Parallel Databases and Map Reduce

Introduction to Databases

CompSci 316 Spring 2017

DUKE
COMPUTER SCIENCE

Announcements (Wed., Apr. 17)

- Homework #4 due Monday, April 24, 11:55 pm
  - No other extra credit problem
- Project Presentations
  - A few project groups still have not signed up – please sign up soon
- Google Cloud code
  - Please redeem your code asap, by May 11
- Wednesday April 19
  - Guest Lecture by Prof. Jun Yang
  - OLAP, Data warehousing, Data mining
  - Included in the final

Where are we now?

We learnt

- ✓ Relational Model, Query Languages, and Database Design
  - ✓ SQL
  - ✓ RA
  - ✓ E/R diagram
  - ✓ Normalization
- ✓ DBMS Internals
  - ✓ Storage
  - ✓ Indexing
  - ✓ Query Evaluation
  - ✓ External sort
  - ✓ Join Algorithms
  - ✓ Query Optimization
- ✓ Transactions
  - ✓ Basic concepts and SQL
  - ✓ Concurrency control
  - ✓ Recovery
- ✓ XML
  - ✓ DTD and XML schema
  - ✓ XPath and XQuery
  - ✓ Relational Mapping

Next

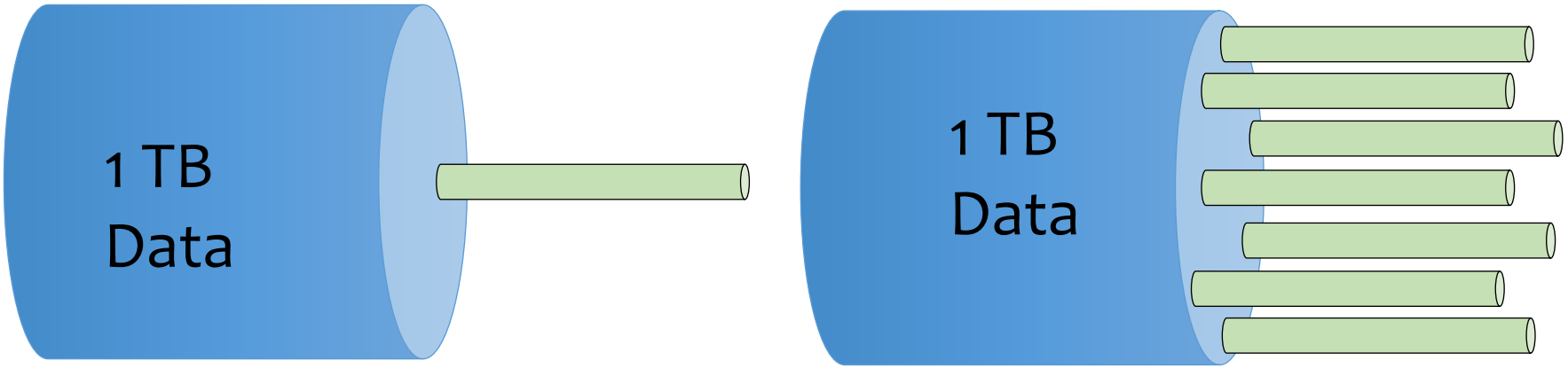
- Parallel DBMS — Today
- Map Reduce — Today
- Data Warehousing, OLAP, mining — Wednesday
- Distributed DBMS — Next Monday
- NOSQL — Next Monday

Parallel DBMS

Why Parallel Access To Data?

At 10 MB/s
1.2 days to scan

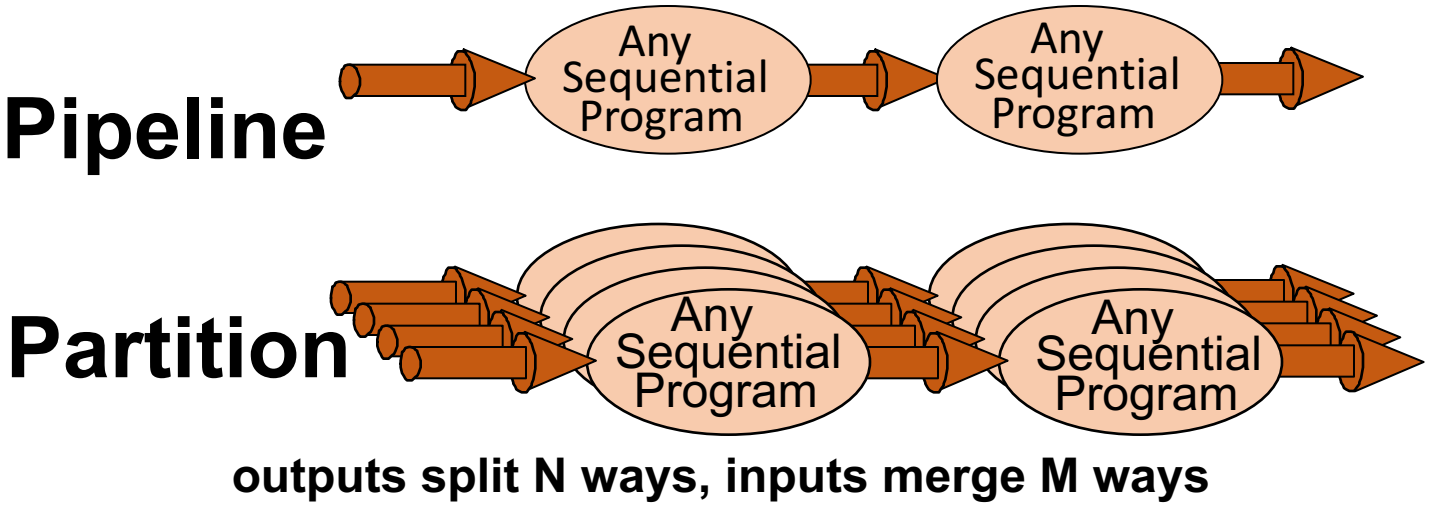
1,000 x parallel
1.5 minute to scan.



Parallelism:
divide a big problem
into many smaller ones
to be solved in parallel.

Parallel DBMS

- Parallelism is natural to DBMS processing
  - Pipeline parallelism: many machines each doing one step in a multi-step process.
  - Data-partitioned parallelism: many machines doing the same thing to different pieces of data.
  - Both are natural in DBMS!



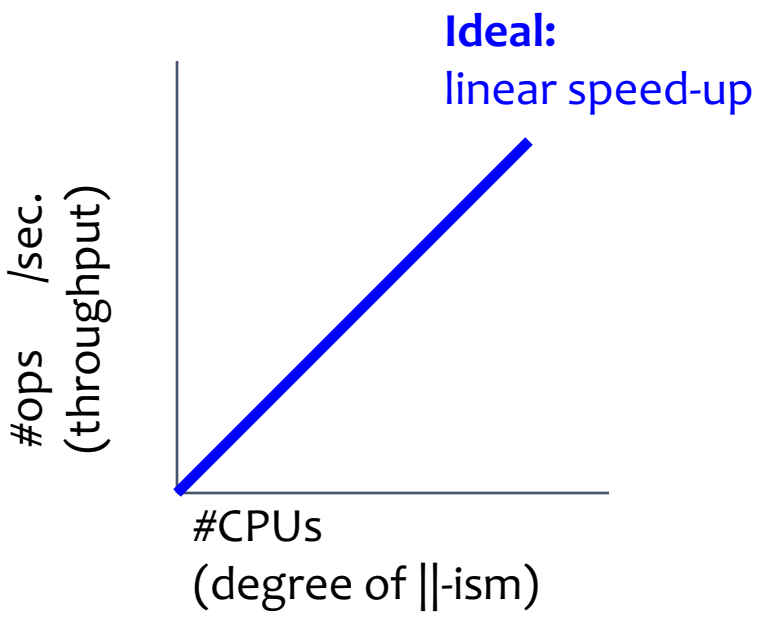
DBMS: The parallel Success Story

- DBMSs are the most successful application of parallelism
  - Teradata (1979), Tandem (1974, later acquired by HP),..
  - Every major DBMS vendor has some parallel server
- Reasons for success:
  - Bulk-processing (= partition parallelism)
  - Natural pipelining
  - Inexpensive hardware can do the trick
  - Users/app-programmers don't need to think in parallel

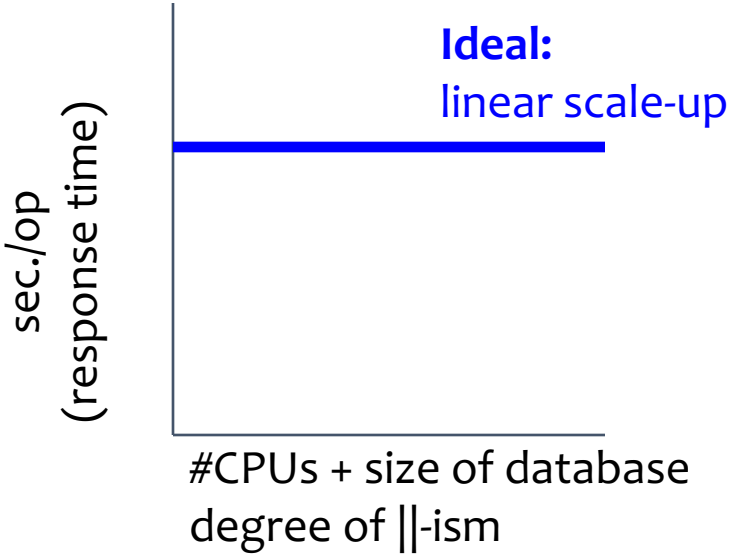
Some || Terminology

Ideal graphs

- Speed-Up
  - More resources means proportionally less time for given amount of data.



- Scale-Up
  - If resources increased in proportion to increase in data size, time is constant.



