Exercise 4: MapReduce and Spark

Concepts and Models of Parallel and Data-centric Programming

Lecture, Summer 2019

Dr. Christian Terboven

i12 High Performance Computing | RWTH AACHEN UNIVERSITY

Task 1: Maximum Temperature

Task 1

- Set of temperature measurements (in $°C$) for different cities
- Number of key-value pairs *(city name, temperature values)*
  - City name is a **`String`**
  - Temperature values a **`String`** with comma-separated floating point values.
- Multiple key-value pairs for each city
- Use the MapReduce programming model to determine the *maximum temperature* for each city.
- Some measurement values are inconsistent: Filter out values where temperature greater than $60\ °C$ or lower than $-90\ °C$.

Example KV pairs:

```
("Aachen", "9.3, 15.2, 24.0")
("Berlin", "4.1, 6.4, 100.4")
("Cologne", "-5.5, 14.2")
("Aachen", "2.0, 16.0, 9.0")
("Berlin", "6.4, 100.4")
("Cologne", "-2.0, -137.3")
```

Task 1.1

Example values:

```
("Aachen", "9.3, 15.2, 24.0")
("Berlin", "4.1, 6.4, 100.4")
("Cologne", "-5.5, 14.2")
("Aachen", "2.0, 16.0, 9.0")
("Berlin", "6.4, 100.4")
("Cologne", "-2.0, -137.3")
```

Task 1.1

- Give the required **`Map()`** and **`Reduce()`** functions in pseudocode to…
  - … filter out invalid values (greater than $60\,°C$ or lower than $-90\,°C$)
  - … determine the maximum temperature for each city
- Furthermore: Denote type signatures of both functions.
- Constraint: No pre-combination of values in the **`Map()`** function.
- Note: Use function **`extractFloats(String valString)`** which extracts list of float values out of comma-separated String

$$Map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$Reduce\big(k_2, list(v_2)\big) \rightarrow list(k_3, v_3)$$

Task 1.1 – Solution

Solution:

Type signatures:

```
Map(String,String) → List<String,Float>
Reduce(String, List<Float>) → List<String,Float>
```

Pseudocode:

```
map(String city, String valString):
    List<Float> valList = extractValues(valString)
    for each temp t in valList:
        if t <= 60.0 and t >= -90.0:
            Emit(city, t)

reduce(String city, Iterator<Float> temps):
    float max = MIN_FLOAT
    for each temp t in temps:
        if t > max:
            max = t
    Emit(city, max)
```

$$Map(k_1, v_1) \rightarrow list(k_2, v_2)$$
$$Reduce\big(k_2, list(v_2)\big) \rightarrow list(k_3, v_3)$$

Task 1.1 – Solution

Alternative Solution:

Type signatures:

```
Map(String,String) → List<String,List<Float>>
Reduce(String, List<List<Float>>) → List<String,Float>
```

Pseudocode:

```
map(String city, String valString):
    Emit(city, extractValues(valString))

reduce(String city, Iterator<List<Float>> temps):
    float max = MIN_FLOAT
    for each tempList tl in temps:
        for each temp t in tl:
            if t <= 60.0 and t >= -90.0 and t > max:
                max = t
    Emit(city, max)
```

But: No pre-filtering done in Map function, serializing a list and sending over wire more complex.

Task 1.2

- Perform the MapReduce computation on the given dataset manually.
  - Apply the defined **`Map()`** function on each input key-value pair
  - Group the corresponding outputs by key
  - Apply the **`Reduce()`** function on it to generate the final output

```
map(String city, String valString):
    List<Float> valList = extractValues(valString)
    for each temp t in valList:
        if t <= 60.0 and t >= -90.0:
            Emit(city, t)

reduce(String city, Iterator<Float> temps):
    float max = MIN_FLOAT
    for each temp t in temps:
        if t > max:
            max = t
    Emit(city, max)
```

Task 1.2 – Solution

```
Map(String,String) → List<String,Float>
Reduce(String, List<Float>) → List<String,Float>
```

Map values:

```
Map("Aachen", "9.3, 15.2, 24.0")  = [("Aachen",9.3),("Aachen",15.2),
                                     ("Aachen",24.0)]
Map("Berlin", "4.1, 6.4, 100.4")  = [("Berlin",4.1),("Berlin",6.4)]
Map("Cologne", "-5.5, 14.2")      = [("Cologne",-5.5),("Cologne",14.2)]
Map("Aachen", "2.0, 16.0, 9.0")   = [("Aachen",2.0),("Aachen",16.0),
                                     ("Aachen",9.0)]
Map("Berlin", "6.4, 100.4")       = [("Berlin",6.4)]
Map("Cologne", "-2.0, -137.3")    = [("Cologne",-2.0)]
```

Group by key:

```
("Aachen", [9.3, 15.2, 24.0, 2.0, 16.0, 9.0])
("Berlin", [4.1, 6.4, 6.4])
("Cologne", [-5.5, 14.2, -2.0])
```

Task 1.2 – Solution

```
Map(String,String) → List<String,Float>
Reduce(String, List<Float>) → List<String,Float>
```

Reduce values:

```
Reduce("Aachen", [9.3, 15.2, 24.0, 2.0, 16.0, 9.0]) = 24.0
Reduce("Berlin", [4.1, 6.4, 6.4])  = 6.4
Reduce("Cologne", [-5.5, 14.2, -2.0]) = 14.2
```

Task 1.3 – Solution

Task 1.3

- Which (combination of) MapReduce design patterns did you use to solve the task?

Solution

- **Numerical Summarization Pattern** and **Filtering Pattern**

$$Combine(k_2, list(v_2)) \rightarrow list(k_2, v_2)$$

Task 1.4 – Solution

Task 1.4

- Is using a **`Combiner()`** function useful?
  - If so, give the function in pseudocode and its type signature.
  - If not, state why a combiner is not applicable.

Solution

- Yes, we can use our defined reducer as combiner (maximum computation associative):
  - Type signature: `Combine(String, List<Float>) → List<String,Float>`
  - Pseudocode:

```
combine(String city, Iterator<Float> temps):
    float max = MIN_FLOAT
    for each temp t in temps:
        if t > max:
            max = t
    Emit(city, max)
```

Task 2: MapReduce and Friends

Task 2

- You are a developer of a social network service.
- If some user visits profile of friend, the friends they both have in common should be displayed.
- Recomputing list of common friends on each profile visit is expensive.
- Use MapReduce to precompute list for each possible combination of two friends and store it.
- Each user and his friends stored as key-value pairs.
- Friend relation is *symmetric*, i.e., if Alice is friend of Bob, then Bob is friend of Alice.

Example KV pairs:

```
Alice -> [Bob, Dave, Eve]
Bob   -> [Alice, Carol, Dave, Eve]
Carol -> [Bob, Dave]
Dave  -> [Alice, Bob, Carol]
Eve   -> [Alice, Bob]
```

Task 2

Example KV pairs:

```
Alice -> [Bob, Dave, Eve]
Bob   -> [Alice, Carol, Dave, Eve]
Carol -> [Bob, Dave]
Dave  -> [Alice, Bob, Carol]
Eve   -> [Alice, Bob]
```

Example output for Alice and Bob:

```
(Alice, Bob) -> [Dave, Eve]
```

Task 2

- Key idea 1: Emit for a given user and his friends list for each combination with a friend the complete friends list.
  - Key: Tuple of combined two users
  - Value: Corresponding friends list
- Example:
  - Input `Alice -> [Bob, Dave, Eve]` leads to `Map()` output:

```
(Alice, Bob)  -> [Bob, Dave, Eve]
(Alice, Dave) -> [Bob, Dave, Eve]
(Alice, Eve)  -> [Bob, Dave, Eve]
```

  - Input `Bob -> [Alice, Carol, Dave, Eve]` leads to `Map()` output:

```
(Bob, Alice)  -> [Alice, Carol, Dave, Eve]
(Bob, Carol)  -> [Alice, Carol, Dave, Eve]
(Bob, Dave)   -> [Alice, Carol, Dave, Eve]
(Bob, Eve)    -> [Alice, Carol, Dave, Eve]
```

Task 2

- Key idea 2: For each combination of two friends, we get **exactly** two lists in the **`Reduce()`** step.
  - Reason: Friend relation is symmetric, so we get one list for **`(Alice, Bob)`** and one list for **`(Bob, Alice)`**
  - Note: As stated in the exercise, the pairs **`(Alice, Bob)`** and **`(Bob, Alice)`** are “equal” and thus are grouped together in the shuffle step.
  - Just calculate the **intersection of the two lists** for each combination to get the list of common friends.

Task 2.1

Give the required **`Map()`** function for the MapReduce computation in pseudocode. You may assume that the signature of the function is

```
Map: (String, List<String>)
      -> List(Pair<String, String>, List<String>)
```

Solution:

Take each friend **`f`** of friends list and emit as key tuple **`(user, f)`** together with friends list **`friends`** as value.

```
map(String user, List<String> friends):
    for each friend f in friends:
        Emit(new Pair(user, f), friends)
```

Task 2.2

- Apply the `Map()` function to each key-value pair given in the example.
- For simplicity, only write down the first letter of each user name.
- After that, group the output by key to perform the shuffle step.

