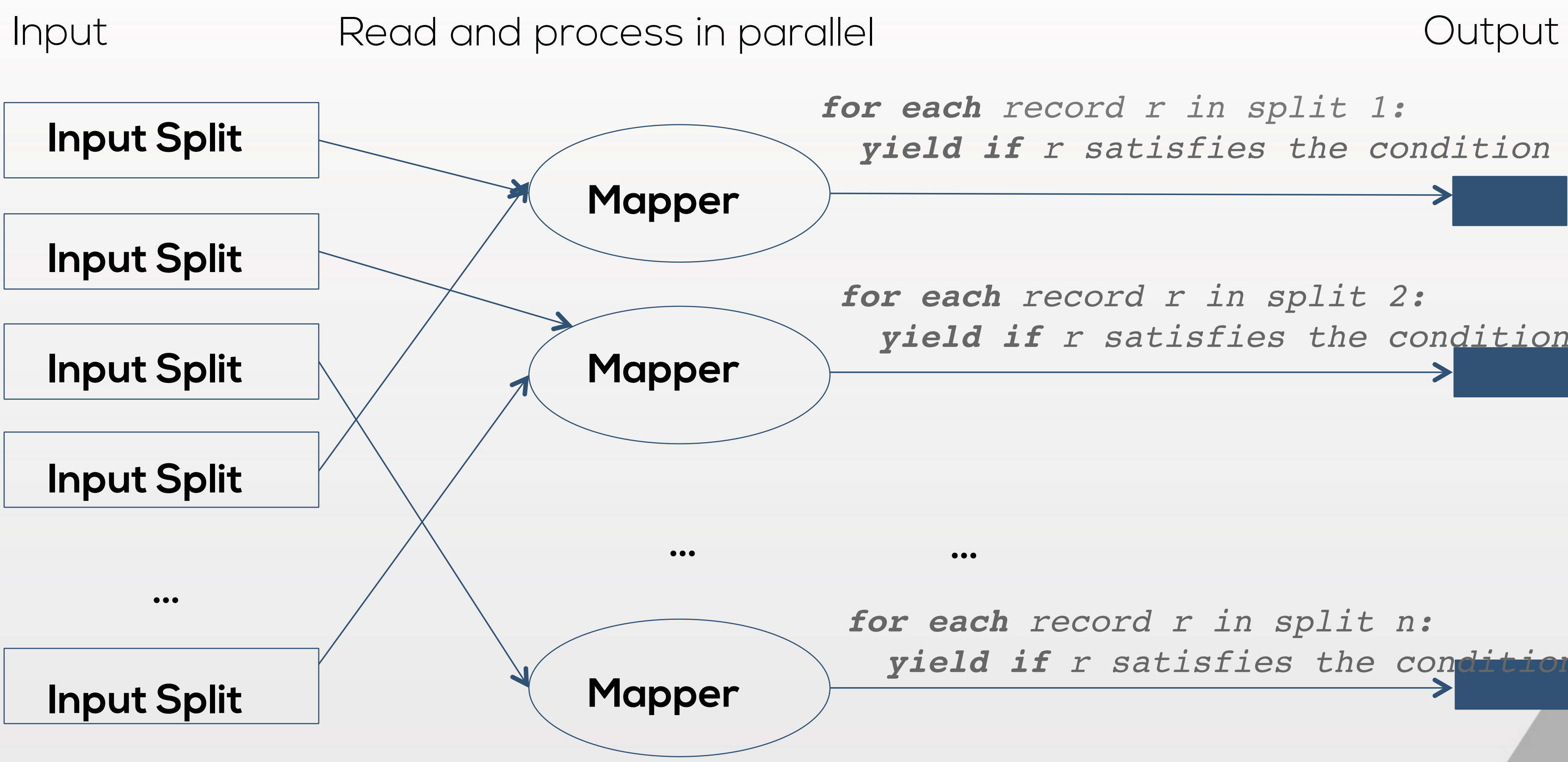
Introduction to Processing Big Data

- Moving Computation to Data
- **Parallel Operations**
- Distributed Aggregations
- Accumulating and Broadcasting
- Working with Pairs of Records
- Multi-Dataset Operations
- From Concepts to Frameworks

Parallel Operations

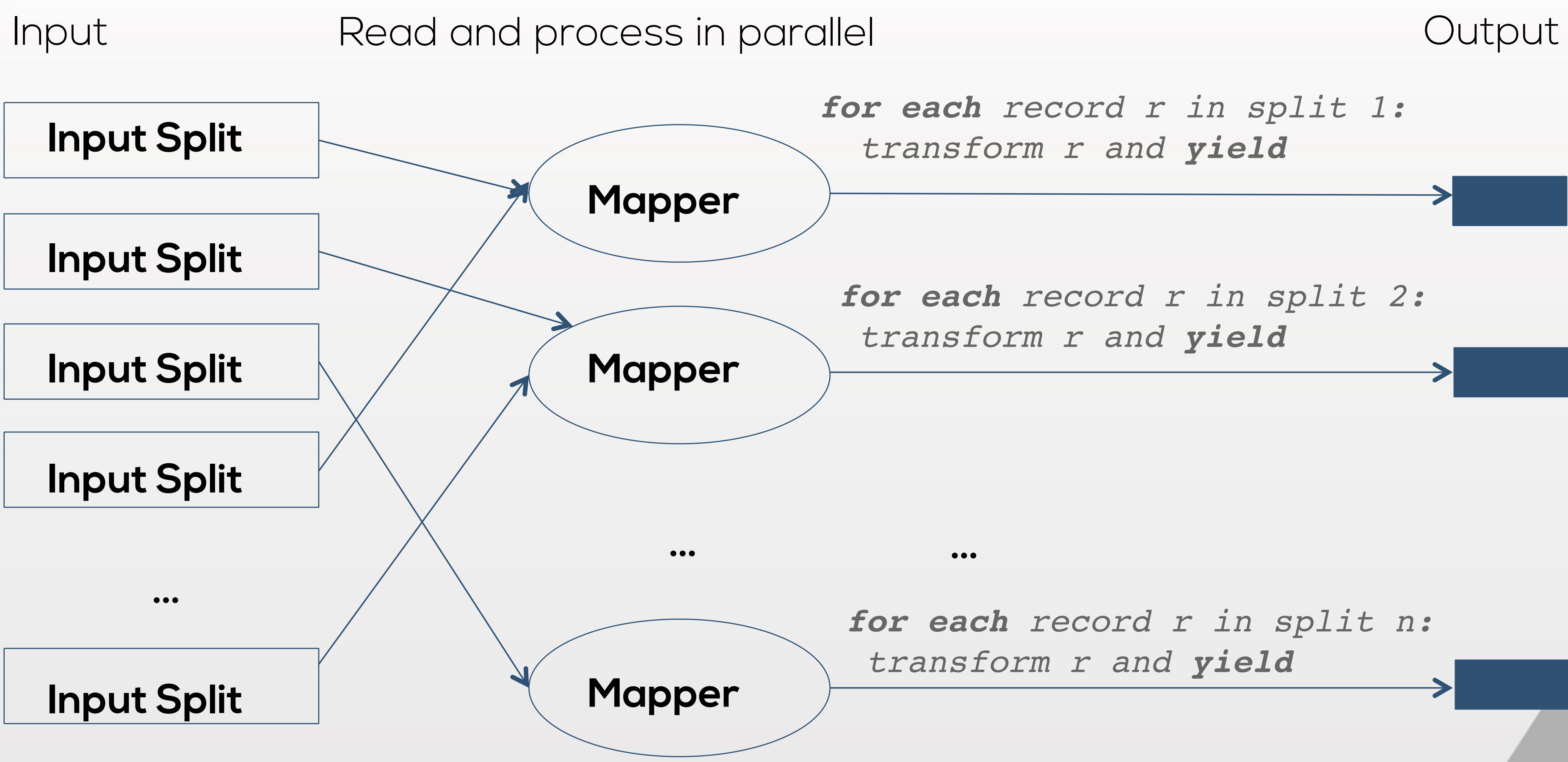
- On a distributed collection of data, operations such as:
  - **Selection**
  - **Projection** can be implemented in parallel

```
//Pseudo-code for selection (filter out records that
//does not satisfy a boolean condition, i.e. where
//clause in SQL)

for each split i in parallel:
  for each record j in split i:
    if the record satisfies the condition
      yield the record
    end-if
  end-for
```