Some || Terminology

In practice

- Due to overhead in parallel processing

- Start-up cost

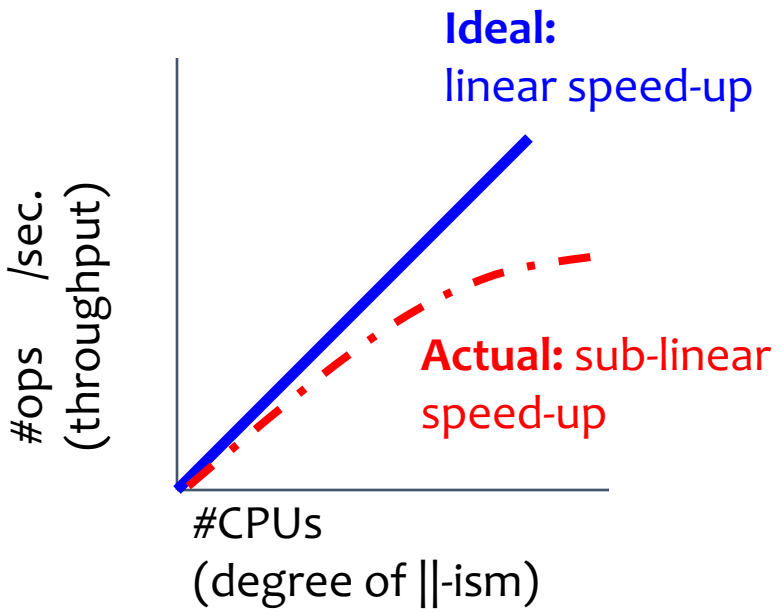
Starting the operation on many processor, might need to distribute data

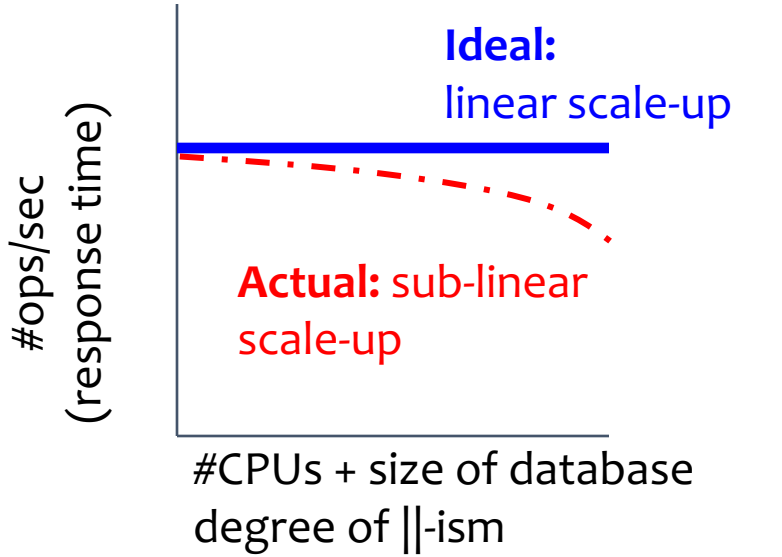
- Interference

Different processors may compete for the same resources

- Skew

The slowest processor (e.g. with a huge fraction of data) may become the bottleneck





Architecture for Parallel DBMS

- Units: a collection of processors
  - assume always have local cache
  - may or may not have local memory or disk (next)

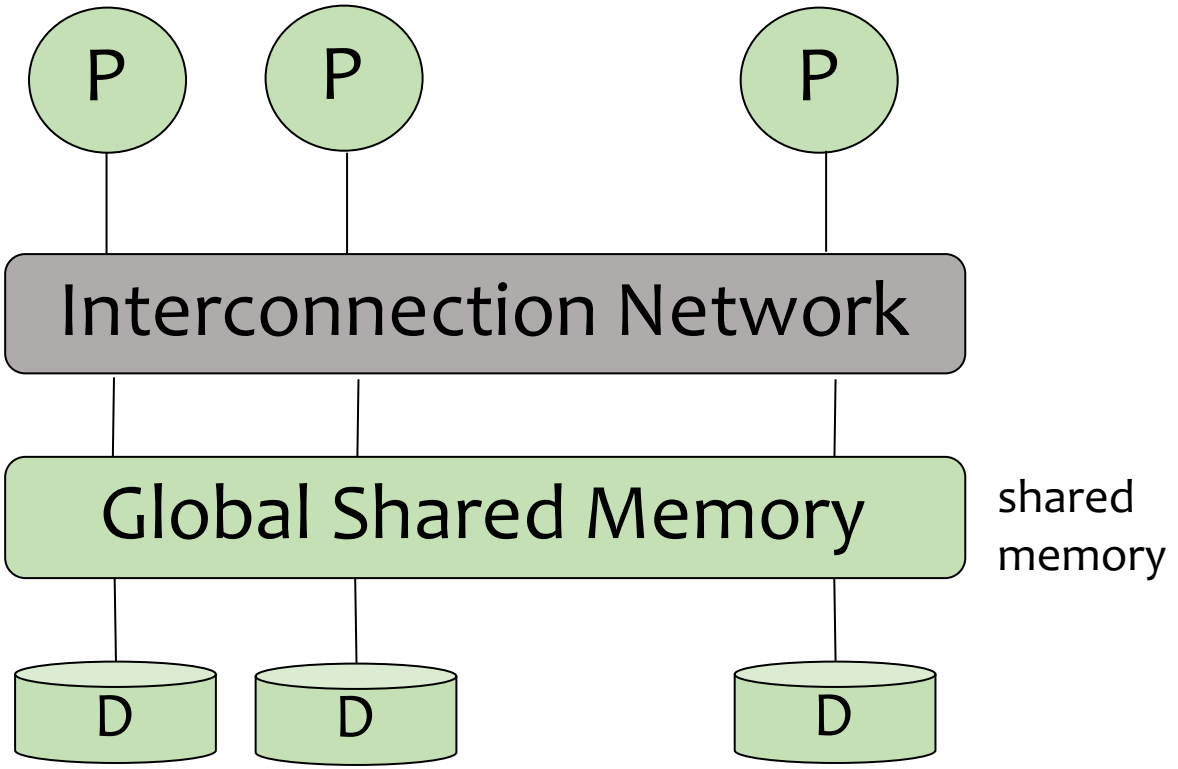
- A communication facility to pass information among processors
  - a shared bus or a switch

- Among different computing units
  - Whether memory is shared
  - Whether disk is shared

Shared Memory

e.g. SMP Server

- Easy to program
- Expensive to build
- Low communication overhead: shared memory
- Difficult to scaleup (memory contention)



