CPT-S 415

Big Data

Yinghui Wu

EME B45

CPT-S 415
Big Data

Parallel Data Management

- ✓ Why parallel DBMS?
- ✓ Architectures
- ✓ Parallelism
  - Intraquery parallelism
  - Interquery parallelism
  - Intraoperation parallelism
  - Interoperation parallelism

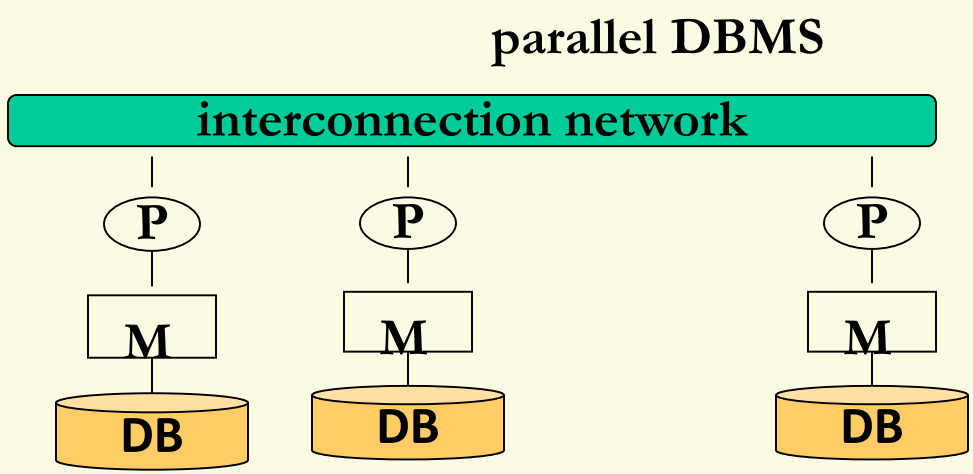
Performance of a database system

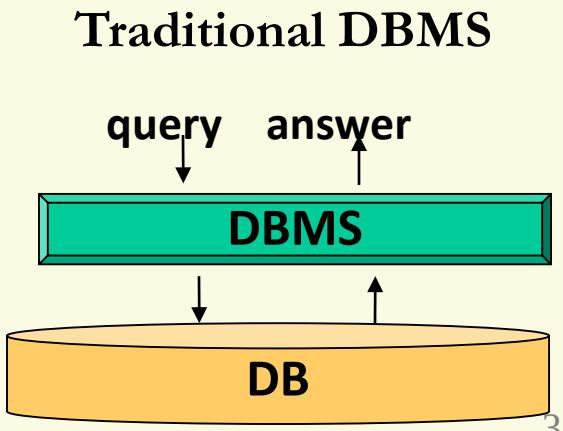
- ✓ Throughput: the number of tasks finished in a given time interval
- ✓ Response time: the amount of time to finish a single task from the time it is submitted

Can we do better given more resources (CPU, disk, …)?

Parallel DBMS: exploring parallelism

- ✓ Divide a big problem into many smaller ones to be solved in parallel
- ✓ improve performance

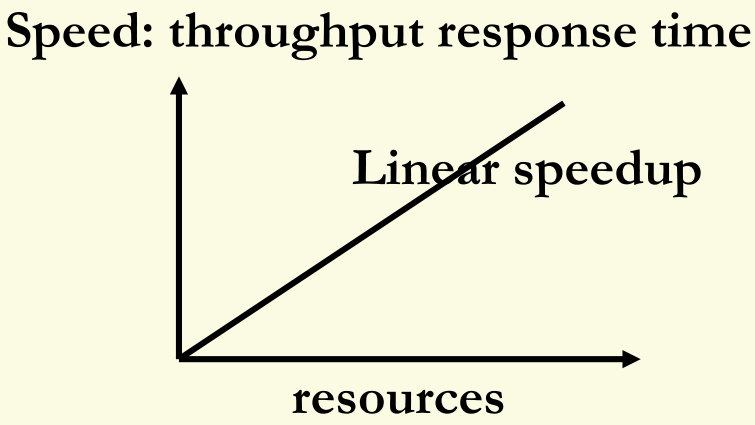




Degree of parallelism -- speedup

Speedup: for a given task, TS/TL,

- ✓ TS: time taken by a traditional DBMS
- ✓ TL: time taken by a parallel DBMS with more resources
- ✓ TS/TL: more sources mean proportionally less time for a task
- ✓ Linear speedup: the speedup is N while the parallel system has N times resources of the traditional system

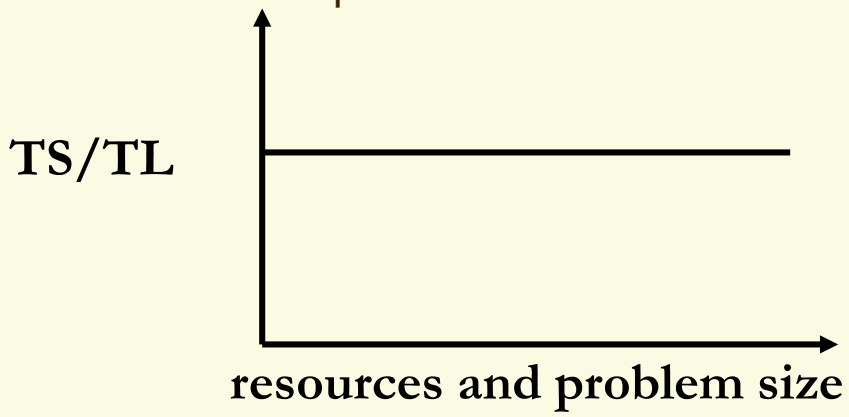


Question: can we do better than linear speedup?

Degree of parallelism -- scaleup

Scaleup: TS/TL; factor that expresses how much more work can be done in the same time period by a larger system

- A task Q, and a task Q_N, N times bigger than Q
- A DBMS M_S, and a parallel DBMS M_L,N times larger
- TS: time taken by M_S to execute Q
- TL: time taken by M_L to execute Q_N
- Linear scaleup: if TL = TS, i.e., the time is constant if the resource increases in proportion to increase in problem size



Why can't it be better than linear scaleup/speedup?

- ✓ Startup costs: initializing each process
- ✓ Interference: competing for shared resources (network, disk, memory or even locks)
- ✓ Skew: it is difficult to divide a task into exactly equal-sized parts; the response time is determined by the largest part
- ✓ Amdahl's law
  - Maximum speedup <= 1/(f+(1-f)/s)
  - f: "sequential fraction" of the program
  - F=0.1; s=10 -> speedup<= 5.26

Question: the more processors, the faster?

How can we leverage multiple processors and improve speedup?

Why parallel DBMS?

- ✓ Improve performance:

Almost died 20 years ago; with renewed interests because

  - Big data -- data collected from the Web
  - Decision support queries -- costly on large data
  - Hardware has become much cheaper

- ✓ Improve reliability and availability: when one processor goes down

Renewed interest: MapReduce

Parallel Database Management Systems

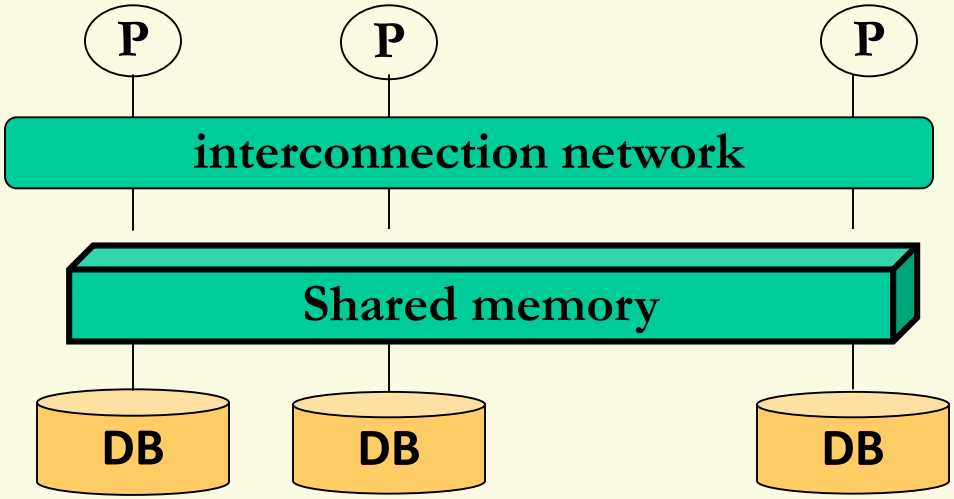
- ✓ Why parallel DBMS?
- ✓ Architectures
- ✓ Parallelism
  - Intraquery parallelism
  - Interquery parallelism
  - Intraoperation parallelism
  - Interoperation parallelism

