Distributed Data Processing Environments

José Orlando Pereira

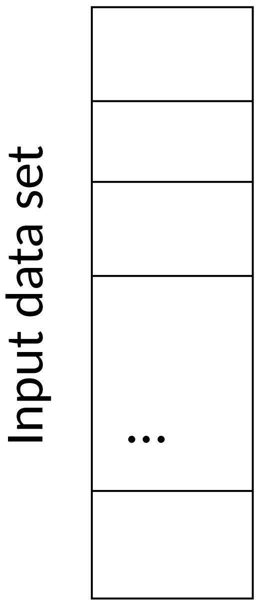
Departamento de Informática
Universidade do Minho

Roadmap

- Problem <u>too big</u> to be efficiently stored and processed by a <u>single server</u>
  - Distributed-parallel data processing

- Problem too complex to be expressed as a single step and/or with a single tool
  - DAG orquestration

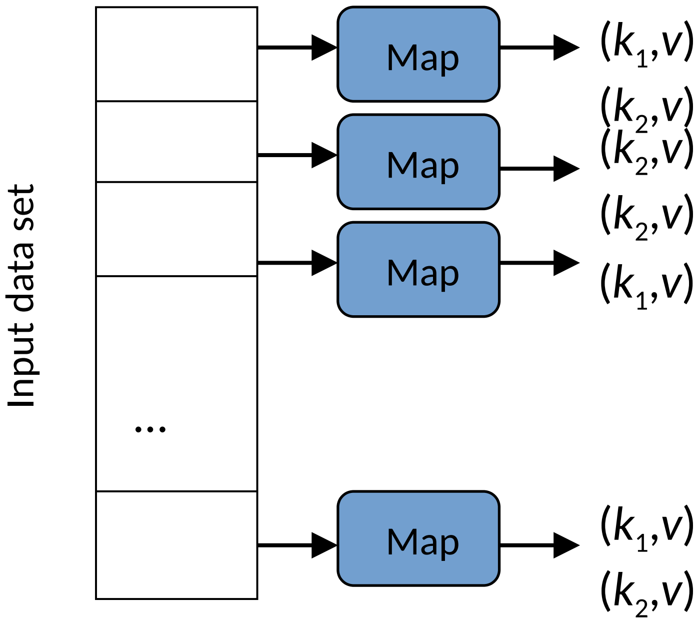
Map Reduce



- Data is <u>split</u> in chunks that can be handled separately
- Done by the framework, but might need help from the programmer

Image adapted from: M. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems (3rd Edition)

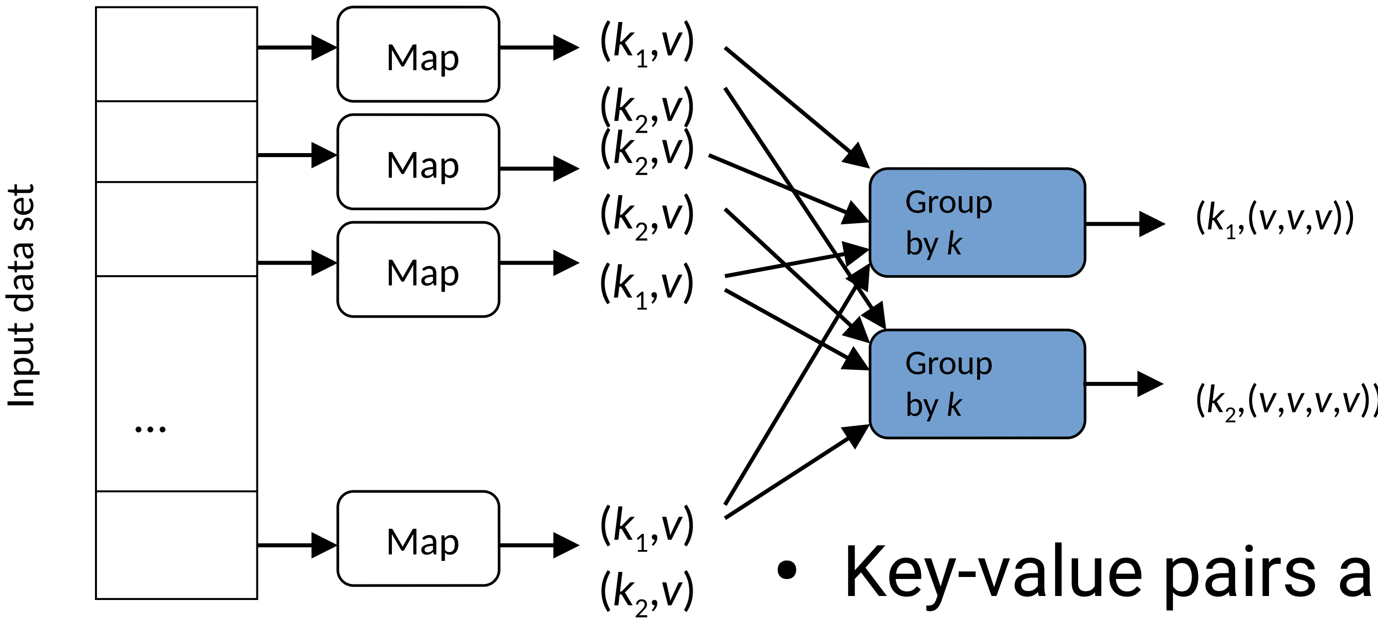
Map Reduce



- Input data items are <u>mapped</u> to zero or more key/value pairs
- Map function is provided by the programmer
  - Can deal with arbitrary and unstructured data formats (e.g., plain text)