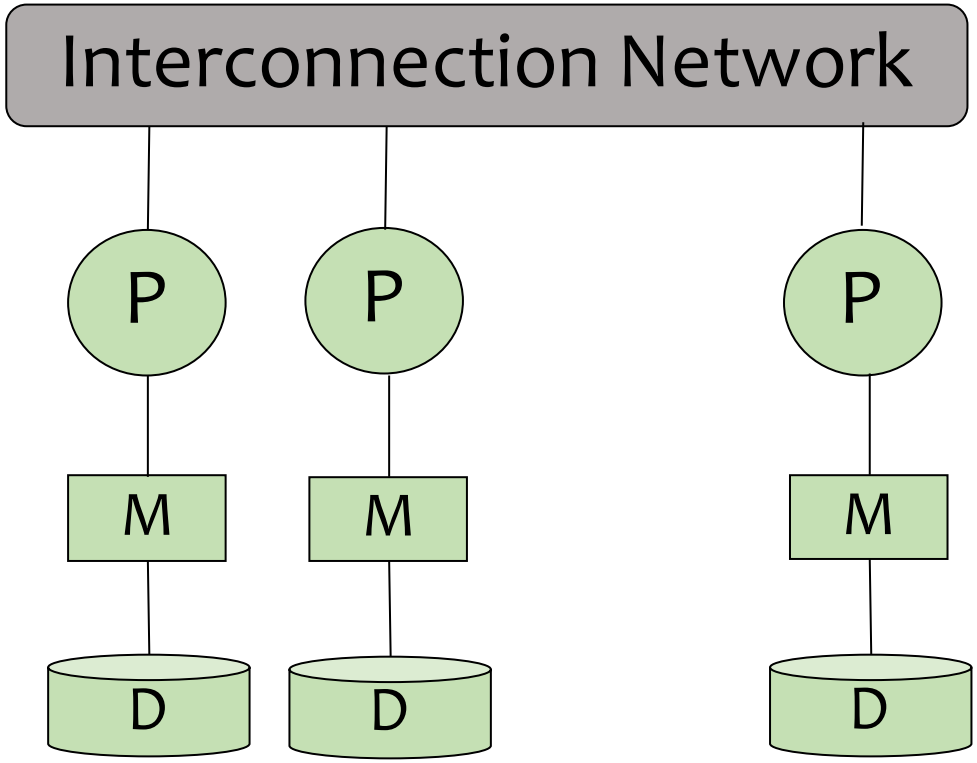
Shared Nothing

e.g. Greenplum

local memory and disk

no two CPU can access the same storage area

all communication through a network connection

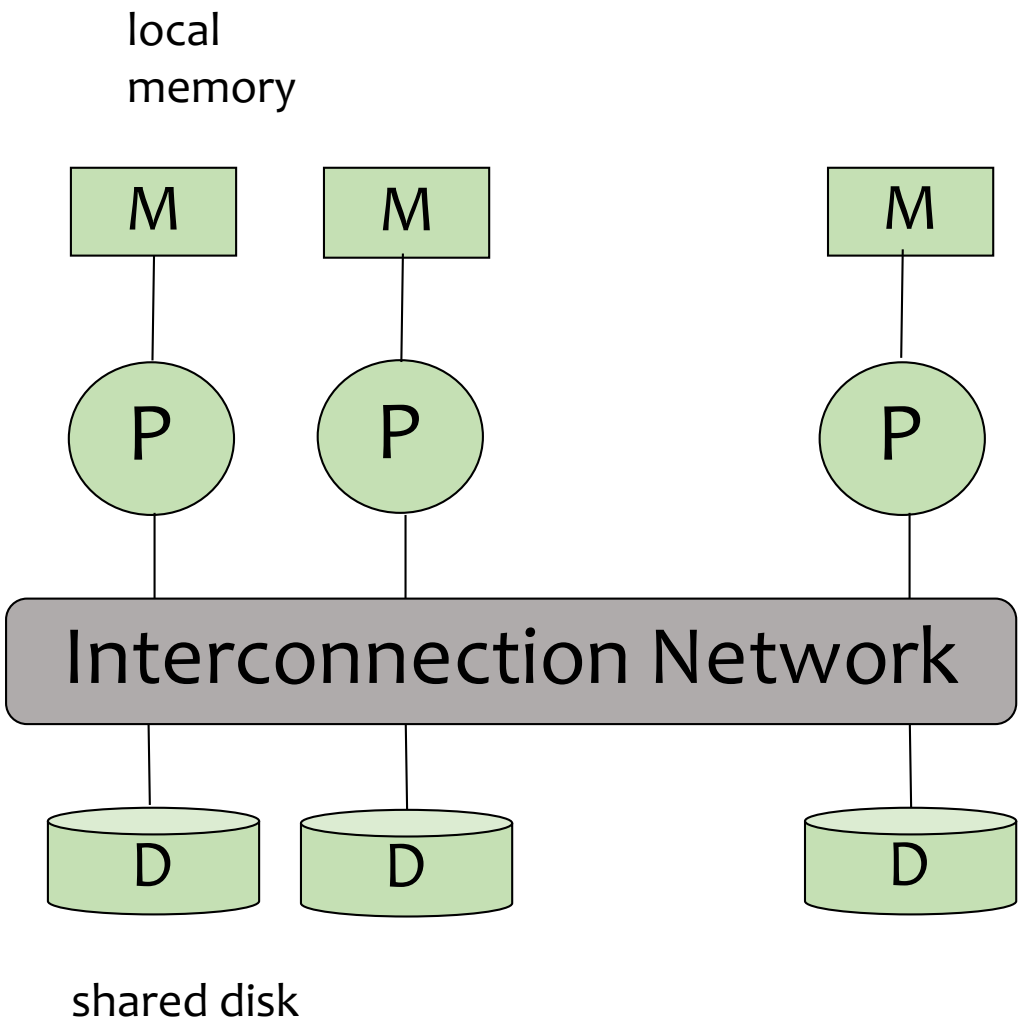


- Hard to program and design parallel algos
- Cheap to build
- Easy to scaleup and speedup
- Considered to be the best architecture

Shared Disk

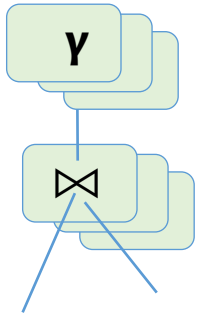
e.g. ORACLE RAC

- Trade-off but still interference like shared-memory (contention of memory and nw bandwidth)

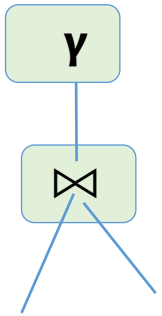


Different Types of DBMS Parallelism

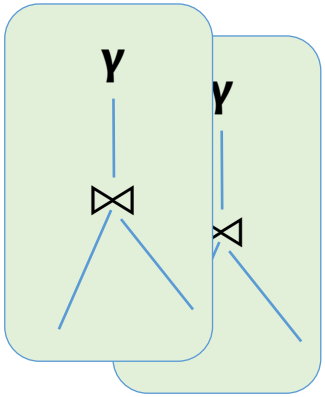
- Intra-operator parallelism
  - get all machines working to compute a given operation (scan, sort, join)
  - OLAP (decision support)



- Inter-operator parallelism
  - each operator may run concurrently on a different site (exploits pipelining)
  - For both OLAP and OLTP (transactiond)



- Inter-query parallelism
  - different queries run on different sites
  - For OLTP



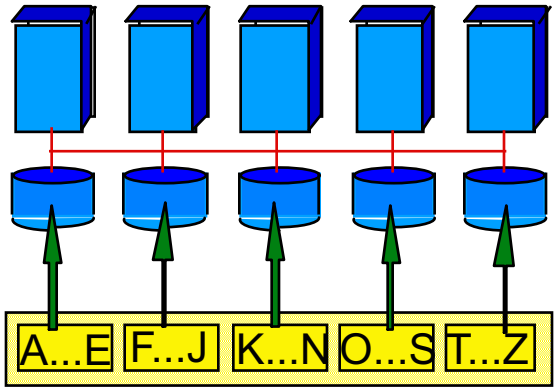
- We'll focus on intra-operator parallelism

Ack:
Slide by Prof. Dan Suciu

Data Partitioning

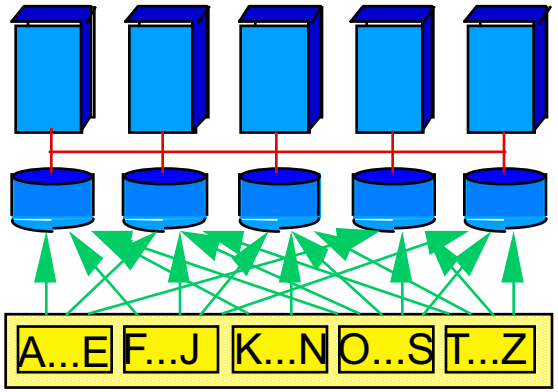
Horizontally Partitioning a table (why horizontal?)**:**

Range-partition



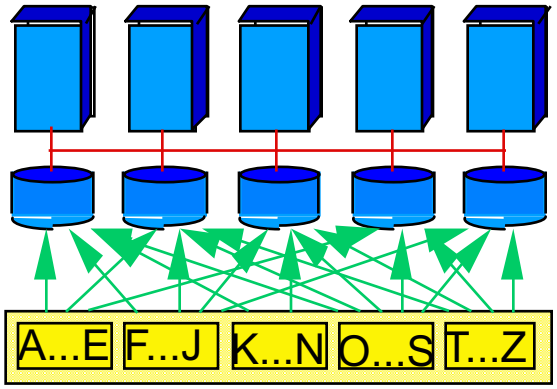
- Good for equijoins, range queries, group-by
- Can lead to data skew

Hash-partition



- Good for equijoins
- But only if hashed on that attribute
- Can lead to data skew

Block-partition or Round Robin



- Send i-th tuple to i-mod-n processor
- Good to spread load
- Good when the entire relation is accessed

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

Example

- R(Key, A, B)

- Can Block-partition be skewed?
  - no, uniform

- Can Hash-partition be skewed?
  - on the key: uniform with a good hash function
  - on A: may be skewed,
    - e.g. when all tuples have the same A-value

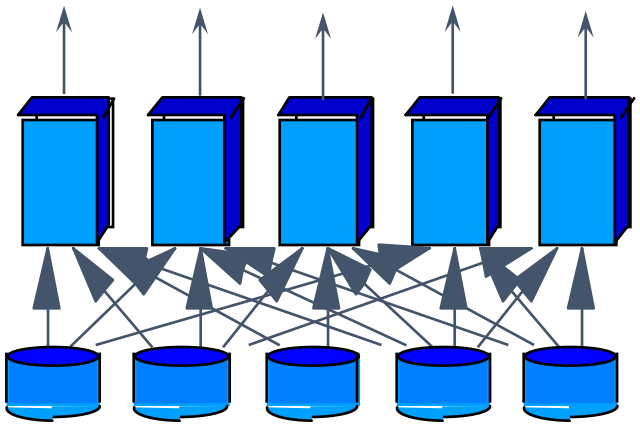
Parallelizing Sequential Evaluation Code

- "Streams" from different disks or the output of other operators
  - are "merged" as needed as input to some operator
  - are "split" as needed for subsequent parallel processing
- Different Split and merge operations appear in addition to relational operators
- No fixed formula for conversion
- Next: parallelizing individual operations

Parallel Scans

- Scan in parallel, and merge.
- Selection may not require all sites for range or hash partitioning
  - but may lead to skew
  - Suppose $\sigma_{A = 10}R$ and partitioned according to A
  - Then all tuples in the same partition/processor
- Indexes can be built at each partition

Parallel Sorting



Idea:

- Scan in parallel, and range-partition as you go
  - e.g. salary between 10 to 210, #processors = 20
  - salary in first processor: 10-20, second: 21-30, third: 31-40, ….
- As tuples come in, begin "local" sorting on each
- Resulting data is sorted, and range-partitioned
- Visit the processors in order to get a full sorted order
- Problem: skew!
- Solution: "sample" the data at start to determine partition points.

