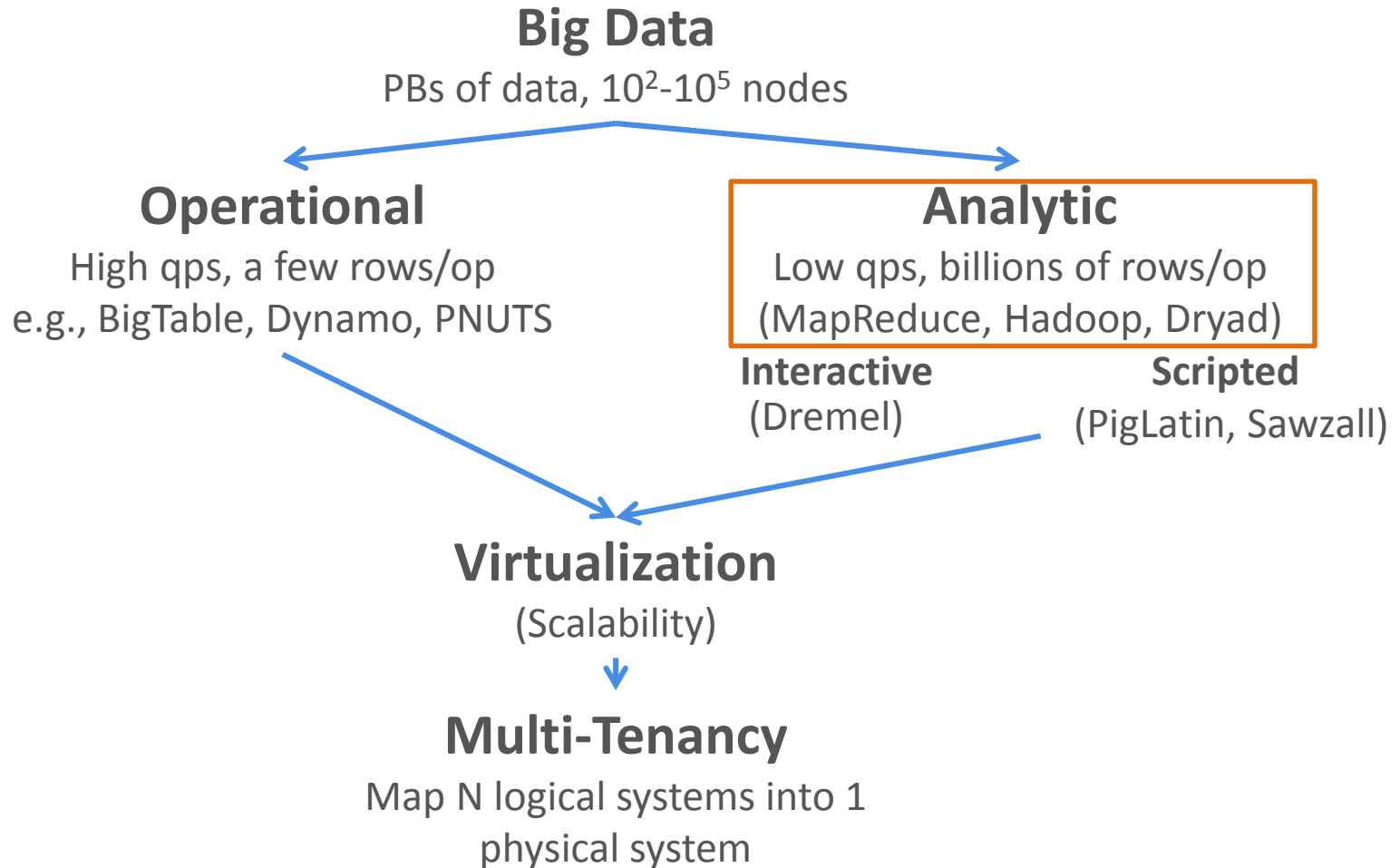




2 Map/Reduce Programming Model



[S. Melnik: The Frontiers of Data Programmability, BTW 2009]



- ◆ MOTIVATION
- ◆ DER MAPREDUCE-ANSATZ
 - ❖ Map- und Reduce-Operationen
 - ❖ Googles Implementierungskonzept
 - ❖ Fehlertoleranz
 - ❖ Beispiele
 - ❖ Implementierungen
- ◆ HADOOP
- ◆ ANSÄTZE FÜR WEITERE SPRACHABSTRAKTIONEN
 - ❖ Sawzall, Dryad, Pig, Dremel
- ◆ KRITIK AN MAPREDUCE
 - ❖ Vergleich zwischen Mapreduce und DBMS
 - ❖ HadoopDB
- ◆ AUSBLICK



Problem: Petabytes an Daten, auf mehrere Computer verteilt, müssen verarbeitet oder analysiert werden

- Daten sind zu groß, um sie auf einer einzigen Maschine sequenziell zu bearbeiten
- Beispiel: Wie oft kommen welche Wörter in Textdateien (z.B. Suchlogs) vor?
- Eigentliche Funktionalität (Wörter zählen) trivial implementierbar
- Aber wie verteilen?
- MapReduce
 - Programmiermodell, das bei der Parallelisierung datenintensiver Anwendungen zum Einsatz kommt
 - Abstrahiert den Verteilungsvorgang (Parallelisierung, Verteilung von Daten, Fehlertoleranz)



Vorteil

- Entwickler von verteilten Anwendungen müssen nur das Framework benutzen, keine Codeänderungen bei Änderung der Client-Anzahl nötig
- Verwendung von handelsüblichen Rechnern möglich
- Keine Notwendigkeit für spezielle High-End Server

Anwendungsgebiete

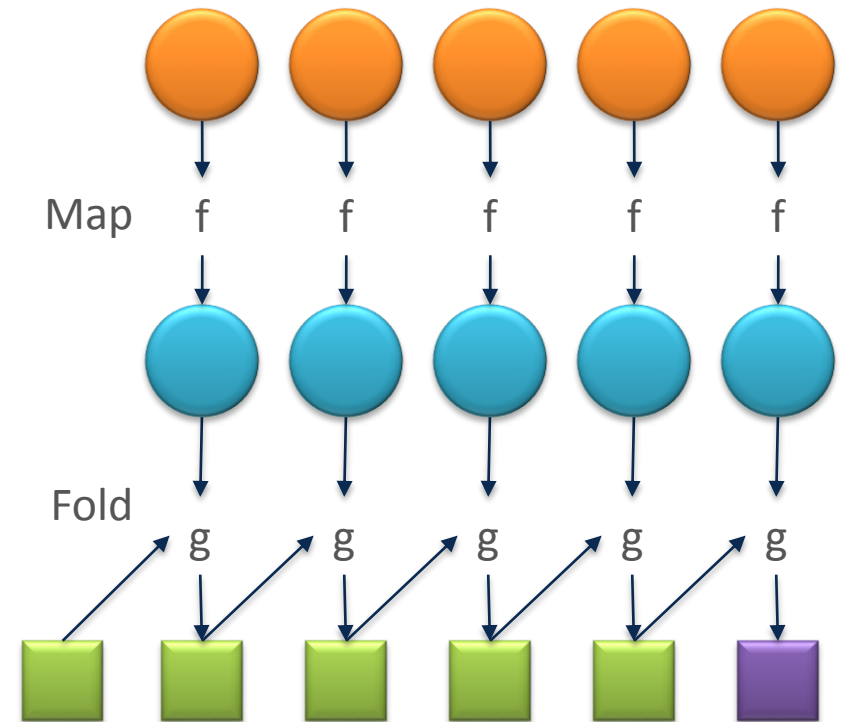
- Bildung verschiedener Statistiken für Web-Seiten
- Invertierter Web-Link Graph (welche Seiten zeigen auf eine bestimmte Seite?)
- Bildung von invertierten Indizes (in welchen Dokumenten kommt ein bestimmtes Wort vor?)
- Verteiltes Sortieren
- Cluster-Algorithmen
- Verarbeitung von Satelliten-Bildern
- Sprachverarbeitung für automatische Übersetzung
- Graphentheorie



Ziel: Verstecke Komplexität der parallelen Programmierung, Datenverteilung, Fehlertoleranz vor dem Entwickler

MapReduce-Ansatz

- Bietet dem Entwickler eine Schnittstelle mit nur zwei Operationen an, die er selbst definieren muss: Map und Reduce
- Inspiriert von Map/Reduce aus funktionalen Sprachen
- Zwei Schritte
 - Map: Generiert aus Eingabedaten eine Sammlung von Zwischenergebnissen
 - Reduce: Generiert daraus aggregierte Ergebnisse





Map erzeugt als Zwischenergebnis eine Sammlung von (Schlüssel, Wert)-Paaren

Beispiel: Zähle Wörter in einem Dokument

```
map(dokumentName, wert) {  
    for each word w in wert  
        emit(w, "1");  
}
```

■ Textinhalt des Dokuments

■ Emittiert für jedes Wort ein Paar, welches das Wort und die Häufigkeit 1 enthält. Für dasselbe Wort, das mehr als einmal vorkommt, werden mehrere Paare erzeugt

- Anmerkung: Die Eingabe (dokumentName, wert) ist auch ein Paar, jedoch haben dessen Elemente einen anderen Definitionsbereich als die Elemente der emittierten Paare



Reduce kombiniert Zwischenergebnisse

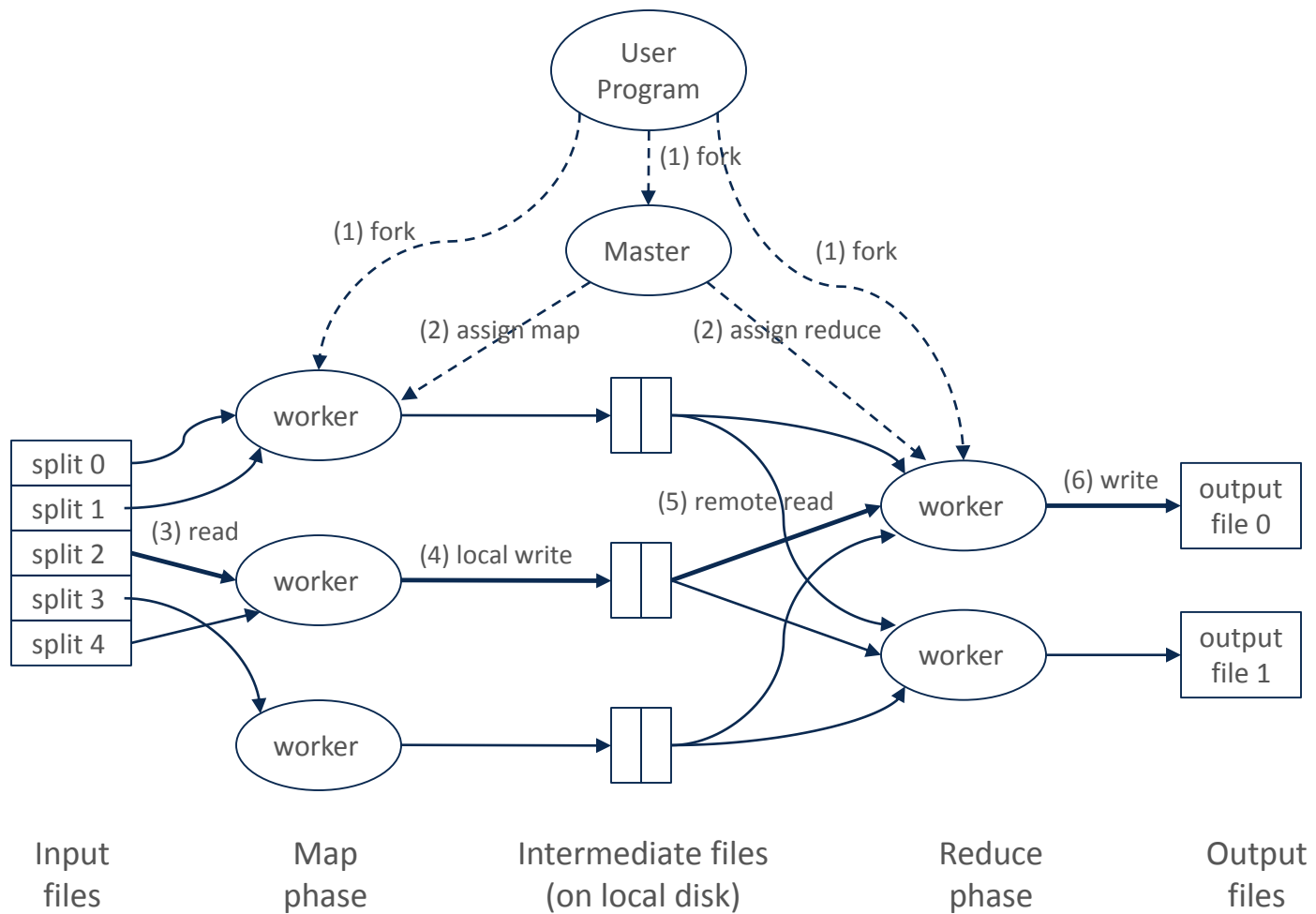
- Beispiel (fortgesetzt)

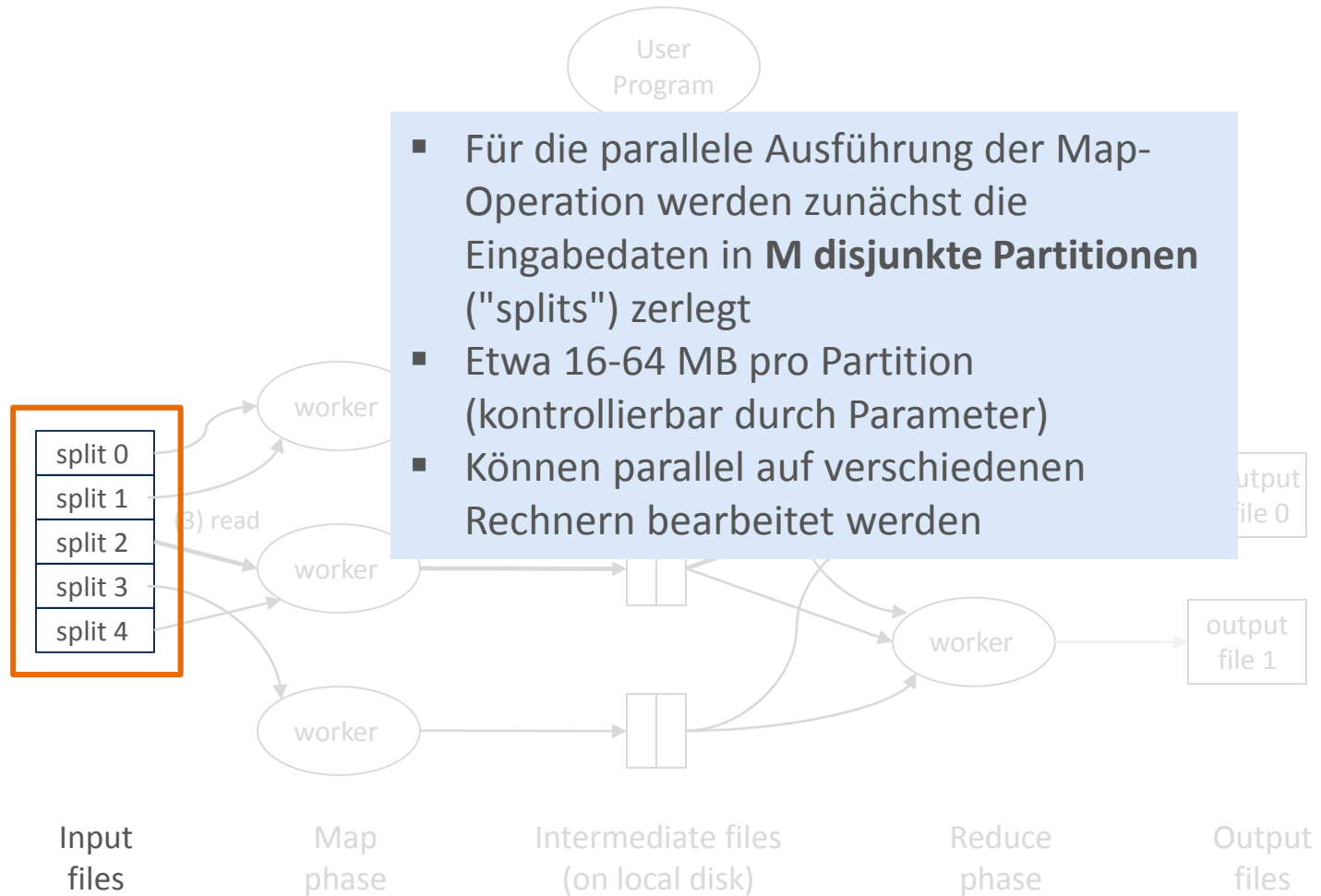
```
reduce(wort, werte) {  
    int ergebnis = 0;  
    for each w in werte  
        ergebnis += w;  
    emit(wort, ergebnis);  
}
```