Map Solution (part 1):

```
Map(A, [B, D, E])    = [((A, B), [B, D, E]),
                        ((A, D), [B, D, E]),
                        ((A, E), [B, D, E])]

Map(B, [A, C, D, E]) = [((B, A), [A, C, D, E]),
                        ((B, C), [A, C, D, E]),
                        ((B, D), [A, C, D, E]),
                        ((B, E), [A, C, D, E])]
```

Task 2.2

Map Solution (part 2):

```
Map(C, [B, D])    = [((C, B), [B, D]),
                     ((C, D), [B, D])]

Map(D, [A, B, C]) = [((D, A), [A, B, C]),
                     ((D, B), [A, B, C]),
                     ((D, C), [A, B, C])]

Map(E, [A, B])    = [((E, A), [A, B]),
                     ((E, B), [A, B])]
```

Task 2.2

```
Map(A, [B, D, E])    = [((A, B), [B, D, E]),
                        ((A, D), [B, D, E]),
                        ((A, E), [B, D, E])]

Map(B, [A, C, D, E]) = [((B, A), [A, C, D, E]),
                        ((B, C), [A, C, D, E]),
                        ((B, D), [A, C, D, E]),
                        ((B, E), [A, C, D, E])]

Map(C, [B, D])    = [((C, B), [B, D]),
                     ((C, D), [B, D])]

Map(D, [A, B, C]) = [((D, A), [A, B, C]),
                     ((D, B), [A, B, C]),
                     ((D, C), [A, B, C])]

Map(E, [A, B])    = [((E, A), [A, B]),
                     ((E, B), [A, B])]
```

Group Solution:

```
(A, B): [[B, D, E], [A, C, D, E]]
(A, D): [[B, D, E], [A, B, C]]
(A, E): [[B, D, E], [A, B]]
(B, C): [[A, C, D, E], [B, D]]
(B, D): [[A, C, D, E], [A, B, C]]
(B, E): [[A, C, D, E], [A, B]]
(C, D): [[B, D],  [A, B, C]]
```

Task 2.3

- Give the required **`Reduce()`** function for the MapReduce computation in pseudocode.
- You may assume that the signature of the function is

```
Reduce: (Pair<String, String>, List<List<String>>)
        -> List(Pair<String, String>, List<String>)
```

- **Hint:** Exploit that the nested input list always consists of two sublists.

Task 2.3

Solution

- Compute intersection of two sublists to get common friends for each combination of two friends.

```
Reduce(Pair<String, String> key, List<List<String>> values):
    list1 = values.get(0)
    list2 = values.get(1)
    intersecList = new List()

    // perform intersection of the sublists of friends
    for each friend l1 in list1:
        for each friend l2 in list2:
            if l1.equals(l2):
                intersecList.add(l1)

    Emit(key, intersecList)
```

Example: `Reduce((A, B), [[B, D, E], [A, C, D, E]]) = ((A, B), [D, E])`

Task 2.4

- Apply the `Reduce()` function to the output you got in Task 2.2 to generate the final output.
- Again, you might only write down the first letter of each user name.

Task 2.4

Group Solution:

```
(A, B): [[B, D, E], [A, C, D, E]]
(A, D): [[B, D, E], [A, B, C]]
(A, E): [[B, D, E], [A, B]]
(B, C): [[A, C, D, E], [B, D]]
(B, D): [[A, C, D, E], [A, B, C]]
(B, E): [[A, C, D, E], [A, B]]
(C, D): [[B, D],  [A, B, C]]
```

Example KV-Pairs:

```
Alice -> [Bob, Dave, Eve]
Bob   -> [Alice, Carol, Dave, Eve]
Carol -> [Bob, Dave]
Dave  -> [Alice, Bob, Carol]
Eve   -> [Alice, Bob]
```

Reduce Solution:

```
Reduce((A, B), [[B, D, E], [A, C, D, E]]) = ((A, B), [D, E])
Reduce((A, D), [[B, D, E], [A, B, C]])    = ((A, D), [B])
Reduce((A, E), [[B, D, E], [A, B]])       = ((A, E), [B])
Reduce((B, C), [[A, C, D, E], [B, D]])    = ((B, C), [D])
Reduce((B, D), [[A, C, D, E], [A, B, C]]) = ((B, D), [A, C])
Reduce((B, E), [[A, C, D, E], [A, B]])    = ((B, E), [A])
Reduce((C, D), [[B, D],  [A, B, C]])      = ((C, D), [B])
```

Task 3: RDD Dependencies in Apache Spark

Task 3.1 (1)

- Consider following two lists of key-value pairs:

```
List 1: [(a, 1), (b, 2), (a, 3), (c, 42), (d, 43), (e, 5))]
List 2: [(b, 3), (b, 6), (d, 10), (d, 13),
         (e, 12), (a, 41), (c, 4), (c, 0)]
```

- Lists should be stored as RDDs
- Custom hash function h_i ($i \in \mathbb{N} \setminus \{0\}$) is used to split data:

$$h_i\colon \mathbb{N} \to \{0, \ldots, i-1\}, \qquad n \mapsto n \bmod i$$

- Compute partitions of the following three RDDs:
  - RDD1 out of List 1 with hash function h_3
  - RDD2 out of List 2 with hash function h_3
  - RDD3 out of List 2 with hash function h_4
- Note: Key entries are characters, use ASCII table to convert them to decimals.

Task 3.1 (2)

- Hash value calculation for character ‘c’:
  - Lookup decimal value in ASCII table: 99
  - $h_3(99) = 99 \bmod 3 = 0$
  - $h_4(99) = 99 \bmod 4 = 3$
- Same calculation for all other characters

Character	Decimal	$h_3(n)$	$h_4(\mathrm{n})$
a	97	1	1
b	98	2	2
c	99	0	3
d	100	1	0
e	101	2	1

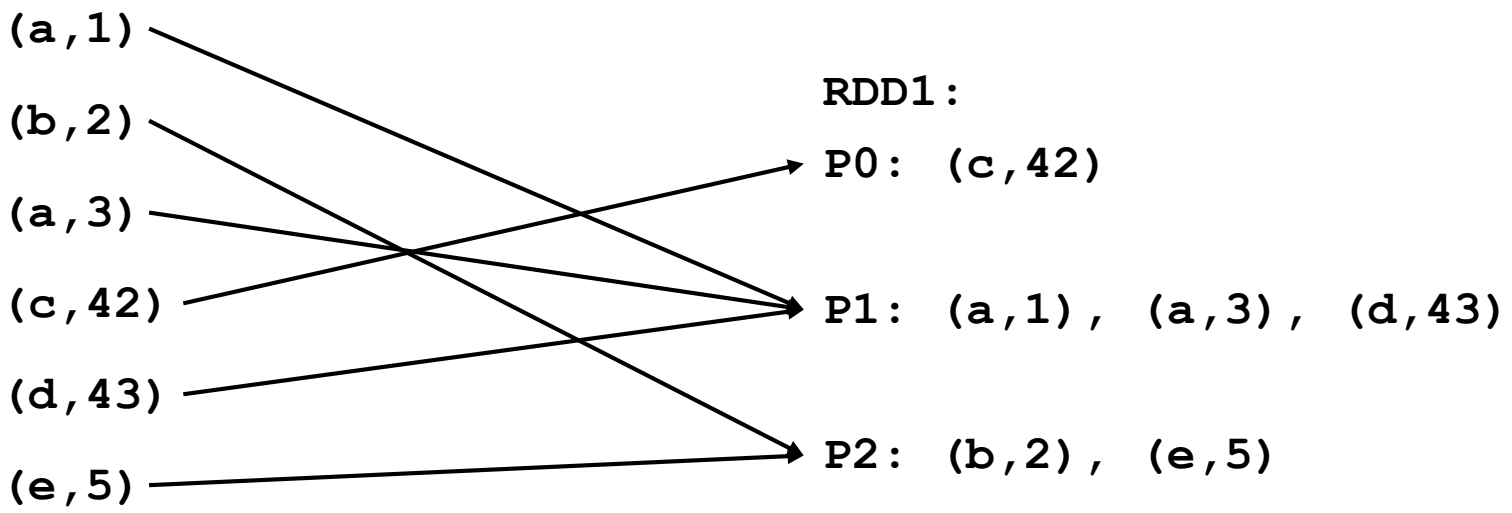
Task 3.1 (3)

- RDD1 out of List 1 with hash function h_3

```
List 1: [(a, 1), (b, 2), (a, 3),
         (c, 42), (d, 43), (e, 5))]
```

- Partitioning: **`P0`** contains all KV-pairs with hash value 0, **`P1`** all KV-pairs with value 1, ...