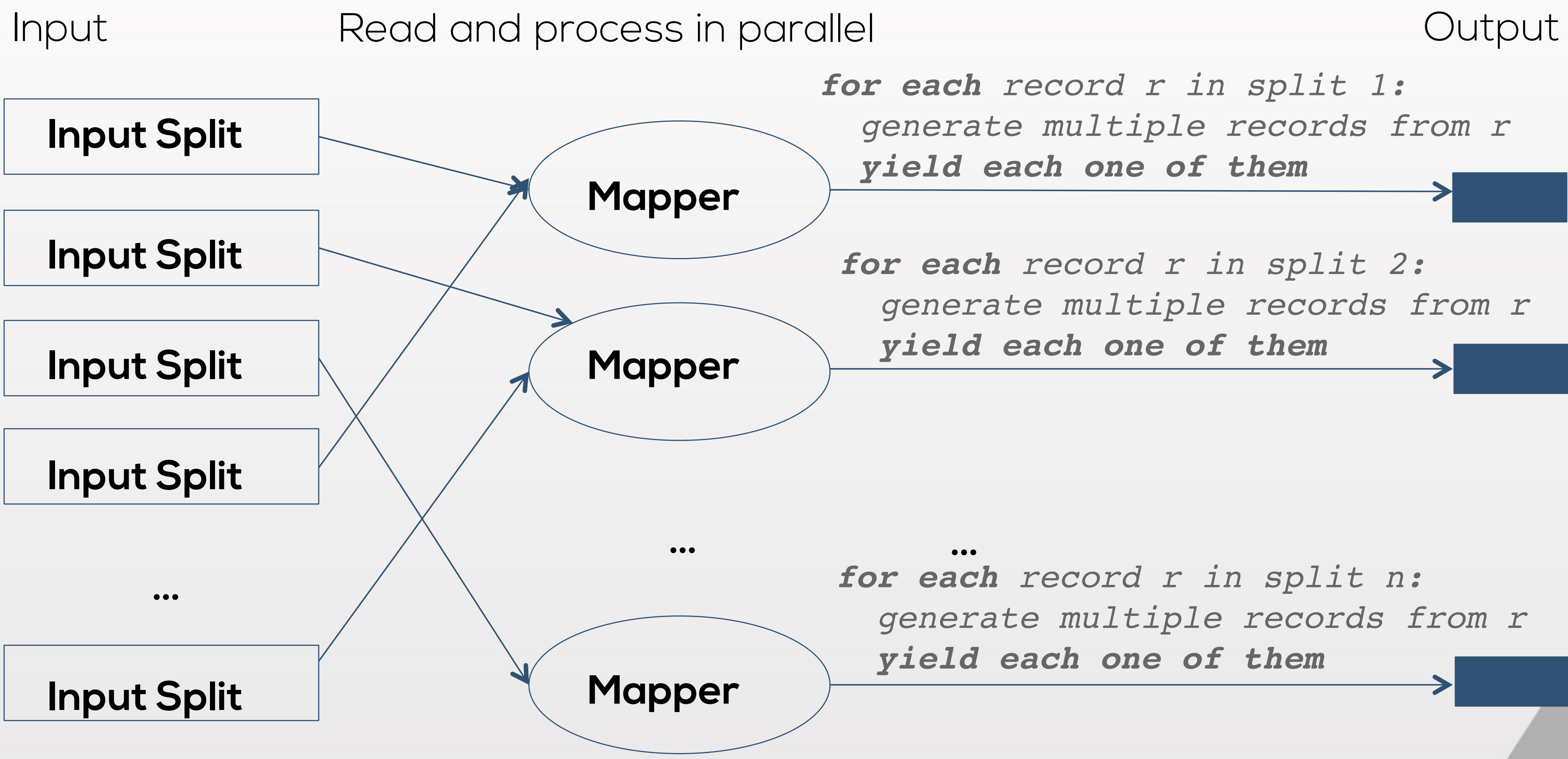
Select in parallel (WHERE predicate):

filter as a higher order function



Project in parallel (SELECT clause):

map as a higher order function



FlatMap in parallel

(in the functional programming sense)

Parallel Operations

- Applying parallel operations is no hassle
  - Provided a parallel computation framework that runs higher order functions on records of a distributed collection

Introduction to Processing Big Data

- Moving Computation to Data
- Parallel Operations
- **Distributed Aggregations**
- Accumulating and Broadcasting
- Working with Pairs of Records
- Multi-Dataset Operations
- From Concepts to Frameworks

Distributed Aggregations

- Often, we need to compute aggregates over the collection
- We consider two strategies for aggregating distributed collections:
  - The aggregate operation is distributive
  - The aggregate operation is holistic