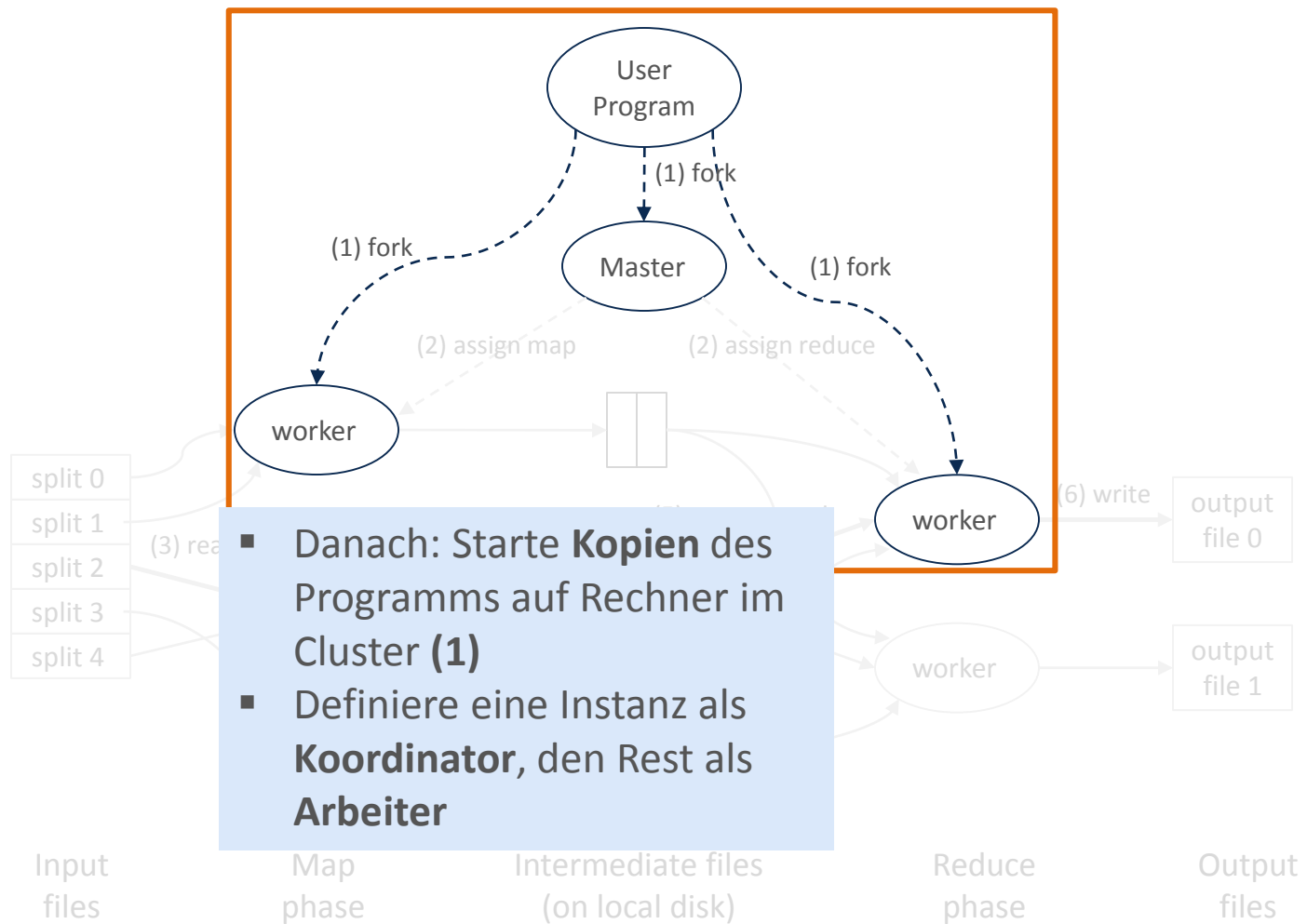
- Liste mit Worthäufigkeiten: Entsteht durch Aufsammeln und Gruppieren mehrerer Paare, die dasselbe Wort als Schlüssel haben (Shuffle)
- Wird der Reduce-Operation als Iterator übergeben. Auf diese Weise sind große Datenmengen handhabbar, die sonst nicht vollständig in den Speicher passen würden

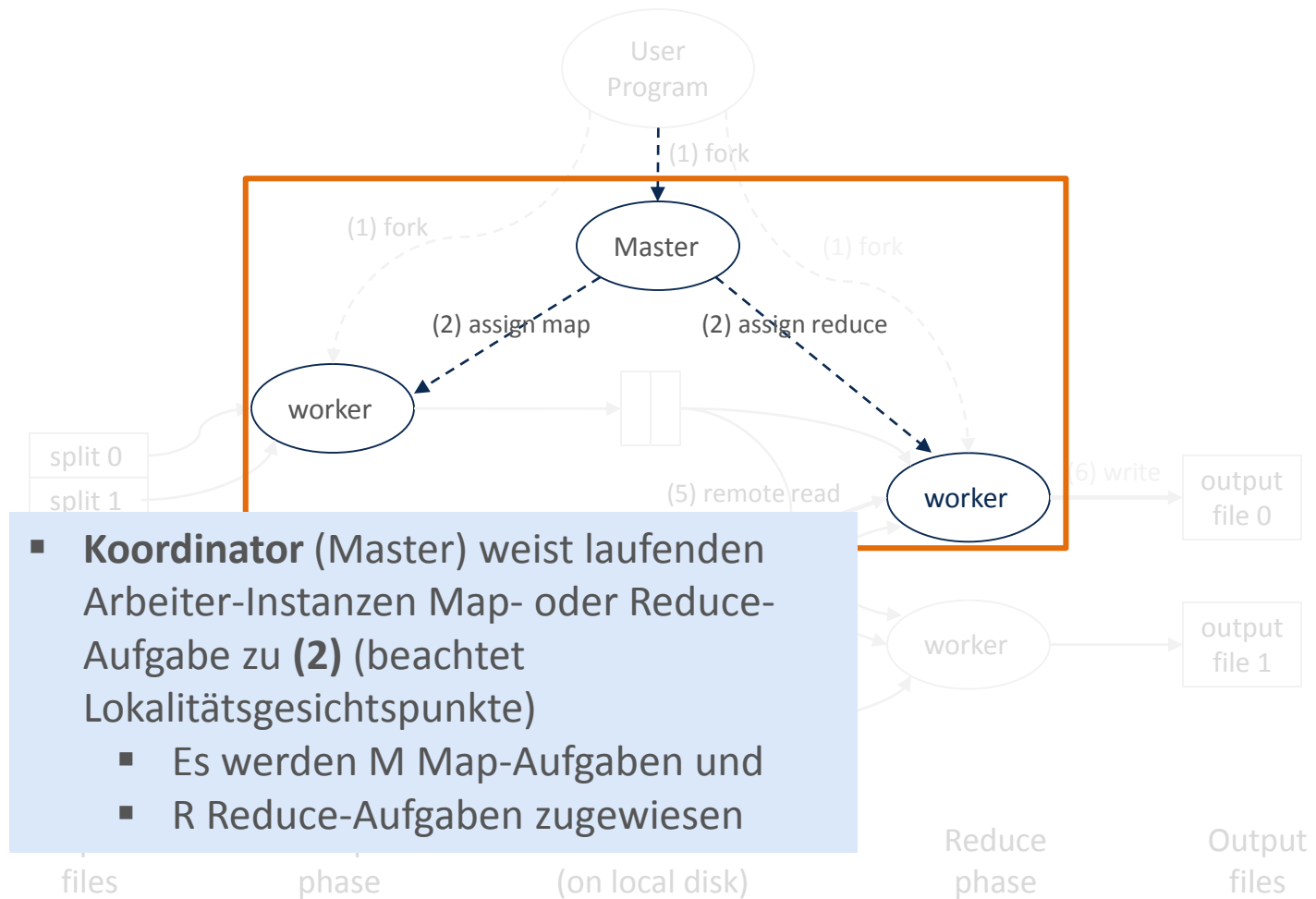
Summiere die Häufigkeiten aus der Werteliste auf und emittiere das Ergebnis

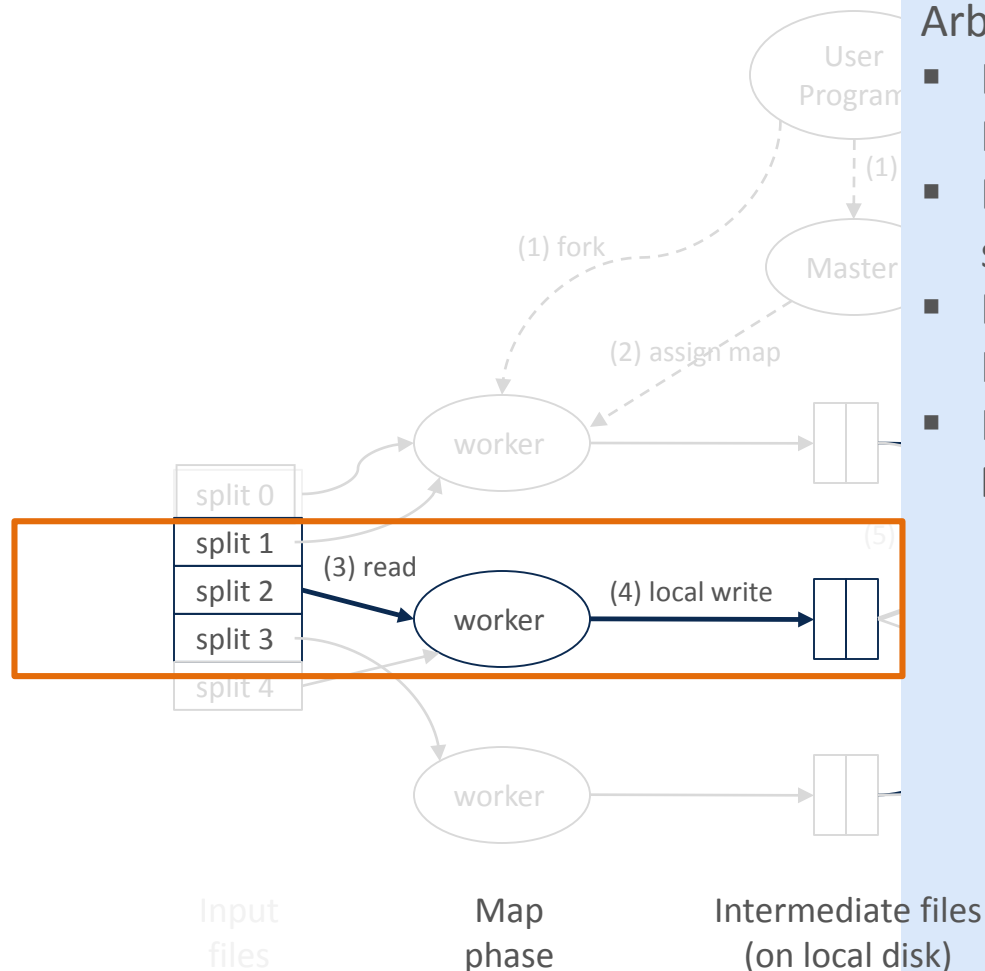
- Anmerkung: Elemente der Eingabe-Paars haben denselben Definitionsbereich wie die des Ausgabe-Paars (Yahoo's Hadoop ist hier etwas großzügiger.)
- Es kann mehrere Arbeiter geben, welche die Reduktionsoperation ausführen und selbst wieder Zwischenergebnisse emittieren











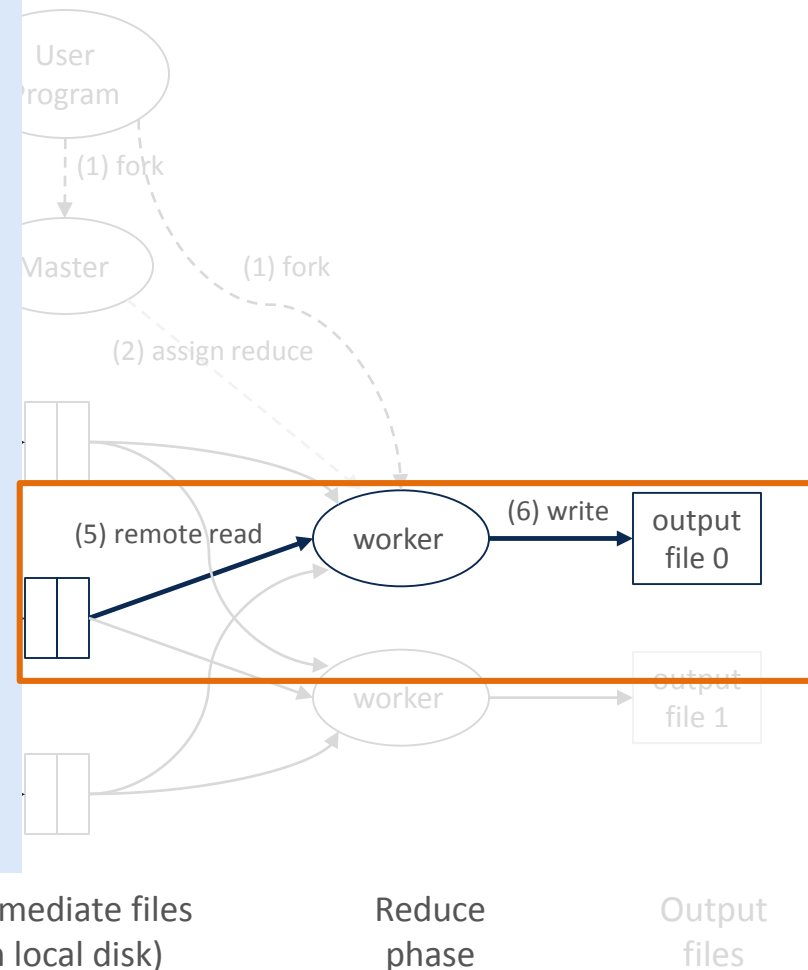
Arbeiter mit Map-Aufgabe

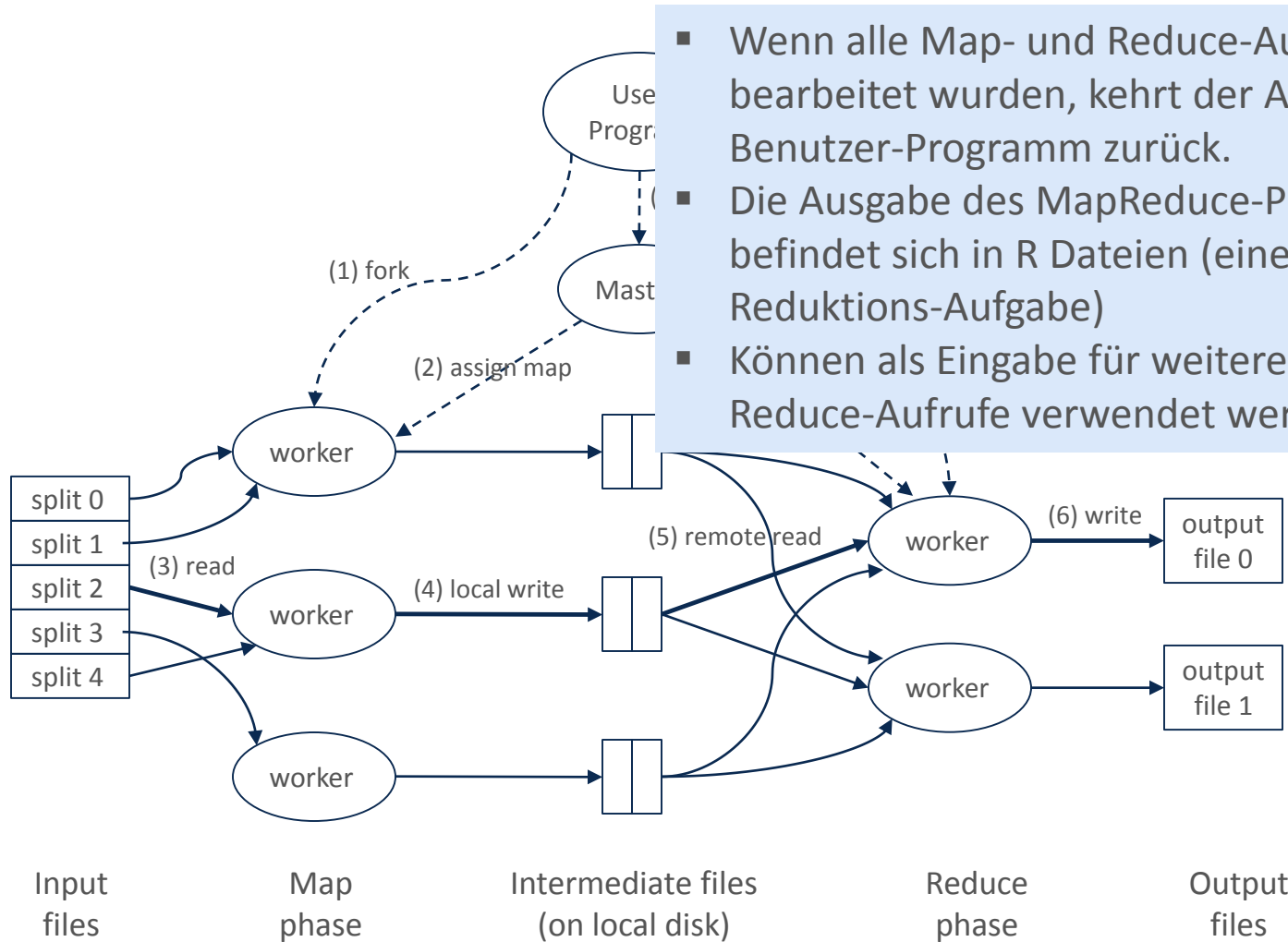
- Liest Inhalte seiner zugewiesenen Partition **(3)**
- Bearbeitet alle **Eingabepaare** mit seiner Map-Funktion
- Die **Ergebnisse** werden im Hauptspeicher gepuffert
- In periodischen Abständen werden **Puffer** auf Festplatte geschrieben **(4)**
 - Anhand einer Partitionierungsfunktion in **R Partitionen** unterteilt (z.B. $\text{hash}(\text{key}) \bmod R$)
 - Der **Ablageort** der Partitionen wird dem **Koordinator gemeldet**, der für die Übergabe an Reduce- Arbeiter verantwortlich ist



Arbeiter mit Reduce-Aufgabe

- Wird vom Koordinator **benachrichtigt**
- Bekommt Ablageort für Zwischenergebnisse (Partitionen), die er von den Festplatten der Worker liest **(5)**
- Shuffle: Sortierung der Paare nach Schlüssel, so dass Paare mit gleichem Schlüssel gruppiert werden
 - Arbeiter iteriert darüber
 - Jeder Schlüssel wird samt der zugehörigen Werte der Reduktionsoperation übergeben
- Das Ergebnis der Reduktionsfunktion wird in einer Ausgabedatei angefügt, die zur Partition assoziiert ist **(6)**





- Wenn alle Map- und Reduce-Aufgaben bearbeitet wurden, kehrt der Aufruf zum Benutzer-Programm zurück.
- Die Ausgabe des MapReduce-Programms befindet sich in R Dateien (eine pro Reduktions-Aufgabe)
- Können als Eingabe für weitere Map-Reduce-Aufrufe verwendet werden



Arbeiter

- Werden **periodisch abgefragt**, ob noch funktionsfähig
- Keine Antwort: Kennzeichnung als nicht funktionsfähig
- Zugewiesene Aufgaben werden vom Koordinator neu verteilt
- Auch erfolgreich beendete Map-Aufgaben werden neu verteilt, da deren Ausgabe auf lokaler Festplatte war
- Reduce-Arbeiter werden über Änderung benachrichtigt

Koordinator

- Zustand wird regelmäßig gesichert („**Checkpointing**“)
- Bei Versagen: Neustart und Wiederherstellung des letzten gesicherten Zustands

Fehlertoleranz ist der zentrale Vorteil von MapReduce!

- Es geht bei MapReduce nicht nur darum, Rechenaufwand zu parallelisieren.



Wie gelangen die Daten zu den Workers?

Verschiebe nicht die Daten zu den Workers, sondern die Workers zu den Daten!

- Daten sind auf den lokalen Festplatten der einzelnen Knoten im Cluster gespeichert
- Starten der Workers an welchen die Daten lokal vorliegen

Daten in einem verteilten Dateisystem gespeichert

- GFS (Google File System) für Googles MapReduce
- HDFS (Hadoop Distributed File System) für Hadoop

Googles MapReduce

- Map speichert Ergebnis im lokalen Dateisystem.
- Reduce holt sich Daten via RPC; schreibt Ergebnis in verteiltes Dateisystem (GFS).