Char	Dec	$h_3(n)$	$h_4(\mathrm{n})$
a	97	1	1
b	98	2	2
c	99	0	3
d	100	1	0
e	101	2	1

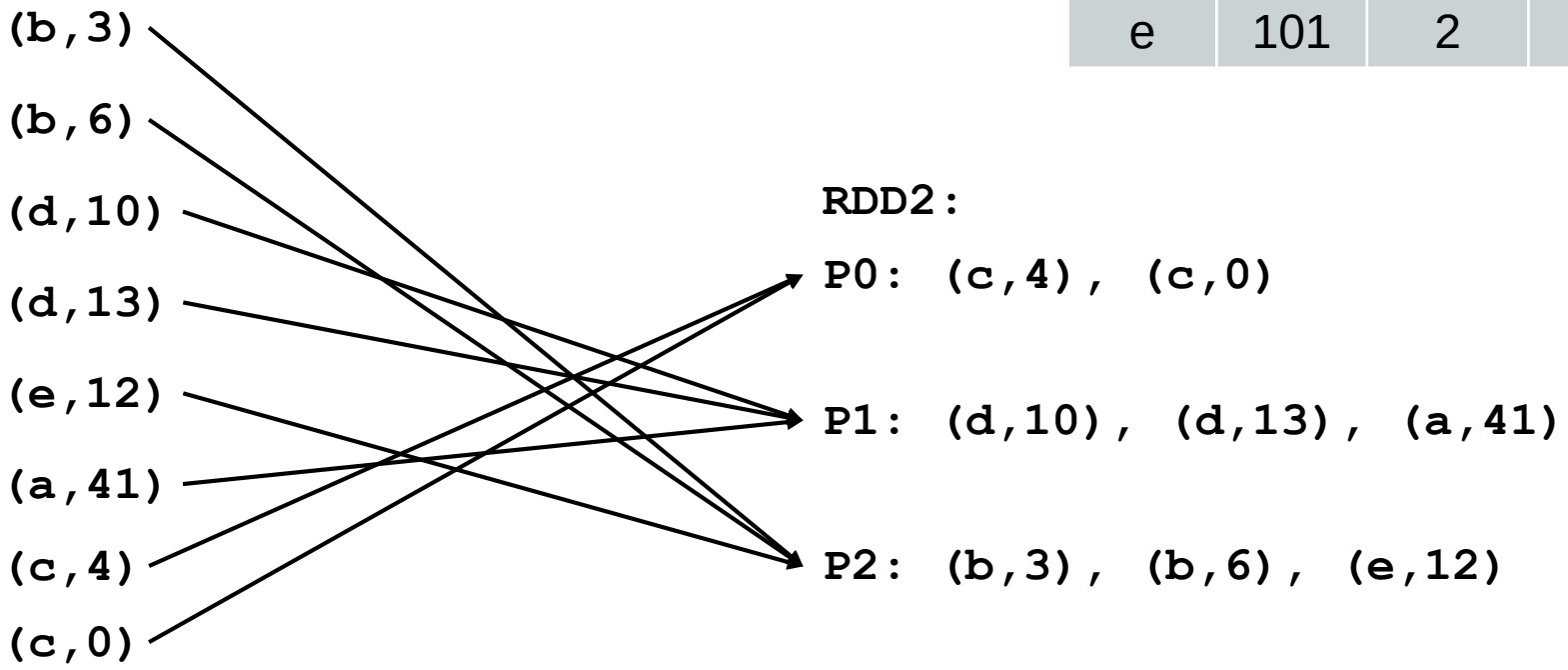


Task 3.1 (4)

- RDD2 out of List 2 with hash function h_3

```
List 2: [(b, 3), (b, 6), (d, 10),
         (d, 13), (e, 12), (a, 41),
         (c, 4), (c, 0)]
```

Char	Dec	$h_3(n)$	$h_4(\mathrm{n})$
a	97	1	1
b	98	2	2
c	99	0	3
d	100	1	0
e	101	2	1

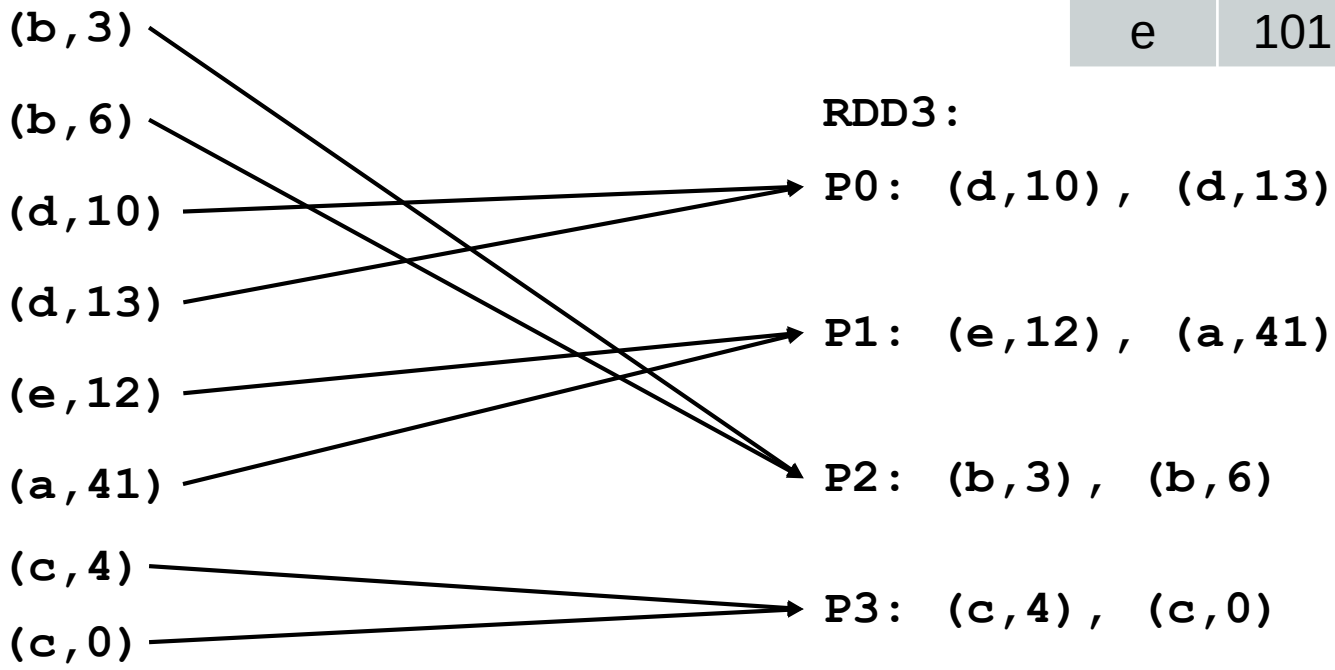


Task 3.1 (5)

- RDD3 out of List 2 with hash function h_4

```
List 2: [(b, 3), (b, 6), (d, 10),
         (d, 13), (e, 12), (a, 41),
         (c, 4), (c, 0)]
```

Char	Dec	$h_3(n)$	$h_4(\mathrm{n})$
a	97	1	1
b	98	2	2
c	99	0	3
d	100	1	0
e	101	2	1



Task 3.2 (1)

```
RDD1:
P0: (c,42)
P1: (a,1),(a,3),(d,43)
P2: (b,2),(e,5)
```

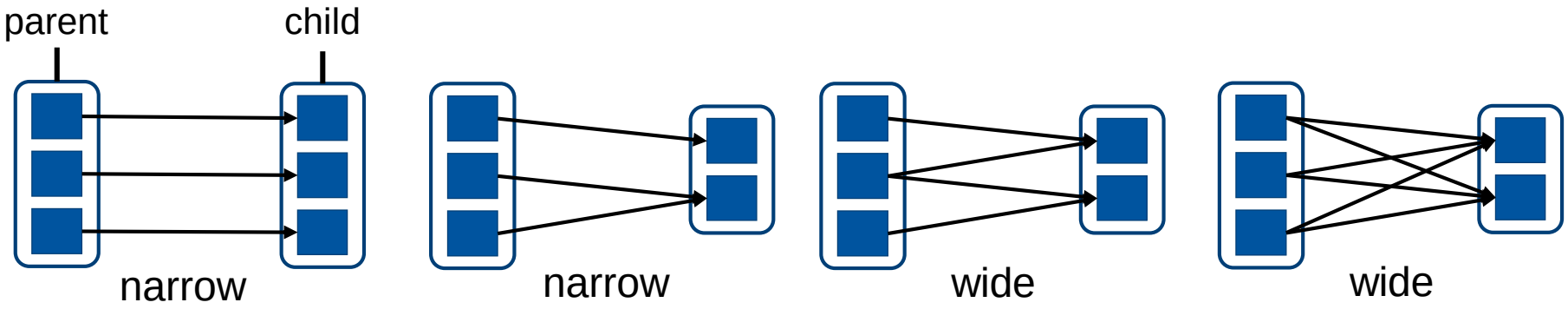
```
RDD2:
P0: (c,4),(c,0)
P1: (d,10),(d,13),(a,41)
P2: (b,3),(b,6),(e,12)
```

```
RDD3:
P0: (d,10),(d,13)
P1: (e,12),(a,41)
P2: (b,3),(b,6)
P3: (c,4),(c,0)
```

- Perform the following transformations on the RDDs by writing down the resulting RDD (use the same notation as in Task 2.1).
  - `mappedRDD = RDD1.mapValues(n -> n + 1)`
  - `unionRDD2 = mappedRDD.union(RDD2)`
  - `unionRDD3 = mappedRDD.union(RDD3)`
  - `joinRDD = mappedRDD.join(RDD2)`
  - `joinUnionRDD = mappedRDD.join(unionRDD3)`
  - `reducedRDD2 = RDD2.reduceByKey((a,b) -> a + b)`
  - `reducedUnionRDD3 = unionRDD3.reduceByKey((a, b) -> a + b)`
- State for each resulting RDD whether it has a narrow or wide dependency on its parent(s).