Distributed Aggregations

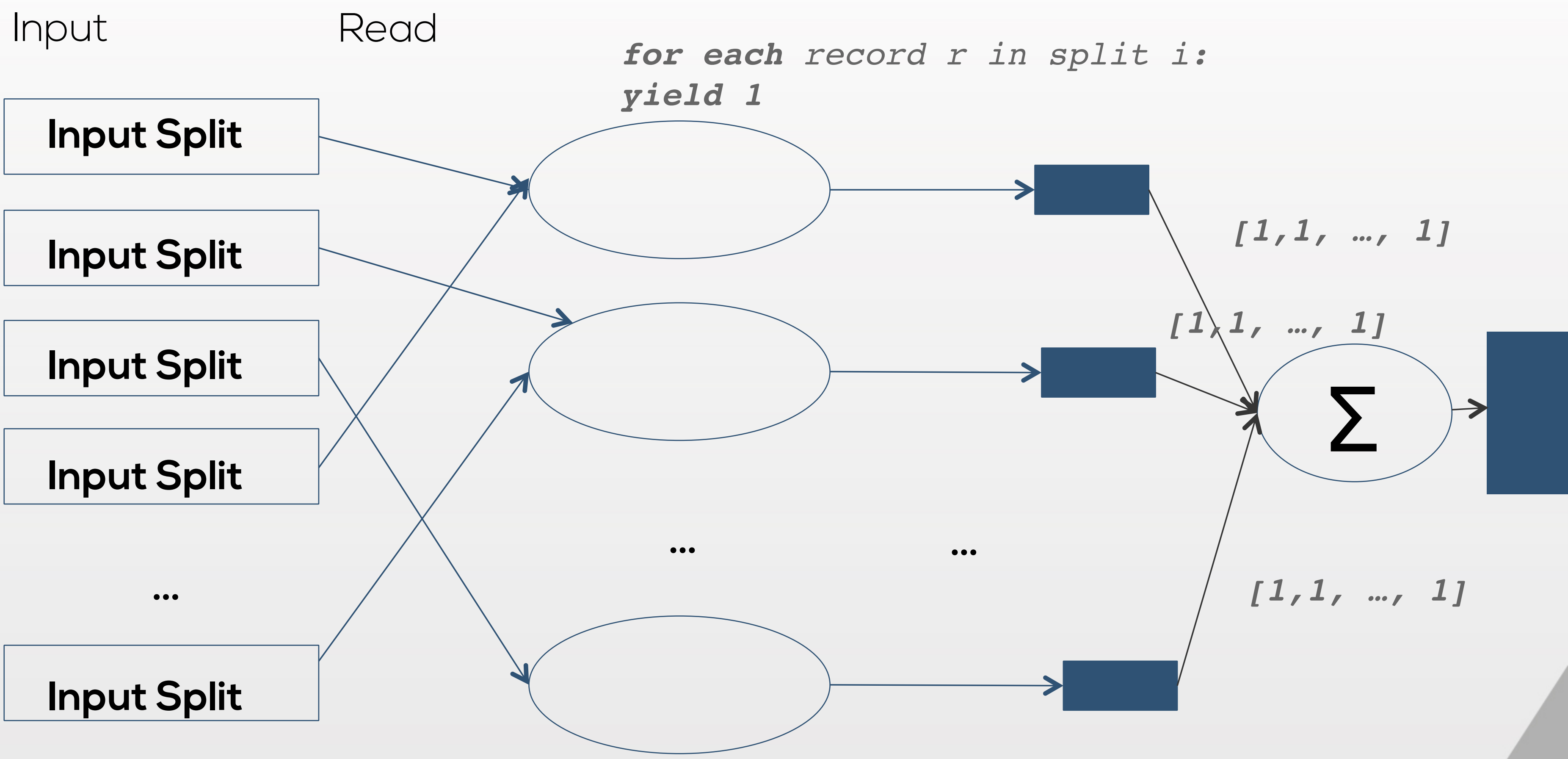
- If the aggregation is holistic (i.e. all data must be seen by the aggregate operation), the amount of communication is large
  - All record-level relevant data should be copied to the same processing unit that computes the aggregate
  - e.g. **median** is such a measure

Distributed Aggregations

```
//Local count function in parallel

for each split i in parallel:
  for each record j in split i:
    yield 1
  end-for
```

Distributed Aggregations

```
//Global aggregate for count

input: [1, 1, 1, …, 1]
yield sum(input)
```



Distributed Count

Distributed Aggregations

- If the aggregation is distributive, we can perform aggregates with minimum amount of communication
- The trick is to
  - compute local aggregates
  - transfer all local aggregates to the same node
  - compute the global aggregate

Distributed Aggregations

- We refer to
  - associative (same aggregate of local aggregates)
  - distributive (different aggregate of local aggregates)
  - algebraic (a function of multiple distributive aggregates)