Shared memory

A common memory

- ✓ Efficient communication: via data in memory, accessible by all
- ✓ Not scalable: shared memory and network become bottleneck -- interference; not scalable beyond 32 (or 64) processors
- ✓ Adding memory cache to each processor? Cache coherence problem when data is updated

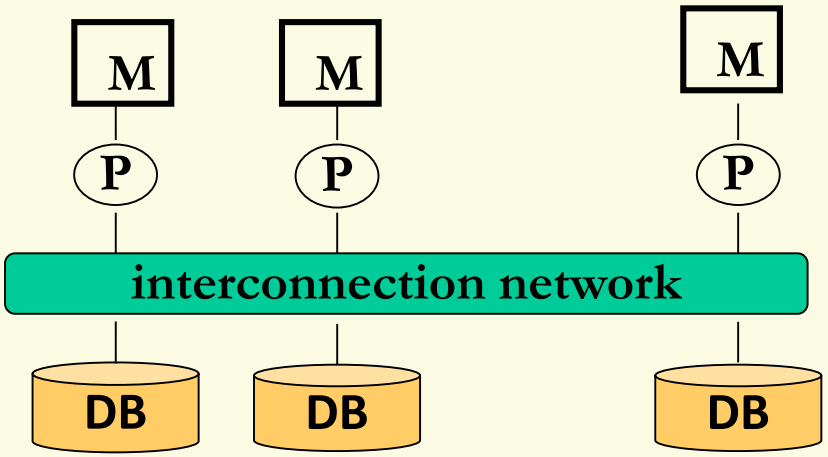
Informix (9 nodes)



Shared disk

- ✓ Fault tolerance: if a processor fails, the others can take over since the database is resident on disk
- ✓ scalability: better than shared memory -- memory is no longer a bottleneck; but disk subsystem is a bottleneck
  - interference: all I/O to go through a single network; not scalable beyond a couple of hundred processors

Oracle RDB (170 nodes)



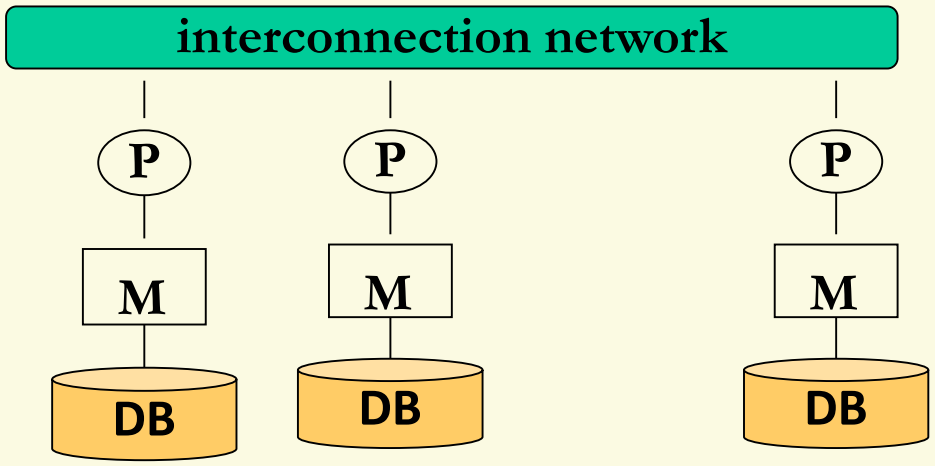
Shared nothing

- ✓ scalable: only queries and result relations pass through the network
- ✓ Communication costs and access to non-local disks: sending data involves software interaction at both ends

Teradata: 400 nodes

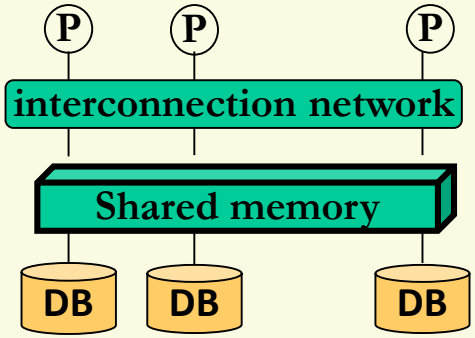
IBM SP2/DB2:128 nodes

Informix SP2: 48 nodes



overview

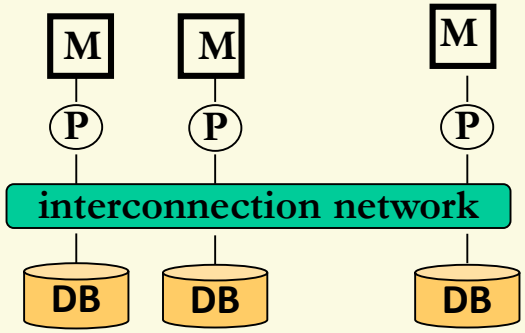
Shared Memory (SMP)



Easy to program
Expensive to build
Difficult to scaleup

Informix, RedBrick
Sequent, SGI, Sun
scale: 9 nodes

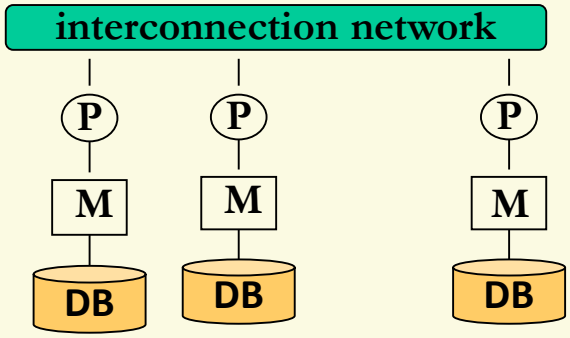
Shared Disk



Better scalability
Fault tolerance

VMScluster,
Oracle (170 nodes)
DEC Rdb (24 nodes)

Shared Nothing (network)



Hard to program
Cheap to build
Easy to scaleup

Teradata: 400 nodes
Tandem: 110 nodes
IBM / SP2 / DB2: 128 nodes
Informix/SP2 48 nodes
ATT & Sybase ? nodes

Architectures of Parallel DBMS

Shared nothing, shared disk, shared memory

Tradeoffs of

- ✓ Scalability
- ✓ Communication speed
- ✓ Cache coherence

MapReduce? 10,000 nodes

Shared-nothing has the best scalability

Parallel Database Management Systems

- ✓ Why parallel DBMS?
- ✓ Architectures
- ✓ Parallelism (II-ism)
- ✓ Parallelism hierarchy
  - Intraquery parallelism
  - Interquery parallelism
  - Intraoperation parallelism
  - Interoperation parallelism

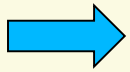
Pipelined II-sim

- The output of operation A is consumed by another operation B, before A has produced the entire output
  Many machines, each doing one step in a multi-step process

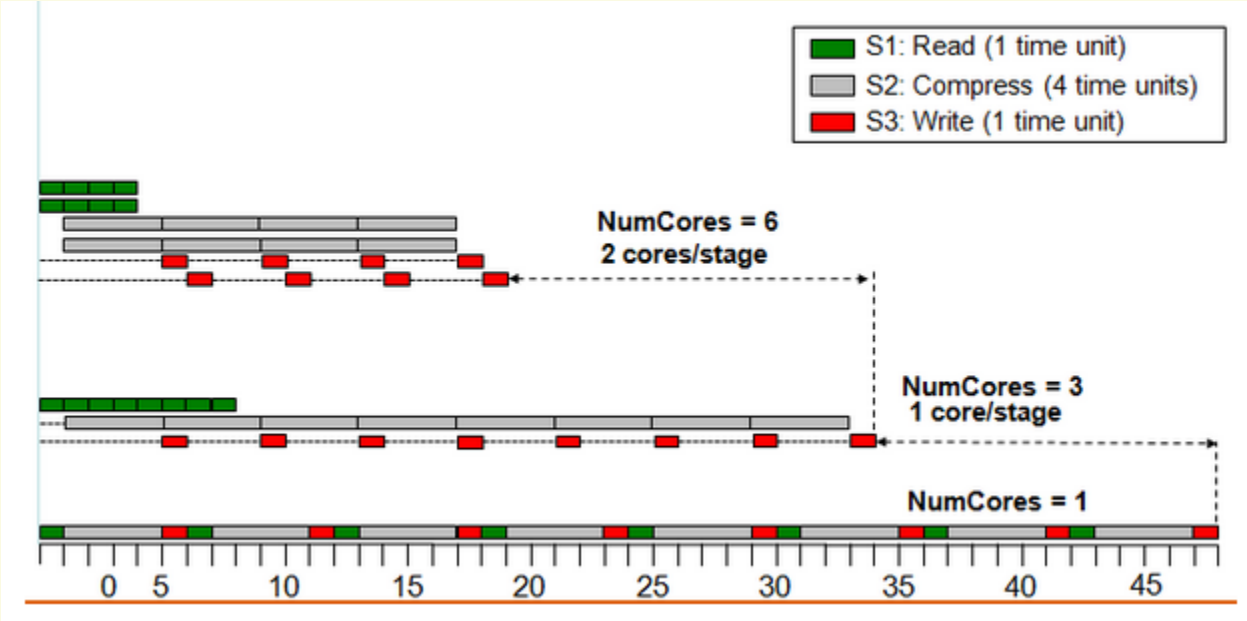
- Does not scale up well when:
  - the computation does not provide sufficiently long chain to provide a high degree of parallelism:
  - relational operators do not produce output until all inputs have been accessed – block, or
  - A's computation cost is much higher than that of B