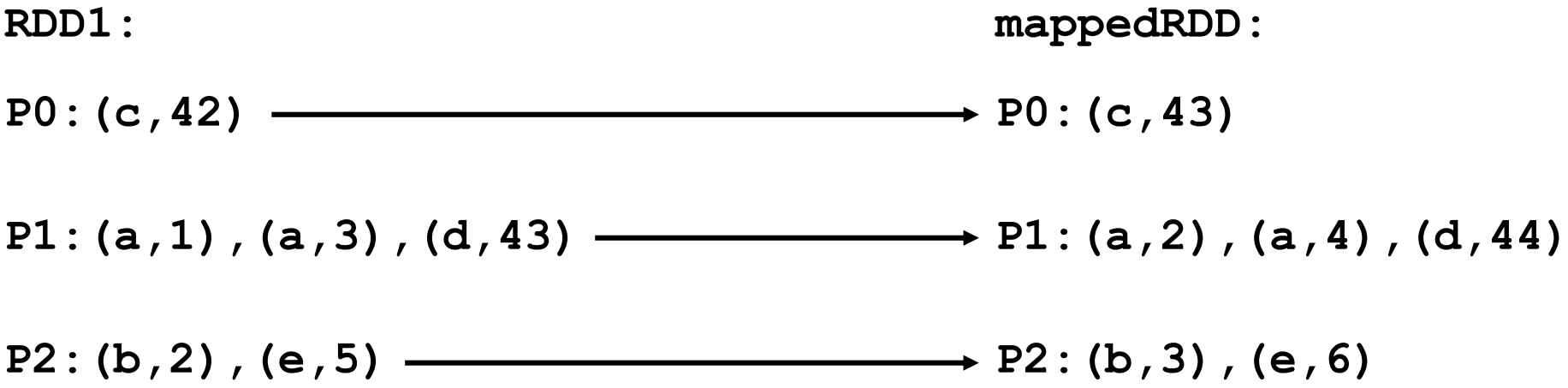
Reminder: Narrow and Wide Dependencies

- Narrow dependency
  - Each partition of parent RDD is used by **at most one** partition of child RDD.
  - Example transformations: *map*, *filter*, *union*
- Wide dependency
  - **Multiple child partitions** may depend on **one partition** of the parent RDD.
  - Example transformations (which *typically* have wide dependencies, but not in every case): *reduceByKey*, *groupByKey*



Task 3.2 (2)

Transformation: `mappedRDD = RDD1.mapValues(n -> n + 1)`

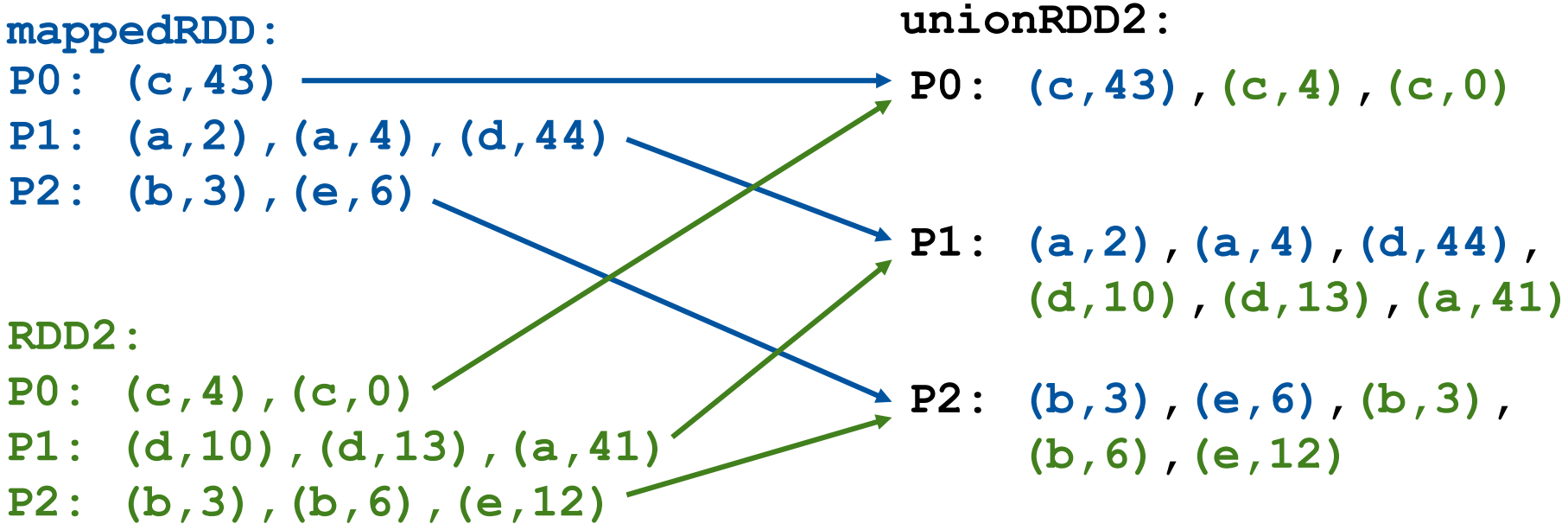


mappedRDD has a narrow dependency on **RDD1**.

Task 3.2 (3)

Transformation: `unionRDD2 = mappedRDD.union(RDD2)`

Note: **`union`** merges the resulting partitions only if the parent RDDs are co-partitioned. Otherwise, it just combines the partitions in a new RDD without merging them and the partitioning is lost.

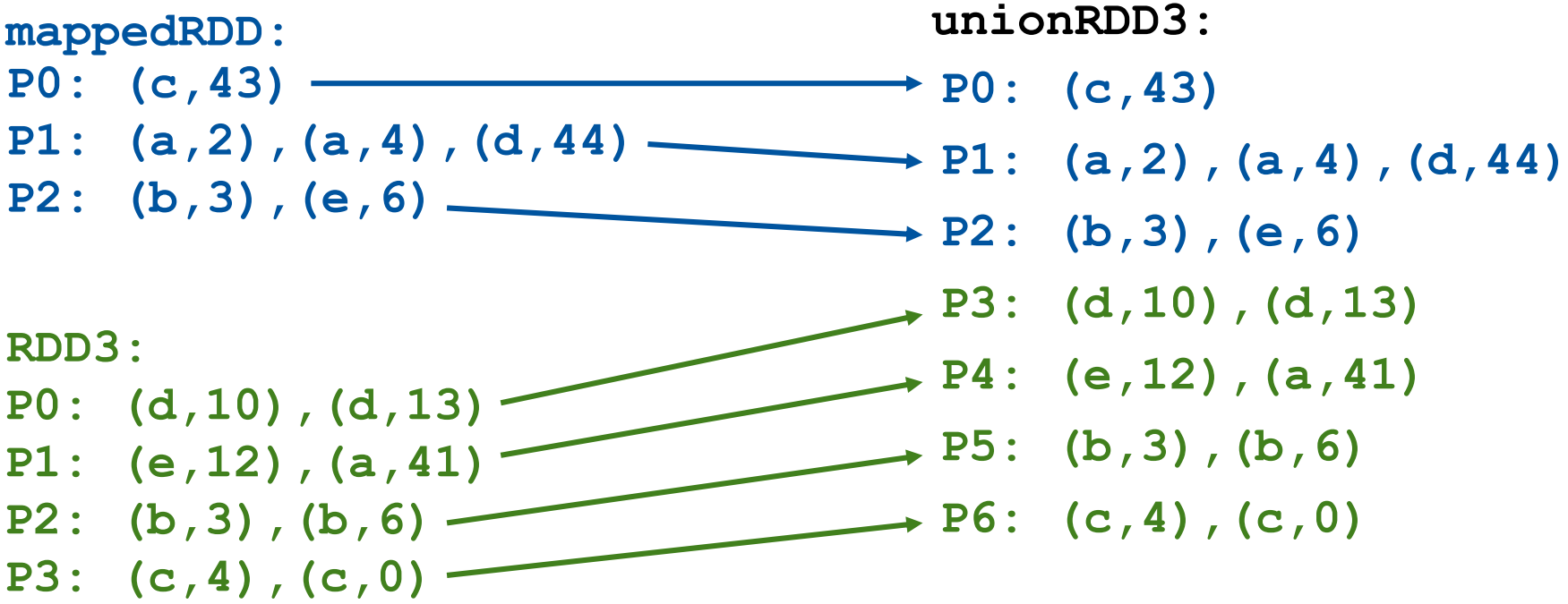


`unionRDD2` has narrow dependencies on **`mappedRDD`** and **`RDD2`**.