Image adapted from: M. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems (3rd Edition)

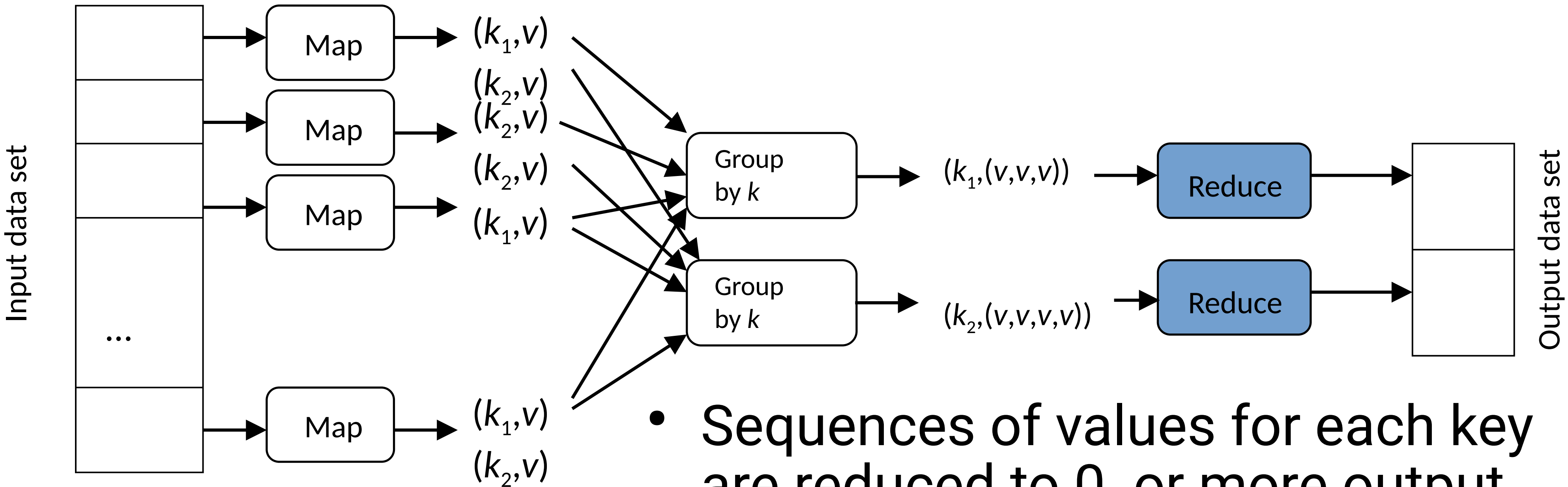
Map Reduce



- Key-value pairs are grouped by key
- Done by the framework

Image adapted from: M. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems (3rd Edition)

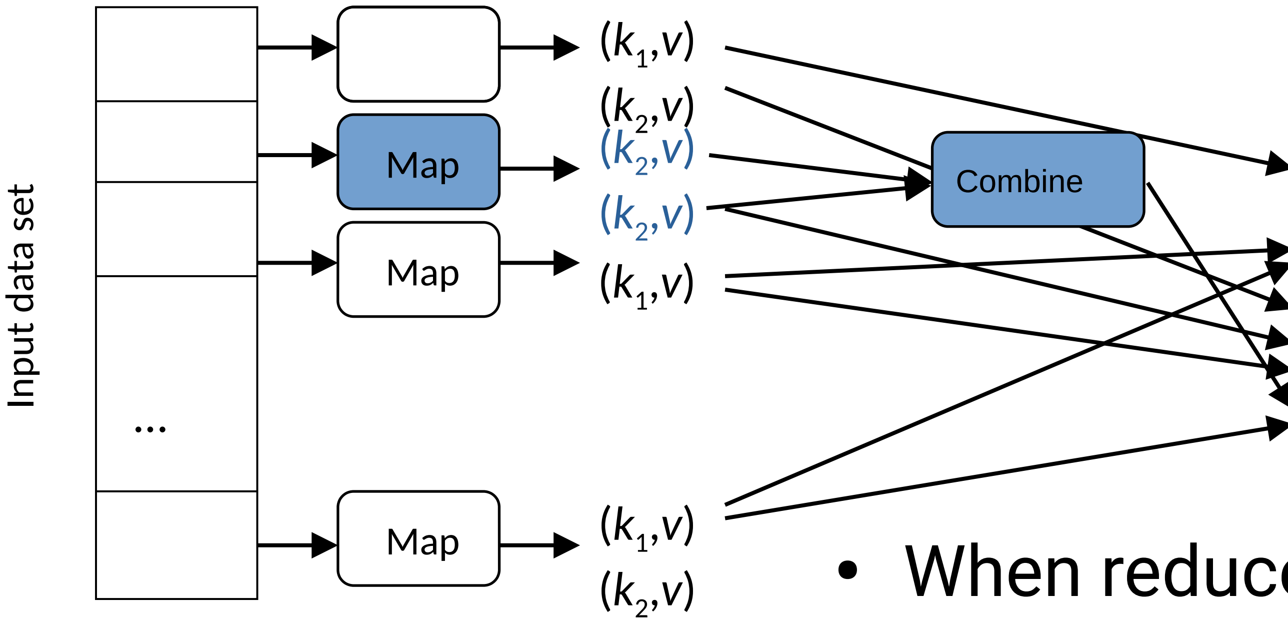
Map Reduce



- Sequences of values for each key are <u>reduced</u> to 0 or more output data items
- Reduce function is provided by the programmer