Pipelined parallelism: compress a file

```
while(!done){
        Read block from file;
        Compress the block;
        Write block to file;
}
```



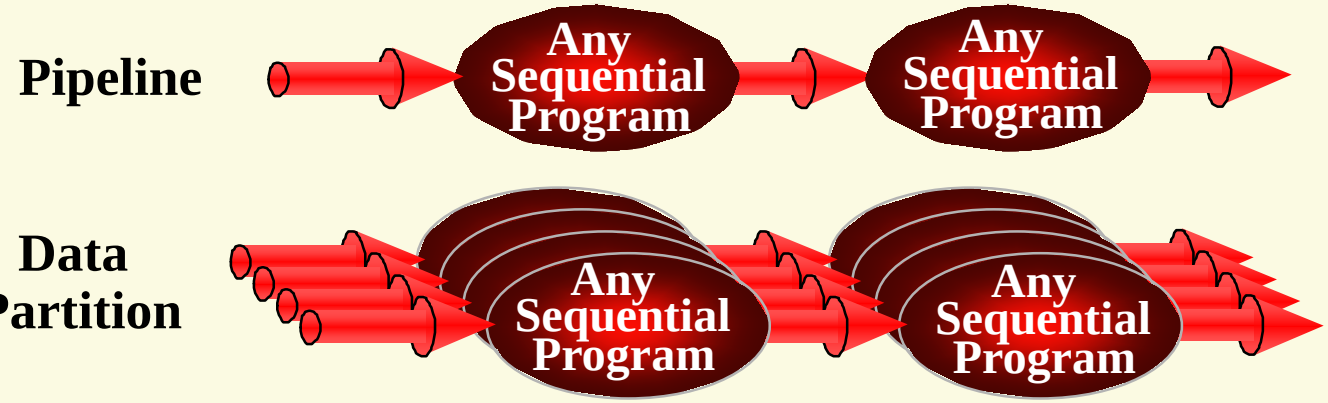
```
while(!program end){
        If work is available in the in-Queue of
        thread
        Compress the block;
        Write block to file;
}
```



Data Partitioned parallelism

- ✓ Many machines performing the same operation on different pieces of data

The parallelism behind MapReduce



Partitioning in RDBMS

Partition a relation and distribute it to different processors

- ✓ Maximize processing at each individual processor
- ✓ Minimize data shipping

Query types:

- ✓ scan a relation,
- ✓ point queries (r.A = v),
- ✓ range queries (v < r.A < v')

Partitioning strategies

N disks, a relation R

- ✓ Round-robin: send the *j*-th tuple of R to the disk number *j mod n*
  - Even distribution: good for scanning
  - Not good for equal joins (point queries) and range queries (all disks have to be involved for the search)

- ✓ Range partitioning: partitioning attribute A, vector $[v_1, \ldots, v_{n-1}]$
  - send tuple t to disk *j* if t[A] in $[v_{j-1}, v_j]$
  - good for point and range queries on partitioning attributes (using only a few disks, while leaving the others free)
  - Execution skew: distribution may not be even, and all operations occur in one or few partitions (scanning)

Partitioning strategies (cont.)

N disks, a relation R

- ✓ Hash partitioning: hash function f(t) in the range of [0, N-1]
  - Send tuple t to disk f(t)
  - good for point queries on partitioning attributes, and sequential scanning if the hash function is even
  - No good for point queries on non-partitioning attributes and range queries

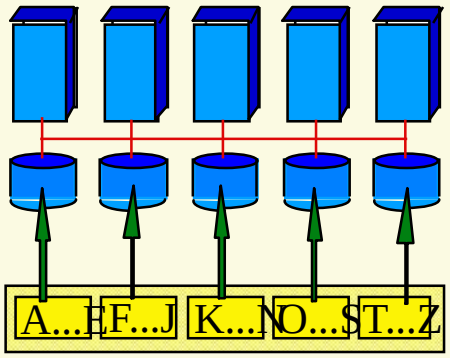
Question: how to partition R1(A, B): {(i, i+1)}, with 5 processors?

- ✓ Round-robin
- ✓ Range partitioning: partitioning attribute A
- ✓ Hash partitioning

Automatic Data Partitioning

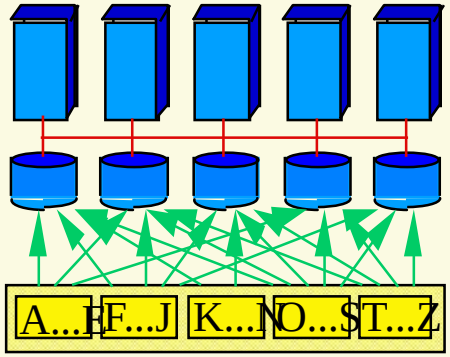
Partitioning a table:

Range



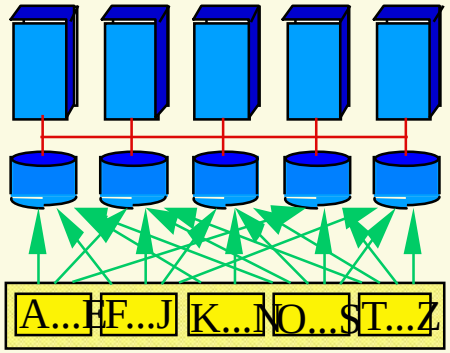
Good for group-by, range queries, and also equip-join

Hash



Good for equijoins

Round Robin



Good to spread load;
Most flexible
Not good for equi-join and range

Shared-disk and -memory less sensitive to partitioning,
Shared nothing benefits from "good" partitioning

Interquery vs. intraquery parallelism

- interquery: different queries or transactions execute in parallel
  - Improve transaction throughput
  - Easy on shared-memory: traditional DBMS tricks will do
  - Shared-nothing/disk: cache coherence problem

  Ensure that each processor has the latest version of the data in its buffer pool --flush updated pages to shared disk before releasing the lock

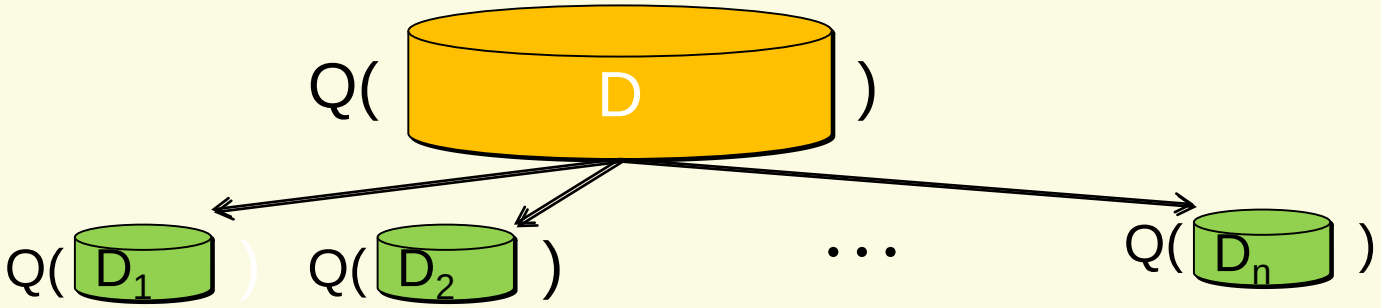
  - Oracle 8 and Oracle Rdb
- Intraquery: a single query in parallel on multiple processors
  - speed up single complex long running queries
  - Interoperation: operator tree
  - Intraoperation: parallelize the same operation on different sets of the same relations: Parallel sorting, Parallel join; Selection, projection, aggregation
  - Informix, Terradata

Parallel query answering

Given data D, and n processors S_1, …, S_n

- ✓ D is partitioned into fragments (D_1, …, D_n)
- ✓ D is distributed to n processors: D_i is stored at S_i

Each processor S_i processes operations for a query on its local fragment D_i, in parallel



Dividing big data into small fragments of manageable size

Relational operators

Recall relational operators.