Task 3.2 (4)

Transformation: `unionRDD3 = mappedRDD.union(RDD3)`

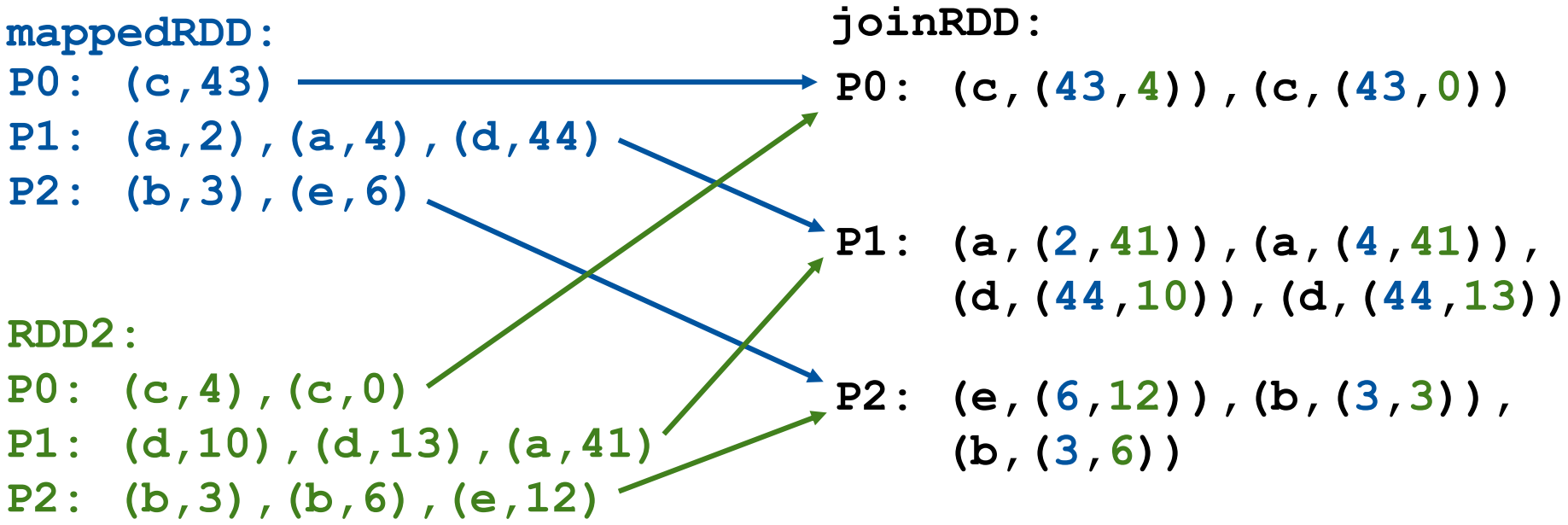
Note: **`union`** merges the resulting partitions only if the parent RDDs are co-partitioned. Otherwise, it just combines the partitions in a new RDD without merging them and the partitioning is lost.



`unionRDD3` has narrow dependencies on **`mappedRDD`** and **`RDD3`**.

Task 3.2 (5)

Transformation: `joinRDD = mappedRDD.join(RDD2)`

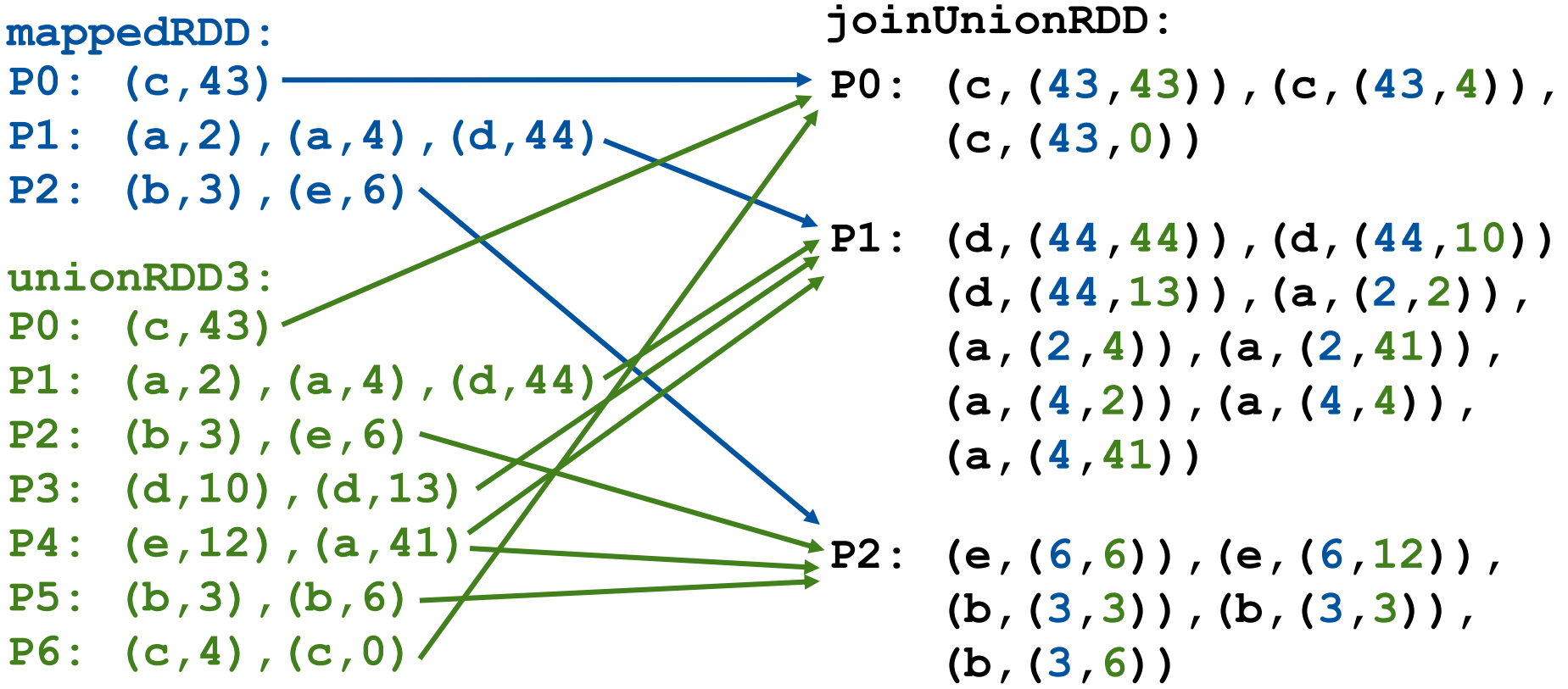


joinRDD has narrow dependencies on both **mappedRDD** and **RDD2**.

Task 3.2 (6)

Transformation: `joinUnionRDD = mappedRDD.join(unionRDD3)`

Note: `unionRDD3` has no partitioner, but `mappedRDD` has.

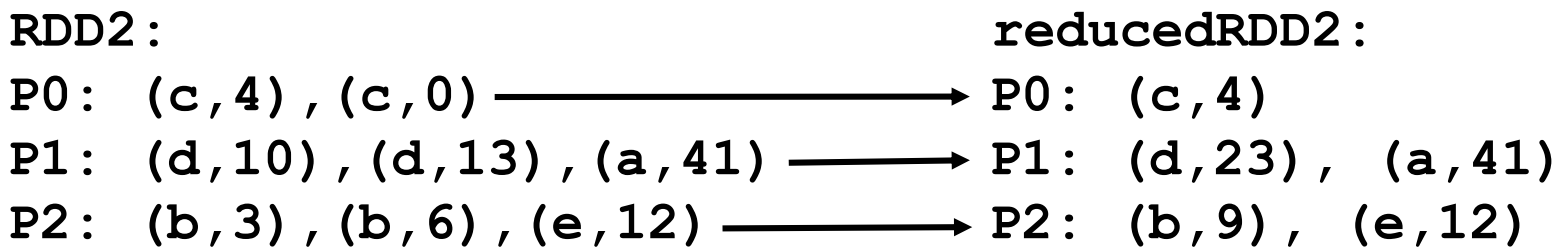


`joinUnionRDD` has narrow dependency on `mappedRDD` and wide dependency on `unionRDD3`.

Task 3.2 (7)

Transformation:

```
reducedRDD2 = RDD2.reduceByKey((a, b) -> a + b)
```



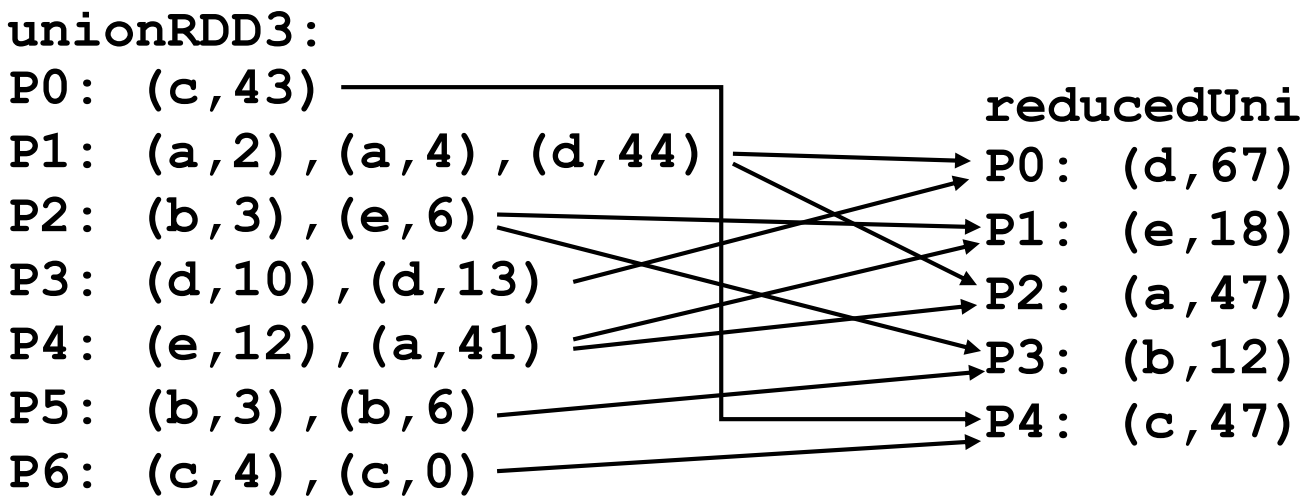
reduceRDD2 has a narrow dependency on **RDD2**.