Image adapted from: M. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems (3rd Edition)

Map Reduce



- When reduce is associative, values for the same key from the same mapper can be <u>combined</u>
- Lessens the data to be grouped

Image adapted from: M. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems (3rd Edition)

Selection and projection

- Example:

  select x, y+1 from … where z = …;

- Can be performed by the Map stage:
  - Return computed key/value for those items that match

Grouped aggregates

- Example:

  select x,sum(y) from … group by x;

- In the Map stage:
  - Return Key=x and Value=y
- Optionally use a Combiner stage
- In the Reduce stage:
  - Iterate over values and return Key,sum(Values)

Map/Broadcast join

- Example:

  select a.y,b.z from A join B on a.x=b.x;

- Assumptions:
  - One data set (B) is small
  - No assumption on number of occurrences
- Before Map, cache B in all workers
- In the Map stage:
  - Lookup a.x in B, getting b.z
  - Return a.y and b.z

Map/Broadcast join

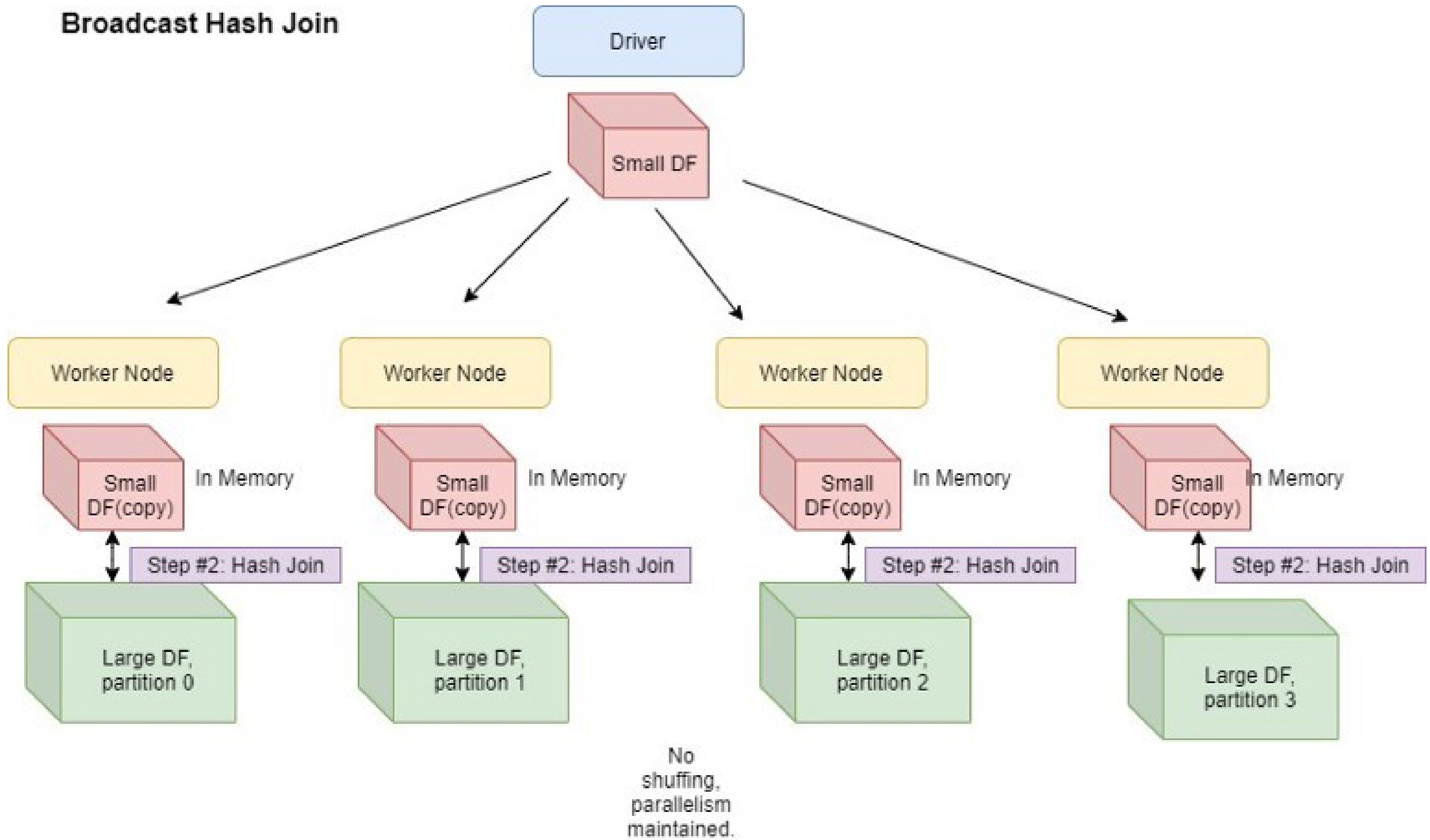


Image from: https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf

Shuffle join

- Example:

  select a.y,b.z from A join B on a.x=b.x;

- Assumptions:
  - Both input data sets are large
  - Small number of ocurrences of each key
- In the Map stage:
  - For A: Return Key=x and Value=(LEFT,y)
  - For B: Return Key=x and Value=(RIGHT,z)
- In the Reduce stage:
  - Collect all (LEFT,…) and (RIGHT,…) pairs
  - Return all combinations