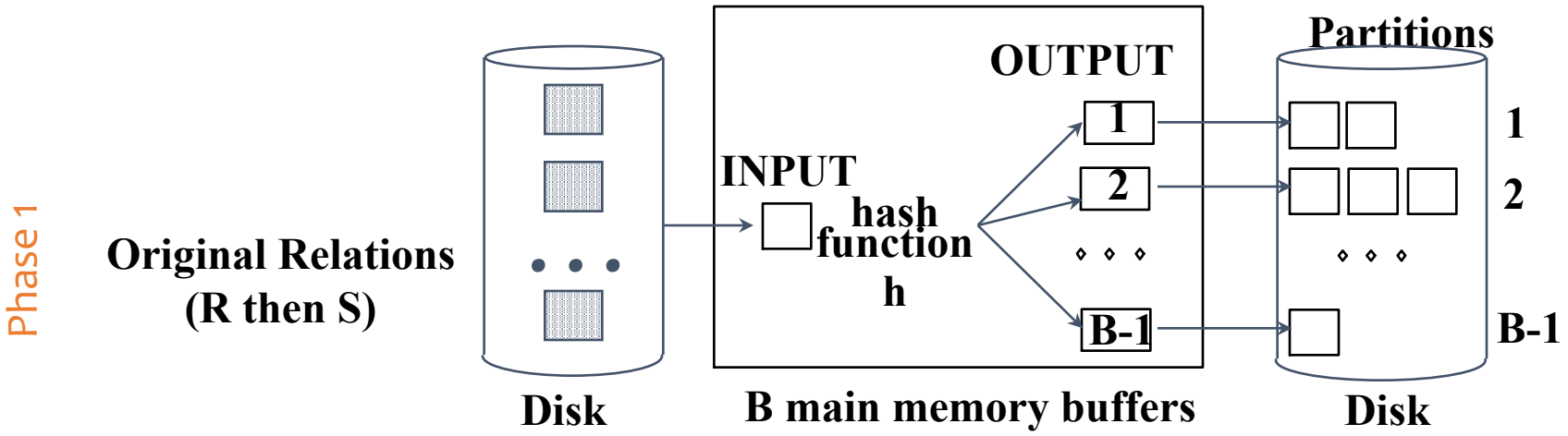
Parallel Joins

- Need to send the tuples that will join to the same machine
  - also for GROUP-BY

- Nested loop:
  - Each outer tuple must be compared with each inner tuple that might join
  - Easy for range partitioning on join cols, hard otherwise

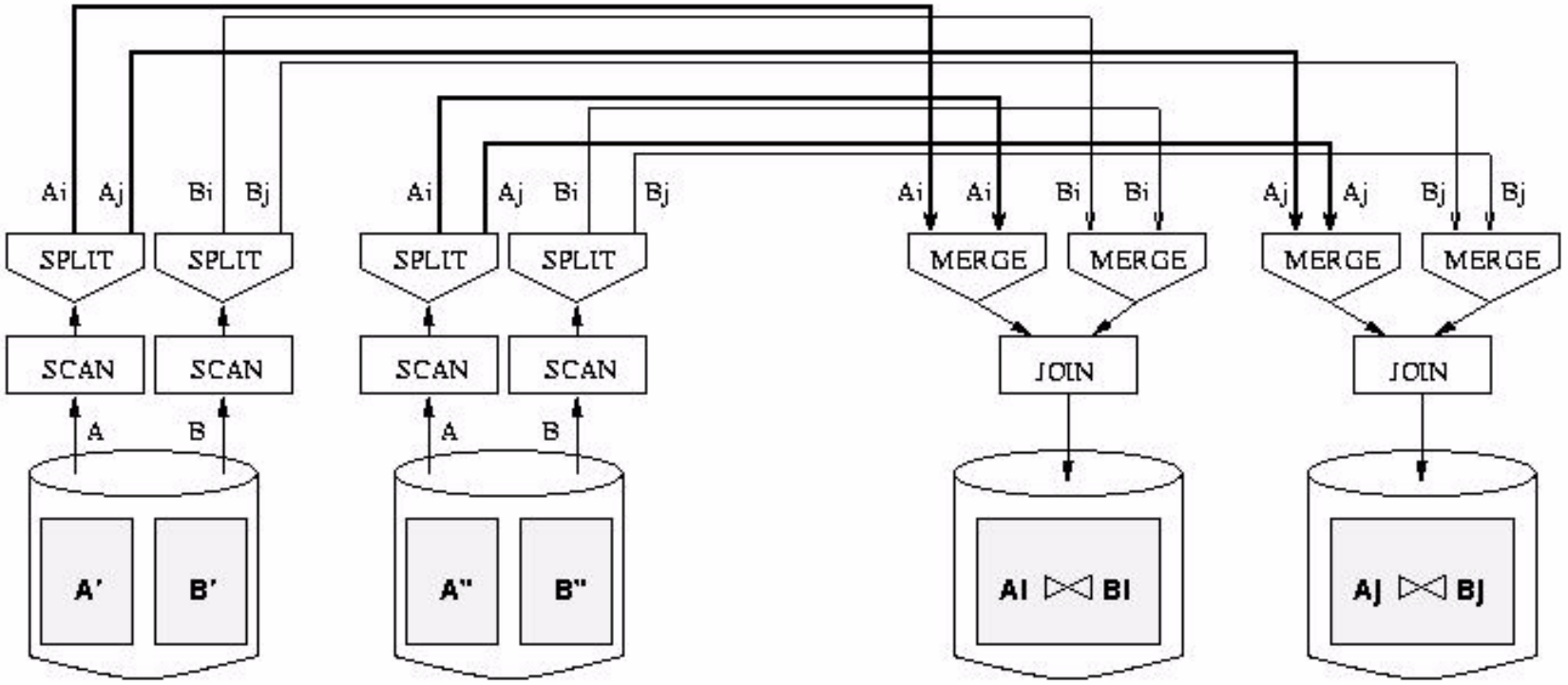
- Sort-Merge:
  - Sorting gives range-partitioning
  - Merging partitioned tables is local

Parallel Hash Join



- In first phase, partitions get distributed to different sites:
  - A good hash function *automatically* distributes work evenly
- Do second phase at each site.
- Almost always the winner for equi-join

Dataflow Network for parallel Join

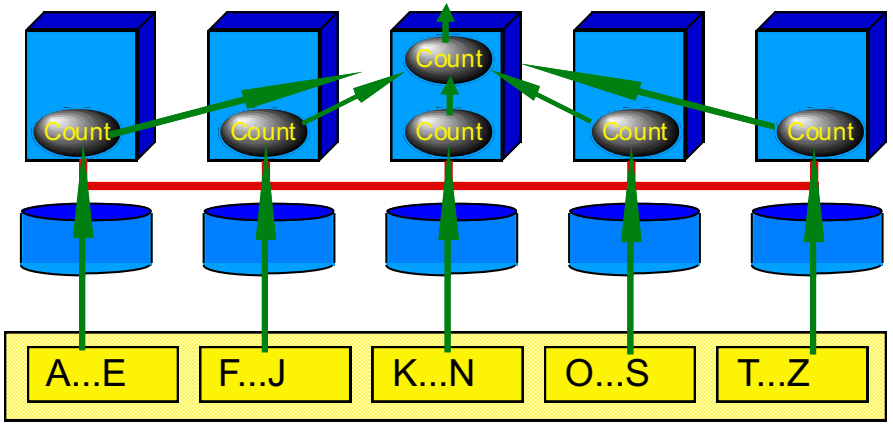


- Good use of split/merge makes it easier to build parallel versions of sequential join code.

Parallel Aggregates

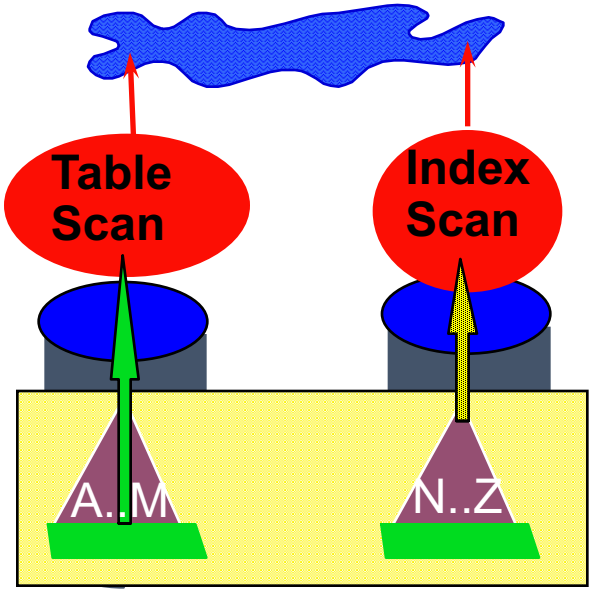
- For each aggregate function, need a decomposition:
  - count(S) = Σ count(s(i)), ditto for sum()
  - avg(S) = (Σ sum(s(i))) / Σ count(s(i))
  - and so on...
- For group-by:
  - Sub-aggregate groups close to the source.
  - Pass each sub-aggregate to its group's site.
    - Chosen via a hash fn.

Which SQL aggregate operators are not good for parallel execution?



Best serial plan may not be best ||

- Why?
- Trivial counter-example:
  - Table partitioned with local secondary index at two nodes
  - Range query: all of node 1 and 1% of node 2.
  - Node 1 should do a scan of its partition.
  - Node 2 should use secondary index.



Example problem: Parallel DBMS

R(a,b) is horizontally partitioned across N = 3 machines.

Each machine locally stores approximately 1/N of the tuples in R.

The tuples are randomly organized across machines (i.e., R is block partitioned across machines).

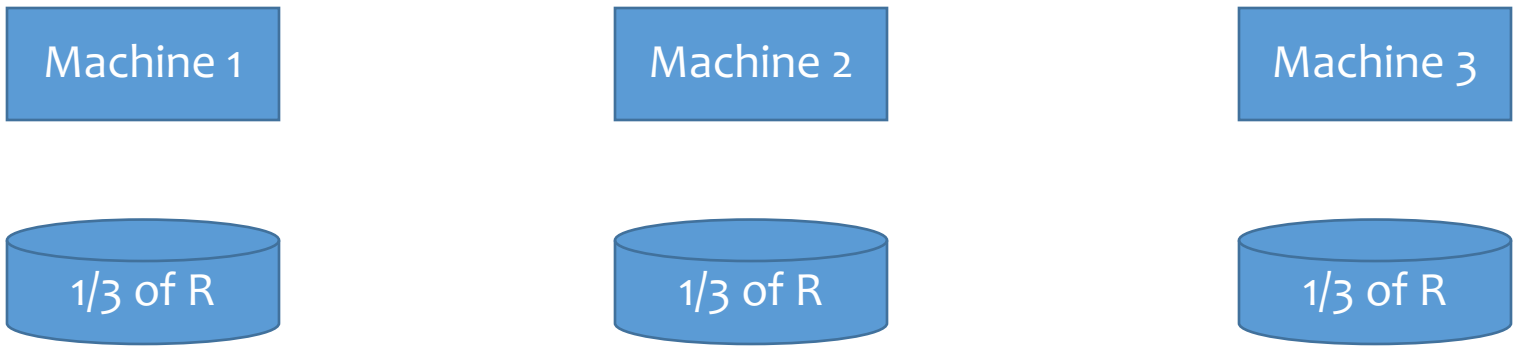
Show a RA plan for this query and how it will be executed across the N = 3 machines.