Task 3.2 (8)

Transformation:

```
reducedUnionRDD3 = unionRDD3.reduceByKey((a, b) -> a + b)
```

Note: **`reduceByKey`** partitions the reduced KV-pairs implicitly. Assumption here: We have one partition for each reduced KV-pair and the hash function is h_5.

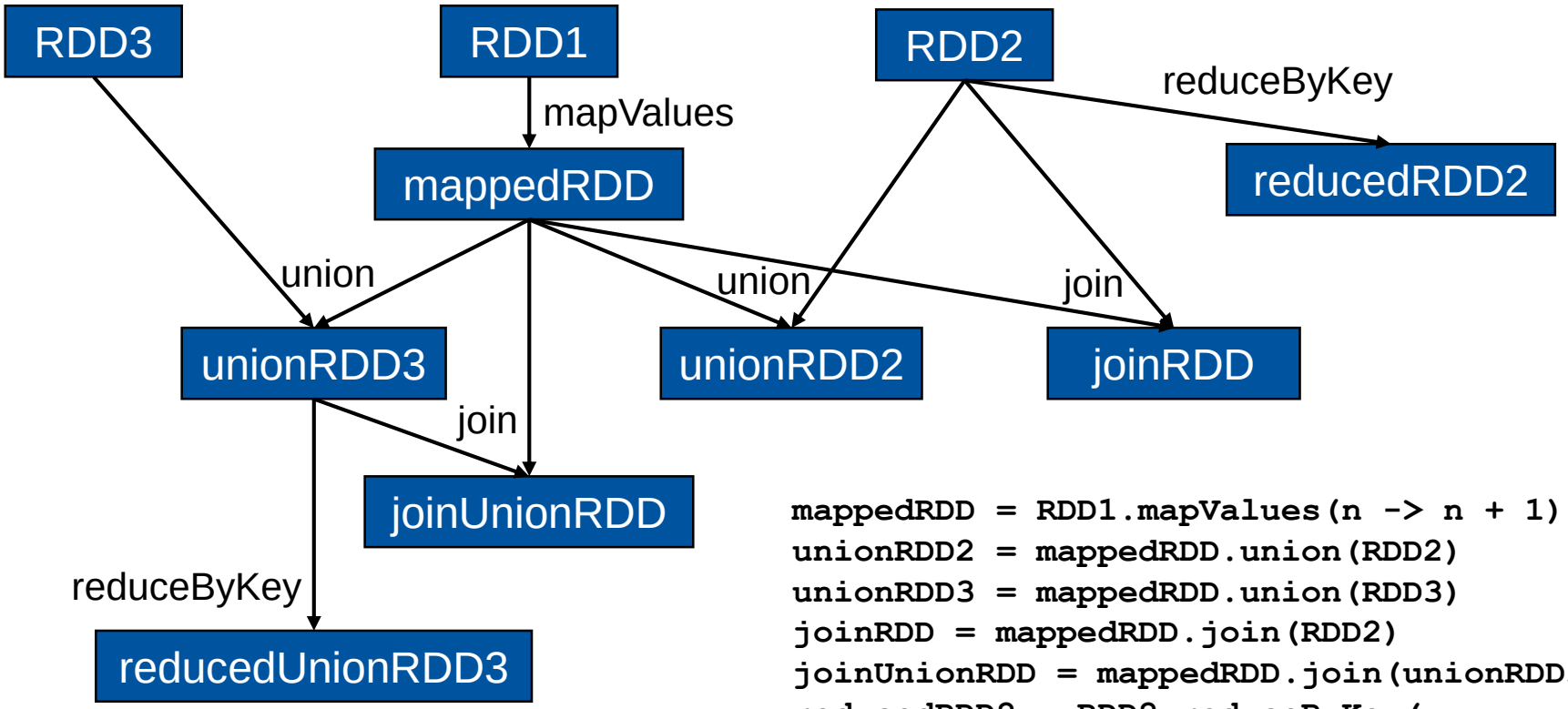


Char	Dec	$h_5(n)$
a	97	2
b	98	3
c	99	4
d	100	0
e	101	1

`reduceUnionRDD3` has a wide dependency on **`unionRDD3`**.

Task 3.3

- Draw the lineage graph of the defined transformations in Task 2.2.




```
mappedRDD = RDD1.mapValues(n -> n + 1)
unionRDD2 = mappedRDD.union(RDD2)
unionRDD3 = mappedRDD.union(RDD3)
joinRDD = mappedRDD.join(RDD2)
joinUnionRDD = mappedRDD.join(unionRDD3)
reducedRDD2 = RDD2.reduceByKey(
                        (a, b) -> a + b)
reducedUnionRDD3 = unionRDD3.reduceByKey(
                        (a, b) -> a + b)
```

Appendix: Running RDD Transformations in the Spark Shell

- You can try out the RDD transformations yourself on our cluster.
- Login on any arbitrary frontend node of the cluster (https://doc.itc.rwth-aachen.de/display/CC/Access)
- If you have not yet sent your TIM-ID to contact@hpc.rwth-aachen.de to get access to the lecture group on the cluster, please do so.
- Run the following commands to set up the environment and start the shell:
  - `module use /home/lect0034/modules`
  - `module load spark`
  - `spark-shell`

Appendix: Running RDD Transformations in the Spark Shell

- In the Spark shell, set up the initial RDDs and partitioners by running the following command:
  - `:load /home/lect0034/spark-exercise/rdd_exercise.scala`
- If you also want to load directly all resulting RDDs of the exercise, then run the following command:
  - `:load /home/lect0034/spark-exercise/rdd_transformations.scala`
- Print the content of an arbitrary RDD using the following command:
  - `printRDD(myRDD)`
- Get the partitioner of an RDD:
  - `myRDD.partitioner`

Appendix: Running RDD Transformations in the Spark Shell

- Print the dependencies of an RDD:
  - `printDeps(myRDD)`
- Quit the shell:
  - `:quit`
- Note: Compared to the Java syntax used in the exercise, you have to replace in the commands the arrow "**->**" with "**=>**".

Task 4: Using a Hadoop Cluster

Task 4.1

Live Demo

Task 4.1

1. Examine the effect of the combiner: How much network I/O and time is saved compared to a run without combiner?
2. In case of a run with combiner: Why is the total number of input records of the combiner larger than the total number of map output records?
3. Why is there (most probably) a number of killed map and reduce tasks shown in the statistics?

Task 4.1

1. Examine the effect of the combiner: How much network I/O (in the shuffle step) and time is saved compared to a run without combiner?

Job Statistics: 200 GB	With Combiner	Without Combiner
Map input records	18,119,633	18,119,633
Map output records	20,196,741,706	20,196,741,706
Map output bytes	304,466,887,520	304,466,887,520
Map output materialized bytes	14,209,779,175 ~ 13.23 GB	344,864,690,932 ~ 333.9 GB
Combine input records	20,692,020,928	0
Combine output records	947,761,943	0
Reduce input groups	104,702,349	104,702,349
Reduce shuffle bytes	14,209,779,175 ~ 13.23 GB	344,864,690,932 ~ 333.9 GB
Reduce input records	452,482,721	20,196,741,706
Reduce output records	104,702,349	104,702,349
Spilled Records	1,400,244,664	60,068,841,090

~300 GB network I/O saved due to combiner (factor of ~25).