operations as trivially distributive aggregations

Distributed Aggregations

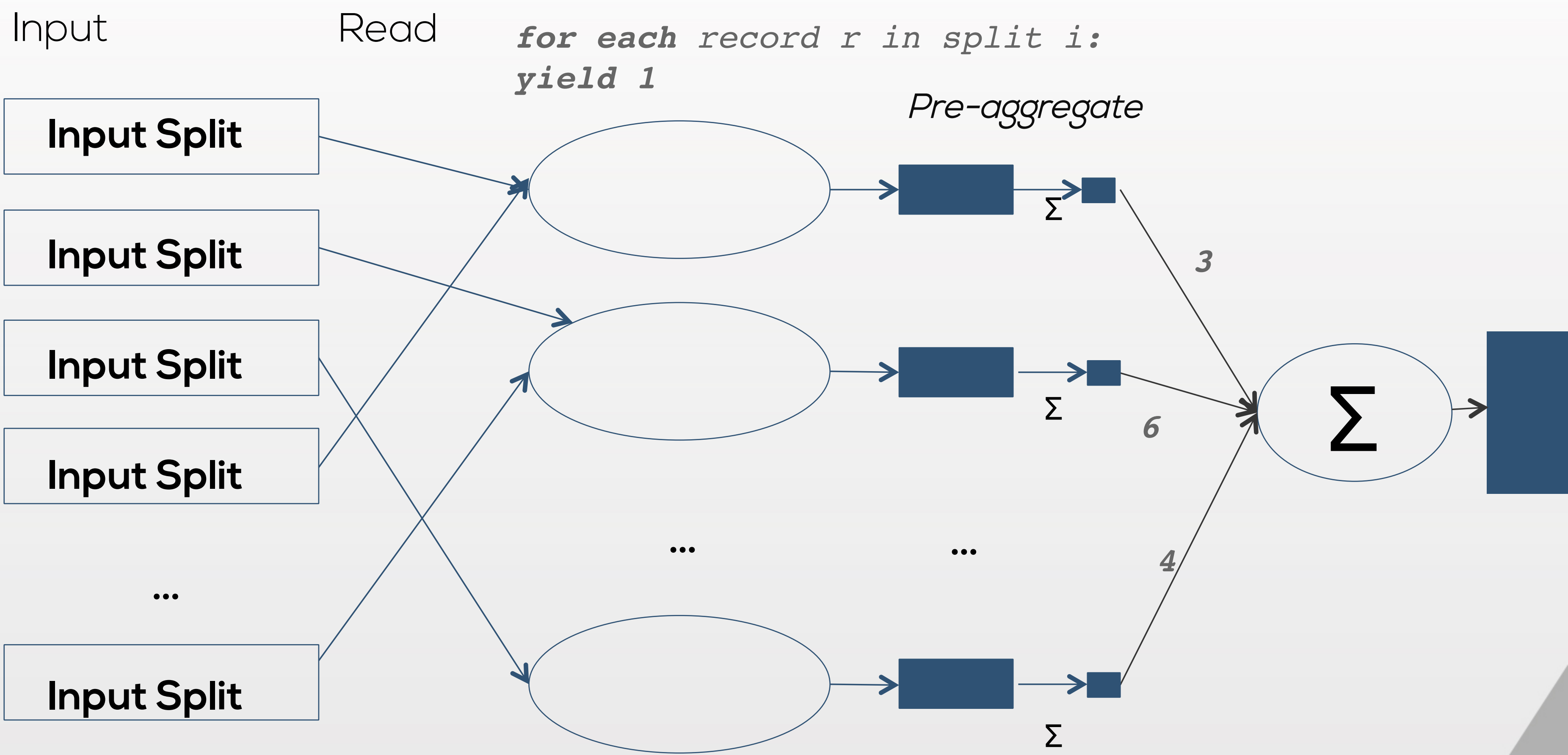
- Many aggregate operations can be distributed (communication) efficiently, such as:
  - **count** (sum of counts)
  - **max** (max of maxes)
  - **sum** (sum of sums)
  - **average** (sum/count)

Distributed Aggregations

```
//Local count function in parallel

for each split i in parallel:
  for each record j in split i:
    yield 1
  end-for

  take a sum of all 1's
```

Distributed Aggregations

```
//Global aggregate for count

input: [c1, c2, …, cm]
yield sum(input)
```



Distributed Count

Introduction to Processing Big Data

- Moving Computation to Data
- Parallel Operations
- Distributed Aggregations
- **Accumulating and Broadcasting**
- Working with Records of Pairs
- Multi-Dataset Operations
- From Concepts to Frameworks

Accumulating

- Sometimes a global counter, available to all parallel execution nodes, can be defined to be incremented by parallel operations, in an atomic way
- An example use case is counting the number of bad records in the collection

Broadcasting

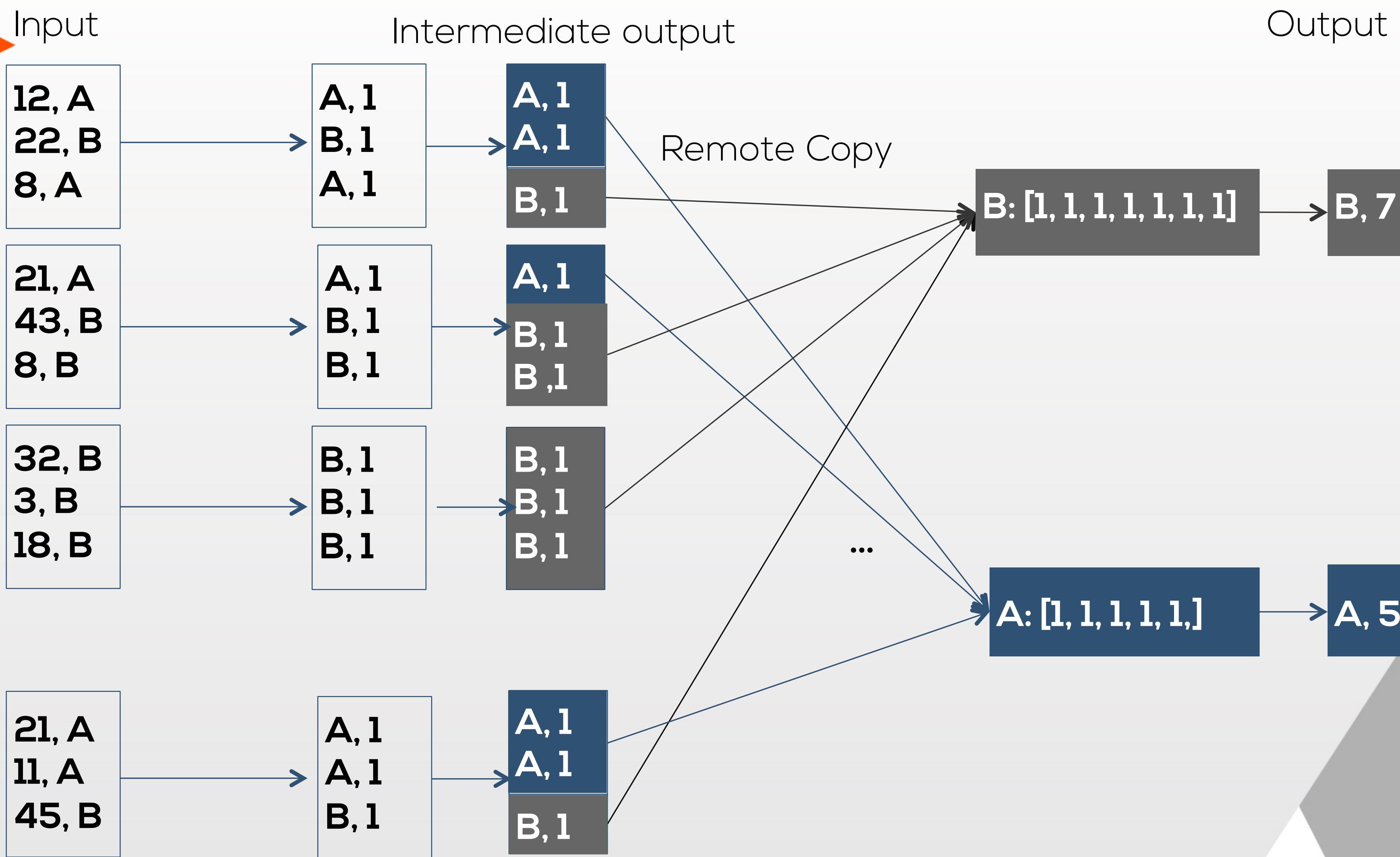
- A small data structure can be broadcast into all parallel execution node
- This might be a small data set or a configuration, for example
- Such variables are naturally read-only