Wetterdaten



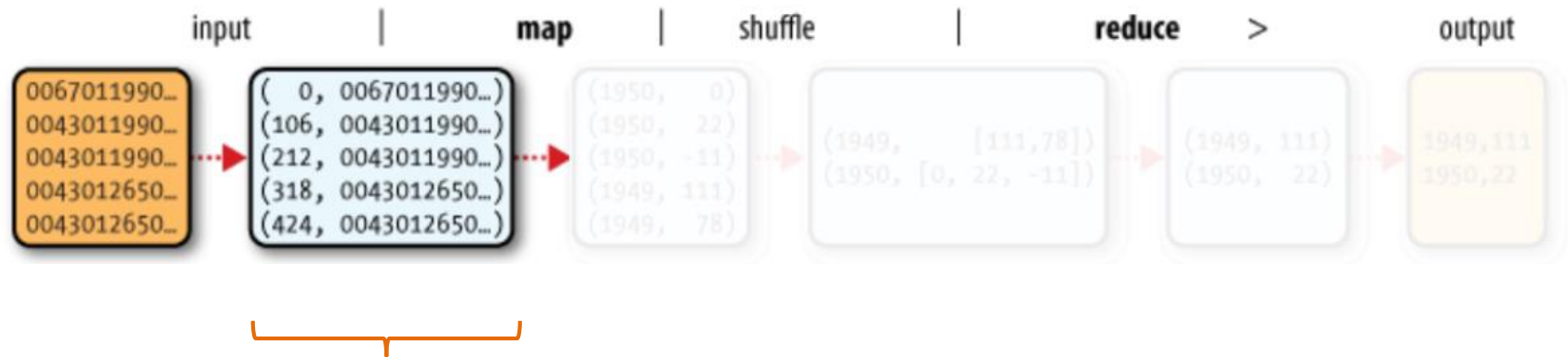
1. Unstrukturierte Wetterdaten einlesen

- 00290290709999991901010106004+64333+023450FM-12+00059
9999V0202701N015919999999N0000001N9-00781+9999910200
1ADDGF1089919999999999999999999999
- 0029029070999999**190101011300**4+64333+023450FM-12+00
9999V0202901N008219999999N0000001N9-**0072**1+999991
1ADDGF1049919999999999999999999999

1901-01-01
13:00
-7,2°C



Wetterdaten

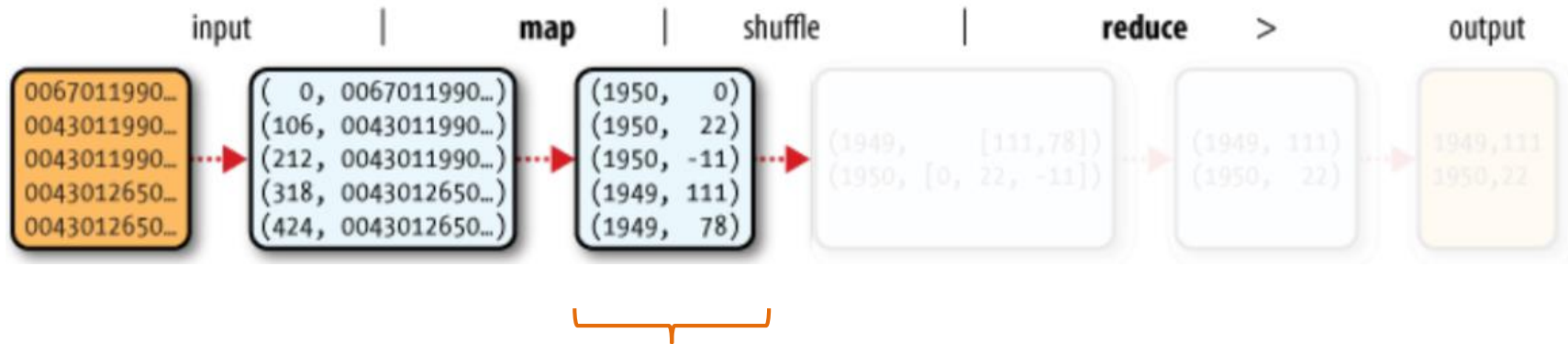


2. Zuordnung von Datei-Inhalt zu Positionen

- Jede Zeile wird anhand des Byte-Offsets identifiziert
- Byte-Offset verweist jeweils auf den Beginn der Zeile
- (k1, v1) = (long, String)



Wetterdaten



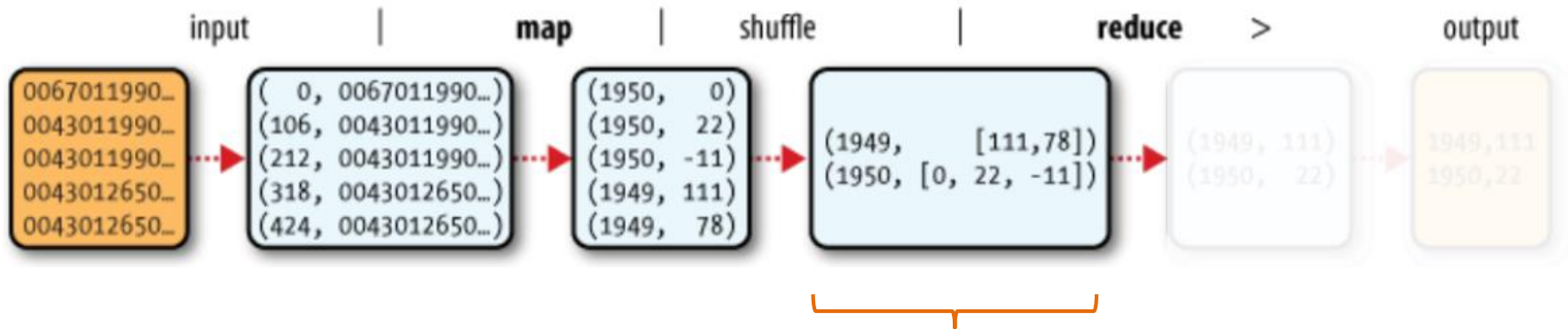
3. Map: Transformieren dieser Schlüssel-/Werte-Paare in Zwischen-Schlüssel-/Werte-Paare

- Benötigte Daten werden aus den Zeilen extrahiert
- Es entstehen viele Key/Value-Paare

Jahr	Temperatur
1950	0
1950	22
1949	111
1949	78



Wetterdaten



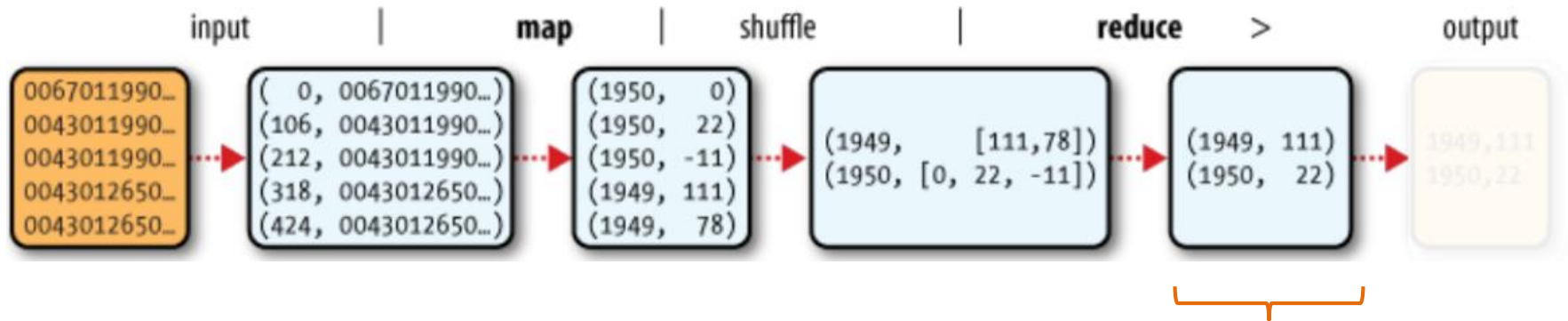
4. Shuffle: Erzeugen von **gruppierten** Schlüssel-/Werte-Paaren

- Sortieren der Schlüssel
- Zuordnen von Werten zu **einem** Schlüssel
- Jeder Mapper schreibt den sortierten Output ins Filesystem
- Pro Jahr wird ein eigener Reducer auf einem Rechner im Cluster ausgeführt

Jahr	Temperatur
1949	111
	78
1950	0
	22



Wetterdaten



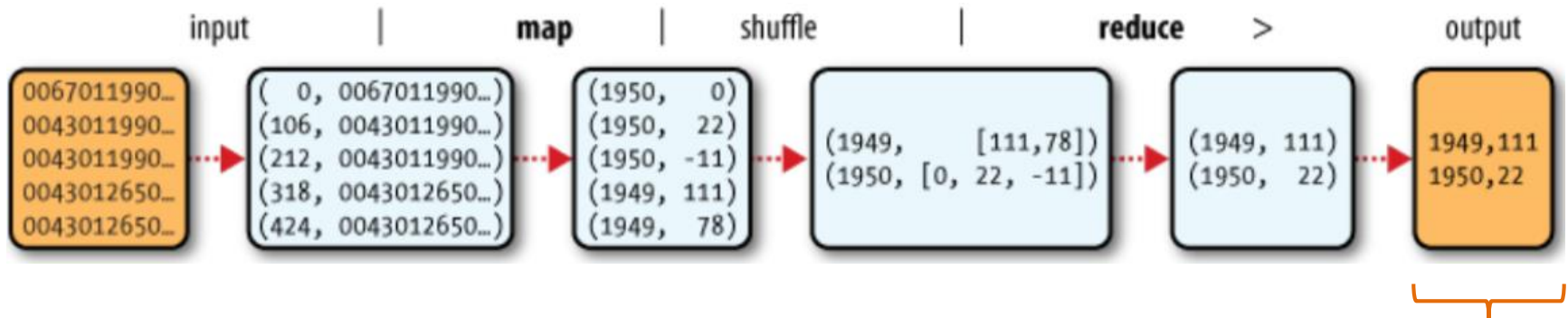
5. Reduce

- Zusammenfassung der Werte (hier: Maximum finden)
- Pro Schlüssel nur noch ein Wert

Jahr	Temperatur
1949	111
1950	22



Wetterdaten




6. Ausgabe in eine Datei



Beispiel aus Sprachverarbeitung in einem Übersetzungssystem

- Zähle, wie oft jede 5-Wörter-Sequenz in einem großen Korpus von Dokumenten vorkommt
- Map: Extrahiere (5-Wörter-Sequenz, Anzahl) von Dokument
- Reduce: Kombiniere Anzahl

Google Übersetzer

Von:  Nach:

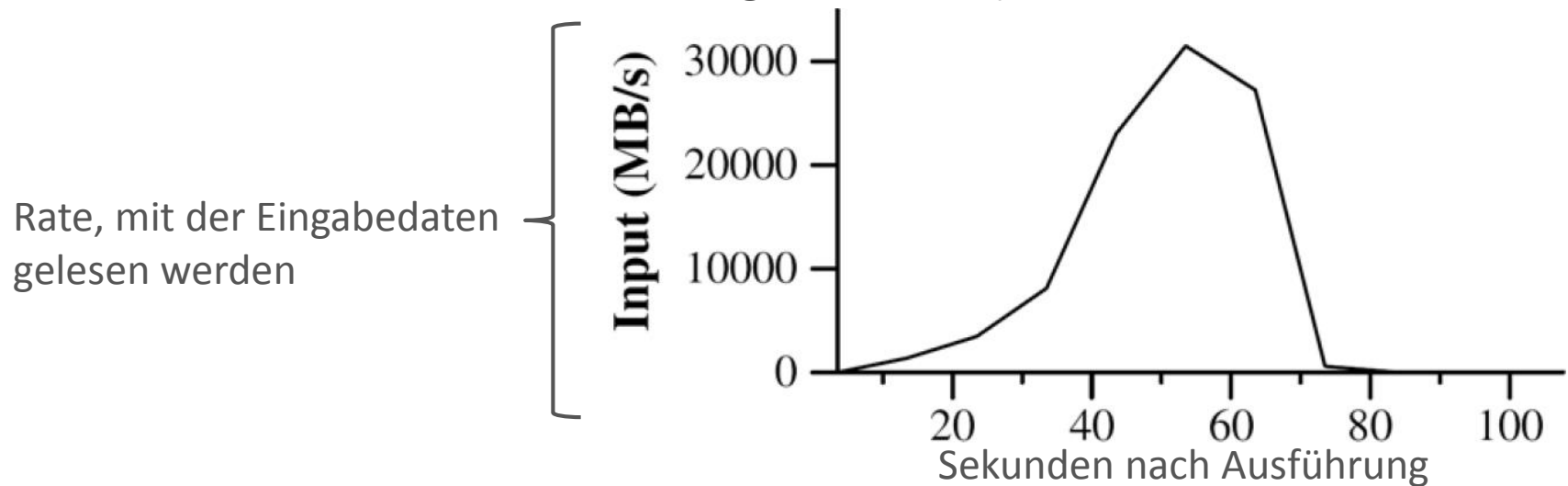
natural language processing

 Anhören



Performanz

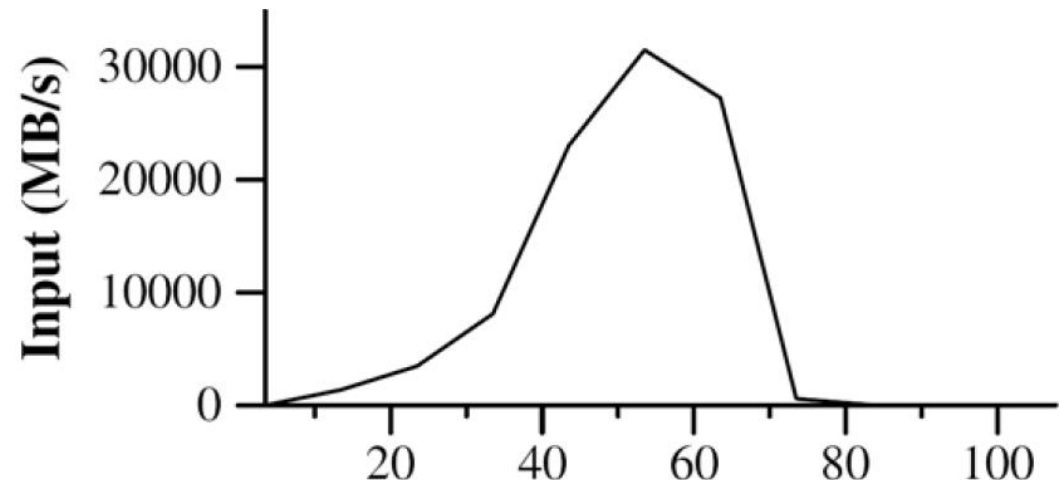
- Ausführung eines auf MapReduce basierenden **grep**-Programms (Global Regular Expression Print)
- Durchsucht 10^{10} 100-Byte lange Datensätze ($\sim 1\text{TB}$) nach einem seltenen, drei Zeichen langem Muster (nur in 92.337 Datensätzen vorhanden)
- Eingabe in $M=15.000$ Partitionen à 64MB unterteilt, Ausgabe in einer einzigen Partition ($R=1$)
- Datentransferrate im zeitlichen Verlauf (Cluster mit 1.800 Rechnern, mit je 2GHz Intel Xeon, 4GB RAM, 2x160GB HD, Gigabit Ethernet)





Performanz (Fortsetzung)

- Rate nimmt anfangs zu, wenn mehr und mehr Rechnern Arbeit zugewiesen wird
- Maximum bei ~30GB/s und 1.764 Arbeitern
- Danach beginnen Map-Aufgaben fertig zu werden
- Gesamtdauer der Ausführung etwa 150s; Mehraufwand beim Start ~1 Minute (Programm auf verschiedenen Rechnern starten, etc.)





Das MapReduce-Konzept ist nicht an eine bestimmte Rechnerarchitektur gebunden

- Die Implementierung der Schnittstelle kann z.B. auf Rechner mit verteiltem Speicher (Cluster) oder gemeinsamen Speicher (Multicore-Rechner) zugeschnitten sein
- Beispiel für Implementierungen
 - Auf Architektur mit gemeinsamen Speicher: Phoenix (Stanford, C++/PThreads)
 - Auf Architektur mit verteiltem Speicher: Google, Hadoop



Proprietäre Entwicklung durch Google

- Implementiert in C++
- Bindings in Java, Python

Open-source-Implementierung in Java (Hadoop)

- Ursprünglich von Yahoo entwickelt, produktiv eingesetzt
- Mittlerweile Apache-Projekt
- <http://hadoop.apache.org/mapreduce/>



Viele Forschungsprototypen

- Umsetzungen für GPUs
 - Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. PACT 2008
- Cell-Prozessoren
 - Marc de Kruijf and Karthikeyan Sankaralingam. MapReduce for the Cell B.E. Architecture. IBM Journal of Research and Development, 53(5), 2009.





Was ist Hadoop?