Shuffle

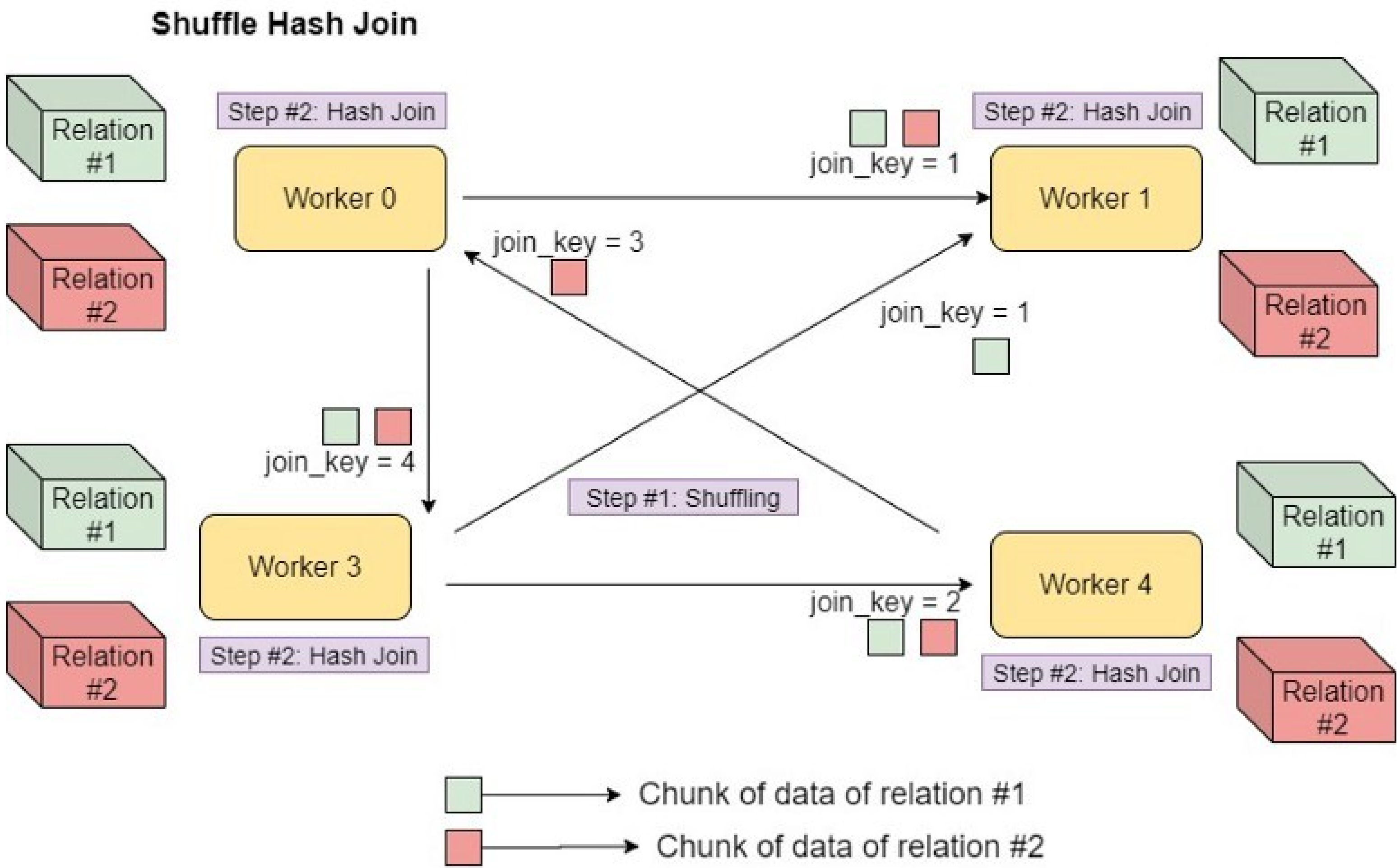


Image from: https://towardsdatascience.com/strategies-of-spark-join-c0e7b4572bcf

Sort

- Example:

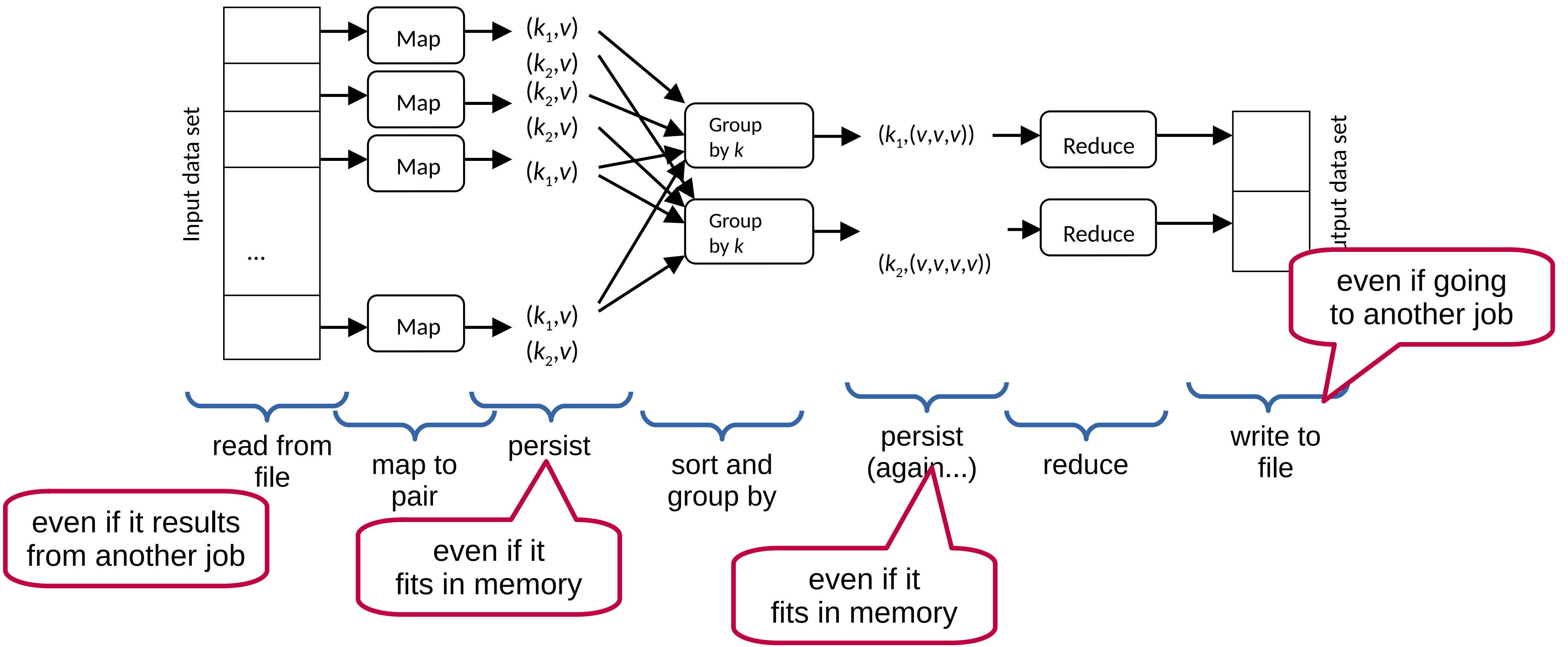
  select x,y from … order by y;

- In the Map stage:
  - Return Key=y and Value=x
- Use one identity reducer task

Chaining

- Multiple MapReduce jobs can be chained to compose operations
- This is necessary when sorting and grouping need to be done on multiple criteria
  - e.g., join on x, group by y