Pick an efficient plan that leverages the parallelism as much as possible.

- **SELECT a, max(b) as topb**
- **FROM R**
- **WHERE a > 0**
- **GROUP BY a**

R(a, b)

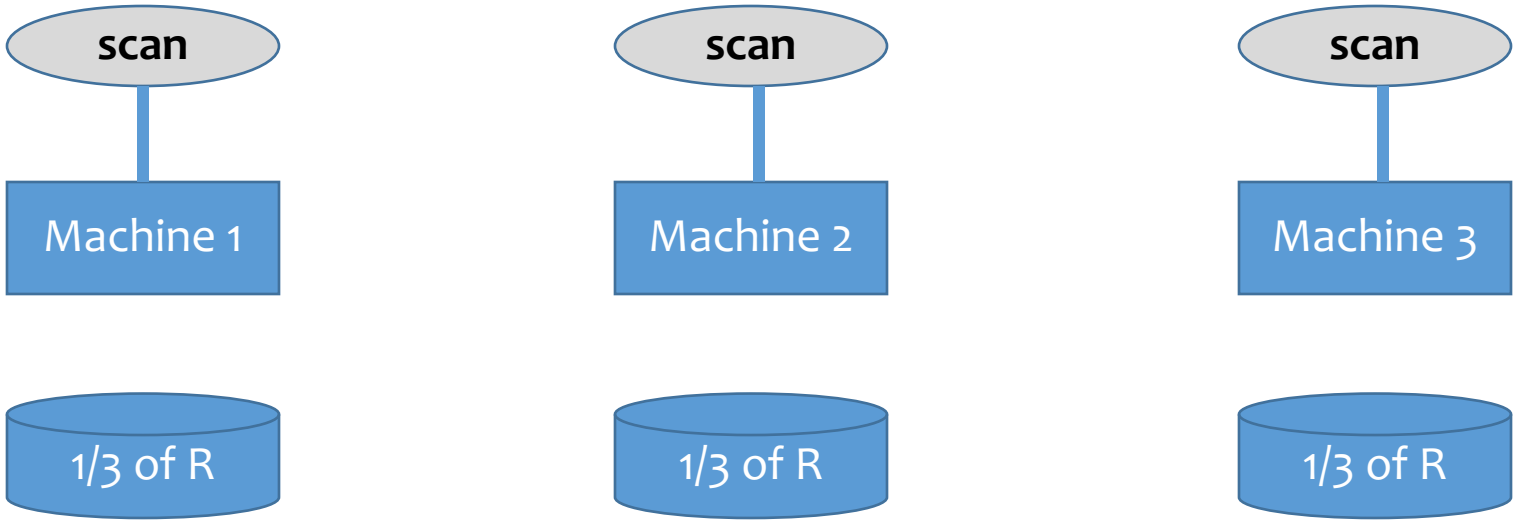
```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```



R(a, b)

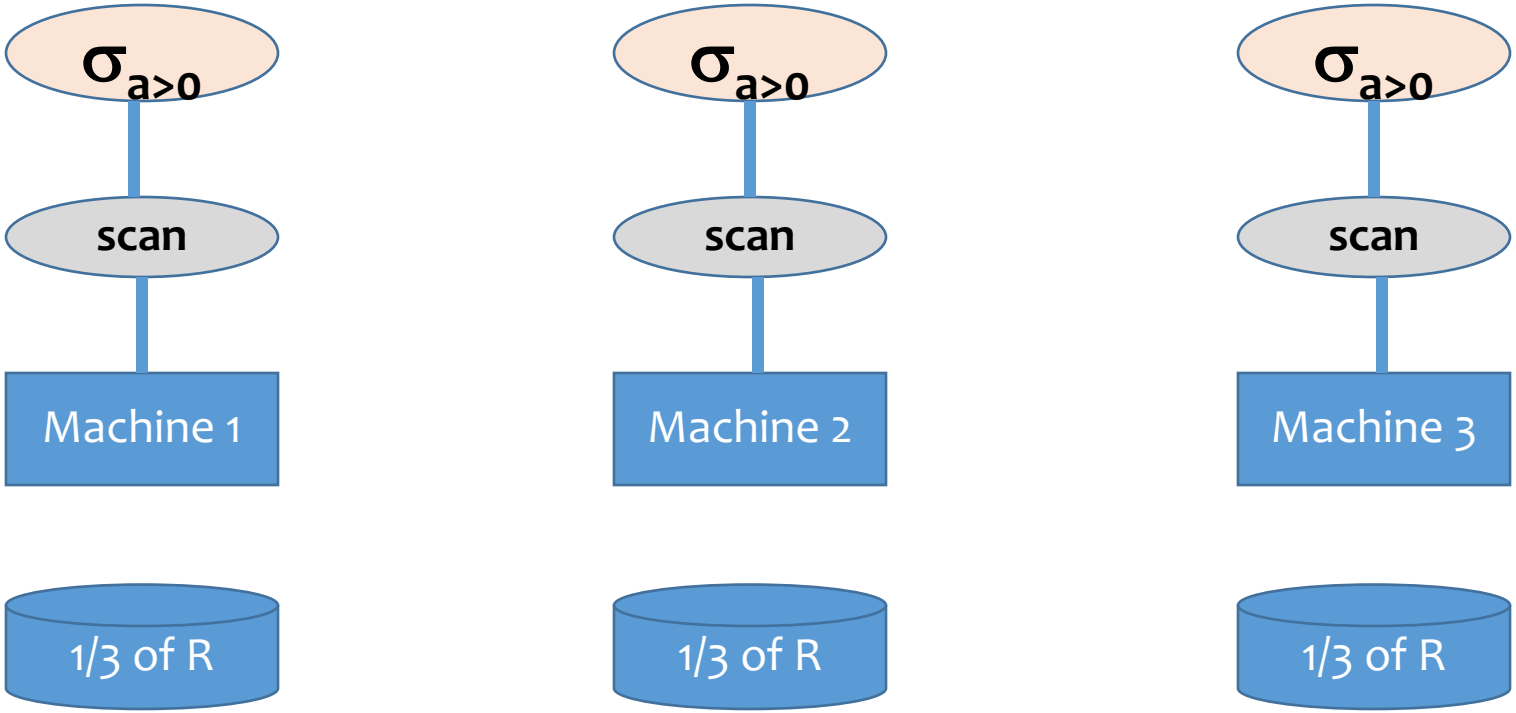
```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

If more than one relation on a machine, then "scan S", "scan R" etc



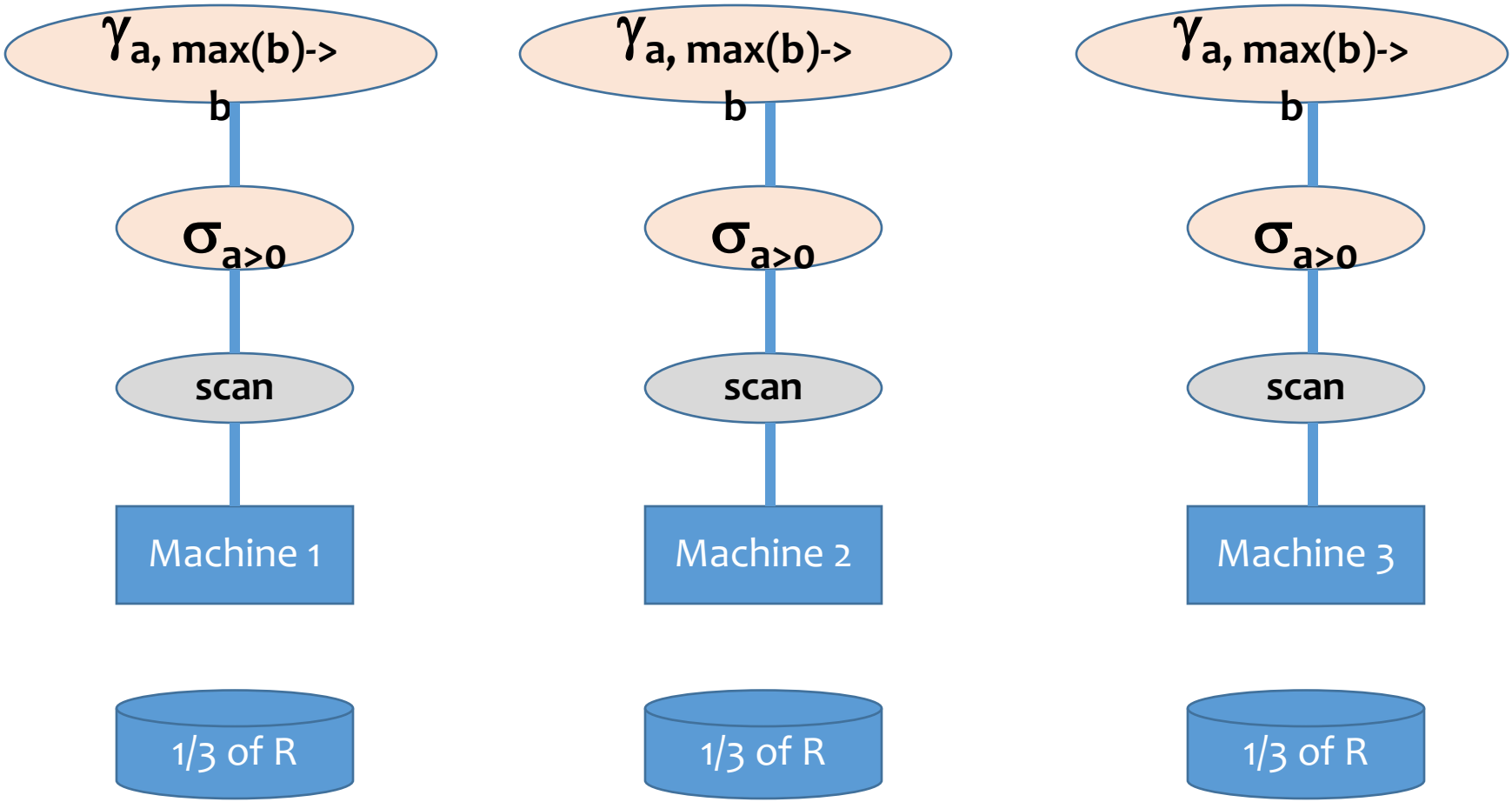
R(a, b)