- ✓ Projection: Π_A R
- ✓ Selection: σ_C R
- ✓ Join: R1 $\bowtie_C$ R2
- ✓ Union: R1 $\cup$ R2
- ✓ Set difference: R1 – R2
- ✓ Group by and aggregate (max, min, count, average)

How to support these operations in a parallel setting?

Intraoperation parallelism -- loading/projection

Π_A R, where R is partitioned across n processors

- ✓ Read tuples of R at all processors involved, in parallel
- ✓ Conduct projection on tuples
- ✓ Merge local results
  - – Duplicate elimination: via sorting

Intraoperation parallelism -- selection

σ_C R, where R is partitioned across n processors

If A is the partitioning attribute

- ✓ Point query: C is A = v
  - a single processor that holds A = v is involved
- ✓ Range query: C is v1 < A and A < v2
  - only processors whose partition overlaps with the range

If A is not the partitioning attribute:

- ✓ Compute σ_C Ri at each individual processor
- ✓ Merge local results

Question: evaluate $\sigma_{2<A \text{ and } A<6}$ R, R(A, B): {(1, 2), (3, 4), (5, 6), (7, 2), (9, 3)}, and R is range partitioned on B to 3 processors

Intraoperation parallelism -- parallel sort

sort R on attribute A, where R is partitioned across n processors

If A is the partitioning attribute: Range-partitioning

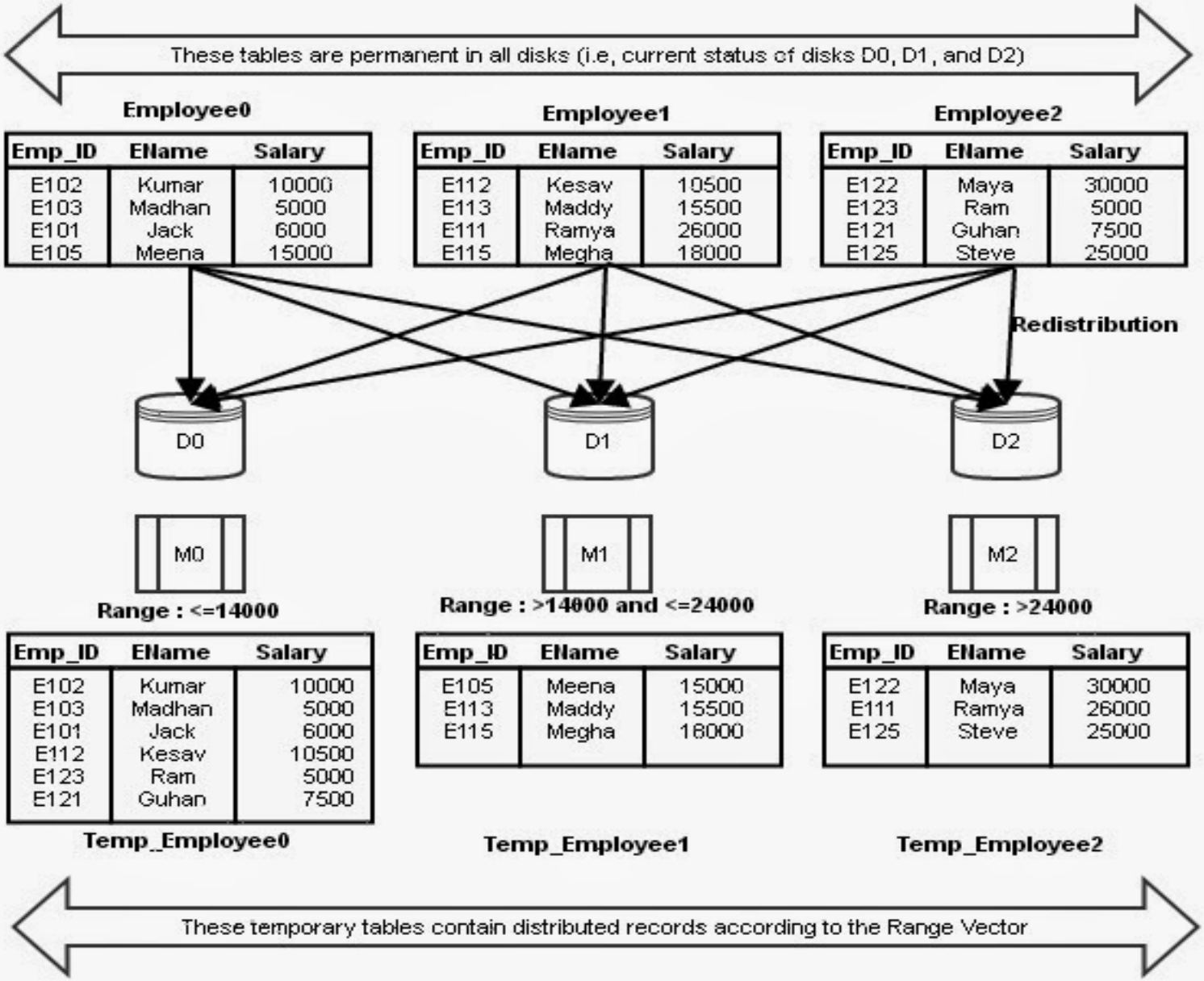
- ✓ Sort each partition
- ✓ Concatenate the results

If A is not the partitioning attribute: Range-partitioning sort

- ✓ Range partitioning R based on A: redistribute the tuples in R
  Every processor works in parallel: read tuples and send them to corresponding processors
- ✓ Each processor sorts its new partition locally when the tuples come in -- data parallelism
- ✓ Merge local results

Problem: skew

Solution: sample the data to determine the partitioning vector


These tables are permanent in all disks (i.e, current status of disks D0, D1, and D2)
Employee0
Emp_ID EName Salary
E102 Kumar 10000
E103 Madhan 5000
E101 Jack 6000
E105 Meena 15000
Employee1
Emp_ID EName Salary
E112 Kesav 10500
E113 Maddy 15500
E111 Ramya 26000
E115 Megha 18000
Employee2
Emp_ID EName Salary
E122 Maya 30000
E123 Ram 5000
E121 Guhan 7500
E125 Steve 25000
Redistribution
D0
D1
D2
M0
M1
M2
Range : <=14000
Range : >14000 and <=24000
Range : >24000
Emp_ID EName Salary
E102 Kumar 10000
E103 Madhan 5000
E101 Jack 6000
E112 Kesav 10500
E123 Ram 5000
E121 Guhan 7500
Temp_Employee0
Emp_ID EName Salary
E105 Meena 15000
E113 Maddy 15500
E115 Megha 18000
Temp_Employee1
Emp_ID EName Salary
E122 Maya 30000
E111 Ramya 26000
E125 Steve 25000
Temp_Employee2
These temporary tables contain distributed records according to the Range Vector

Intraoperation parallelism -- parallel join

$R1 \bowtie_C R2$

- ✓ Partitioned join: for equi-joins and natural joins
- ✓ Fragment-and replication join: inequality
- ✓ Partitioned parallel hash-join: equal or natural join
  - where R1, R2 are too large to fit in memory
  - Almost always the winner for equi-joins

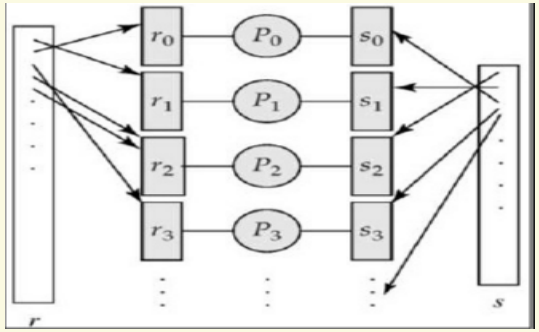
Partitioned join

$R1 \bowtie_{R1.A = R2.B} R2$

- ✓ Partition R1 and R2 into n partitions, by the same partitioning function in R1.A and R2.B, via either
  - range partitioning, or
  - hash partitioning
- ✓ Compute $R^i1 \bowtie_{R1.A = R2.B} R^i2$ locally at processor i
- ✓ Merge the local results

Question: how to perform partitioned join on the following, with 2 processors?