Introduction to Processing Big Data

- Moving Computation to Data
- Parallel Operations
- Distributed Aggregations
- Accumulating and Broadcasting
- **Working with Records of Pairs**
- Multi-Dataset Operations
- From Concepts to Frameworks

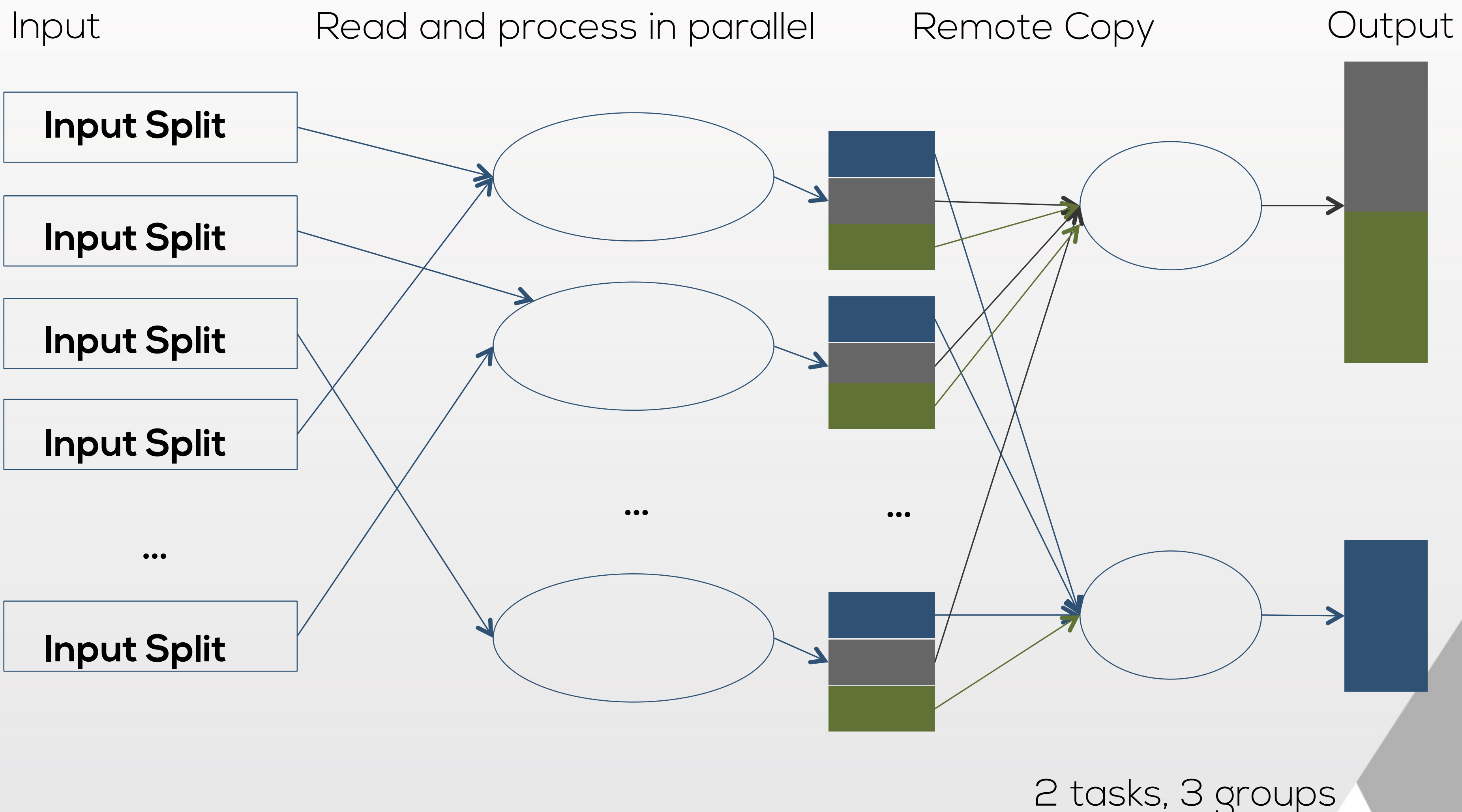
Working with Records of Pairs

- Often, we work with datasets of records of pairs:
  - records are in `<k, v>` form
  - records are mapped into `<k, v>` form
  - the required aggregate is by `K`
- In this case, the aggregate operations can be performed on a per `K` basis
- To have multiple aggregation units running in parallel:
  - the results from the local operations should be somehow grouped by the `K`
  - all local results with the same `K` should be copied to the same aggregation unit