- Hadoop ist eine Implementierung des MapReduce-Konzepts
- Open Source Projekt der Apache Software Foundation
- Download für Unix/Linux verfügbar
- Programmierung mit Java, Python, C++, etc. möglich

Hadoop Komponenten

- Hadoop Common
 - Stellt Grundfunktionen bereit: implementierungsneutrale File-System-Schnittstelle, Schnittstelle für die RPC-Kommunikation im Cluster
- Hadoop Distributed FileSystem (HDFS)
 - Primäre Dateisystem von Hadoop, folgt dem Vorbild des Google-Dateisystems
 - Eingabe-Dateien müssen erst in das HDFS kopiert werden, bevor sie verwendet werden können
 - Master-Slave Struktur, Fehlertoleranz durch dreifache Redundanz
 - Performanz auf sequentielles Lesen und Schreiben ganzer Blöcke optimiert
 - Es werden auch andere Dateisysteme unterstützt (z.B. CloudStore, S3)
- Hadoop MapReduce
 - Bietet alle Funktionen um nach dem MapReduce-Programmiermodell zu entwickeln



	2008	2009
Webmap	~70 hours runtime ~300 TB shuffling ~200 TB output 1480 nodes	~73 hours runtime ~490 TB shuffling ~280 TB output 2500 nodes
Sort benchmarks (Jim Gray contest)	1 Terabyte sorted <ul style="list-style-type: none">▪ 209 seconds▪ 900 nodes	1 Terabyte sorted <ul style="list-style-type: none">▪ 62 seconds, 1500 nodes 1 Petabyte sorted <ul style="list-style-type: none">▪ 16.25 hours, 3700 nodes
Largest cluster	2000 nodes <ul style="list-style-type: none">▪ 6PB raw disk▪ 16TB of RAM▪ 16K CPUs	4000 nodes <ul style="list-style-type: none">▪ 16PB raw disk▪ 64TB of RAM▪ 32K CPUs▪ (40% faster CPUs too)



Relationale und DM Operatoren in MapReduce



Selection / projection / aggregation

- SQL Query:

```
SELECT year, SUM(price)
FROM sales
WHERE area_code = "US"
GROUP BY year
```

- Map/Reduce job:

```
map(key, tuple) {
    int year = YEAR(tuple.date);
    if (tuple.area_code = "US")
        emit(year, {'year' => year, 'price' =>
tuple.price });
}
```

```
reduce(key, tuples) {
    double sum_price = 0;
    foreach (tuple in tuples) {
        sum_price += tuple.price;
    }
    emit(key, sum_price);
}
```

Sorting

- SQL Query:

```
SELECT *
FROM sales
ORDER BY year
```

- Map/Reduce job:

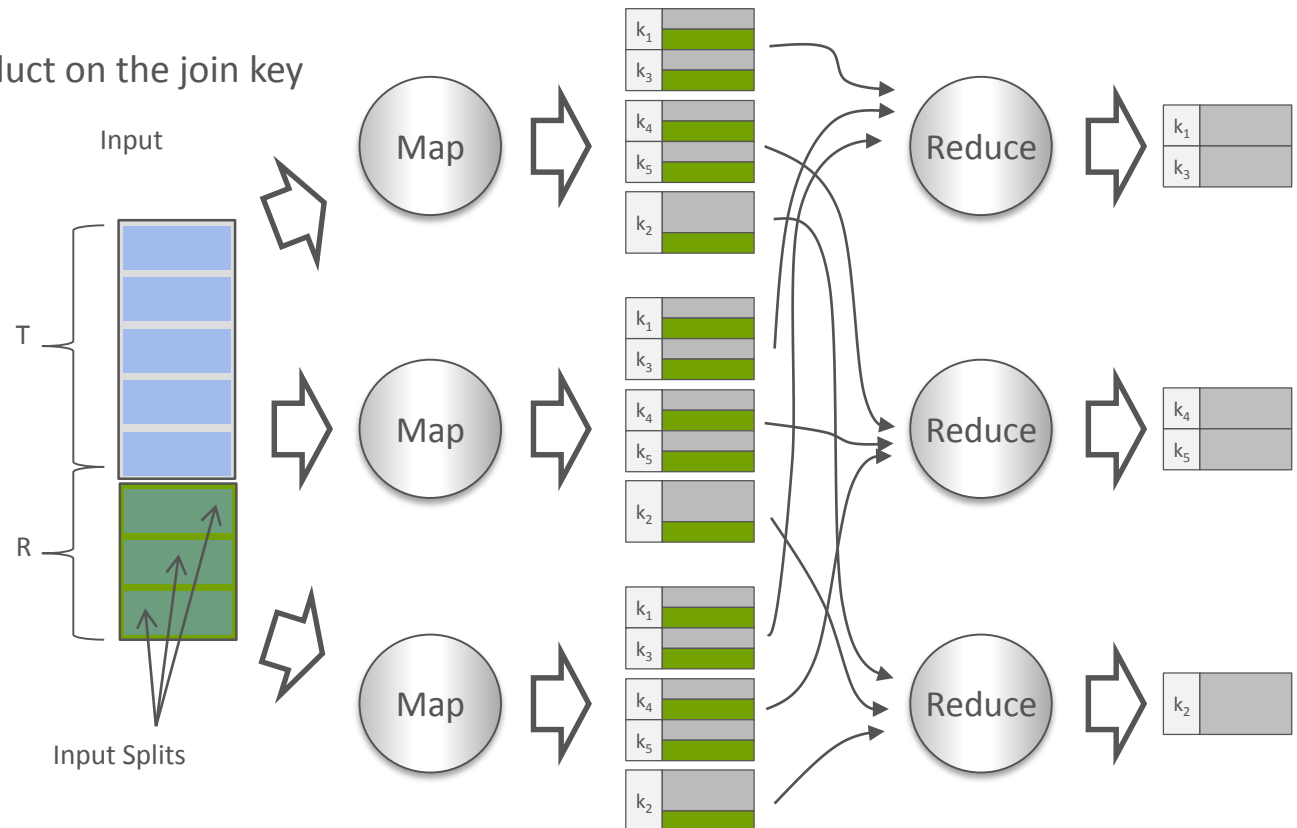
```
map(key, tuple) {
    emit(YEAR(tuple.date) DIV 10, tuple);
}

reduce(key, tuples) {
    emit(key, sort(tuples));
}
```



Problem: map and reduce process usually only a single input set

- feed both input sets into the same MapReduce program
- Map:
 - performs a repartition of both input sets based on the join key:
 - tuples which need to be joined are sent to the same reducer
- Reduce:
 - Performs a cross product on the join key





Application Use case: Entity Resolution

- Detection of entities in one or more sources that refer to the same real-world object
- Runtime-intensive task $\rightarrow O(n^2)$ entity comparisons
- Blocking
 - Semantically grouping of similar entities in blocks
 - Based on blocking keys derived from entities attributes
 - Restrict entity comparisons to entities from the same block
- Parallelization
 - MapReduce
 - Exploitation cloud infrastructures
- Work done by Lars Kolb
 - Uni Leipzig

Canon VIXIA HF S10 Camcorder - 1080p - 8.59 MP - 10 x optical zoom **\$975** new
Flash card, 32 GB, 1 yr warranty, F41 8.3.0
The VIXIA HF S10 delivers brilliant video and photos through a Canon exclusive 8.59 megapixel CMOS image sensor and the latest version of Canon's advanced image processor, ...
★★★★★ 12 reviews - [Add to Shopping List](#) [Compare prices](#)

Canon (VIXIA) HF S10 iVIS Dual Flash Memory Camcorder **\$899.00** new
Canon HF S10 iVIS Dual Flash memory CamcorderSPECIAL SALE PRICE: \$899
Display both English/Japanese + we supply all English manuals in English as PDF. ...
[Add to Shopping List](#) [Made in Japan Online](#)

Canon VIXIA HF S10 **\$999.00** new
Dual Flash Memory High Definition Camcorder The Next Step Forward in HD Video
Canon has a well-known and highly-regarded reputation for optical excellence, ...
[Add to Shopping List](#) [Performance Audio](#) [2 seller ratings](#)

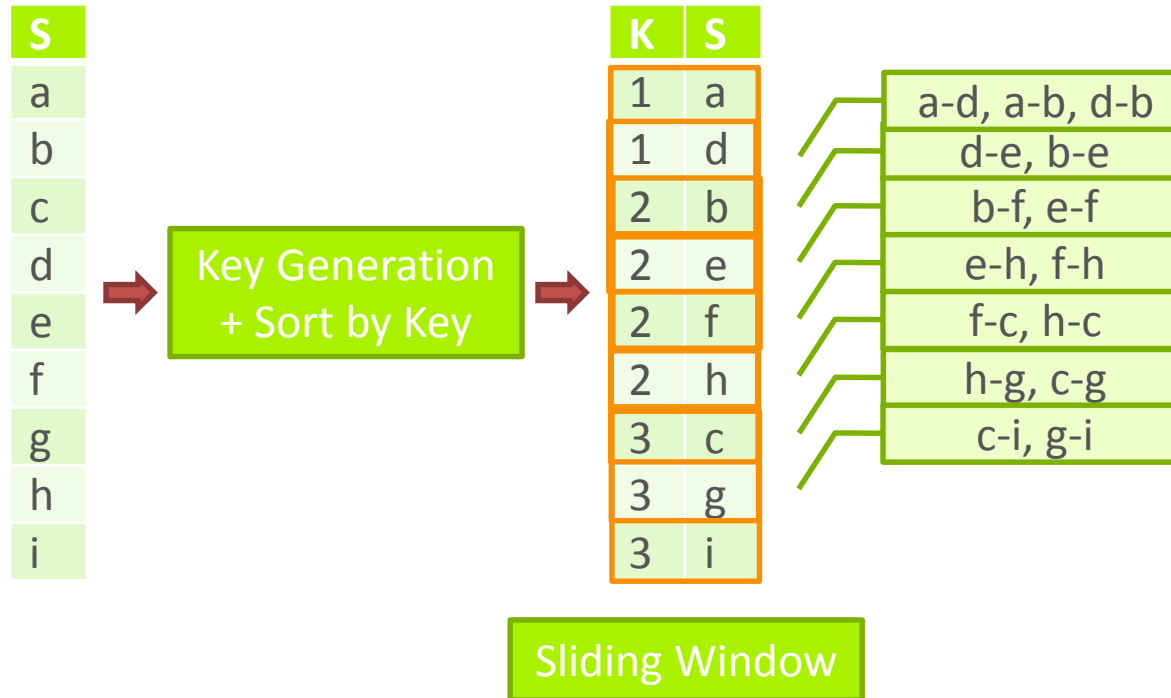
Canon VIXIA HF S100 Flash Memory Camcorder **\$899.95** new
***Canon Video HF S100 Instant Rebate Receive \$200 with your purchase of a new Canon VIXIA HF S100 Flash Memory Camcorder. (Price above includes \$200 ...
[Add to Shopping List](#) [Arlingtoncamera.com](#) [5 seller ratings](#)

Canon Vixia Hf S10 Care & Cleaning **\$2.99** new
Care & Cleaning Digital Camera/Camcorder Deluxe Cleaning Kit with LCD Screen Guard Canon VIXIA HF S10 Camcorders Care & Cleaning.
[Add to Shopping List](#) [shop.com](#) [★★★★☆ 38 seller ratings](#)

> Sorted Neighborhood -



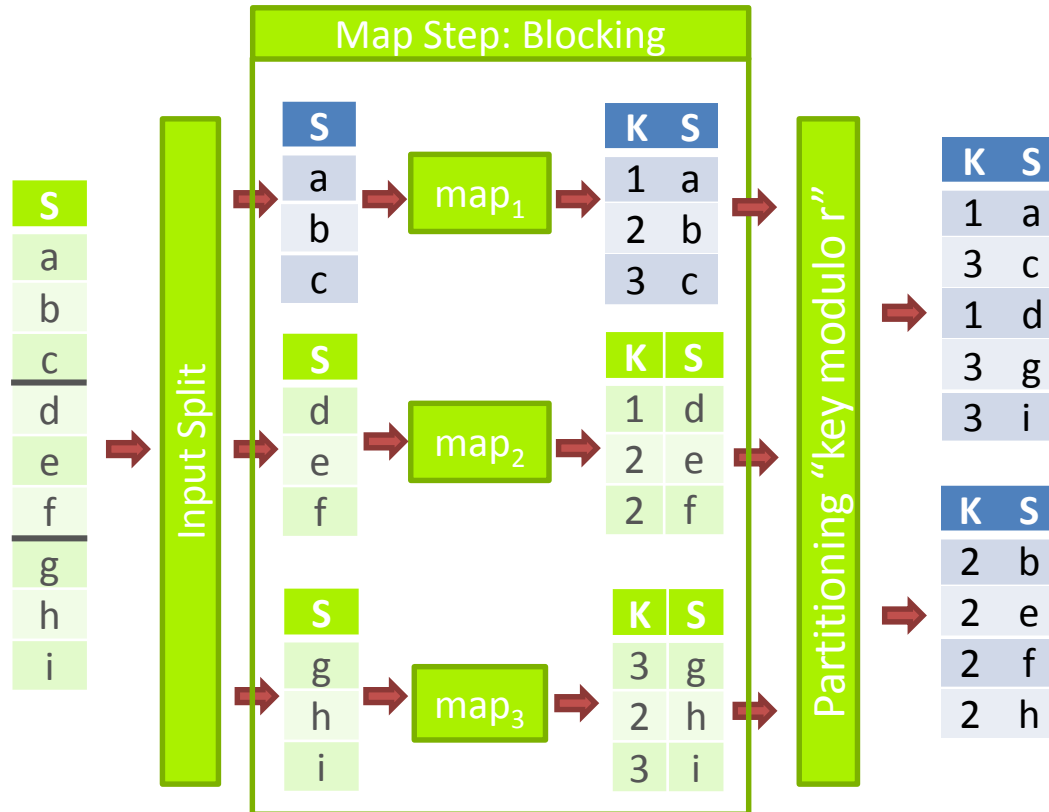
Running Example ($w=3$)



- Determine blocking key for each entity and sort entities by blocking key
- Move window of fixed size w over sorted records and compare all entities within window
- All entities within a distance of $w-1$ are compared
- $O(n^2) \rightarrow O(n) + O(n \cdot \log n) + O(n \cdot w)$

> Entity Resolution with MapReduce

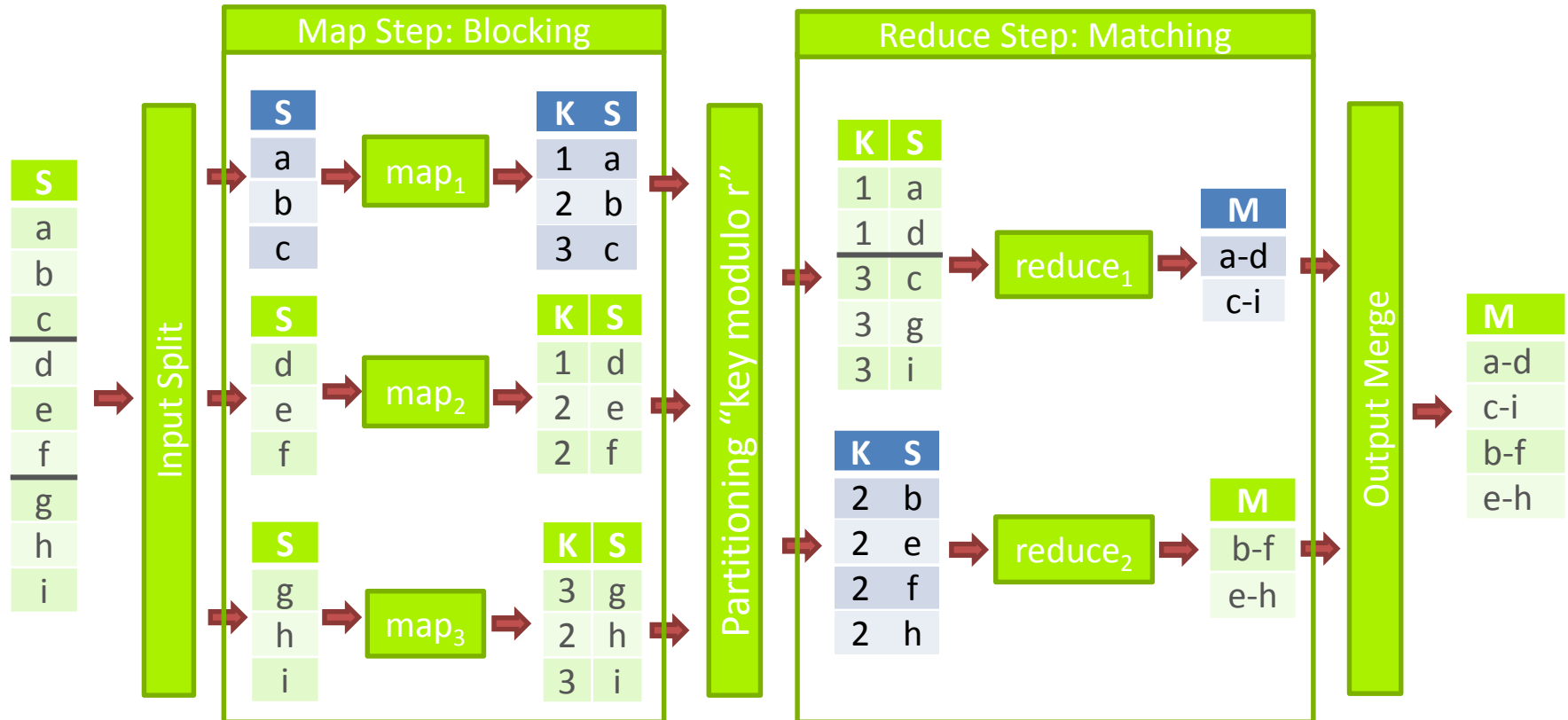
($m=3$, $r=2$)



- Map phase
 - Input data partitioned in m partitions
 - Each processed by one map task that calls **map** for each input record ("blocking")
 - UDF **part** partitions map output and distributes it to the r reduce tasks

> Entity Resolution with MapReduce

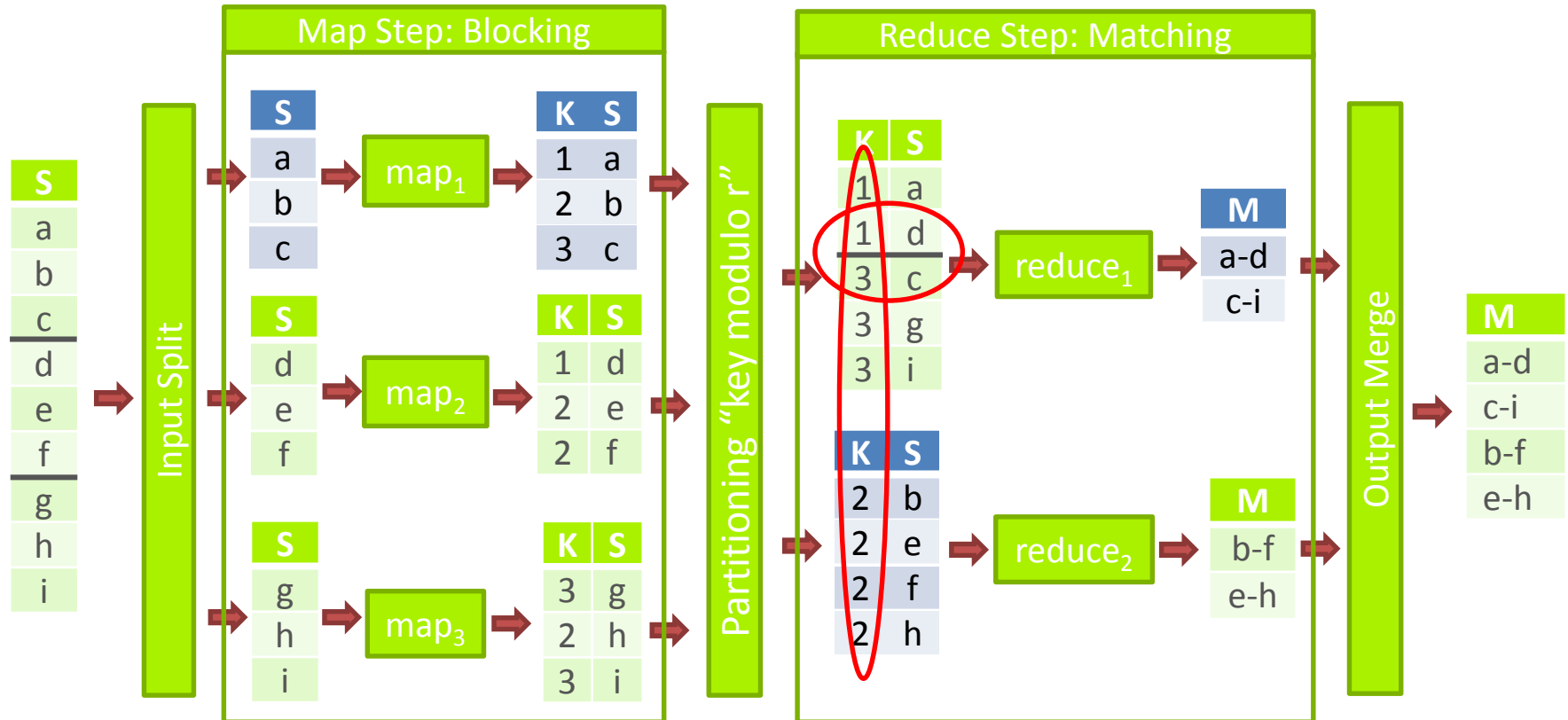
($m=3, r=2$)