- ✓ R1(A, B): {(1, 2), (3, 4), (5, 6)}
- ✓ R2(B, C): {(2, 3), {3, 4)}



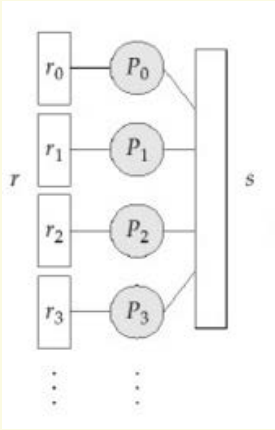
Fragment and replicate join

$R1 \bowtie_{R1.A < R2.B} R2$

- ✓ Partition R1 into n partitions, by any partitioning method, and distribute it across n processors
- ✓ Replicate the other relation R2 across all processors
- ✓ Compute $R^j1 \bowtie_{R1.A < R2.B} R2$ locally at processor j
- ✓ Merge the local results

Question: how to perform fragment and replicate join on the following, with 2 processors?

- ✓ R1(A, B): {(1, 2), (3, 4), (5, 6)}
- ✓ R2(B, C): {(2, 3), {3, 4)}



Partitioned parallel hash join

R1 $\bowtie_{R1.A = R2.B}$ R2, where R1, R2 are too large to fit in memory

- ✓ Hash partitioning R1 and R2 using hash function h1 on partitioning attributes A and B, leading to k partitions
- ✓ For i in [1, k], process the join of i-th partition $R^i1 \bowtie R^i2$ in turn, one by one in parallel
  - Hash partitioning R^i1 using a second hash function h2 , build in-memory hash table (assume R1 is smaller)
  - Hash partitioning R^i2 using the same hash function h2
  - When R2 tuples arrive, do local join by probing the in-memory table of R1

Break a large join into smaller ones

Intraoperation parallelism -- aggregation

Aggregate on the attribute B of R, grouping on A

- ✓ decomposition
  - count(S) = Σ count(Si); similarly for sum
  - avg(S) = (Σ sum(Si) / Σ count(Si))

Strategy:

- ✓ Range partitioning R based on A: redistribute the tuples in R
- ✓ Each processor computes sub-aggregate -- data parallelism
- ✓ Merge local results as above

Alternatively:

- ✓ Each processor computes sub-aggregate -- data parallelism
- ✓ Range partitioning local results based on A: redistribute partial results
- ✓ Compose the local results

Aggregation -- exercise

Describe a good processing strategy to parallelize the query:

```
select   branch-name, avg(balance)
from     account
group by     branch-name
```

where the schema of account is

(account-id, branch-name, balance)

Assume that n processors are available

Aggregation -- answer

Describe a good processing strategy to parallelize the query:

```
select   branch-name, avg(balance)
from     account
group by     branch-name
```

- ✓ Range or hash partition account by using branch-name as the partitioning attribute. This creates table account_j at each site j.
- ✓ At each site j, compute sum(account_j) / count(account_j);
- ✓ output sum(account_j) / count(account_j) – the union of these partial results is the final query answer

interoperation parallelism

Execute different operations in a single query in parallel

Consider R1 ⋈ R2 ⋈ R3 ⋈ R4

- ✓ Pipelined:
  - temp1 ← R1 ⋈ R2
  - temp2 ← R3 ⋈ temp1
  - result ← R4 ⋈ temp2
- ✓ Independent:
  - temp1 ← R1 ⋈ R2
  - temp2 ← R3 ⋈ R4
  - result ← temp1 ⋈ temp2 -- pipelining

Cost model

- ✓ Cost model: partitioning, skew, resource contention, scheduling
  - Partitioning: Tpart
  - Cost of assembling local answers: Tasm
  - Skew: max(T0, …, Tn)
  - Estimation: Tpart + Tasm + max(T0, …, Tn)

  May also include startup costs and contention for resources (in each Tj)
- ✓ Query optimization: find the “best” parallel query plan
  - Heuristic 1: parallelize all operations across all processors -- partitioning, cost estimation (Teradata)
  - Heuristic 2: best sequential plan, and parallelize operations -- partition, skew, … (Volcano parallel machine)

Practice: validation of functional dependencies

- A functional dependency (FD) defined on schema R: $X \rightarrow Y$
  - For any instance D of R, D satisfies the FD if for any pair of tuples t and t', if t[X] = t'[X], then t[Y] = t'[Y]
  - Violations of the FD in D:

    { t | there exists t' in D, such that t[X] = t'[X], but $t[Y] \neq t'[Y]$ }
- Develop a parallel algorithm that given D and an FD, computes all the violations of the FD in D
  - Partitioned join
  - Partitioned and replication join

Which one works better?

- Questions: what can we do if we are given a set of FDs to validate?

Something for you to take home and think about

Practice: implement set difference

- Set difference: R1 – R2
- Develop a parallel algorithm that R1 and R2, computes R1 – R2, by using:
  - partitioned join
  - partitioned and replicated
- Questions: what can we do if the relations are too large to fit in memory?

Is it true that the more processors are used, the faster the computation is?

MapReduce

MapReduce

- ✓ A programming model with two primitive functions:

Google

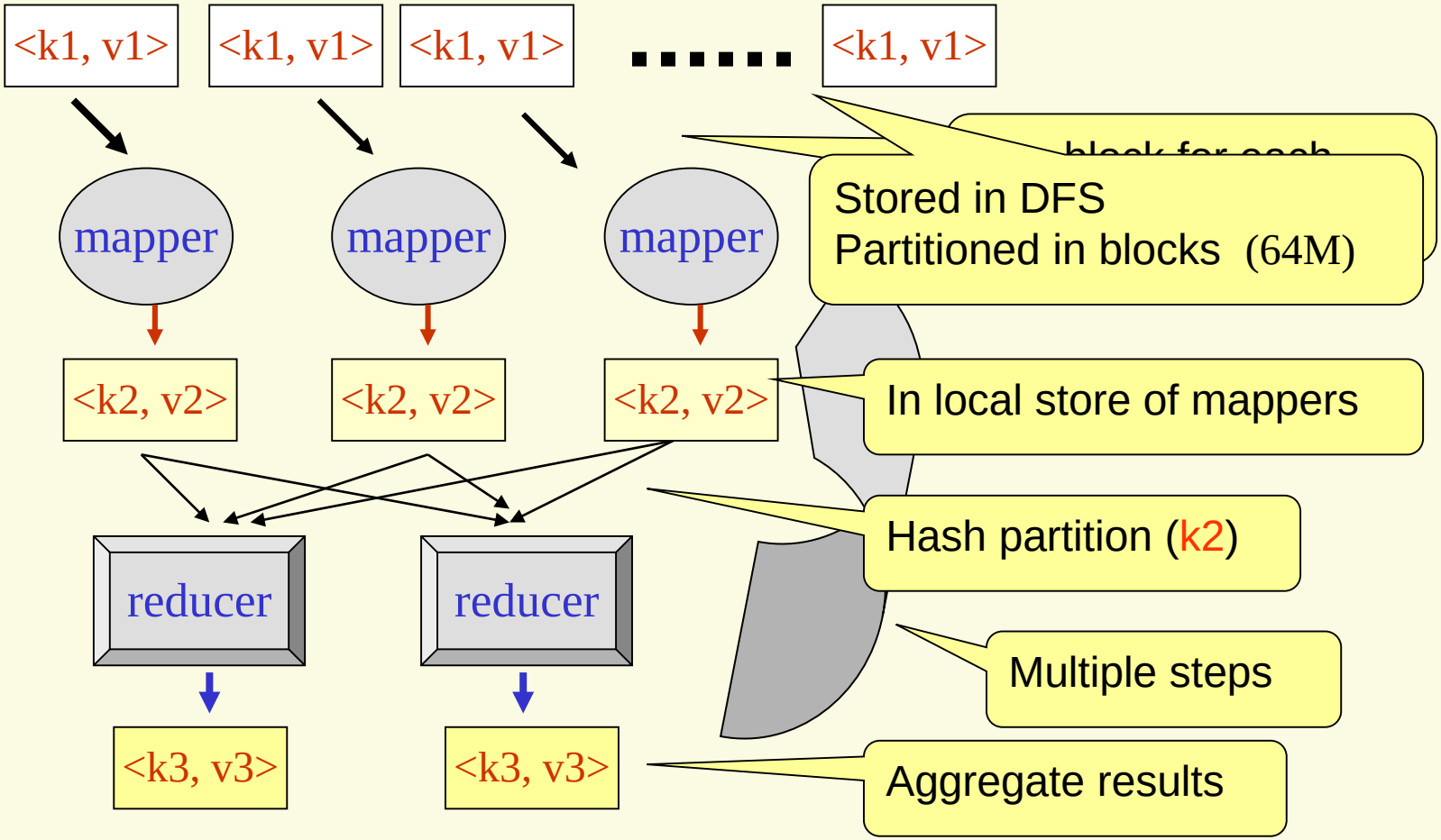
- ✓ Map: <k1, v1> → list (k2, v2)
- ✓ Reduce: <k2, list(v2)> → list (k3, v3)