Counting the numbers of A's and B's

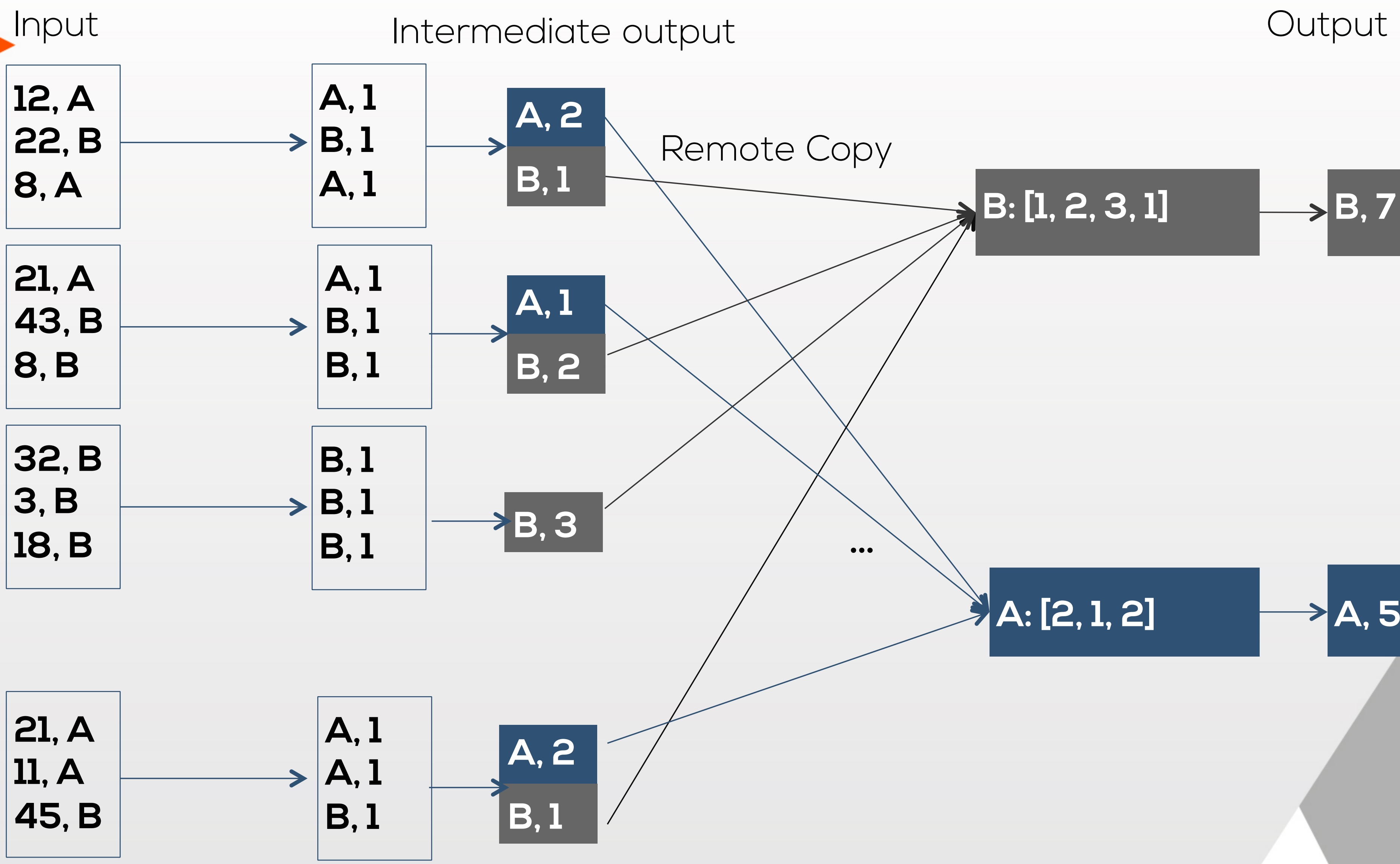
GROUP-BY-COUNT



Working with Records of Pairs (key-values)

Working with Records of Pairs

- Again, we can avoid copying all per-record results to the aggregation units if the aggregation permits to do so
- We could have computed the local sums in the previous example, and reduce the amount of integers copied over network



Counting the numbers of A's and B's
(with pre-aggregation) -- GROUP-BY-COUNT

Working with Records of Pairs

- Notice that when we work with records of pairs, the skew of the dataset by K's greatly affect:
  - the amount of data processed by
  - the amount of data copied into
  - the amount of work done by

an aggregation unit

Introduction to Processing Big Data

- Moving Computation to Data
- Parallel Operations
- Distributed Aggregations
- Working with Records of Pairs
- **Multi-Dataset Operations**
- From Concepts to Frameworks

Multi-Dataset Operations

- Sometimes, we would like to perform operations yielding a single dataset from multiple datasets, such as:
  - Set operations like union and intersection
  - Cross-dataset operations like cartesian product and join
- Without loss of generality, we will assume 2 datasets, split into blocks on the same cluster
  - We may further assume a 'combined split' per node, including records from both splits of the underlying datasets

Multi-Dataset Operations

- Union (duplicate records allowed) can trivially be parallelized, and there is no need of aggregation
  - Union can be implemented without a need of copying intermediate data over network
  - It is a **map-only** job

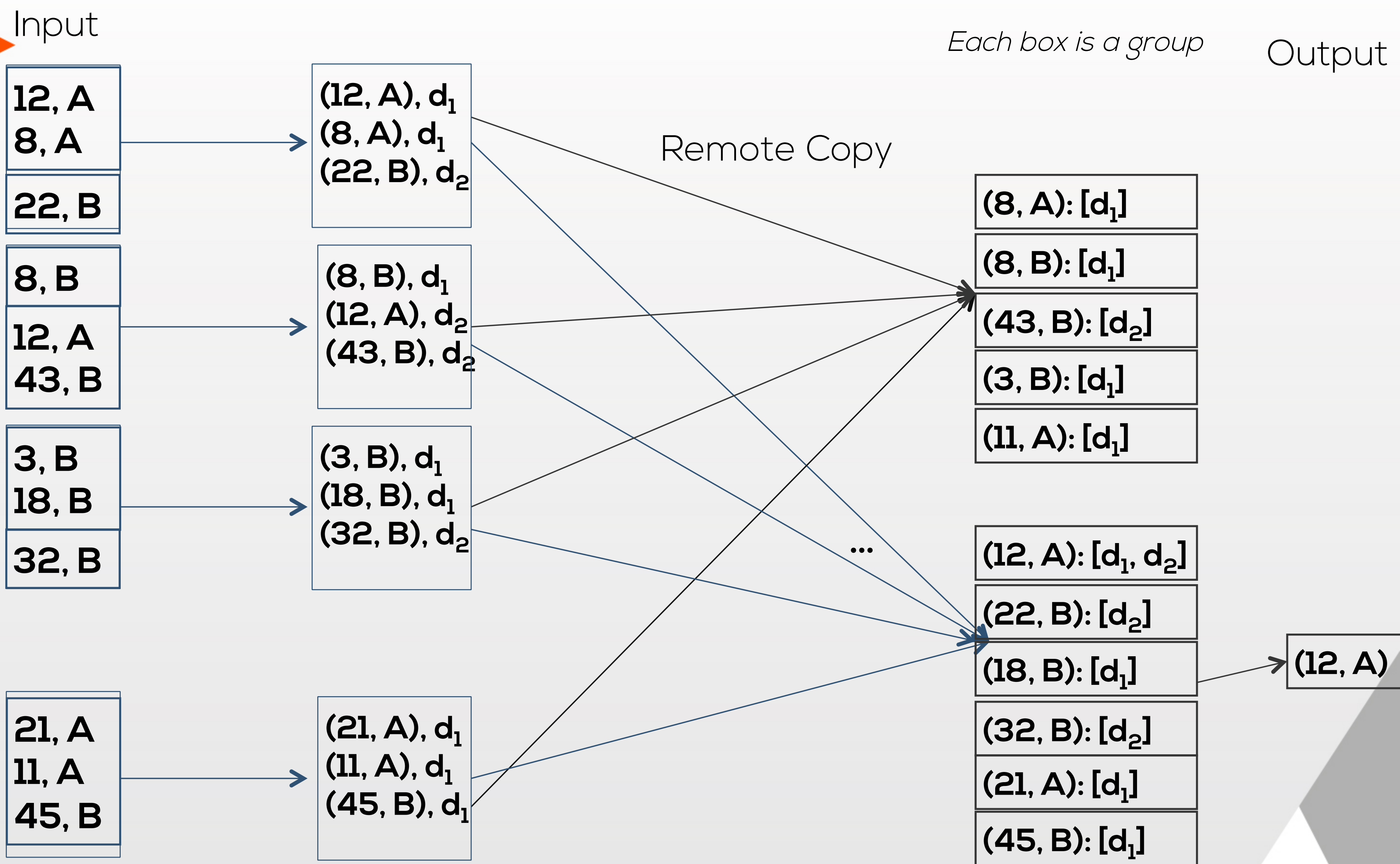
Input	Output
12, A 8, A 22, B	12, A 8, A 22, B
8, B 12, A 43, B	8, B 12, A 43, B
3, B 18, B 32, B	3, B 18, B 32, B
21, A 11, A 45, B	21, A 11, A 45, B