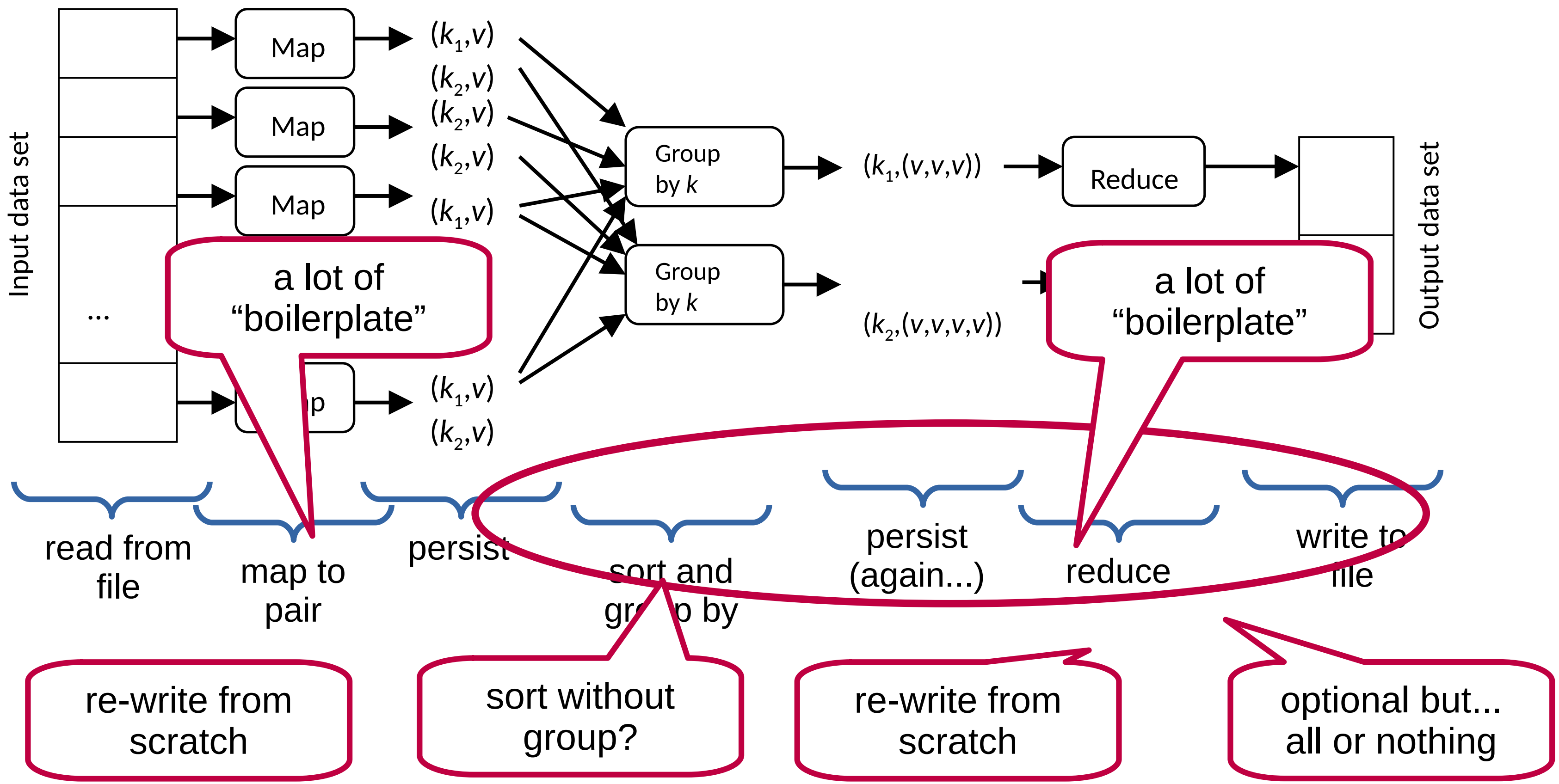
MR: Efficiency limitations



Need to avoid writing to disk so much!

Image adapted from: M. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems (3rd Edition)

MR: Usability limitations



Need to freely compose stages!

Image adapted from: M. Tamer Özsu, Patrick Valduriez. Principles of Distributed Database Systems (3rd Edition)

Distributed dataflow

- Generalized MapReduce:
  - Map, shuffle, reduce, and persistence stages
  - Can be arbitrarily composed
- Efficiency improved with:
  - Caching
  - Query optimization and code generation

Example

```java
public class GroupByTest {
    public static void main(String[] args) throws Exception {
        int numMappers = 100;
        int numKVPairs = 10000;
        int valSize = 1000;
        int numReducers = 36;

        SparkConf conf = new SparkConf().setMaster("local").setAppName("GroupBy Test");
        JavaSparkContext sc = new JavaSparkContext(conf);

        List<Integer> data = IntStream.range(0, numMappers).boxed().collect(Collectors.toList());
        JavaPairRDD<Integer, byte[]> pairs1 = sc.parallelize(data, numMappers)
                .flatMapToPair(p -> {
                    Random ranGen = new Random();
                    Stream<Tuple2<Integer, byte[]>> arr1 = IntStream.range(0, numKVPairs).mapToObj(i -> {
                        byte[] byteArr = new byte[valSize];
                        ranGen.nextBytes(byteArr);
                        return new Tuple2<>(ranGen.nextInt(), byteArr);
                    });
                    return arr1.iterator();
                });

        long count1 = pairs1.count();
        long count2 = pairs1.groupByKey(numReducers).count();

        System.out.println(count1+" "+count2);
    }
}
```