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```



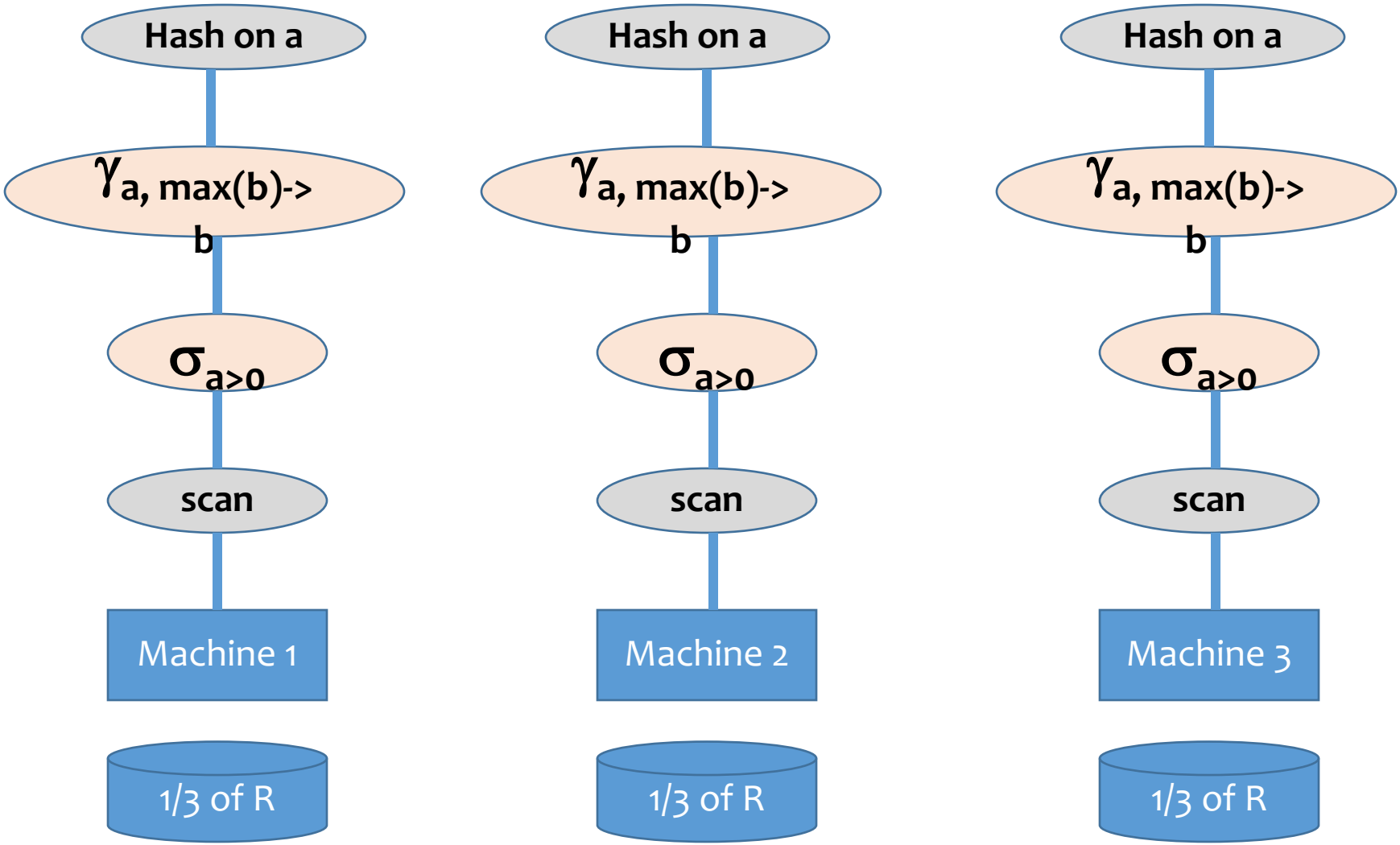
R(a, b)

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY  a
```



R(a, b)

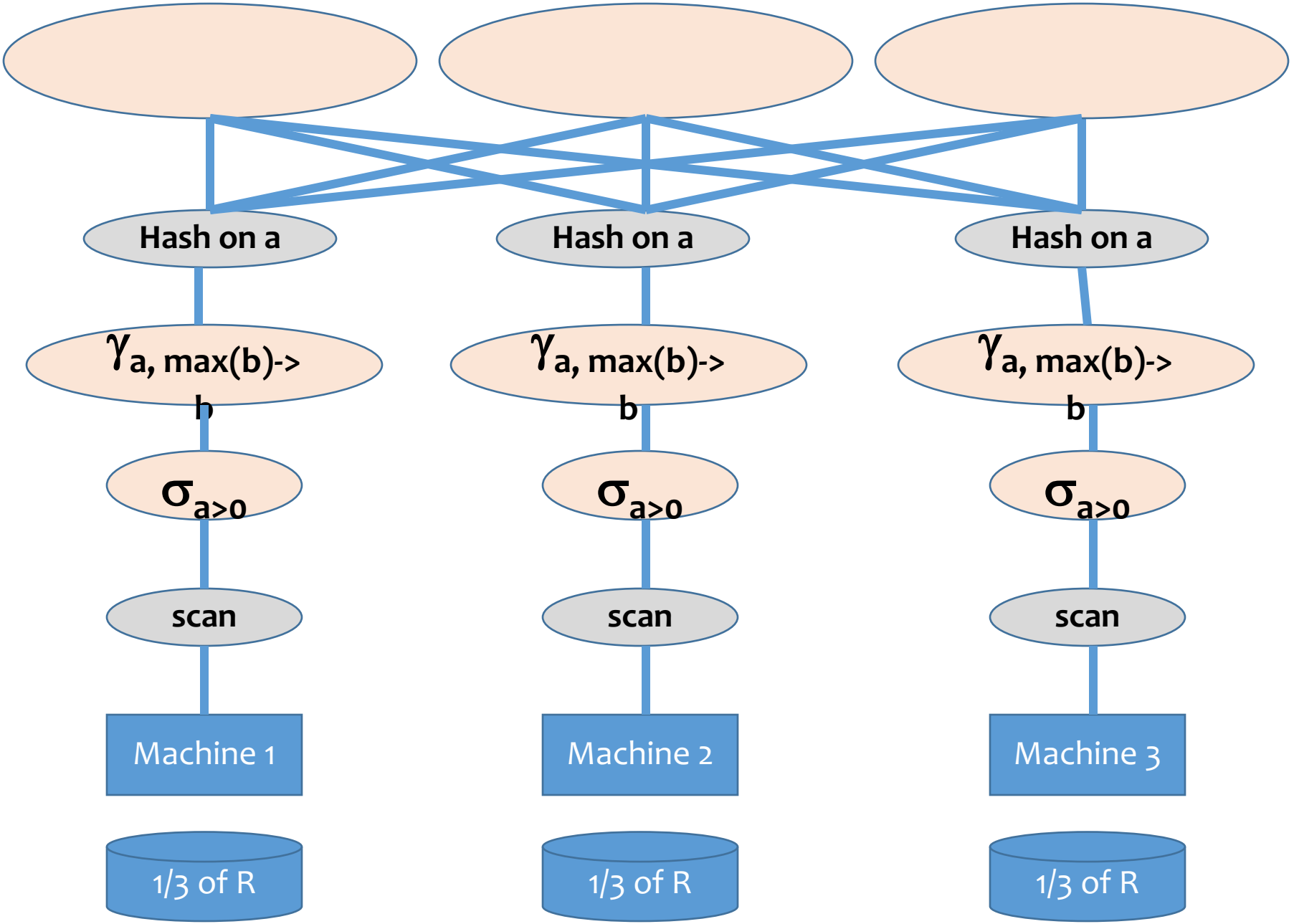
```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