UNION

Multi-Dataset Operations: Union

- Things are not that easy with intersection
  - A record belonging to the resulting dataset can occur in different splits
  - thus we need to copy the intermediate records to the aggregation units, and only keep a record if it is contained in both datasets
    - This is a large amount of network

analyticscenter



INTERSECTION

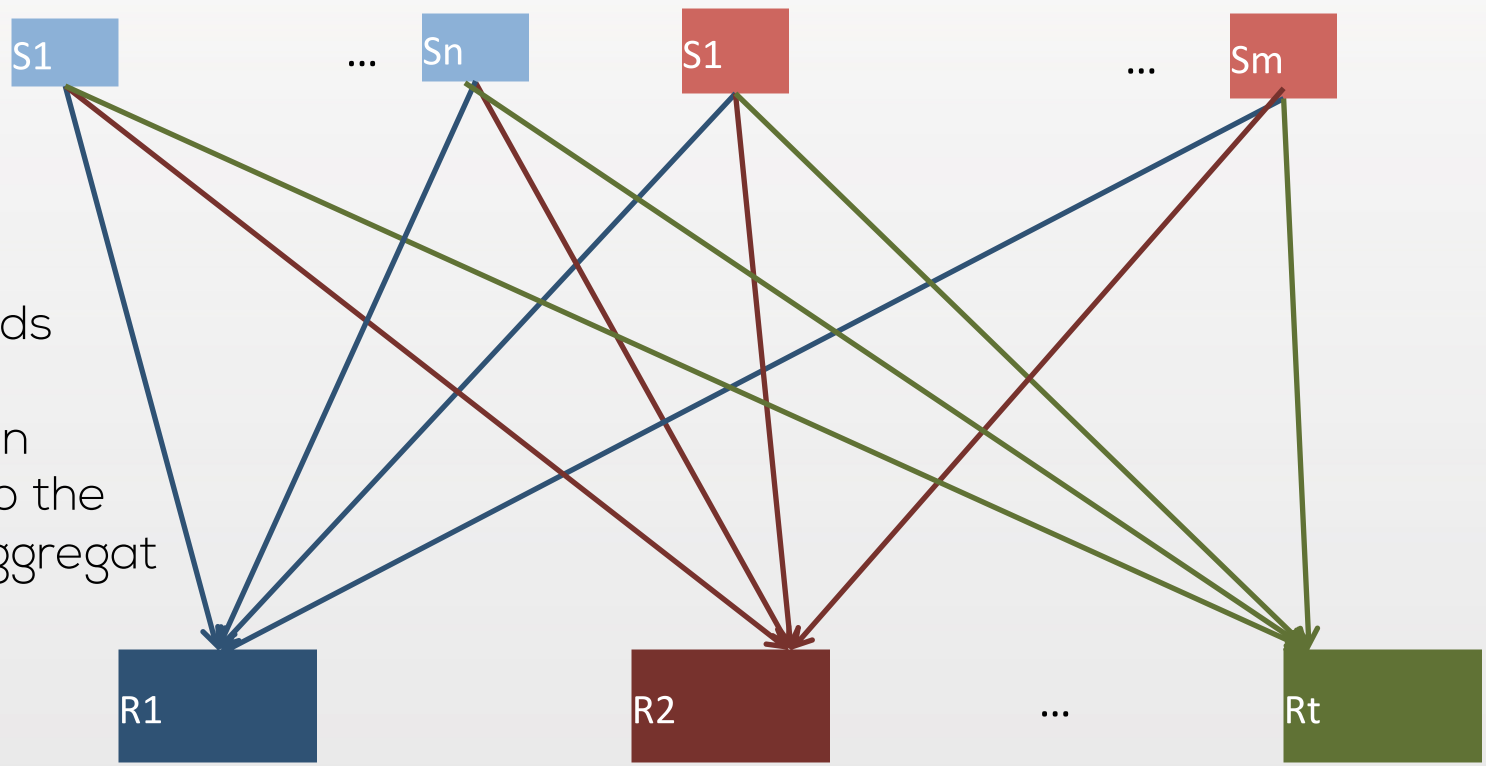
Multi-Dataset Operations on Records of Pairs: Joins

- Join-like operations can greatly be optimized depending on:
  - The size of the datasets
  - Whether or not the datasets are sorted, or how they are initially partitioned into the nodes

Multi-Dataset Operations on Records of Pairs: Shuffle Joins

- When we have no idea of how datasets are partitioned, and the sizes of both are big, we resort to shuffle joins
  - The parallel step would yield
    - the join key
    - the appropriate projection of the row
    - the dataset id
  - The records with the same join key would be copied into the same aggregation units, and these units would yield the cartesian product of all the records after repartitioning the group of projected rows with the same keys per dataset