- ✓ Input: a list <k1, v1> of key-value pairs
- ✓ Map: applied to each pair, computes key-value pairs <k2, v2>
  - The intermediate key-value pairs are hash-partitioned based on k2. Each partition (k2, list(v2)) is sent to a reducer
- ✓ Reduce: takes a partition as input, and computes key-value pairs <k3, v3>

The process may reiterate – multiple map/reduce steps

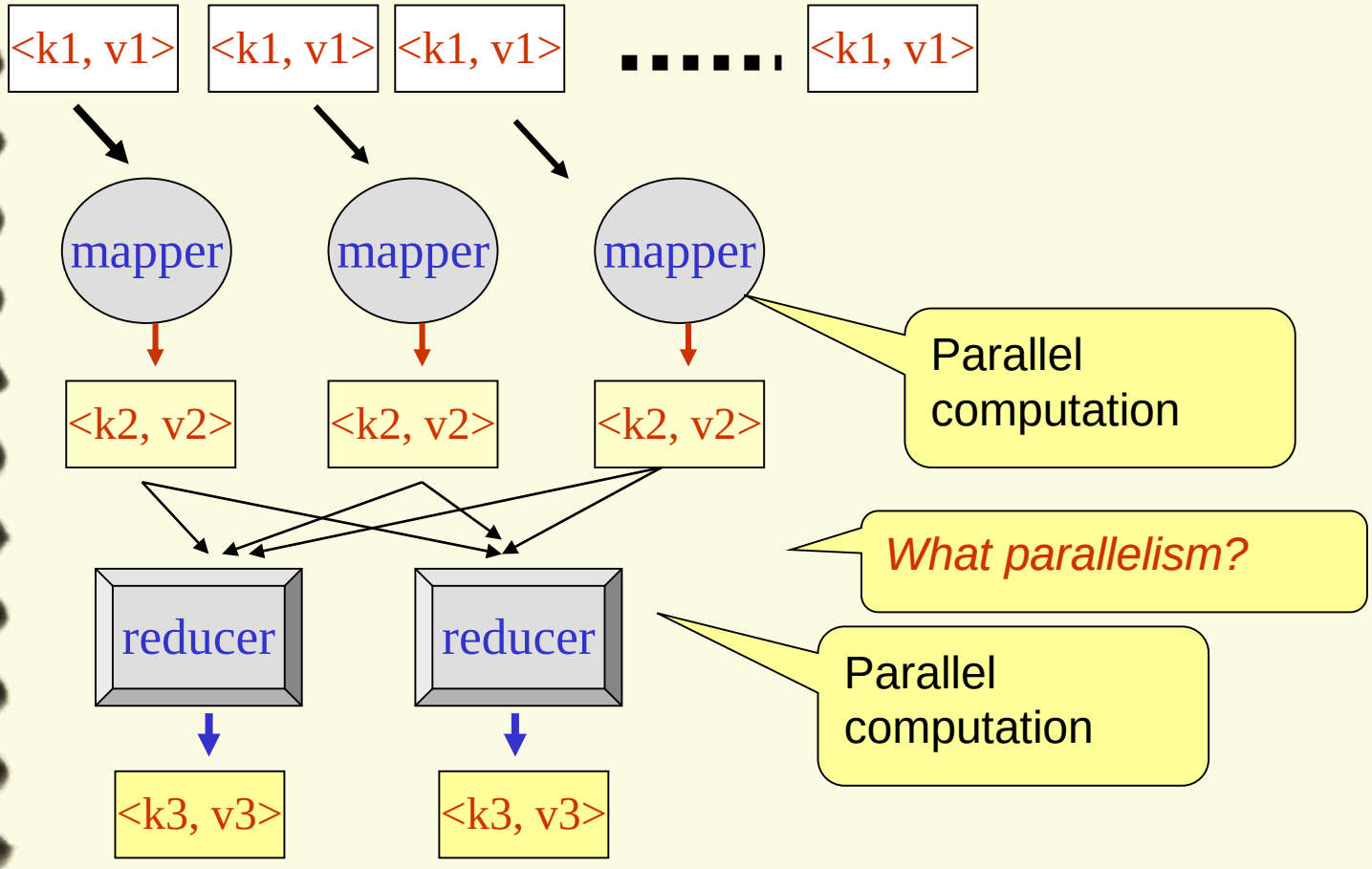
How doe

Architecture (Hadoop)



No need to worry about how the data is stored and sent

parallelism



Data partitioned parallelism

Popular in industry

- ✓ Apache Hadoop, used by Facebook, Yahoo, …
  - Hive, Facebook, HiveQL (SQL)
  - PIG, Yahoo, Pig Latin (SQL like)
  - SCOPE, Microsoft, SQL Cassandra, Facebook, CQL (no join)
  - HBase, Google, distributed BigTable
  - MongoDB, document-oriented (NoSQL)
- ✓ Scalability
  - Yahoo!: 10,000 cores, for Web search queries (2008)
  - Facebook: 100 PB, about half a PB per day
  - Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3); New York Time used 100 EC2 instances to process 4TB of image data, $240

Study Spark: https://spark.apache.org/

Advantages of MapReduce

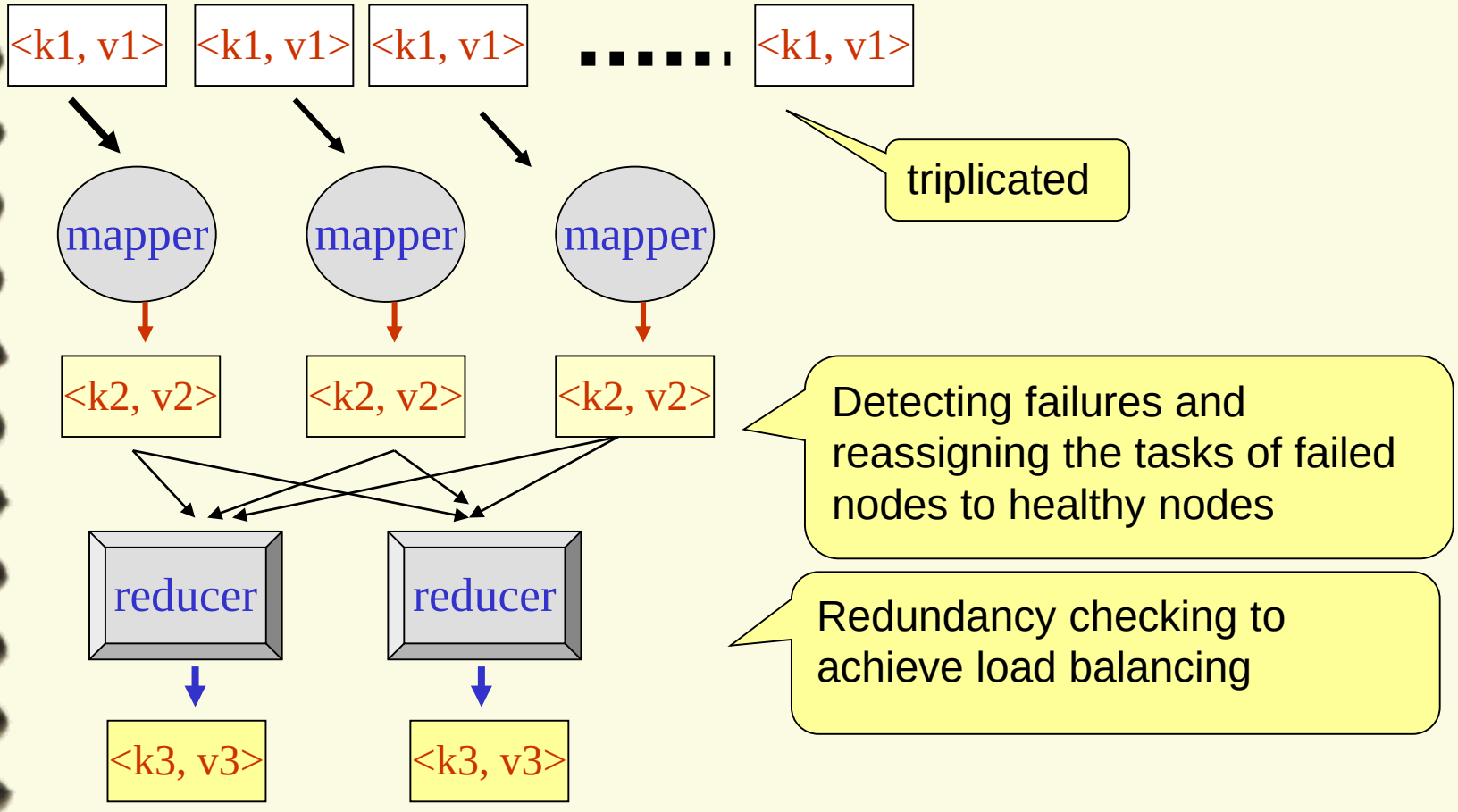
- ✓ Simple: one only needs to define two functions

no need to worry about how the data is stored, distributed and how the operations are scheduled