Example

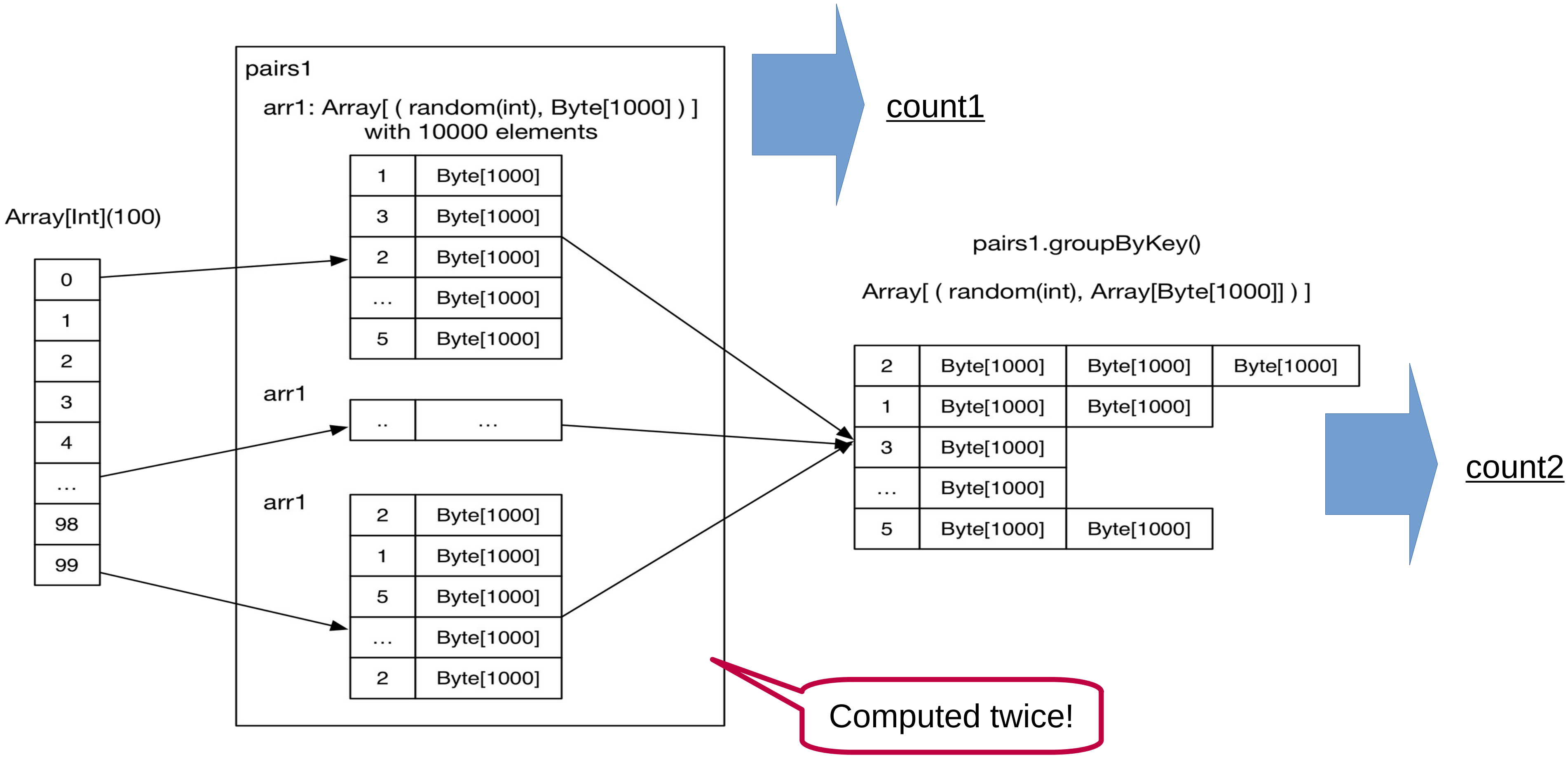


Image from: https://github.com/JerryLead/SparkInternals

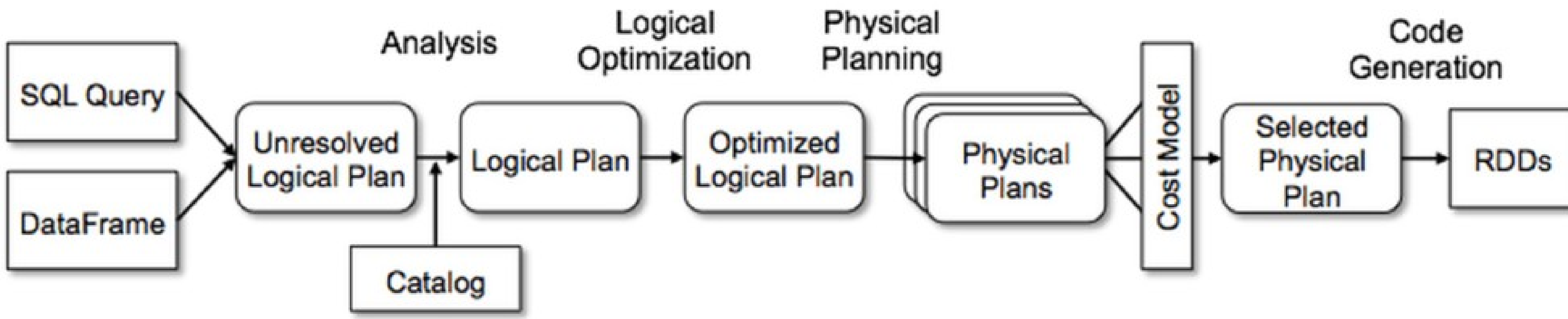
Caching example

```java
public class GroupByTest {
    public static void main(String[] args) throws Exception {
        int numMappers = 100;
        int numKVPairs = 10000;
        int valSize = 1000;
        int numReducers = 36;

        SparkConf conf = new SparkConf().setMaster("local").setAppName("GroupBy Test");
        JavaSparkContext sc = new JavaSparkContext(conf);

        List<Integer> data = IntStream.range(0, numMappers).boxed().collect(Collectors.toList());
        JavaPairRDD<Integer, byte[]> pairs1 = sc.parallelize(data, numMappers)
                .flatMapToPair(p -> {
                    Random ranGen = new Random();
                    Stream<Tuple2<Integer, byte[]>> arr1 = IntStream.range(0, numKVPairs).mapToObj(i -> {
                        byte[] byteArr = new byte[valSize];
                        ranGen.nextBytes(byteArr);
                        return new Tuple2<>(ranGen.nextInt(), byteArr);
                    });
                    return arr1.iterator();
                }).cache();

        long count1 = pairs1.count();
        long count2 = pairs1.groupByKey(numReducers).count();

        System.out.println(count1+" "+count2);
    }
}
```

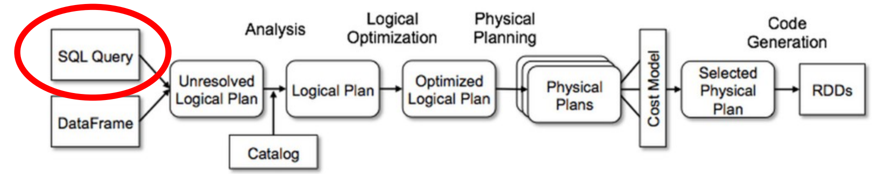
Store interim result for both jobs

Adapted from: https://github.com/JerryLead/SparkInternals

Query processing



Example

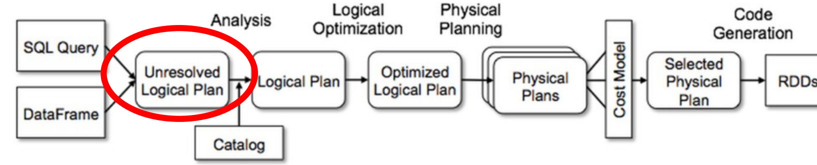


- Consider the following code:

```
Dataset<Row> ds = spark.sql("select town from " +
        "post_code natural join invoice " +
        "natural join item where desc='stuff'");

ds.show();

ds.explain(true);
```