Multi-Dataset Operations on Records of Pairs: Shuffle Joins

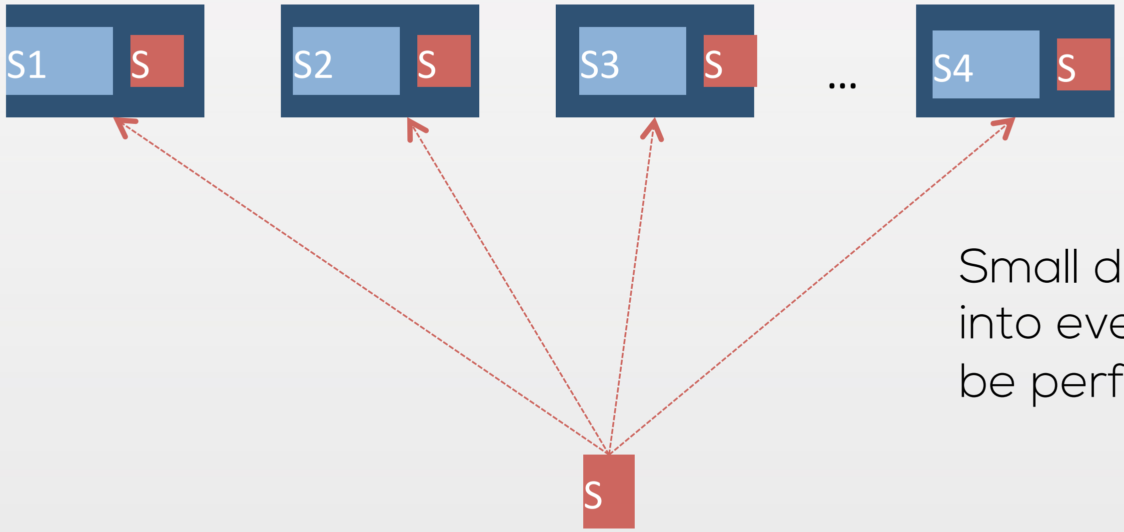


All records with the same join key go to the same aggregation task

Multi-Dataset Operations on Records of Pairs: Broadcast Joins

- If we knew that one of the datasets is small
  - We can broadcast the small dataset into all nodes
  - The join can be performed in parallel, without need of any copying

Multi-Dataset Operations on Records of Pairs: Broadcast Joins

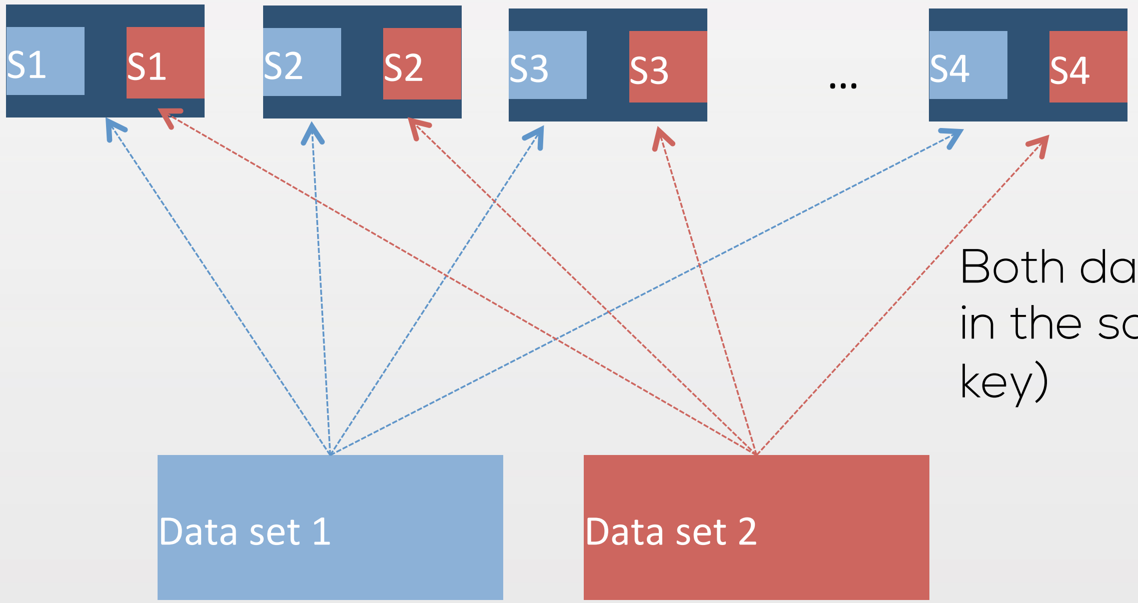


Small data set is broadcast into every node, and join can be performed in-place

Multi-Dataset Operations on Records of Pairs: Merge Joins

- If the datasets are partitioned (and sorted) in the same way, we ensure that the rows that should be joined are always included in the splits on the same node
- Then we can perform local joins and report the resulting dataset

Multi-Dataset Operations on Records of Pairs: Merge Joins



Both data sets are partitioned in the same way (by the join key)

Introduction to Processing Big Data

- Moving Computation to Data
- Parallel Operations
- Distributed Aggregations
- Accumulating and Broadcasting
- Working with Records of Pairs
- Multi-Dataset Operations
- **From Concepts to Frameworks**

From Concepts to Frameworks

- The described concepts are implemented by many frameworks, including
  - Apache Hadoop MapReduce
  - Apache Spark
- We only need to implement the local operations and the rest are performed automatically by the framework:
  - Parallel execution
  - Copying intermediate data
  - Ensuring all records with the same key are collected together
  - Execution of the aggregate operations

MapReduce

- Hadoop MapReduce allows us writing distributed programs, which process vast amounts of data
  - in a parallel and distributed manner
  - on large (of thousands of nodes) clusters
  - reliably

MapReduce

- MapReduce defines an InputFormat for splitting data into chunks, and iterating over records of pairs of these chunks
- MapReduce framework then;
  - Runs **user defined map functions** on the input records in a completely parallel manner
  - **Shuffles** (ensuring map outputs with the same key are kept together) the intermediate map outputs and copies them to the **Reducers**
  - **Sorts** and passes the map outputs with the same key **to the user defined reduce** functions
  - Reports the **Reducer** outputs
- The framework takes care of scheduling and monitoring tasks, and re-executing the failed ones

MapReduce

- Typically, the input(s) and output formats are files stored in the HDFS
  - Although this is the typical case, as long as the correct I/O formats defined, other sources and sinks are available
  - Typical MapReduce inputs are in the form of:
    - Plain text files (possibly compressed)
    - SequenceFiles
    - Avro Files
    - Parquet Files
    - ORC Files
    - HBase tables

  each of which is associated with different `InputFormat`s

analyticscenter

MapReduce

- Typically, the storage nodes (i.e., the DataNodes in the HDFS cluster) and compute nodes of MapReduce are the same
  - The MapReduce framework and the HDFS are running on the same cluster of nodes
  - That allows Map tasks to run on local data –**data locality**

MapReduce

- Applications specify
  - the I/O Formats,
  - input/output locations
  - *map* and *reduce* functions by implementing the appropriate interfaces (**`Mapper#map`**, **`Reducer#reduce`**)
    - To transform/filter records, **`map`** functions are defined to run on each input record
    - **`reduce`** functions are defined such that they run on groups of values attached to a key

MapReduce

- In MapReduce, broadcasting can be performed using the **DistributedCache**
- Accumulators can be implemented using MapReduce **Counters**

Processing Big Data End of Chapter