- ✓ scalability: a large number of low end machines
  - scale out (scale horizontally): adding a new computer to a distributed software application; lost-cost "commodity"
  - scale up (scale vertically): upgrade, add (costly) resources to a single node
- ✓ independence: it can work with various storage layers
- ✓ flexibility: independent of data models or schema

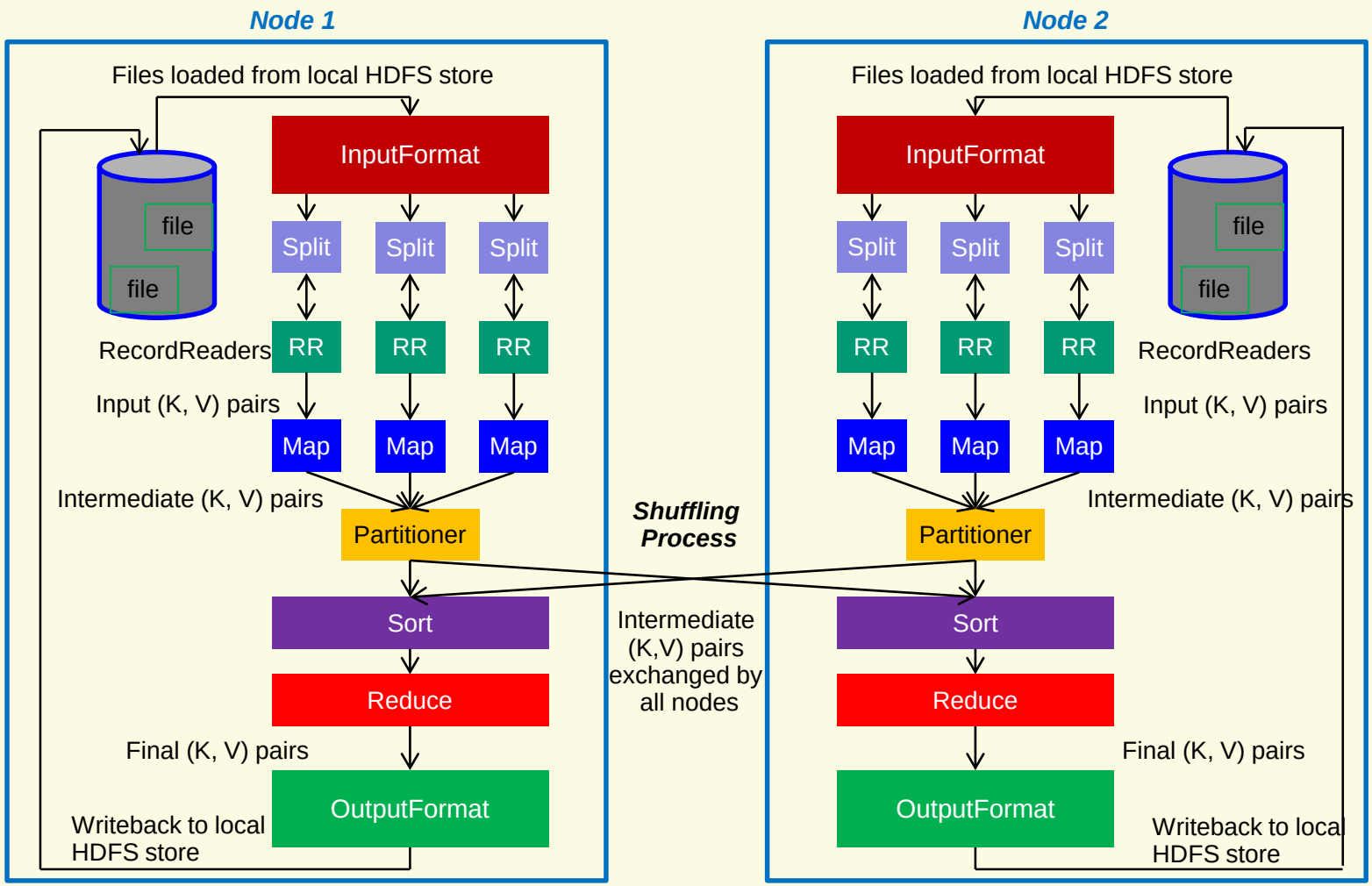
Fault tolerance: why?

Fault tolerance



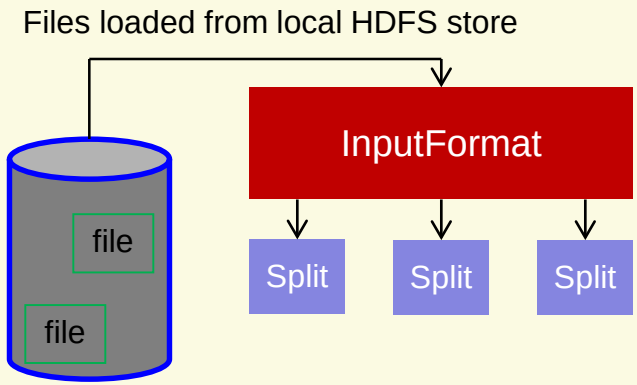
Able to handle an average of 1.2 failures per analysis job

Hadoop MapReduce: A Closer Look



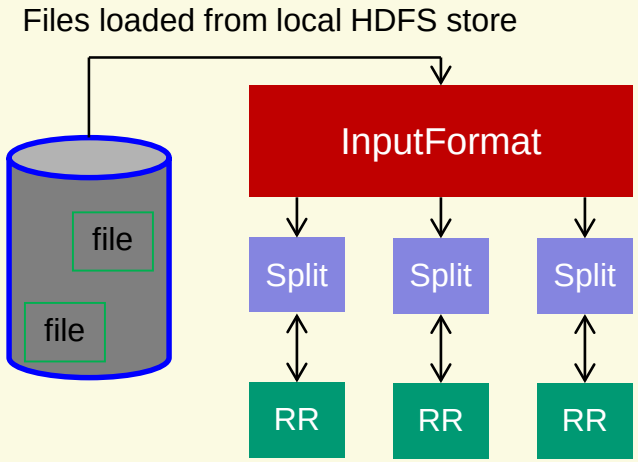
Input Splits

- An input split describes a unit of work that comprises a single map task in a MapReduce program
- By default, the InputFormat breaks a file up into 64MB splits
- By dividing the file into splits, we allow several map tasks to operate on a single file in parallel
- If the file is very large, this can improve performance significantly through parallelism
- Each map task corresponds to a single input split



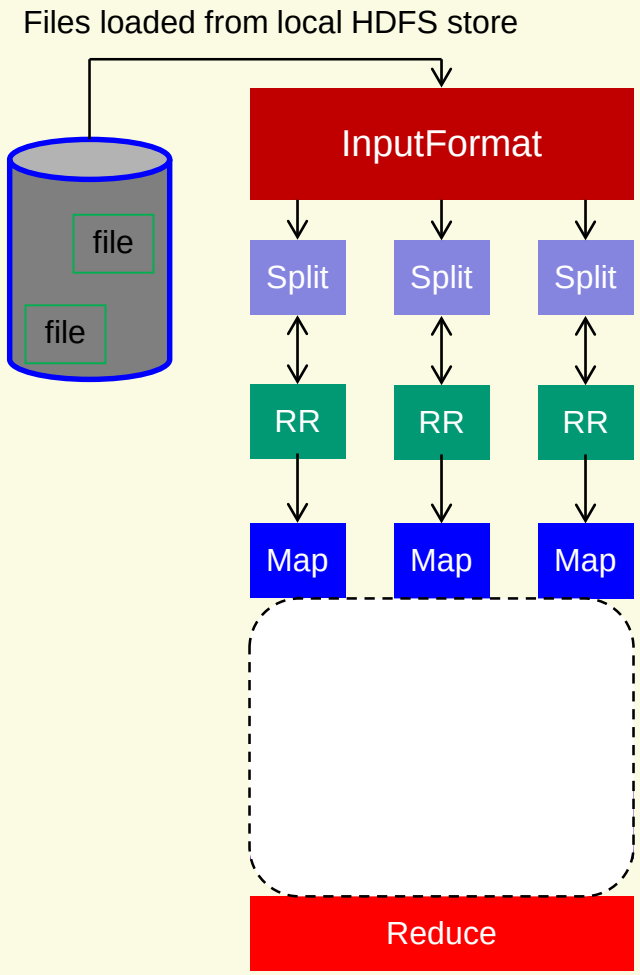
RecordReader

- The input split defines a slice of work but does not describe how to access it
- The RecordReader class actually loads data from its source and converts it into (K, V) pairs suitable for reading by Mappers
- The RecordReader is invoked repeatedly on the input until the entire split is consumed
- Each invocation of the RecordReader leads to another call of the map function defined by the programmer