- Reduce phase
 - Sorting of key-value pairs by key
 - Grouping of key-value pairs by key
 - Invocation of **reduce** for each group ("matching")

> Entity Resolution with MapReduce

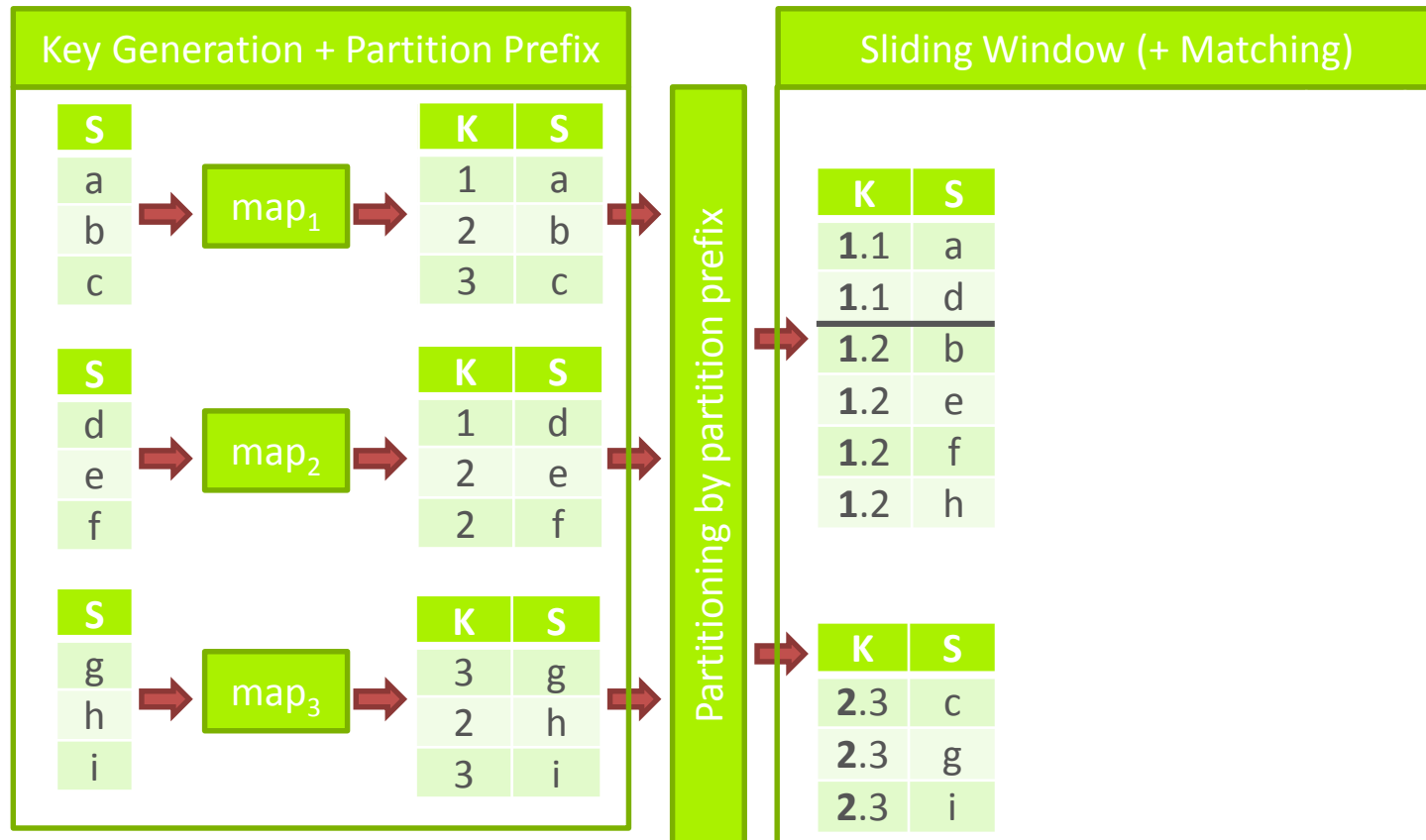
($m=3, r=2$)



- **Challenge 1:** SN requires totally sorted list of entities
 - All entities assigned to reduce task R_i have smaller blocking key than all entities assigned to reduce task R_{i+1}
 - **"Sorted reduce partitions" (SRP)**
 - Must be ensured by **part** → range partitioning

> Sorted Neighborhood with MapReduce

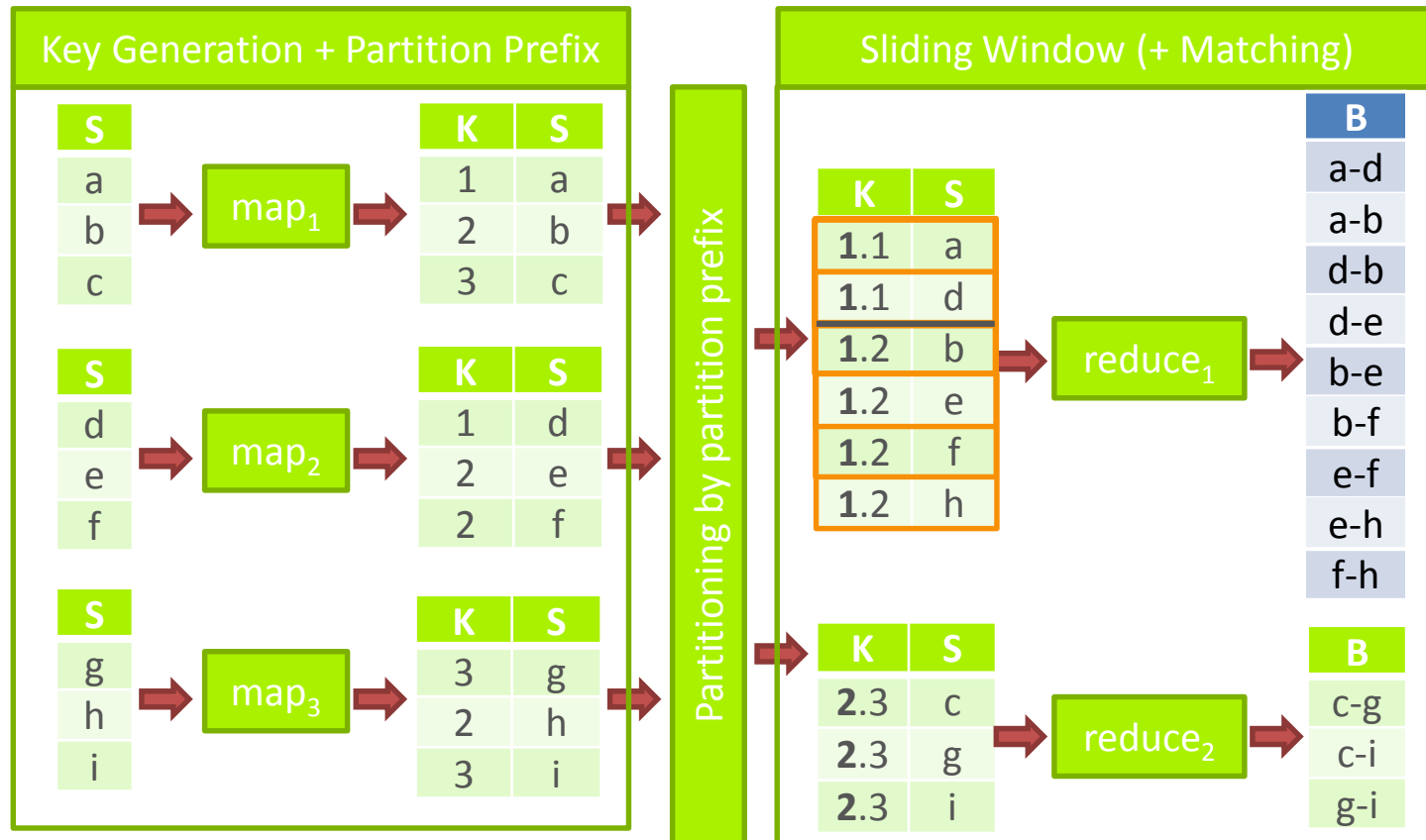
– SRP



- **map** outputs composite key: *partitionPrefix.blockKey*
 - $\text{partitionPrefix}(k) = 1$ if $k \leq 2$, otherwise 2 (range partitioning)
- **part**(*partitionPrefix.blockKey*) = *partitionPrefix*
 - Key-value pairs are sorted and grouped by composed key

> Sorted Neighborhood with MapReduce

– SRP



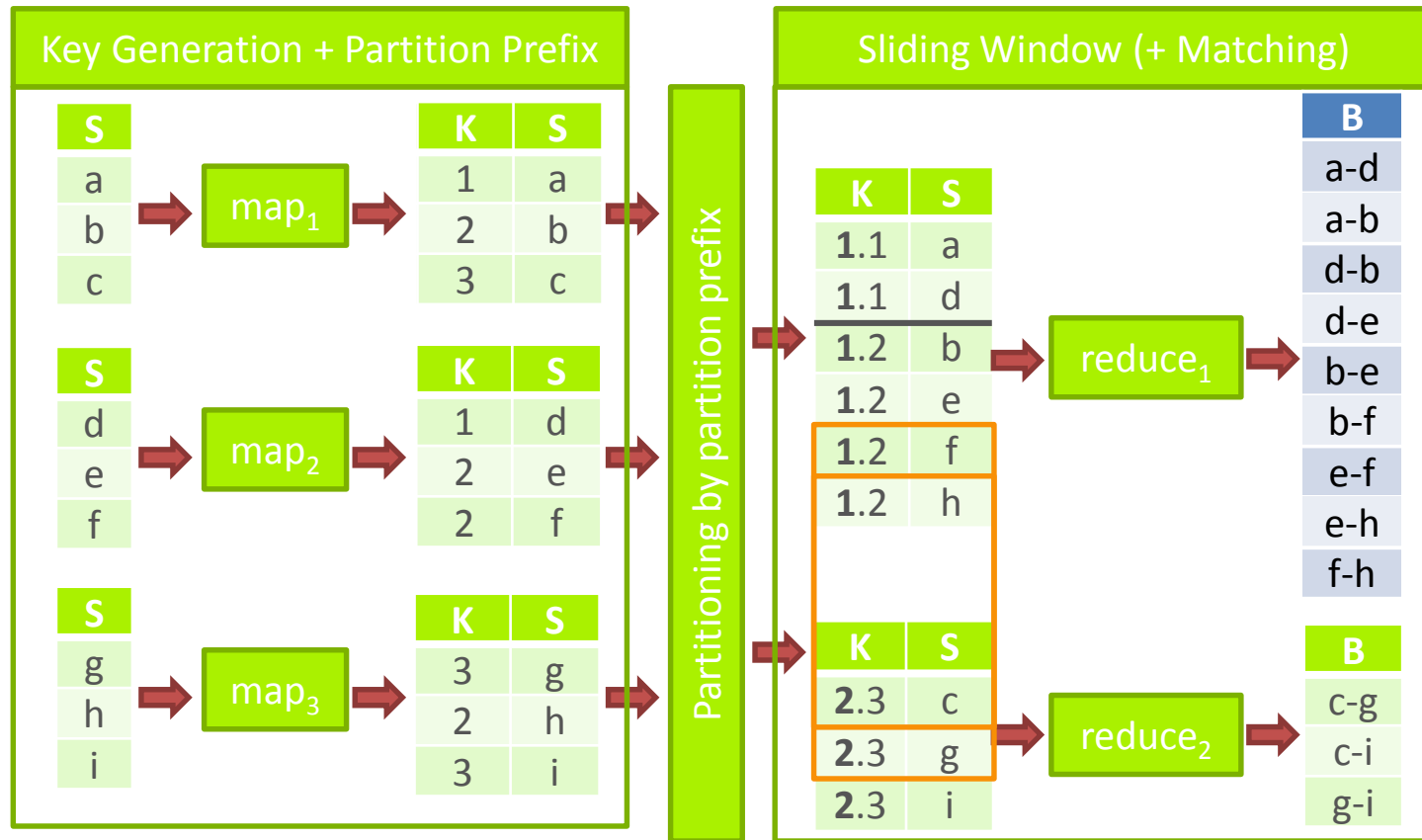
- **reduce:**

```

forEach(entity ∈ list(valuetmp))
    match(buffer, entity); //match all buffered entities with entity
    buffer.append(entity);
    if(buffer.size()==w) buffer.removeFirst();
    
```

> Sorted Neighborhood with MapReduce

– SRP



f-c ?
h-c ?
h-g ?

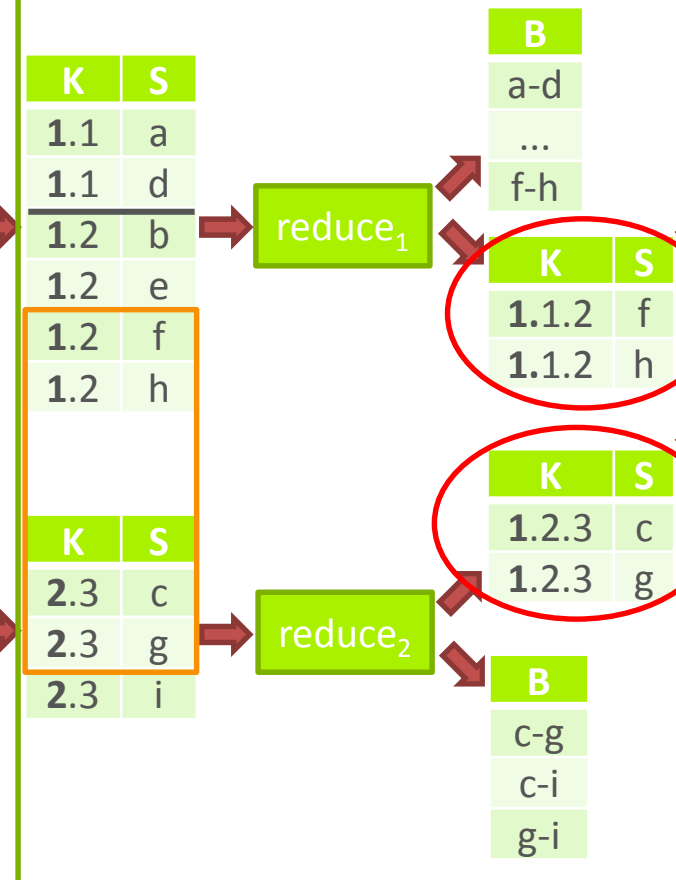
- **Challenge 2:** Boundary Entities
 - Comparison of entities entities that are assigned to different reduce tasks

> Sorted Neighborhood with MapReduce

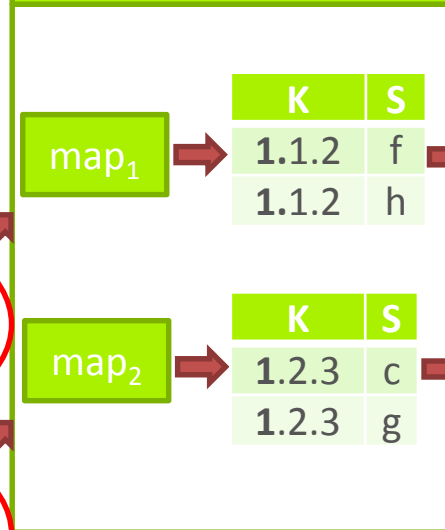
– JobSN



Sliding Window (+ Matching) + Boundary Prefix

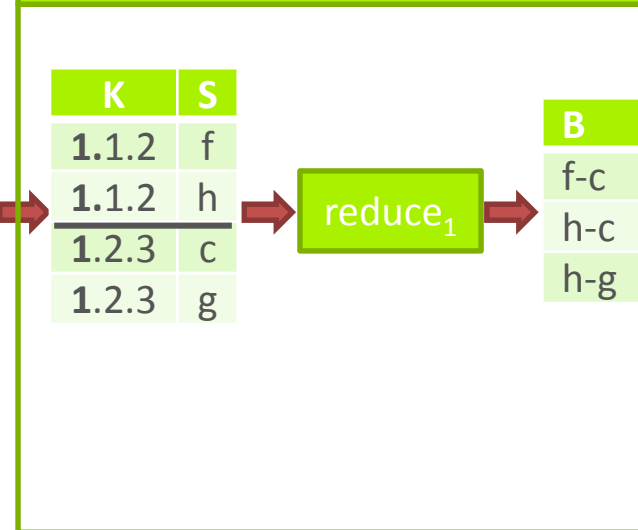


Identity



Partitioning by boundary prefix

Sliding Window (+ Matching)



• SN realization using two consecutive jobs

• Job1:

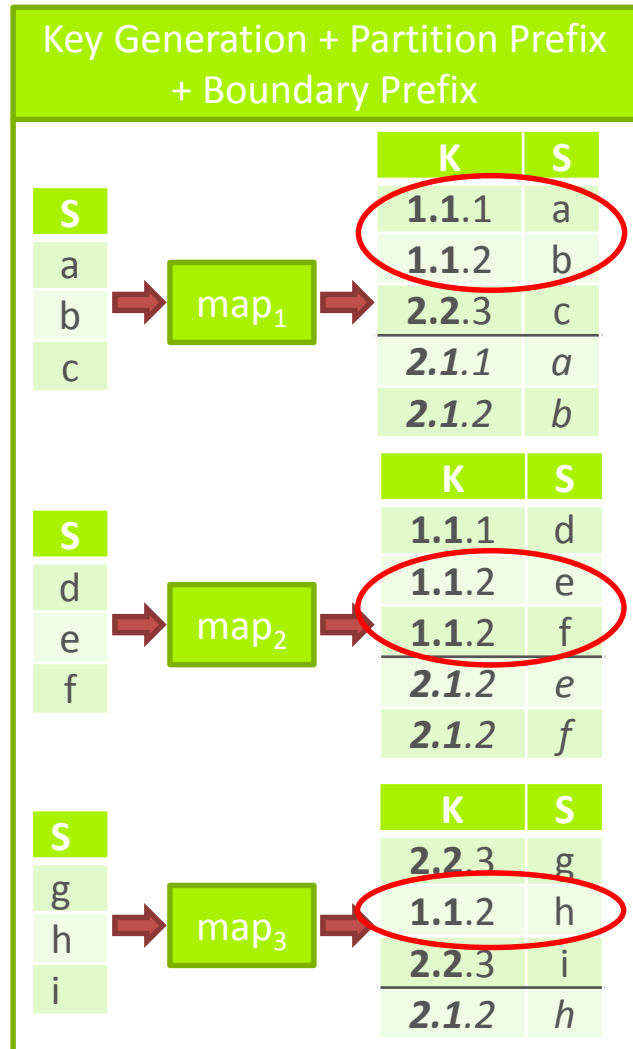
- SRP + additional output of boundary entities
- Keys of the additionally outputted entities are prefixed with an additional boundary component