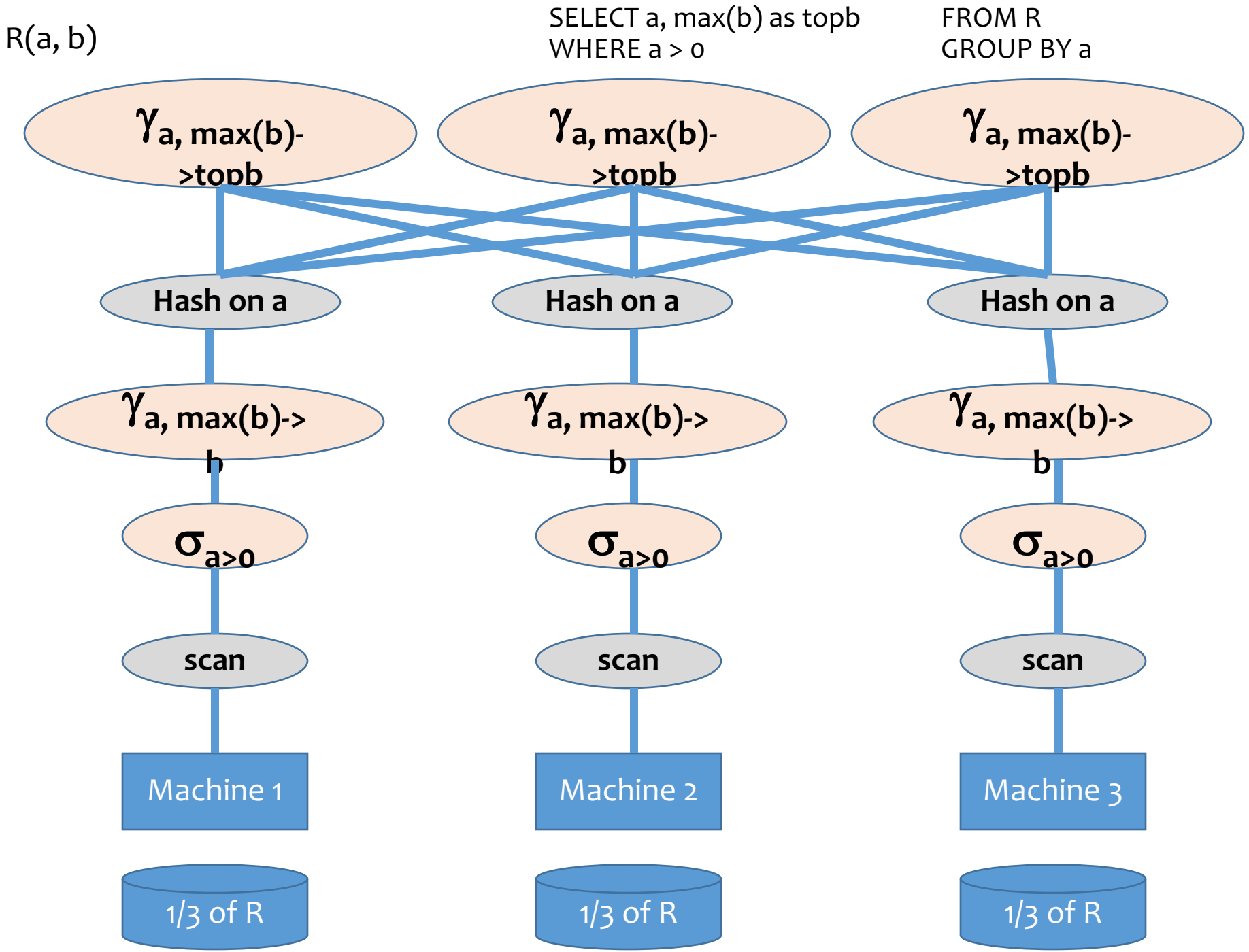


R(a, b)

SELECT a, max(b) as topb
WHERE a > 0

FROM R
GROUP BY a




R(a, b)
SELECT a, max(b) as topb
WHERE a > 0
FROM R
GROUP BY a
γa, max(b)->topb
γa, max(b)->topb
γa, max(b)->topb
Hash on a
Hash on a
Hash on a
γa, max(b)->b
γa, max(b)->b
γa, max(b)->b
σa>0
σa>0
σa>0
scan
scan
scan
Machine 1
Machine 2
Machine 3
1/3 of R
1/3 of R
1/3 of R

Same Example Problem: Map Reduce

Explain how the query will be executed in MapReduce (recall Lecture-3)

- **SELECT a, max(b) as topb**
- **FROM R**
- **WHERE a > 0**
- **GROUP BY a**

Specify the computation performed in the map and the reduce functions

Map

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- Each map task
  - Scans a block of R
  - Calls the map function for each tuple
  - The map function applies the selection predicate to the tuple
  - For each tuple satisfying the selection, it outputs a record with key = a and value = b

•When each map task scans multiple relations, it needs to output something like
key = a and value = (‘R’, b)
which has the relation name ‘R’

Shuffle

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- The MapReduce engine reshuffles the output of the map phase and groups it on the intermediate key, i.e. the attribute a

•Note that the programmer has to write only the map and reduce functions, the shuffle phase is done by the MapReduce engine (although the programmer can rewrite the partition function), but you should still mention this in your answers

Reduce

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- Each reduce task
  - computes the aggregate value **max(b) = topb** for each group (i.e. ***a***) assigned to it (by calling the reduce function)
  - outputs the final results: **(a, topb)**

A local combiner can be used to compute local max before data gets reshuffled (in the map tasks)

- Multiple aggregates can be output by the reduce phase like **key = a and value = (sum(b), min(b))** etc.

- Sometimes a second (third etc) level of Map-Reduce phase might be needed

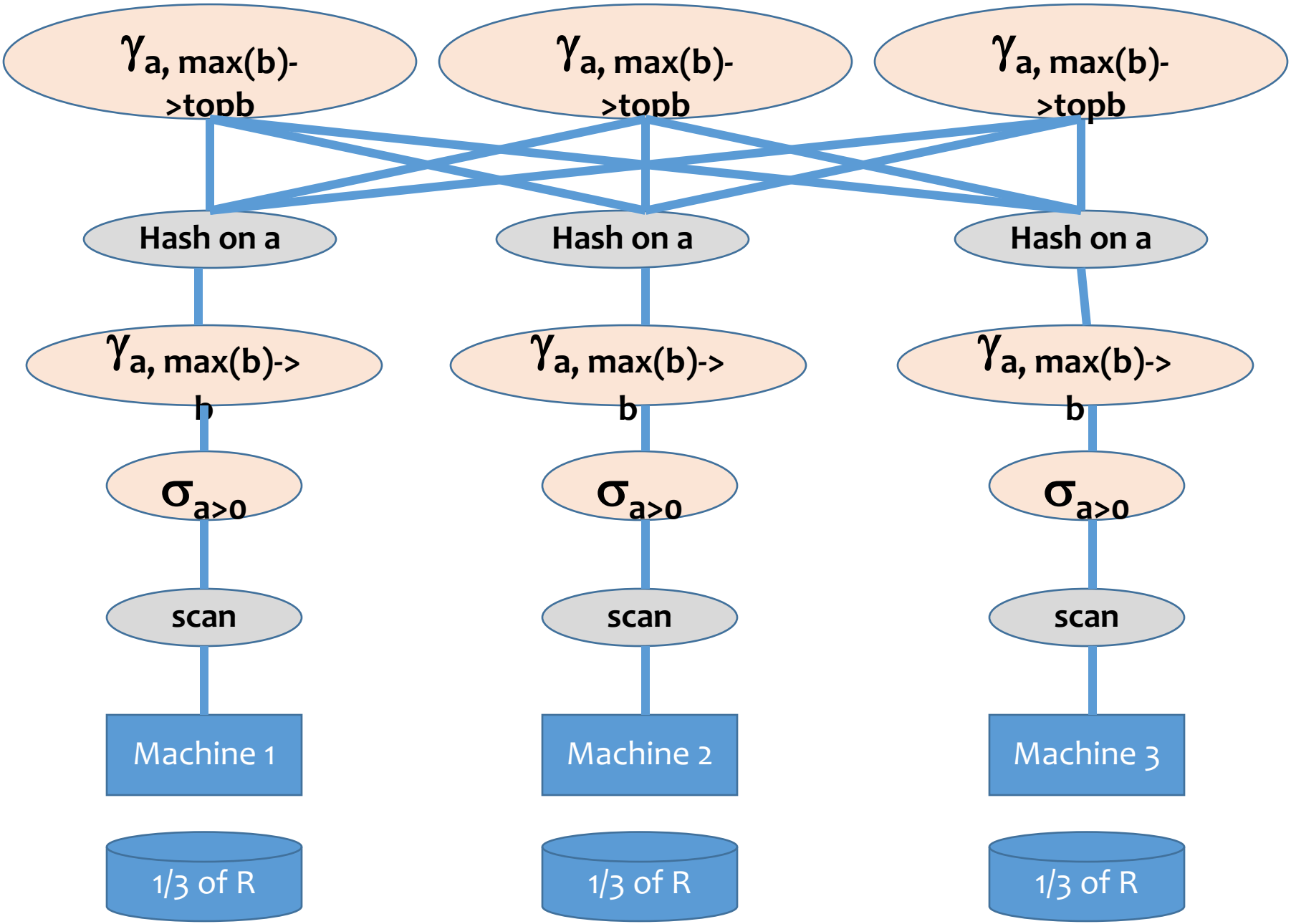
```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

Benefit of hash-partitioning

- What would change if we hash-partitioned R on R.a before executing the same query on the previous parallel DBMS

Prev: block-partition

SELECT a, max(b) as topb
WHERE a > 0
FROM R
GROUP BY a



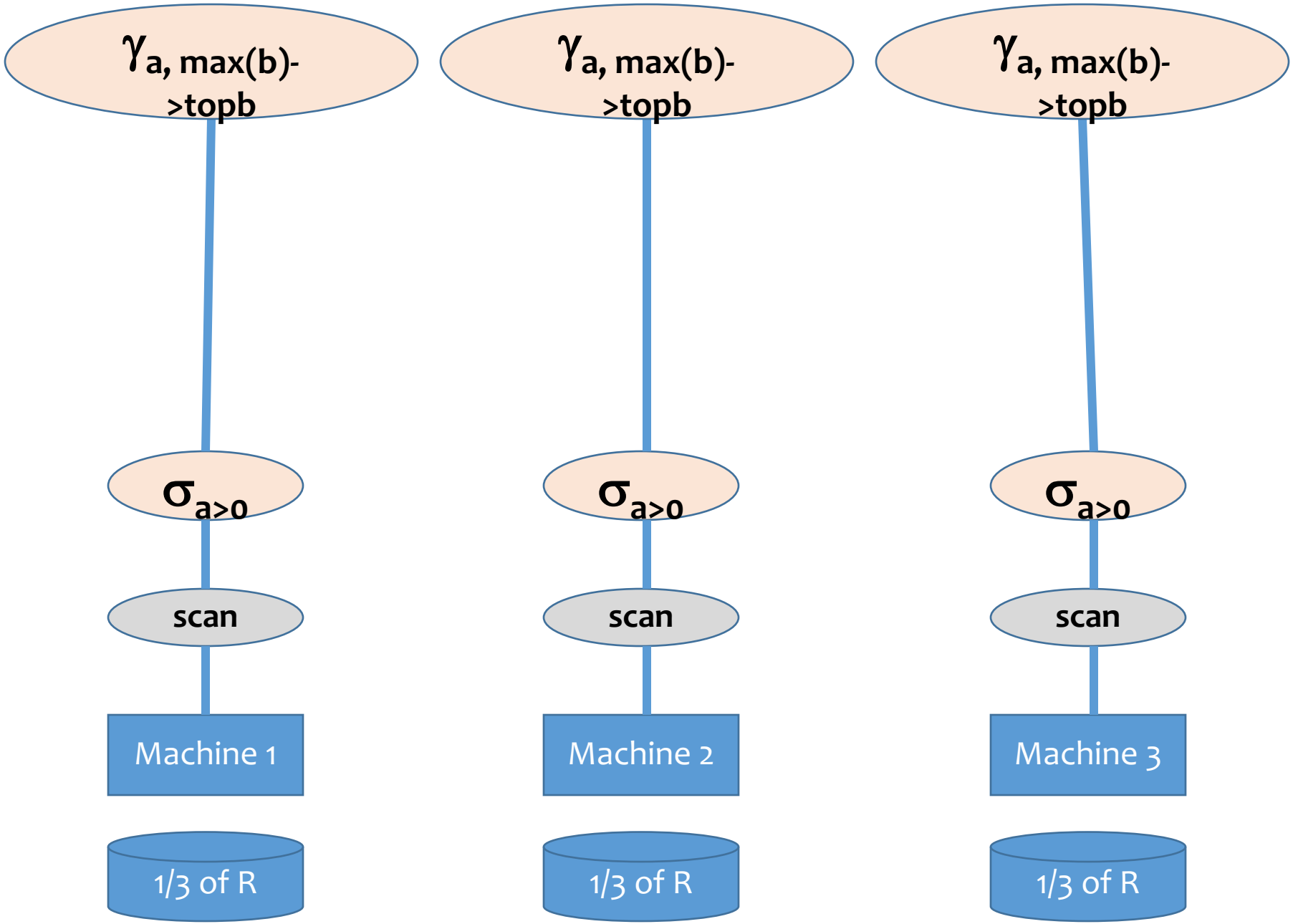
```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- It would avoid the data re-shuffling phase
- It would compute the aggregates locally