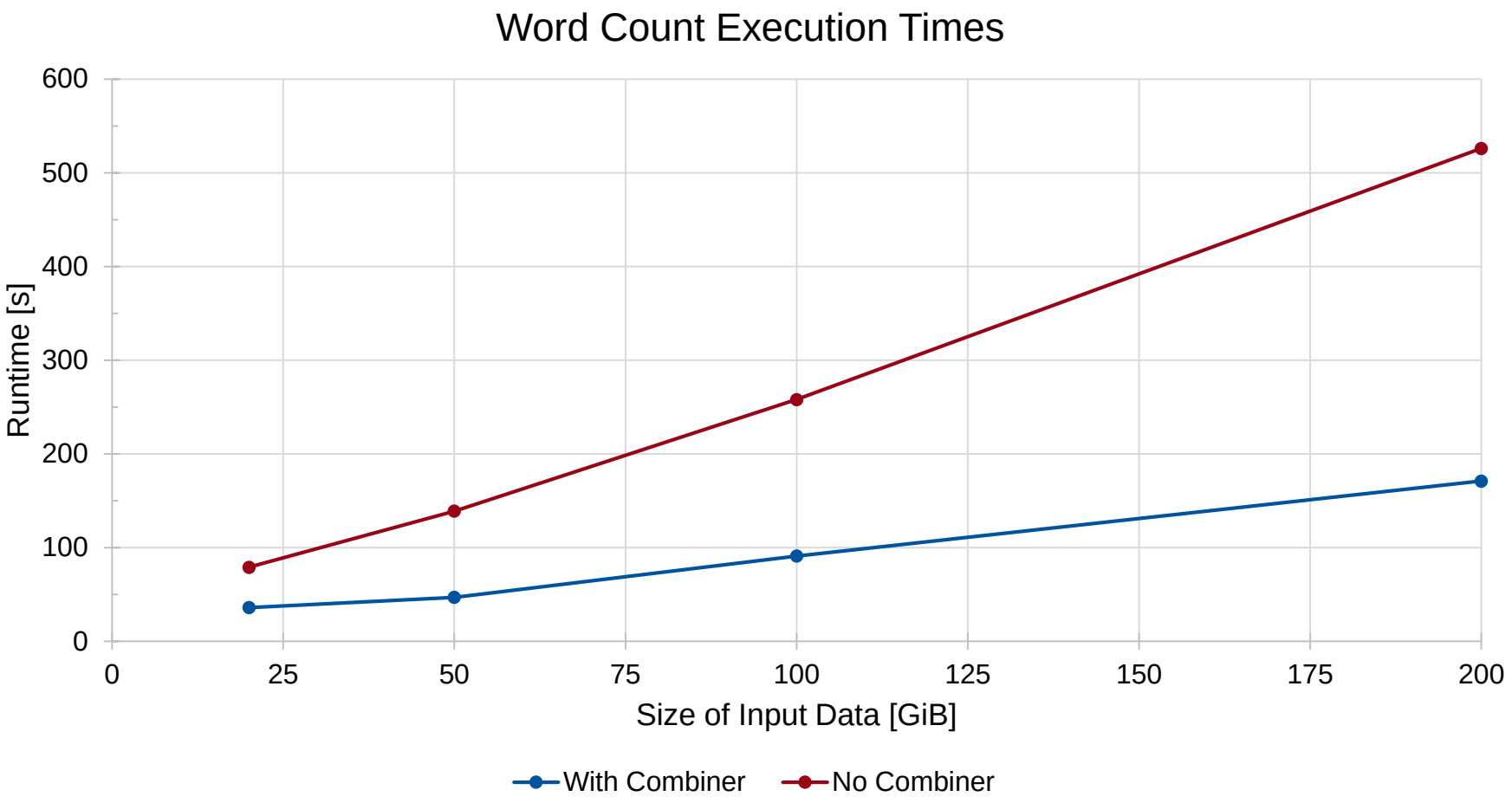
Task 4.1

1. Examine the effect of the combiner: How much network I/O (in the shuffle step) and time is saved compared to a run without combiner?

Job Statistics: 200 GB	With Combiner	Without Combiner
Killed map tasks	1	1
Killed reduce tasks	1	10
Launched map tasks	1801	1801
Launched reduce tasks	400	410
Data-local map tasks	1800	1796
Rack-local map tasks	1	5
Total time spent by all map tasks (s)	3618s	4752s
Total time spent by all reduce tasks (s)	1626s	6490s
Wall-clock time complete job (s)	171s	526s

- **1134 seconds saved (factor ~ 1.31) in total time of map tasks**
- **4864 seconds saved (factor ~3.99) in total time of reduce tasks**
- **355 seconds saved (factor ~3.07) in wall-clock time**

Task 4.1 – Scalability



Setup: 11 CLAIX2018 MPI nodes (1 master, 10 workers), SSD Local Storage, Hadoop 2.8.5

Task 4.1

2. In case of a run with combiner: Why is the total number of input records of the combiner larger than the total number of map output records (the same also holds for Combine output records and Reduce input records)?

Job Statistics: 200 GB	With Combiner	Without Combiner
Map input records	18,119,633	18,119,633
Map output records	20,196,741,706	20,196,741,706
Combine input records	20,692,020,928	0
Combine output records	947,761,943	0
Reduce input records	452,482,721	20,196,741,706
Reduce output records	104,702,349	104,702,349

Short Answer:

- Combiner can run in multiple stages on the Map output records.

Long Answer:

- See next slide

Task 4.1 – Long Answer: Combiner Invocations

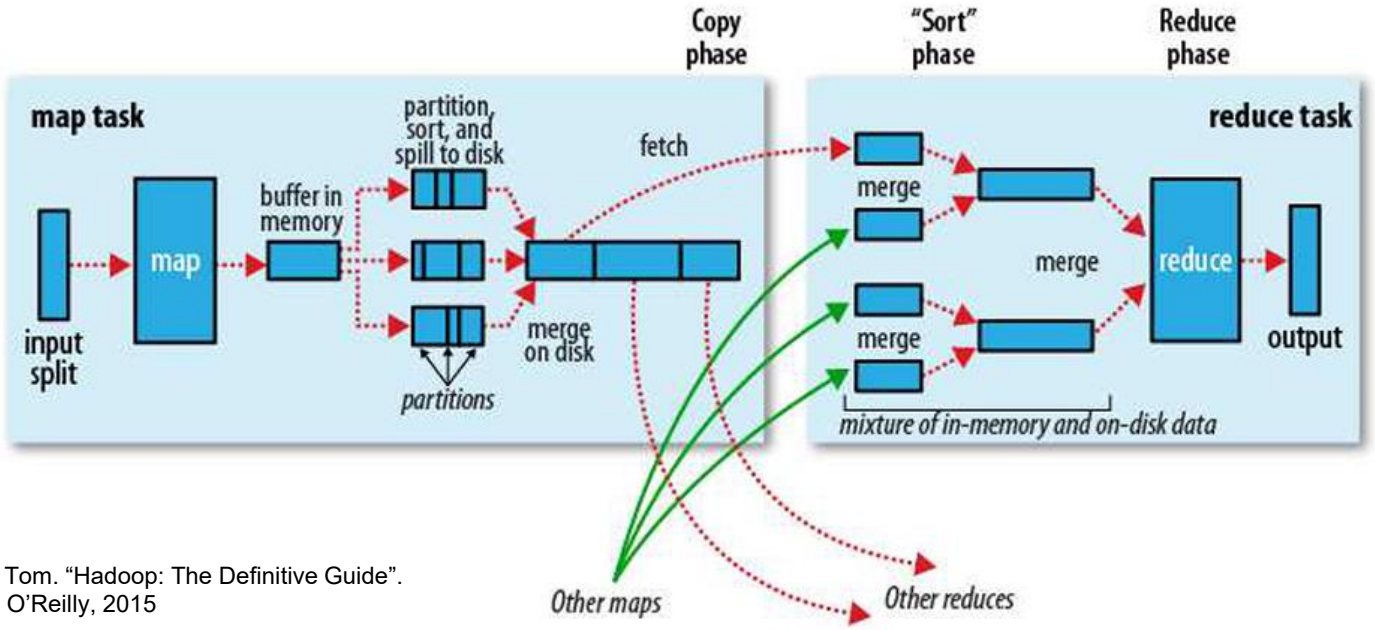


Illustration: White, Tom. "Hadoop: The Definitive Guide". 4th Edition, p.208, O'Reilly, 2015

- After "partition" and "sort", before "spill to disk": Combiner invocation
- "Merge on disk" step: All spills on disk are merged, combiner is invoked again to combine *across spills*.
- In other words: We potentially run the combiner on already combined data in the "merge on disk" step.

Task 4.1

3. Why is there (most probably) a number of killed map and reduce tasks shown in the statistics?

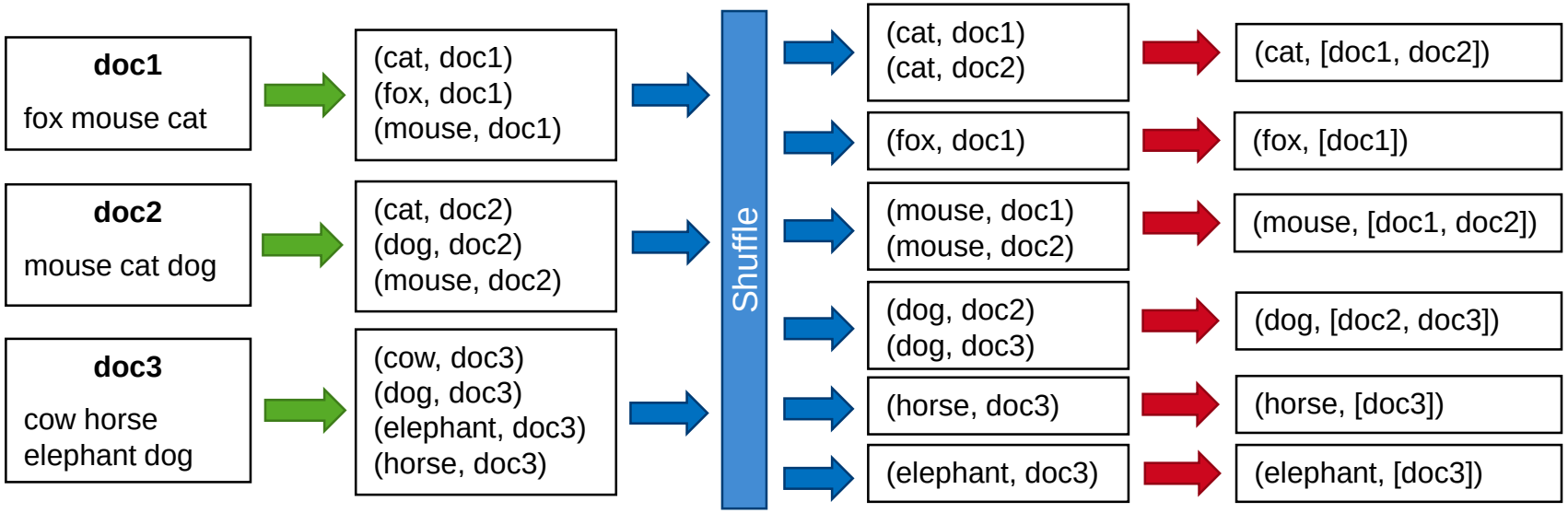
Job Statistics	With Combiner	Without Combiner
Killed map tasks	1	1
Killed reduce tasks	1	10
Launched map tasks	1801	1801
Launched reduce tasks	400	410