• Job2:

- SN for boundary entities
- $\text{part}(\text{boundary.partitionIndex.blockKey}) = \text{boundary} \% r$
- Sort and group by composed key

> Sorted Neighborhood with MapReduce

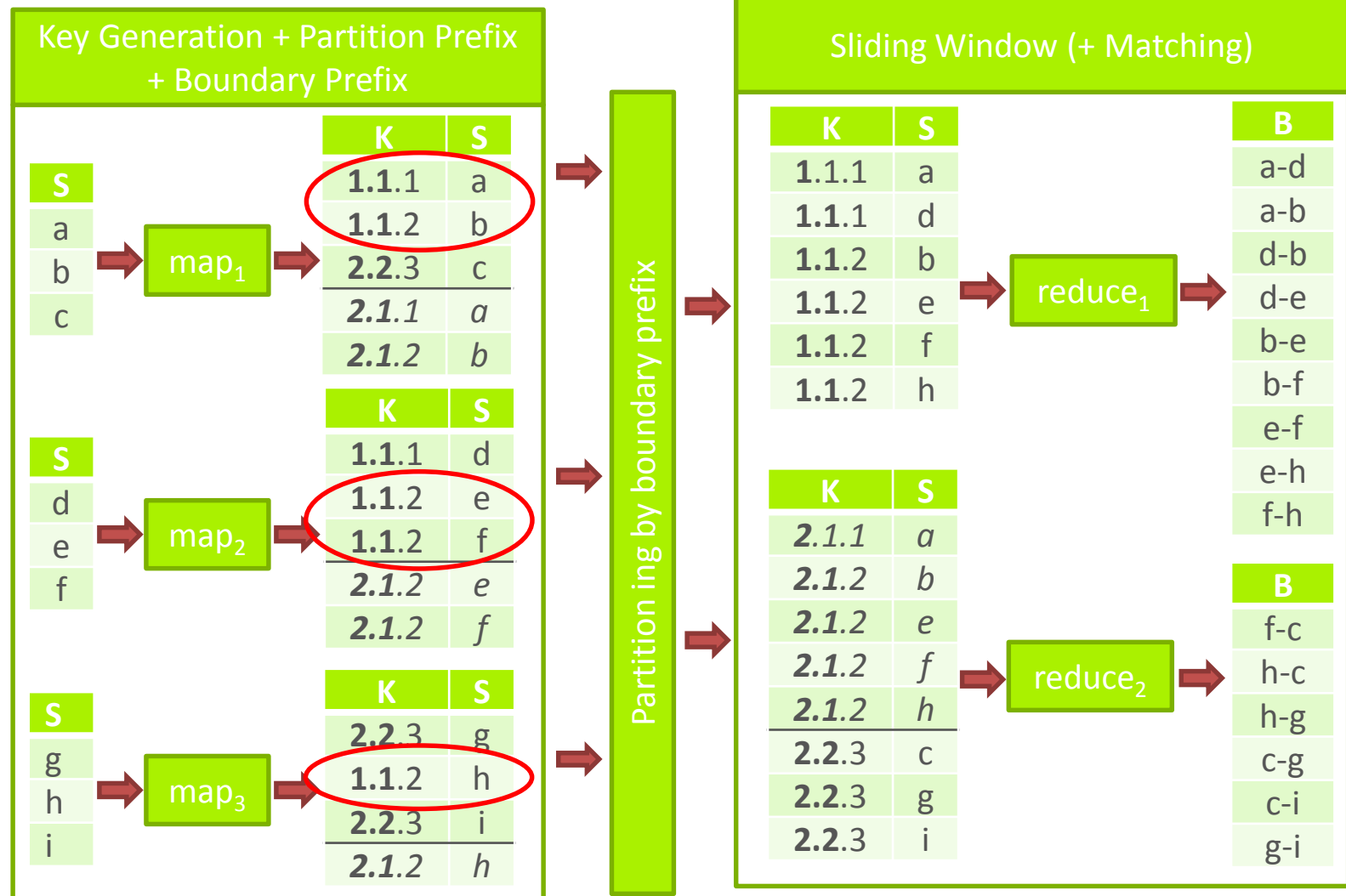
- RepSN



- SN realization using data replication
 - Reduce task $i > 1$ needs last $w-1$ entities of previous partition in front of its input
 - Potential boundary entities are replicated by the map tasks (two key-value pairs)
 - Replica of entity that is assigned to reduce task R_i is assigned to R_{i+1}
- Implementation
 - Map key prefixed with boundary component (like JobSN)
 - $\text{boundary} = \text{partitionPrefix} + 1$ for replicated entities ($\text{boundary} = \text{partitionPrefix}$ otherwise)
 - **part**(*boundary.partitionPrefix.blockKey*) = *boundary*

> Sorted Neighborhood with MapReduce

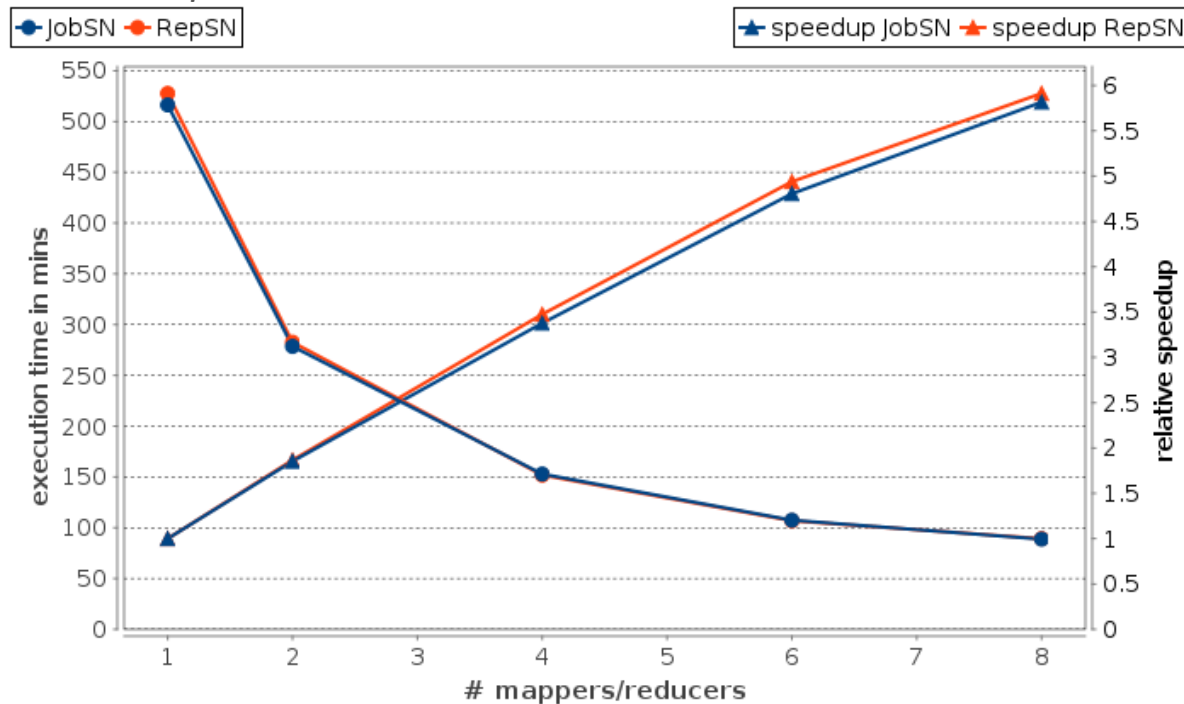
- RepSN





1.4m publication records, blocking by title.substring(2), w=1000

4 Dual core nodes, Hadoop 0.20.2



Runtime reduction: 9h to 1.5h → relative speedup of almost 6

Runtime of the implementations differ only slightly

- JobSN faster for small degree of parallelism
- RepSN completes faster gebinning with $m=r=4$



Clustering = data mining task

- group objects in a way that
 - within a group all objects are similar (or close) to one another
 - and all objects are different from objects in other groups

K-Means

1. Initially, K data objects (points) are randomly chosen as initial centroids (cluster center points)
2. Assign each data object to its closest centroid based on a given distance function such as euclidian distance
3. Update the centroids by recomputing the mean coordinates of all members of its group
4. Repeat step 2-3 until centroids do not change anymore



Well-suited for parallelization

- partition the data
- calculate the centers for each partition
- calculate the centers from the average of partition centers

Data structures:

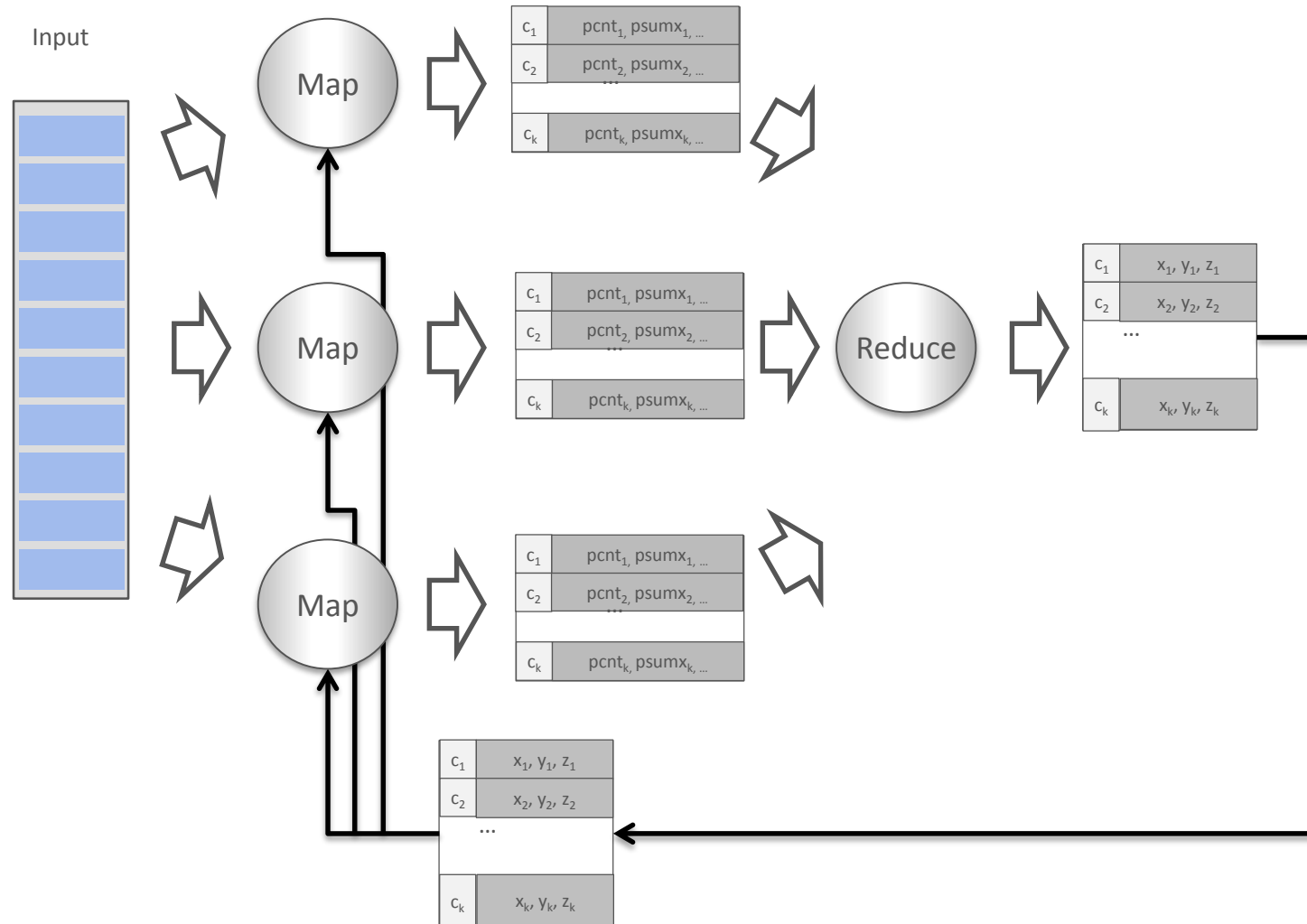
- Global file with centroid information c_1, \dots, c_k with $c_i = (x_i, y_i, z_i)$

Map:

- Process partition and maintain for each centroid c_i the following statistics:
 - $pcnt_i$
 - $psumx_i$
 - $psumy_i$
 - $psumz_i$

Reduce: aggregation

> K-Means in MapReduce



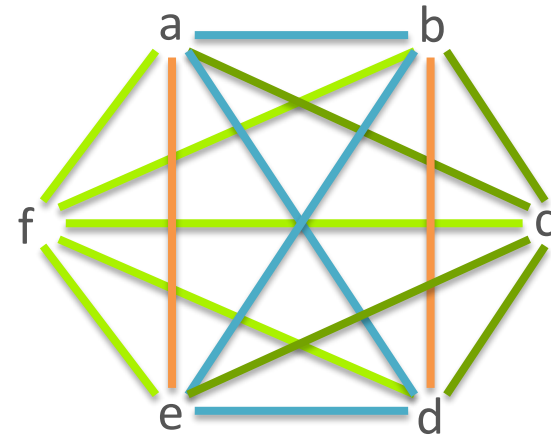


Set S

- 6 elements: $\{a, b, c, d, e, f\}$
- 15 two-element subsets (pairs): $\{a,b\}; \{a,c\}; \dots$

Function $\text{comp}: S \times S \rightarrow \square$

- Needs to be evaluated on all pairs
- Needs to be evaluated in parallel



Why Parallel?

- S may be large (cardinality) \rightarrow complexity grows with the square of it's size
- S may be big (storage) $\rightarrow S$ does not fit in memory (or even on the node)
- comp may be expensive

How Parallel?

- How to partition the Cartesian product $S \times S$?



Evaluated function $comp$ is symmetric

Execution model – MR environment

- Independent nodes – connected by a (possibly slow) network
- All nodes execute tasks in parallel – each node processes local data
- No online communication or shared memory

Data organization

- Input data stored in files – distributed on the participating nodes
- No random access to single elements
- Output written locally (possibly aggregated)

Data representation

- Each record consists of unique identifier and (possibly large) element data
- Computation results can be stored efficiently



Data representation

- Each record consists of unique identifier and (possibly large) element data
- Computation results can be stored efficiently

element
identifier

element
data

computation
results

```
s1 xxxxxxxxxxxxxxxxxxxxxxxxxxxx (s2, comp(s1,s2); s3, comp(s1, s3); ...)  
s2 xxxxxxxxxxxxxxxxxxxxxxxxxxxx (s1, comp(s2,s1); s3, comp(s2, s3); ...)  
s3 xxxxxxxxxxxxxxxxxxxxxxxxxxxx (s1, comp(s3,s1); s2, comp(s3, s2); ...)
```



1. *Build subsets of the dataset and ship them to the nodes*

- Defines the set of elements that a certain node works with (working set)
- Independent tasks require element replication

2. *Perform pairwise element computation on all subsets in parallel*

- Evaluate some (or all) pairs in each subset

3. *Aggregate results of the comparisons per element*

- Depends on the application needs

How are subsets built?

How are pairs determined?



1. *Build subsets of the dataset and ship them to the nodes*

- Defines the set of elements that a certain node works with (working set)
- Independent tasks require element replication

2. *Perform pairwise element computation on all subsets in parallel*

- Evaluate some (or all) pairs in each subset

3. *Aggregate results of the comparisons per element*

- Depends on the application needs

Job 1

Job 2

Map

Reduce



Algorithm 1 Distribution and Pairwise Comparison

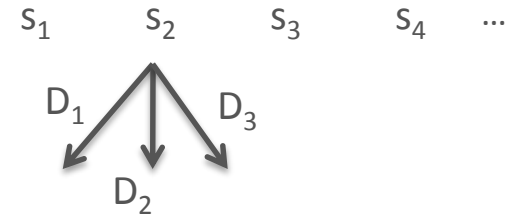
```

begin function map(id(element), element)
  [subset] ← getSubsets(id(element))
  for all  $D \in [subset]$  do
    emit( $D, element$ )
  end for
end function map
  
```

```

begin function reduce(D, [element])
  [(i, j)] = getPairs(D, [element])
  for all  $(i, j) \in [(i, j)]$  do
     $r \leftarrow$  evaluate( $element_i, element_j$ )
    addResult( $element_i, (element_j, r)$ )
    addResult( $element_j, (element_i, r)$ )
  end for
  for all  $s \in [element]$  do
    emit(id(s), s)
  end for
end function reduce
  
```

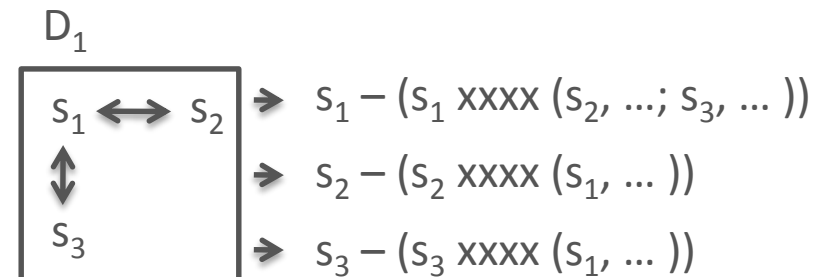
map



sort/shuffle



reduce



> Distribution Schemes: Problem Definition



Having ...

- $S = \{s_1, s_2, \dots, s_v\}$ – a set of elements
- $\mathcal{D} = \{D_1, D_2, \dots, D_b\}$ – a collection of distinct subsets of S (working sets)
- $P = \{P_1, P_2, \dots, P_b\}$ – a collection of relations (pairs), one for each working set

... how can \mathcal{D} and P be constructed, such that ...

- a) work, that is done in parallel is well balanced and
- b) each pair of elements is evaluated *exactly* once among all nodes?