Shows how the query is
processed in the Spark SQL pipeline

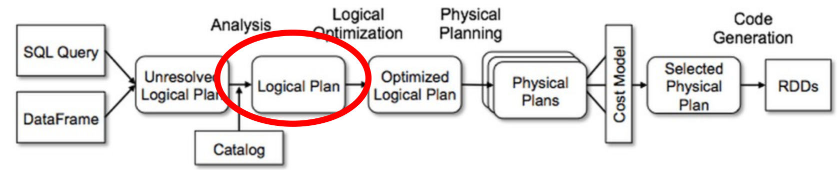
Example



- Parsed logical plan:

```
'Project ['town]
+- 'Filter ('desc = stuff)
   +- 'Join NaturalJoin(Inner)
      :- 'Join NaturalJoin(Inner)
      :  :- 'UnresolvedRelation [post_code], [], false
      :  +- 'UnresolvedRelation [invoice], [], false
      +- 'UnresolvedRelation [item], [], false
```

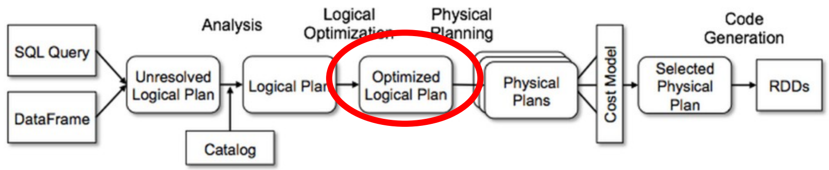
Example



- Analyzed logical plan:

```
town: string
Project [town#59]
+- Filter (desc#39 = stuff)
   +- Project [item#17, code#58, town#59, invoice#16, desc#39]
      +- Join Inner, (item#17 = item#38)
         :- Project [code#58, town#59, invoice#16, item#17]
         :  +- Join Inner, (code#58 = code#18)
         :     :- SubqueryAlias post_code
         :     :  +- Relation[code#58,town#59] csv
         :     +- SubqueryAlias invoice
         :        +- Relation[invoice#16,item#17,code#18] csv
         +- SubqueryAlias item
            +- Relation[item#38,desc#39] csv
```

Example



- Optimized logical plan:

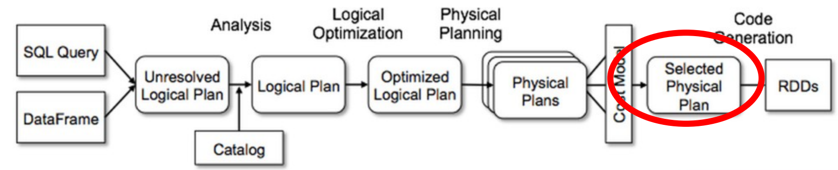
```
Project [town#59]
+- Join Inner, (item#17 = item#38)
   :- Project [town#59, item#17]
   :  +- Join Inner, (code#58 = code#18)
   :     :- Filter isnotnull(code#58)
   :     :  +- Relation[code#58,town#59] csv
   :     +- Project [item#17, code#18]
   :        +- Filter (isnotnull(code#18) AND isnotnull(item#17))
   :           +- Relation[invoice#16,item#17,code#18] csv
   +- Project [item#38]
      +- Filter ((isnotnull(desc#39) AND (desc#39 = stuff)) AND isnotnull
         +- Relation[item#38,desc#39] csv
```

- Using ds.explain("cost") shows additional statistics

Code generation

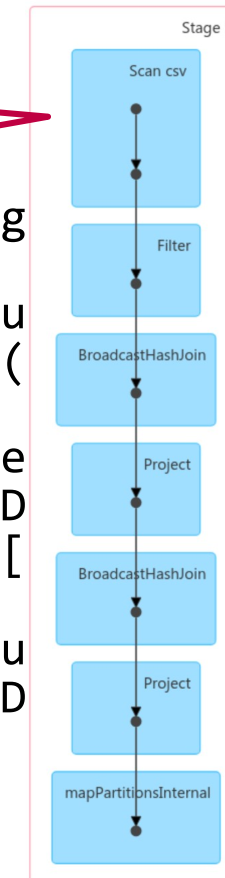
- Converts a physical plan into actual code that can be executed
- Simple code generation:
  - Produces sequence of generic RDD transformations
- Whole stage code generation:
  - Produces custom RDD transformation combining all operations in each stage
  - Resulting code is very different from manually written code…

Example



- Physical plan:

Compiles to a single Map step!




```
*(3) Project [town#59]
+- *(3) BroadcastHashJoin [item#17], [item#38], Inner, BuildRig
   :- *(3) Project [town#59, item#17]
   :  +- *(3) BroadcastHashJoin [code#58], [code#18], Inner, Bu
   :     :- BroadcastExchange HashedRelationBroadcastMode(List(
   :     :  +- *(1) Filter isnotnull(code#58)
   :     :     +- FileScan csv [code#58,town#59] Batched: false
   :        +- FileScan csv [item#17,code#18] Batched: false, D
   +- BroadcastExchange HashedRelationBroadcastMode(List(input[
      +- *(2) Project [item#38]
         +- *(2) Filter ((isnotnull(desc#39) AND (desc#39 = stu
            +- FileScan csv [item#38,desc#39] Batched: false, D
```

- Using ds.explain(“codegen”) shows final Java code

Summary

- Distributed-parallel processing is applicable for extremely large datasets
- SQL compilation is extended to use distributed operators:
  - Data exchange as a new dimension for optimization