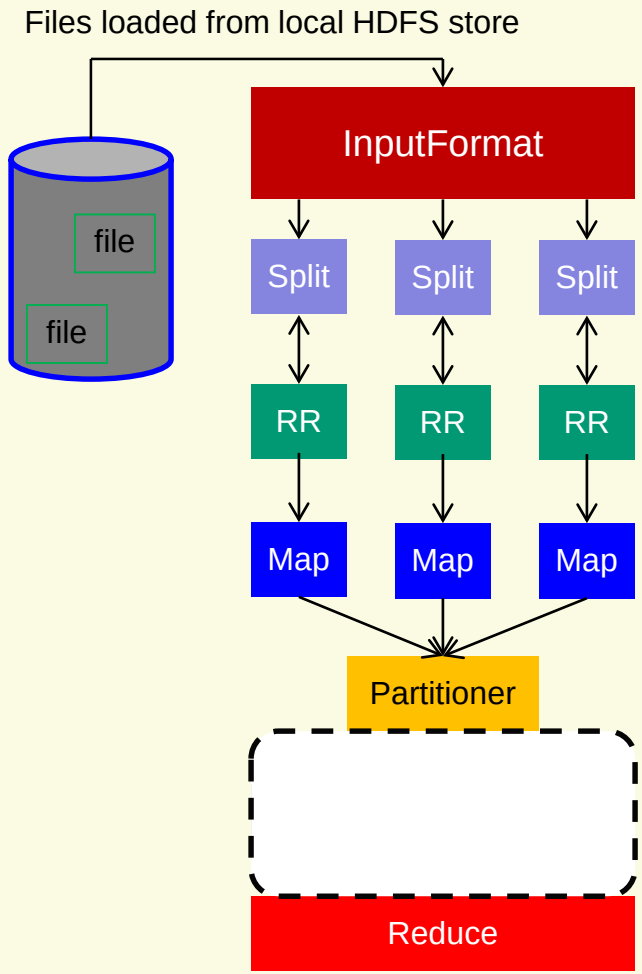
Mapper and Reducer

- The Mapper performs the user-defined work of the first phase of the MapReduce program
- A new instance of Mapper is created for each split
- The Reducer performs the user-defined work of the second phase of the MapReduce program
- A new instance of Reducer is created for each partition
- For each key in the partition assigned to a Reducer, the Reducer is called once



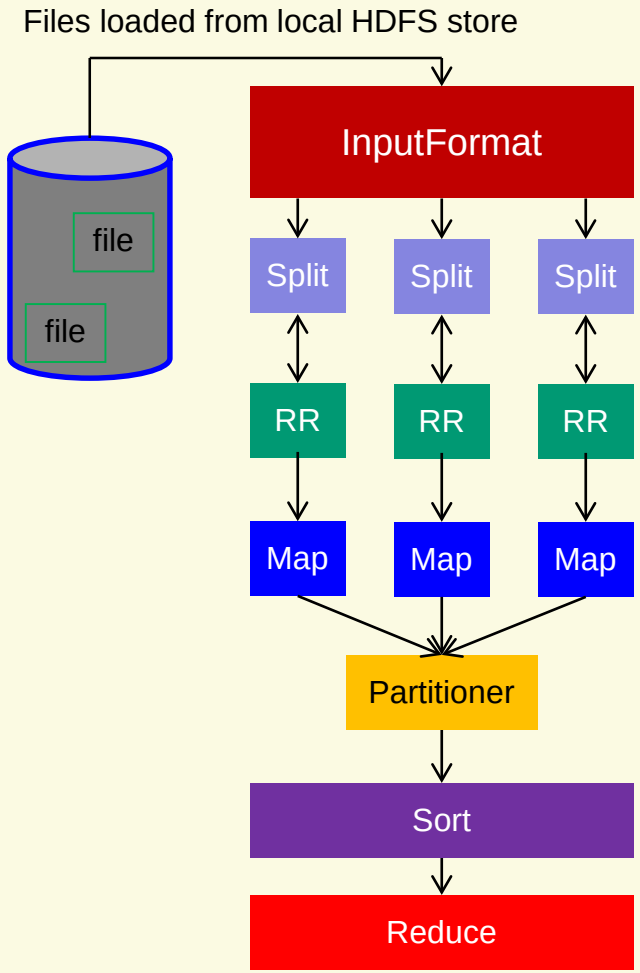
Partitioner

- Each mapper may emit (K, V) pairs to any partition
- Therefore, the map nodes must all agree on where to send different pieces of intermediate data
- The partitioner class determines which partition a given (K,V) pair will go to
- The default partitioner computes a hash value for a given key and assigns it to a partition based on this result



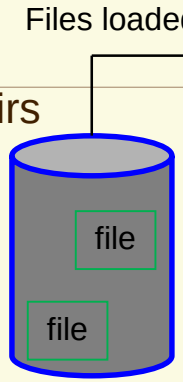
Sort

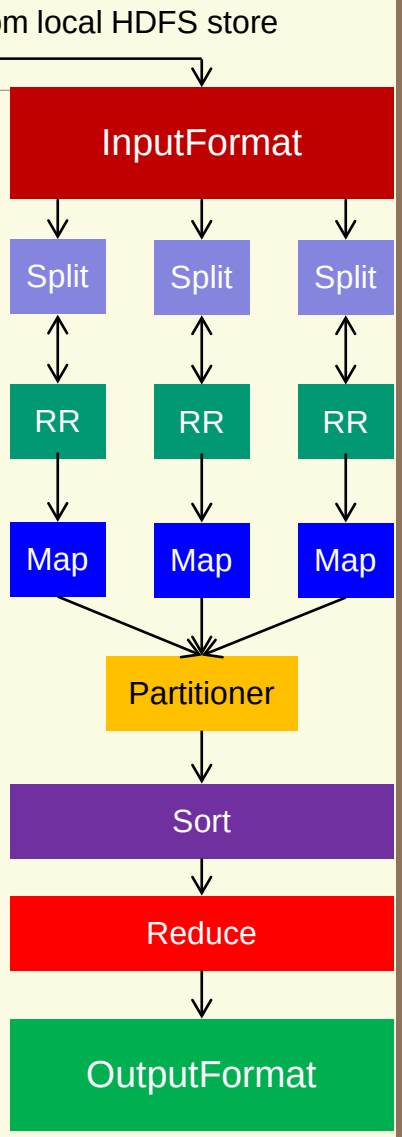
- Each Reducer is responsible for reducing the values associated with (several) intermediate keys
- The set of intermediate keys on a single node is automatically sorted by MapReduce before they are presented to the Reducer



OutputFormat

- The OutputFormat class defines the way (K,V) pairs produced by Reducers are written to output files
- The instances of OutputFormat provided by Hadoop write to files on the local disk or in HDFS
- Several OutputFormats are provided by Hadoop:





OutputFormat	Description
TextOutputFormat	Default; writes lines in "key \t value" format
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Generates no output files

Word count

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

Summary and review

- ✓ What is linear speedup? Linear scaleup? Skew?
- ✓ What are the three basic architectures of parallel DBMS? What is interference? Cache coherence? Compare the three
- ✓ Describe main partitioning strategies, and their pros and cons
- ✓ Compare and practice pipelined parallelism and data partition parallelism
- ✓ What is interquery parallelism? Intraquery?
- ✓ Parallelizing a binary operation (e.g., join)
  - Partitioned
  - Fragment and replicate
- ✓ Parallel hash join
- ✓ Parallel selection, projection, aggregate, sort

Reading list

- ✓ Read distributed database systems, before the next lecture
  - –Database Management Systems, 2nd edition, R. Ramakrishnan and J. Gehrke, Chapter 22.
  - –Database System Concept, 4th edition, A. Silberschatz, H. Korth, S. Sudarshan, Part 6 (Parallel and Distributed Database Systems)
- ✓ Take a look at NoSQL databases, eg:
  - –http://en.wikipedia.org/wiki/NoSQL

What is the main difference between NoSQL and relational (parallel) databases?