Formally:

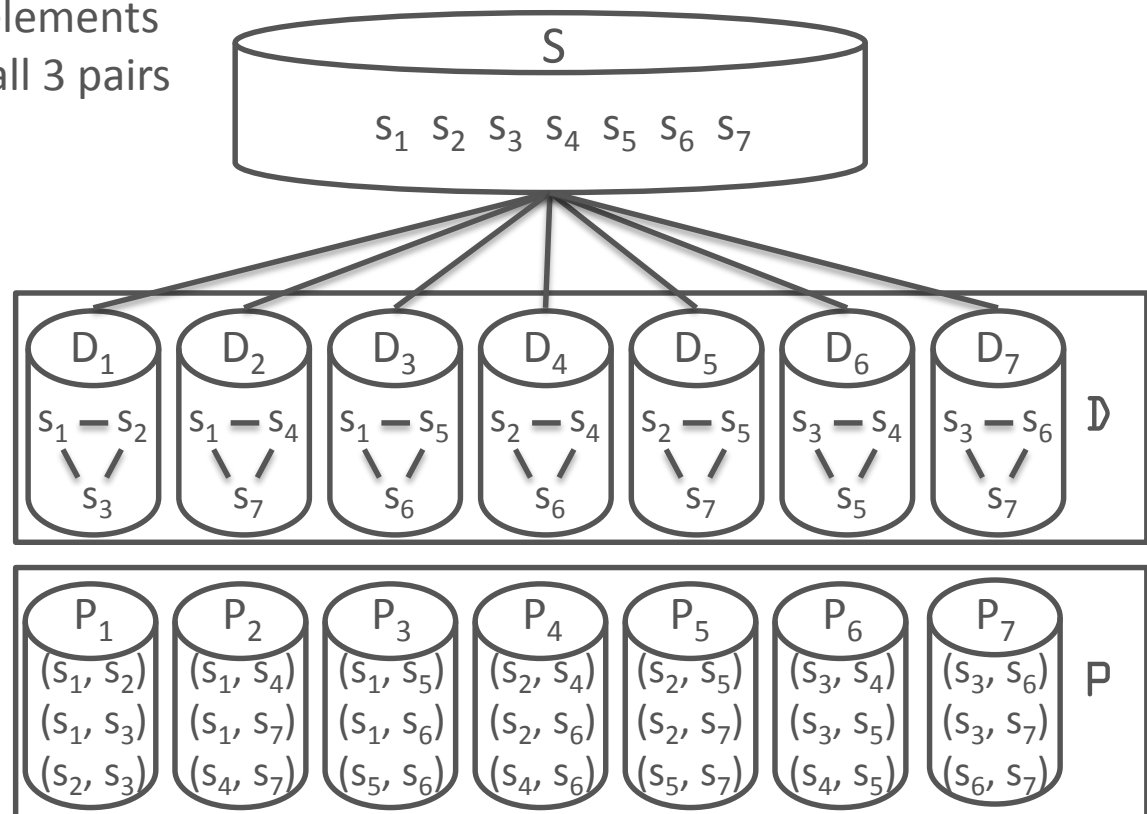
- a) all working sets in \mathcal{D} are equal (or similar) in size
- b) For any two elements s_i and s_j ($i > j$) in S , there is exactly one working set D_k with a pair relation P_k to fulfill: $s_i \in D_k$, $s_j \in D_k$, and $(s_i, s_j) \in P_k$

> Example



Solution parameters

- S contains $v = 7$ elements
- \mathcal{D} contains $b = 7$ subset
- Each subset contains $k = 3$ elements
- Each pair relation contains all 3 pairs
- -- > 7 independent tasks





Broadcast Approach

- Replication of the whole dataset and broadcast to all nodes: $D_1 = \dots = D_b = S$
- Construction of P crucial to ensure, that each pair is evaluated only once
- Enumeration and partitioning of all pairs

$\begin{smallmatrix} i \\ j \end{smallmatrix}$	1	2	3	4	5	6	7	...
1		p=1	2	4	7	11	16	...
2			3	5	8	12	17	...
3				6	9	13	18	...
4					10	14	19	...
5						15	20	...
6							21	...
7								...
...								

$$p(i, j) = \frac{(i-1)(i-2)}{2} + j$$

$$P_l = \{(i(p), j(p)) \mid (l-1)h + 1 \leq p \leq \min(lh, v)\}$$



- Smart enumeration of (upper right) matrix to get $n \times n$ blocks
- Each parallel instance must hold at most $2n$ elements at a time
- Parallel instance computes at most n^2 pairs in its block

$\begin{smallmatrix} i \\ j \end{smallmatrix}$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2															
3															
4															
5															
6															
7															
8															
9															
10															
11															
12															
13															
14															
15															

- h – blocking factor
- $\frac{1}{2} h(h+1)$ blocks/tasks over all
- each element is used in h blocks



- Smart enumeration of (upper right) matrix to get $n \times n$ blocks
- Each parallel instance must hold at most $2n$ elements at a time
- Parallel instance computes at most n^2 pairs in its block

i \ j	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1															
2															
3															
4															
5															
6															
7															
8															
9															
10															
11															
12															
13															
14															
15															

- h – blocking factor
- $\frac{1}{2} h(h+1)$ blocks/tasks over all
- each element is used in h blocks



Combinatorial Designs (a.k.a. t-designs, block designs, Steiner systems)

Parameters: (v, k, λ) -design

- v – number of elements in the dataset
- k – (exact!) number of elements per block (subset)
- λ – number of blocks that contain a certain pair (equal to 1 in this case)
- b – total number of blocks
- r – number of blocks for each element

Necessary conditions:

- $r(k-1) = \lambda(v-1)$
- $bk = vr$

Theorem (Fisher's inequality):

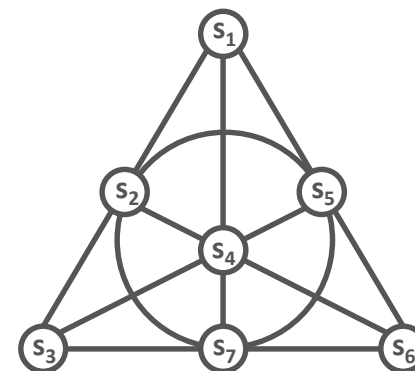
- *If there is a (v, k, λ) -design, then $b \geq v$*

Projective planes:

- $(q^2 + q + 1, q + 1, 1)$ -design
- q prime (power)

Example: $(7, 3, 1)$ -design (Fano plane)

$$\begin{array}{ccc} v = 7 & k = 3 & \\ \lambda = 1 & b = 7 & r = 3 \end{array}$$



> Comparison of Approaches



	Broadcast	Block	Design
Number of Tasks (p)	arbitrary ✓	$\frac{h(h+1)}{2}$	$q^2 + q + 1 \geq v$, q prime ✗
Communication Costs	$2vp$ ✗	$2vh$ ✓	$\approx 2v\sqrt{v}$ (max $2vn$) ✗
Replication Factor	p ✓	h ✓	$\approx \sqrt{v}$ ✗
Working Set Size	v ✗	$2 \left\lceil \frac{v}{h} \right\rceil$ ✓	$\approx \sqrt{v}$ ✓
Evaluations per Task	$\frac{v(v-1)}{2p}$ ✓	$\left\lceil \frac{v}{h} \right\rceil^2$ ✓	$\approx \frac{v-1}{2}$ ✓

Parameters

- v – number of elements in the dataset
- n – number of nodes
- p – number of tasks
- h – blocking factor



Instance Main Memory

- Working set should fit in main memory to avoid heavy I/O
- Maximum *working set size*: max_{ws}
- Ranges from 200MB to several gigabytes (VMs share hardware, instances share VM)
- Crucial for the broadcast approach

Intermediate Storage

- Storage required by the dataset and all its replications
- Maximum intermediate storage: max_{is}
- Directly limits the allowed *replication factor*
- Crucial for the design approach

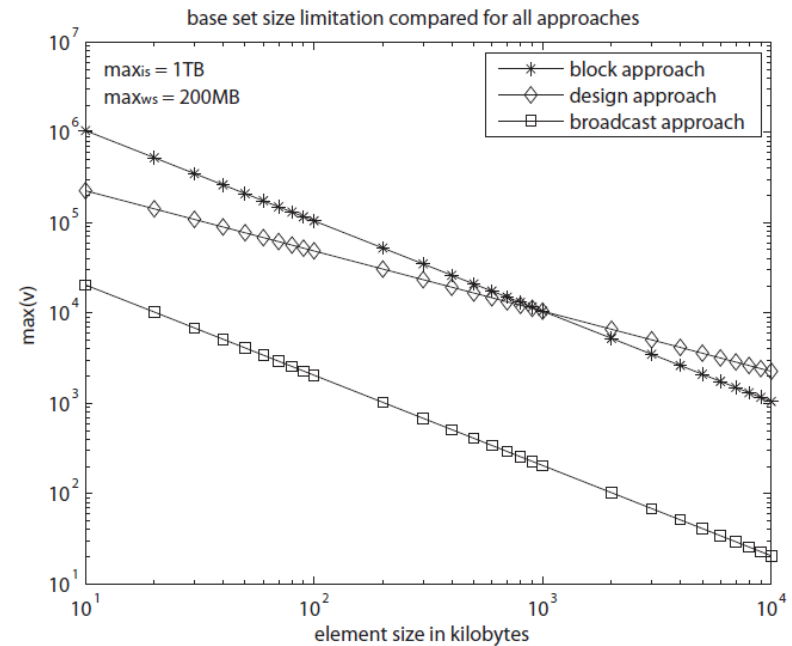
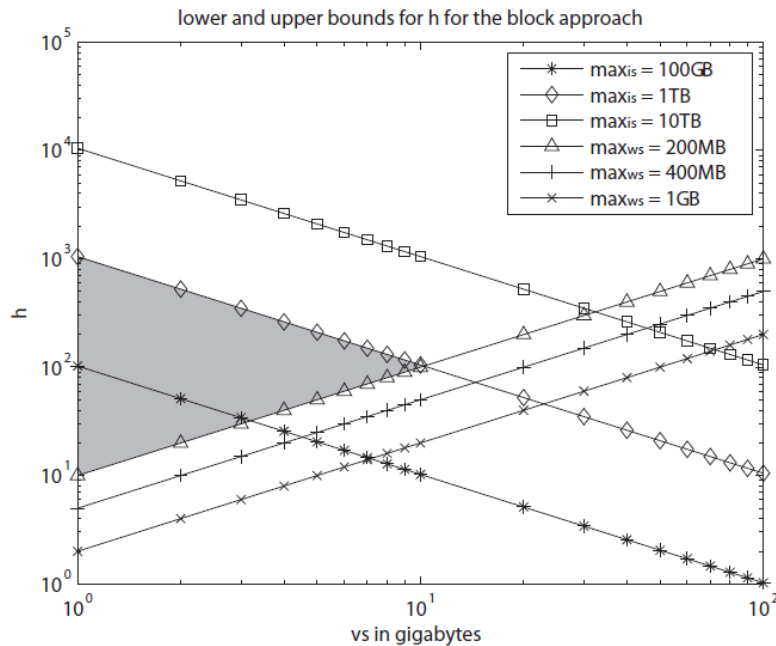


Block Approach Flexibility

- Blocking factor allows to influence characteristics

$$\frac{2vs}{h} \leq \max_{ws} \quad \text{and} \quad vsh \leq \max_{is} \quad \rightarrow \quad \frac{2vs}{\max_{ws}} \leq h \leq \frac{\max_{is}}{vs}$$

$$vs \leq \sqrt{\frac{\max_{ws} \max_{is}}{2}}$$





Benchmark to test the performance of distributed systems

Goal: Sort one Petabyte of 100 byte numbers

Implementation in Hadoop:

- Identity Mapper
- Identity Reducer
- Range-Partitioner that splits the data in equal ranges (one for each participating node)

Sort is basically "Range partitioning sort"

- The partitions are sorted partially after the mapper
- Are merged before the by the reducer.

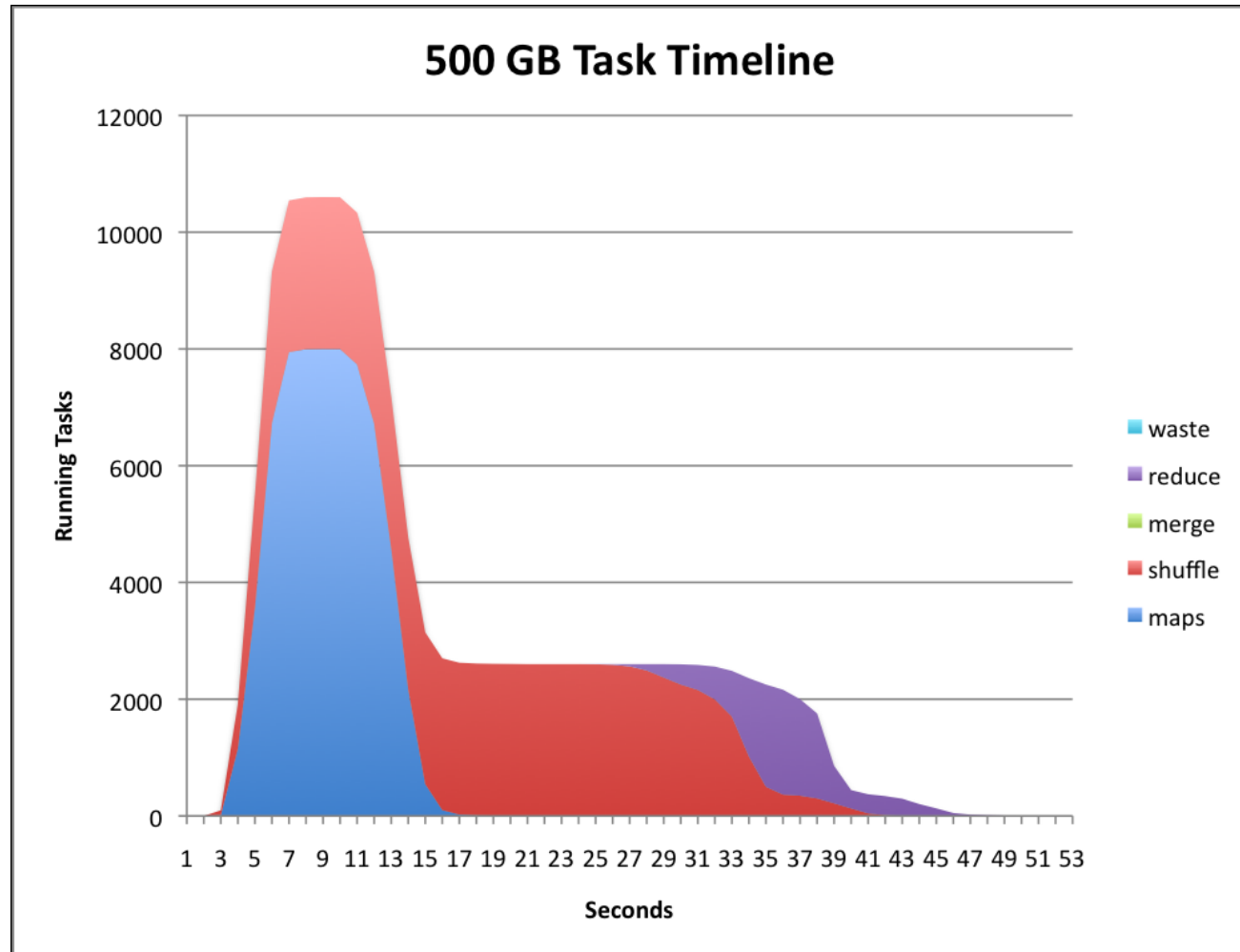
> Petabyte sorting benchmark

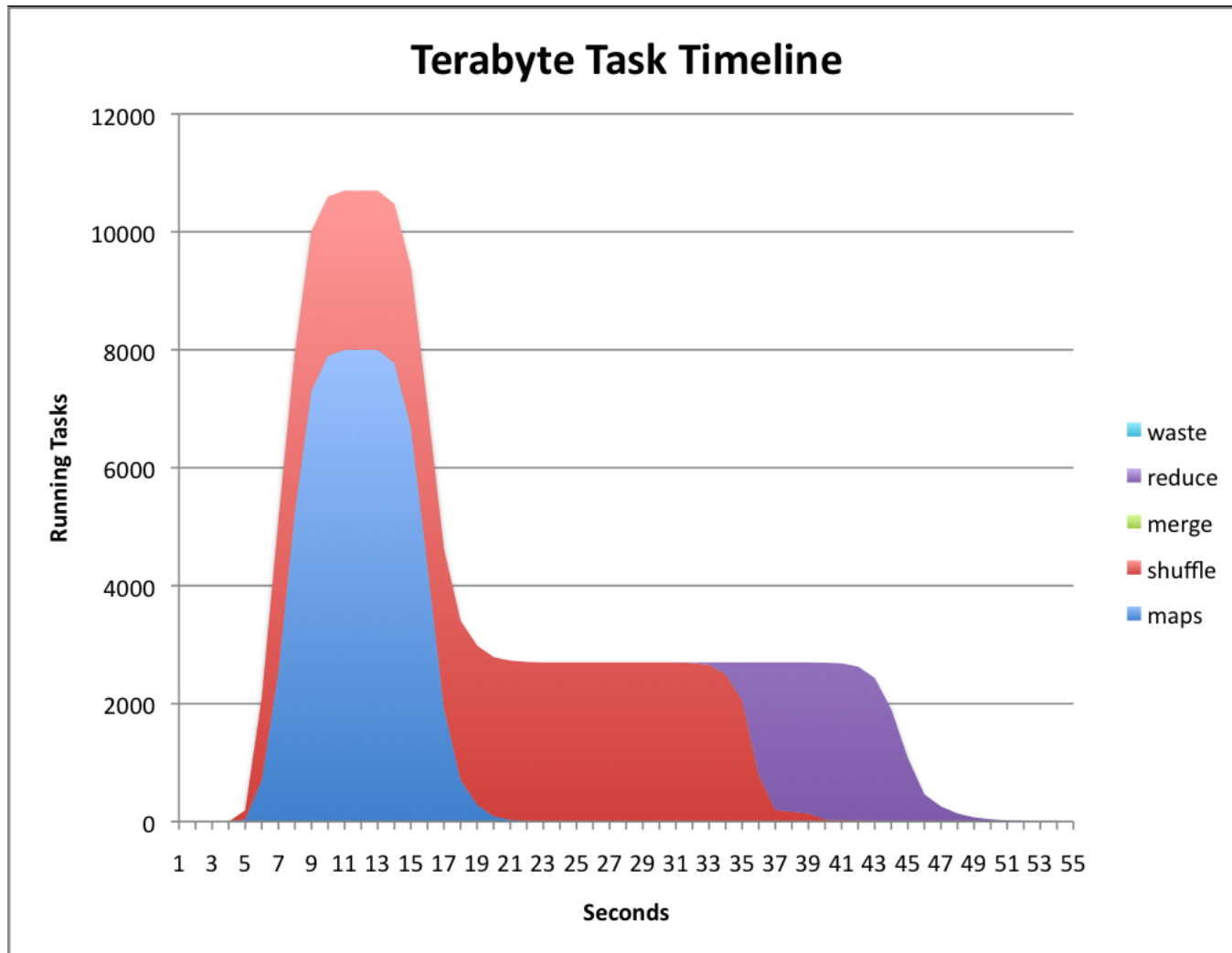


Bytes	Nodes	Maps	Reduces	Replication	Time
500,000,000,000	1406	8000	2600	1	59 seconds
1,000,000,000,000	1460	8000	2700	1	62 seconds
100,000,000,000,000	3452	190,000	10,000	2	173 minutes
1,000,000,000,000,000	3658	80,000	20,000	2	975 minutes