Answer:

- Speculative execution
  - Backup task or original task might be killed
  - See lecture slides MapReduce II, 32 / 33 for more details

Task 4.2: Inverted Index

- Inverted index: Lookup all matching documents for a given search word
  - *Map*: Parse each document, emit sequence of $(word, document\ ID)$
  - *Reduce*: For all pairs for a given word emit $(word, list(document\ ID))$
  - Result: Given some search word, all matching documents can be returned



Task 4.2: Inverted Index

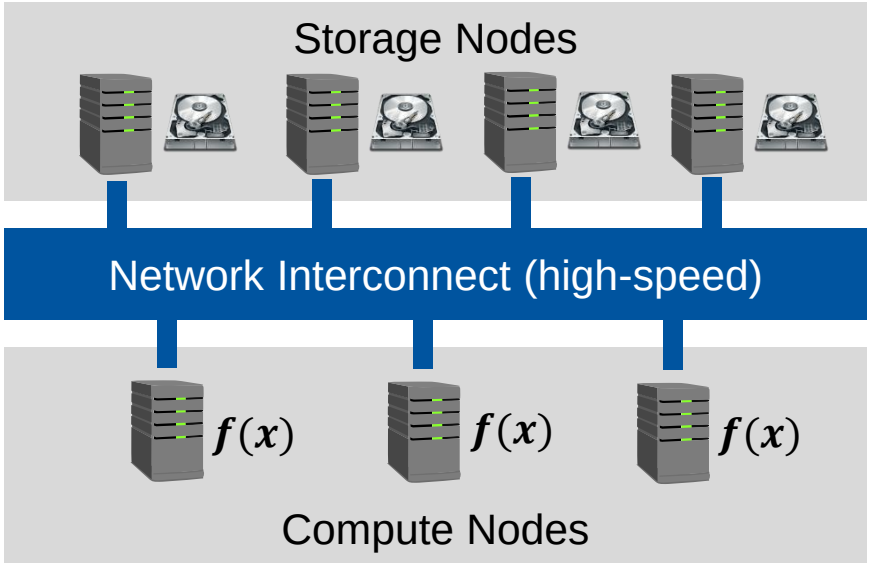
Live Demo

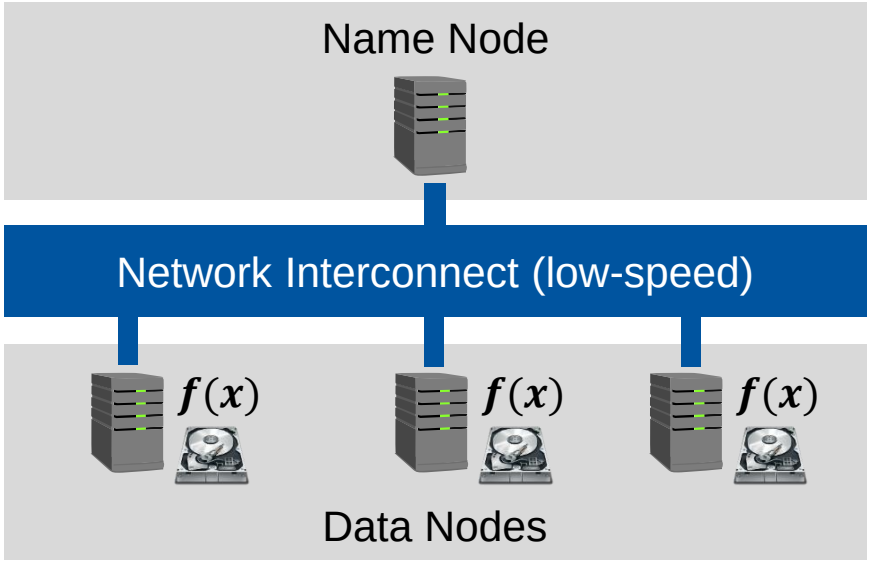
(see Moodle course room for code solution)

Task 4.3

- We discussed **architectural differences** in the file system of a supercomputer and a big data cluster in the lecture.
- However: It is possible that **HDFS data nodes do not have any local storage**, but instead **write “local” data to the parallel file system Lustre**.
- Explain how this affects the data locality feature of HDFS and the data accesses in the three phases (map, shuffle, reduce) of the MapReduce computation.

Task 4.3 – File System Comparison





Lustre, GPFS (Supercomputers)

- Separate storage and computation
- Compute nodes retrieve data from storage nodes via interconnect
- High-speed interconnect efficient even for data-intensive jobs

HDFS (Big Data Clusters)

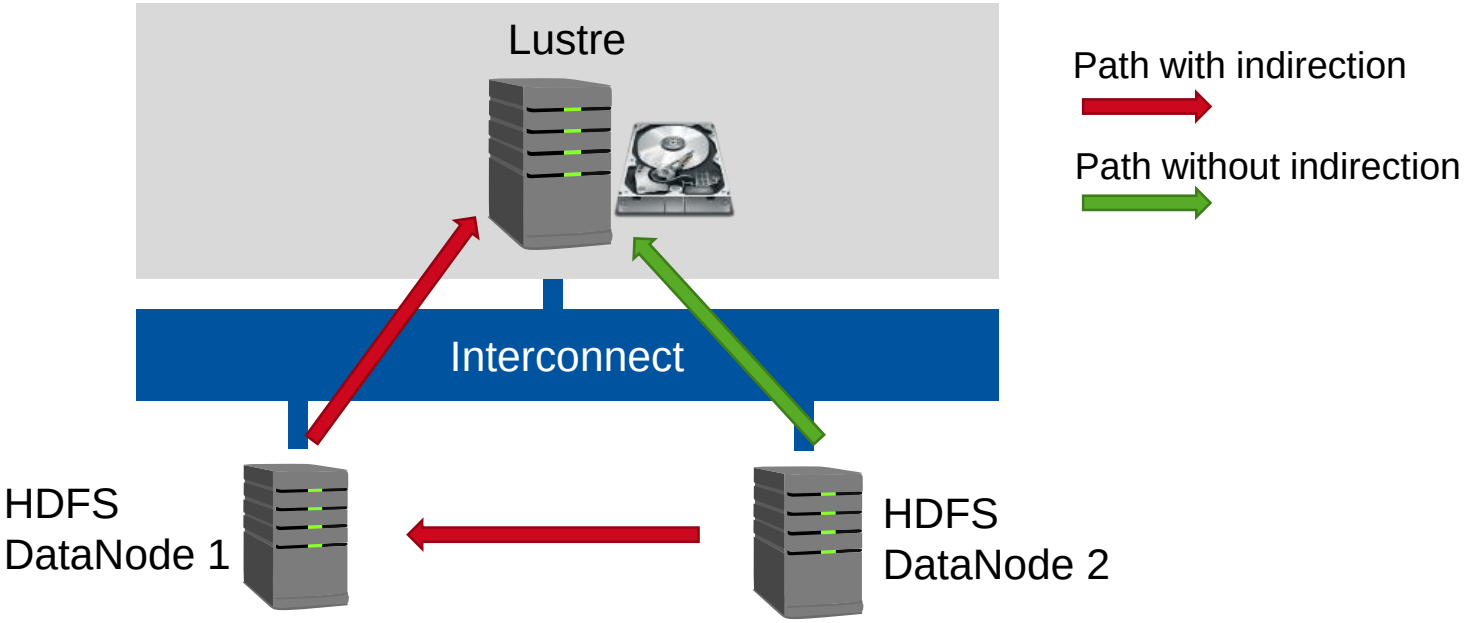
- Focus on data locality
- Each data node: Locally attached storage
- Datasets distributed among nodes
- Job assignments consider data locality

Task 4.3

- HDFS on top of Lustre
  - HDFS assumes that nodes have locally attached storage.
  - However: There is no local storage, all data accesses lead to network I/O.
  - No data locality possible.
  - Even worse: HDFS adds useless indirections in the data accesses.
- Map phase
  - Actual idea: Map tasks work on local data (if possible).
  - Here: Map tasks effectively read data from Lustre → High network I/O
- Shuffle phase
  - Outputs of the Map tasks are written back to Lustre. → High network I/O
  - Reduce tasks copy their required data from the nodes that performed the Map tasks.
  - Problem: Reduce tasks assume that only the node that computed output is in possession of data.

Task 4.3

- But: In Lustre, output data is globally available for all nodes.
- However: Reduce tasks will ask machine which executed Map task for the output data, which itself will retrieve the data from Lustre.
- Result: Useless indirection, because the reduce task could read data directly from Lustre.
- Example: DataNode 2 wants to get outputs of DataNode 1



Task 4.3

- Reduce phase
  - Results are written directly to Lustre instead of local storage
  - Distribution of the result is also needed in case of a usual HDFS setup (returning result, HDFS replication, …)
  - No effects (performance degradations, indirections) for this phase