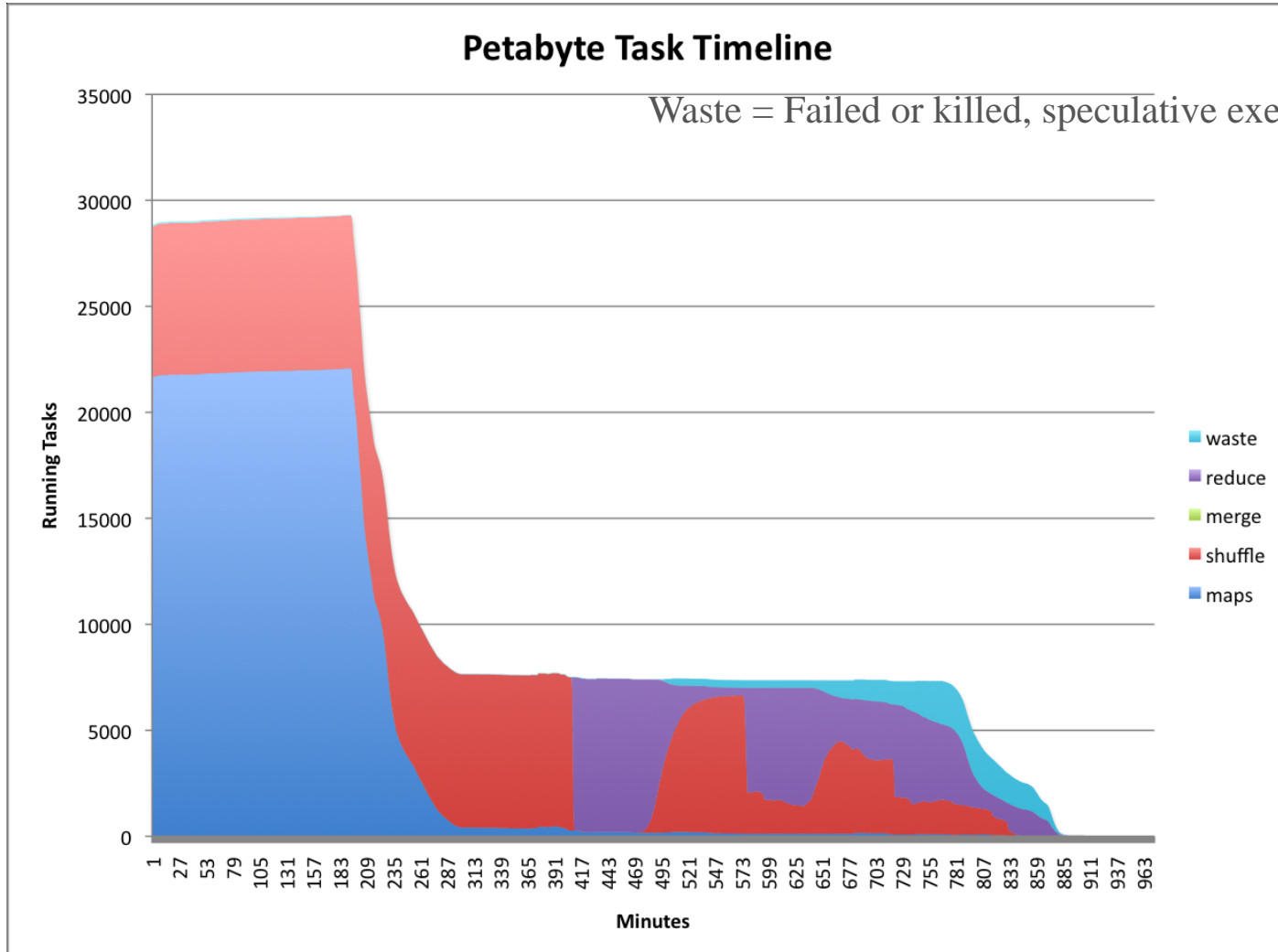
- Per node: 2 quad core Xeons @ 2.5GHz, 4 SATA disks, 16GB RAM, 1 Gigabit Ethernet.
- Per Rack: 40 nodes, 8 gigabit Ethernet uplinks.

> Cluster Utilization during Sort





> Cluster Utilization during Sort





Kritik an MapReduce



MapReduce-Kontroverse in der Datenbank-Gemeinde

- MapReduce wiederholt bereits gemachte Fehler
- „Brute-force-Ansatz“, keine Indizierung o.ä.
- Performanz unklar
- Kein „Schema“ für Daten
- Konstrukte nicht abstrakt genug
- Keine angemessene Abfragesprache

	Parallele DBMS	MapReduce
Schemaunterstützung	X	-
Indexe	X	-
Programmiermodel	Deklarativ (SQL)	Beschreibung eines Algorithmus (C/C++, Java, ...)
Optimierung	X	-
Flexibilität	-	X
Fehlertoleranz	-	X

David J. DeWitt and Michael Stonebraker, MapReduce: A major step backwards, *The Database Column*, Januar 2008

<http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html>

<http://www.databasecolumn.com/2008/01/mapreduce-continued.html>



Andrew Pavlo , Erik Paulson , Alexander Rasin , Daniel J. Abadi , David J. DeWitt , Samuel Madden , Michael Stonebraker, A comparison of approaches to large-scale data analysis. SIGMOD, June 29-July 02, 2009, Providence, Rhode Island, USA.

Hadoop

- 0.19.0
- Java 1.6

DBMS-X

- Parallele shared-nothing, zeilenbasierte Datenbank
- Hash-partitioniert, Indexe angelegt
- Kompression aktiviert

Vertica

- Parallele shared-nothing, spaltenbasierte Datenbank (Version 2.6)
- Kompression aktiviert
- Standardeinstellung (+Hint, dass jeweils nur eine Anfrage ausgeführt wird)
- Keine Sekundärindexe

Alle drei Systeme wurde auf einem Cluster bestehend aus 100 Knoten installiert

Durchführung von

- Load, Selektion, Aggregation und Join



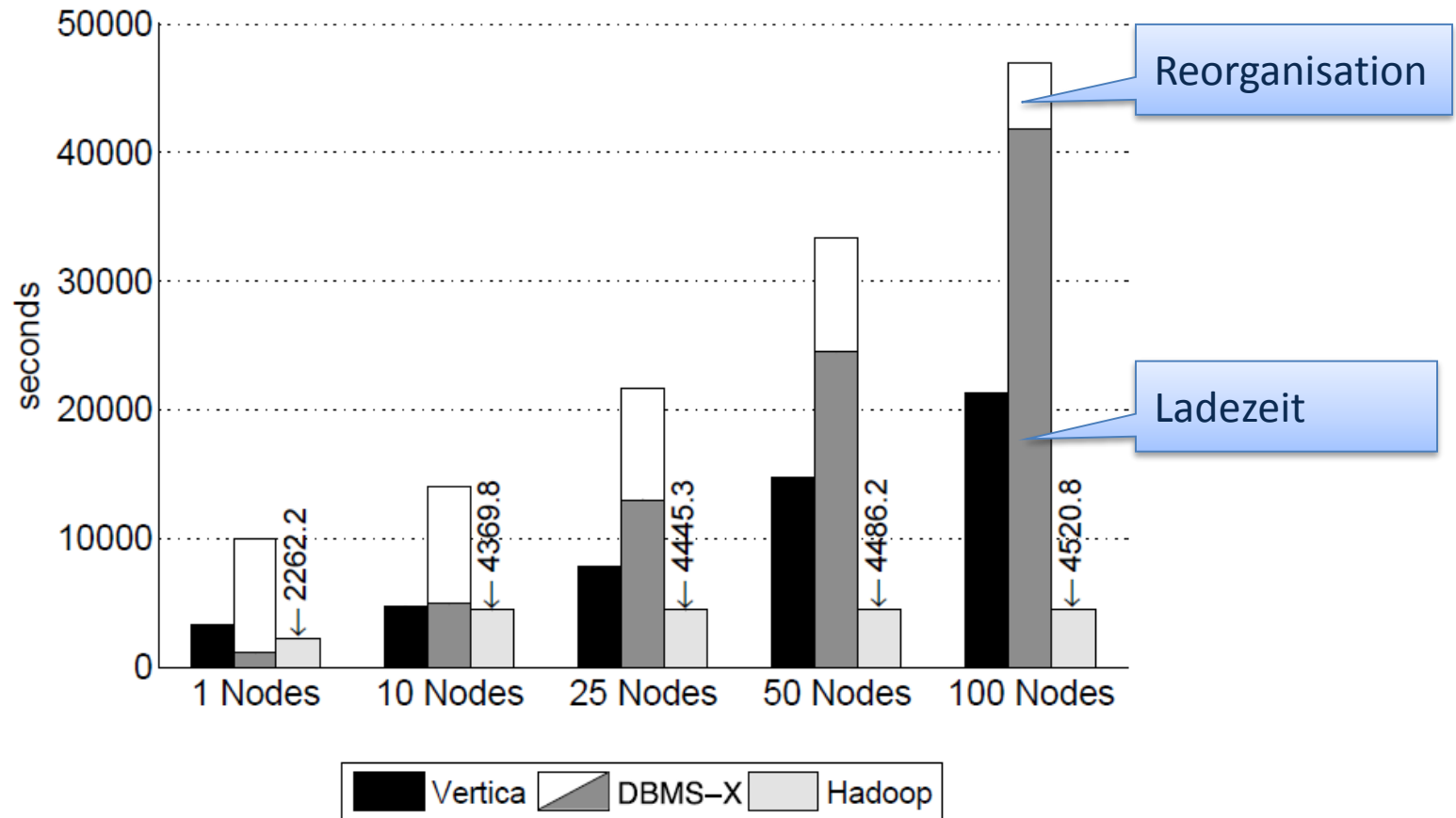
Datensatz

```
CREATE TABLE Documents (  
    url VARCHAR(100)  
        PRIMARY KEY,  
    contents TEXT );
```

```
CREATE TABLE Rankings (  
    pageURL VARCHAR(100)  
        PRIMARY KEY,  
    pageRank INT,  
    avgDuration INT );
```

```
CREATE TABLE UserVisits (  
    sourceIP VARCHAR(16),  
    destURL VARCHAR(100),  
    visitDate DATE,  
    adRevenue FLOAT,  
    userAgent VARCHAR(64),  
    countryCode VARCHAR(3),  
    languageCode VARCHAR(6),  
    searchWord VARCHAR(32),  
    duration INT );
```

- Dokumente : 600.000 unterschiedliche Dokument an einem Knoten
- 155 Mio. UserVisits-Tupel (20GB/Knoten)
- 18 Mio. Ranking-Tupel (1GB/Knoten)





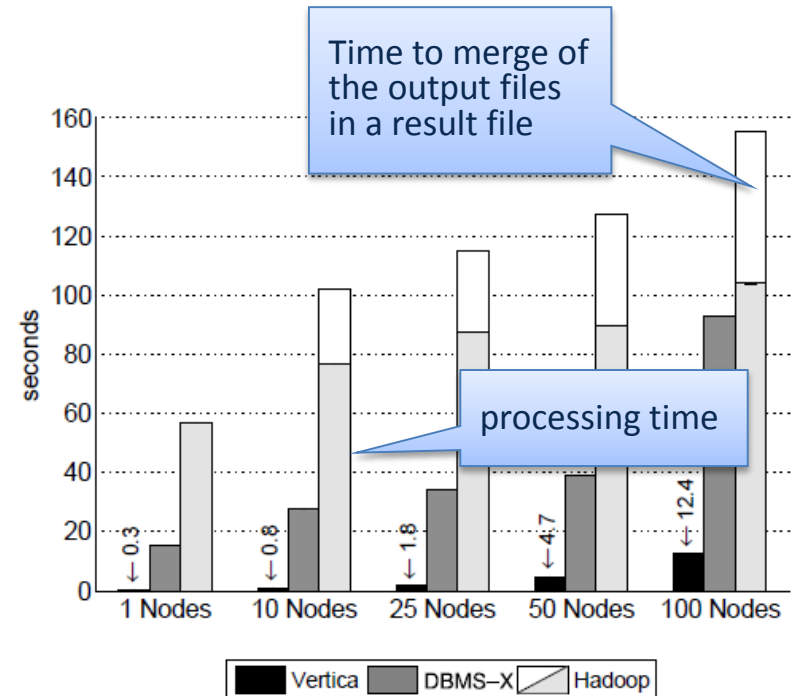
Ziel

- Finde die pageURLs in der Ranking-Tabelle (1GB/node) deren pageRank einen gewissen Schwellwert überschreiten

Anfrage

- SELECT** pageURL, pageRank
FROM Rankings **WHERE** pageRank > x
- x = 10, resultiert in ca. 36.000 Zeilen pro Knoten

Für MapReduce, wurde die Selektion in Java implementiert



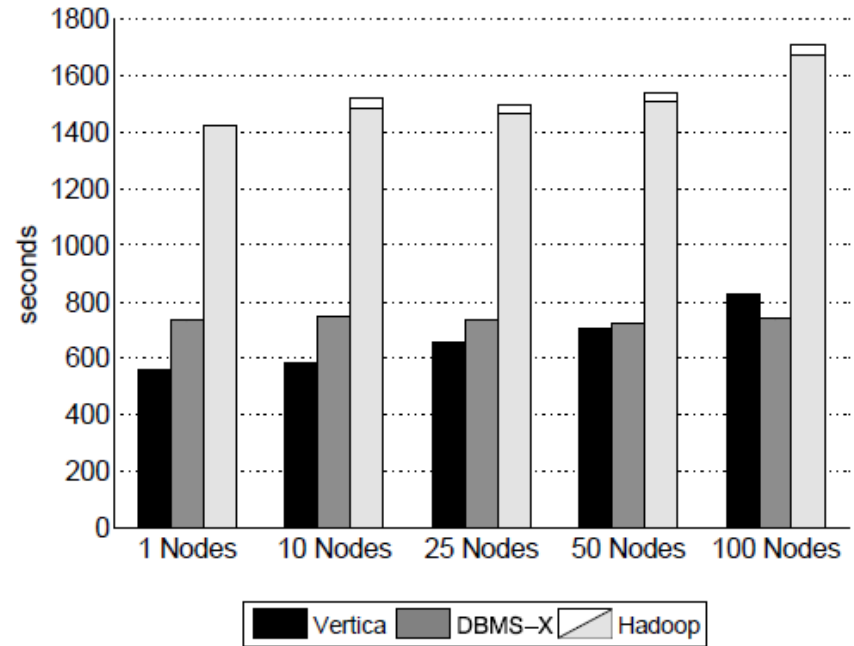


Ziel

- Berechnung des gesamten Werbeumsatzes (adRevenue) für jede sourceIP in der Tabelle UserVisits (20GB/Knoten).

Anfrage

- **SELECT** sourceIP, **SUM**(adRevenue)
FROM UserVisits **GROUP BY** sourceIP
- Die Anfrage erzeugt 2,5 Mio. Ergebnistupel



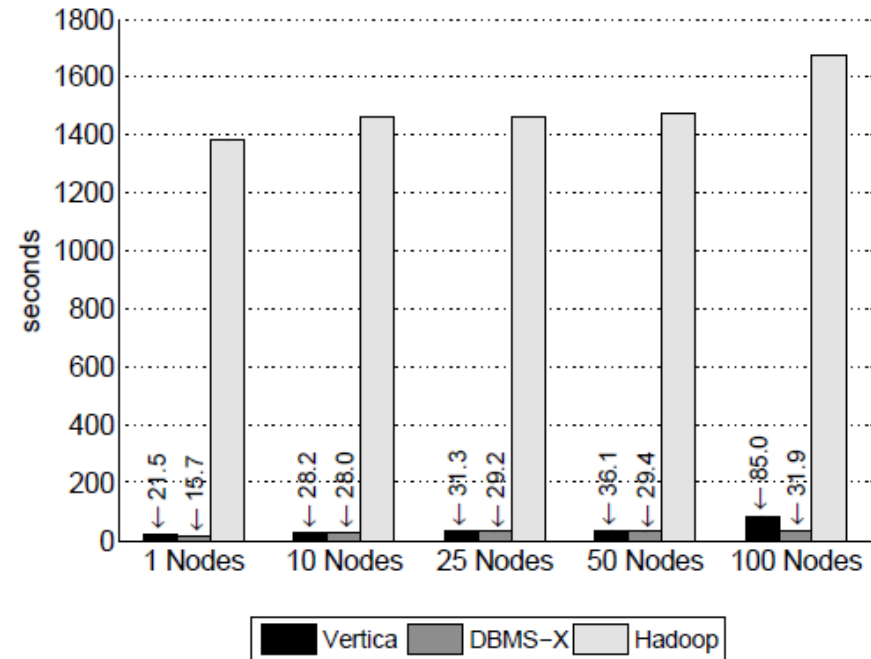


Query

- Join of tables Ranking with Uservisits and computation of the overall turnover and average pageRank
- Descending sort and return of tuple with highest turnover

SQL specification of the query

- **SELECT INTO** Temp sourceIP,
AVG(pageRank) **AS** avgPageRank,
SUM(adRevenue) **AS** totalRevenue
FROM Rankings **AS** R, UserVisits **AS** UV
WHERE R.pageURL = UV.destURL
AND UV.visitDate **BETWEEN** Date('2011-01-15')
AND Date('2011-01-22')
GROUP BY UV.sourceIP
- **SELECT** sourceIP, totalRevenue, avgPageRank
FROM Temp
ORDER BY totalRevenue **DESC LIMIT** 1





Ladezeit sind bei MapReduce um Größenordnungen besser (im Prinzip nicht vorhanden)

Bei den Ausführungszeiten schneiden DBMS weitaus besser ab (1-2 Größenordnungen)

- Vorverarbeitung beschleunigt Ausführung
- z. B. haben Vertica und DBMS-X einen Index auf der pageRank-Spalte

*→ MapReduce ist gut für sogenannte **on-demand** Berechnungen (on-off processing), DBMS für sich wiederholende Datenanalysen*