Hash-partition on a for R(a, b)

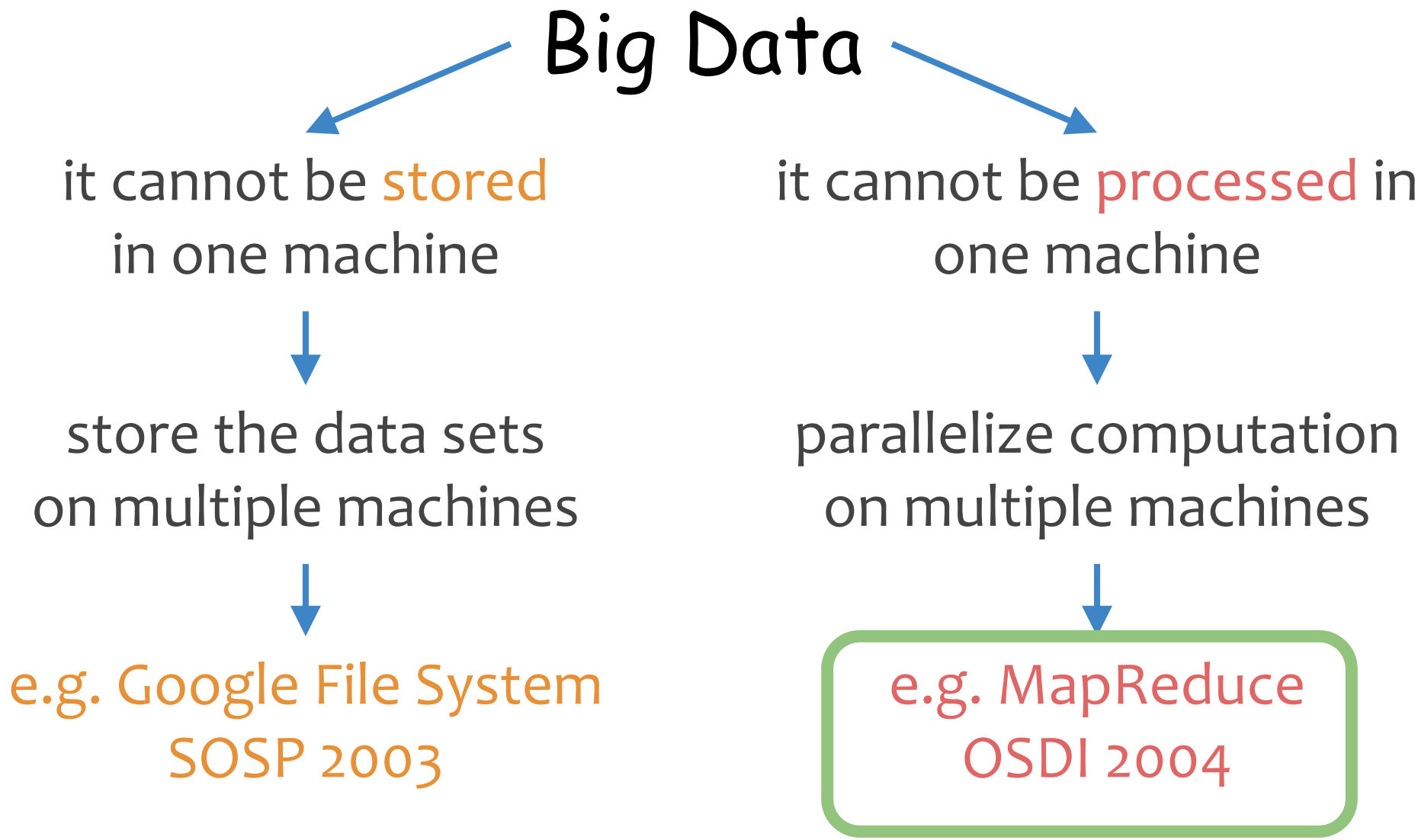
SELECT a, max(b) as topb
WHERE a > 0

FROM R
GROUP BY a

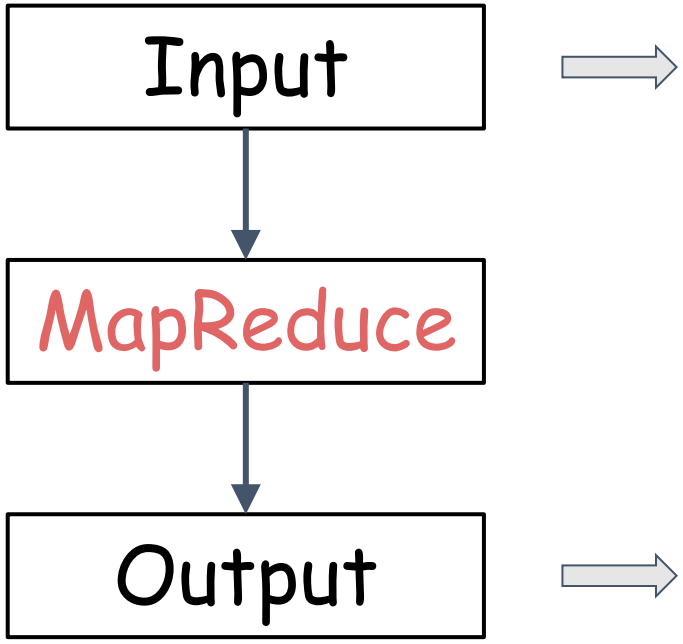


Map Reduce

Slides by Junghoon Kang


Big Data
it cannot be stored in one machine
store the data sets on multiple machines
e.g. Google File System SOSP 2003
it cannot be processed in one machine
parallelize computation on multiple machines
e.g. MapReduce OSDI 2004

Where does Google use MapReduce?



- crawled documents
- web request logs

- inverted indices
- graph structure of web documents
- summaries of the number of pages crawled per host
- the set of most frequent queries in a day

What is MapReduce?

It is a programming model
that processes large data by:
apply a function to each logical record in the input (map)
categorize and combine the intermediate results
into summary values (reduce)

Google's MapReduce is inspired by map and reduce functions in functional programming languages.

For example,
in Scala functional
programming language,

```
scala> val lst = List(1,2,3,4,5)

scala> lst.map( x => x + 1 )

reso: List[Int] = List(2,3,4,5,6)
```

For example,
in Scala functional
programming language,

```
scala> val lst = List(1,2,3,4,5)

scala> lst.reduce( (a, b) => a + b )

reso: Int = 15
```

Ok, it makes sense
in one machine.

Then, how does Google extend
the functional idea
to multiple machines
in order to
process large data?

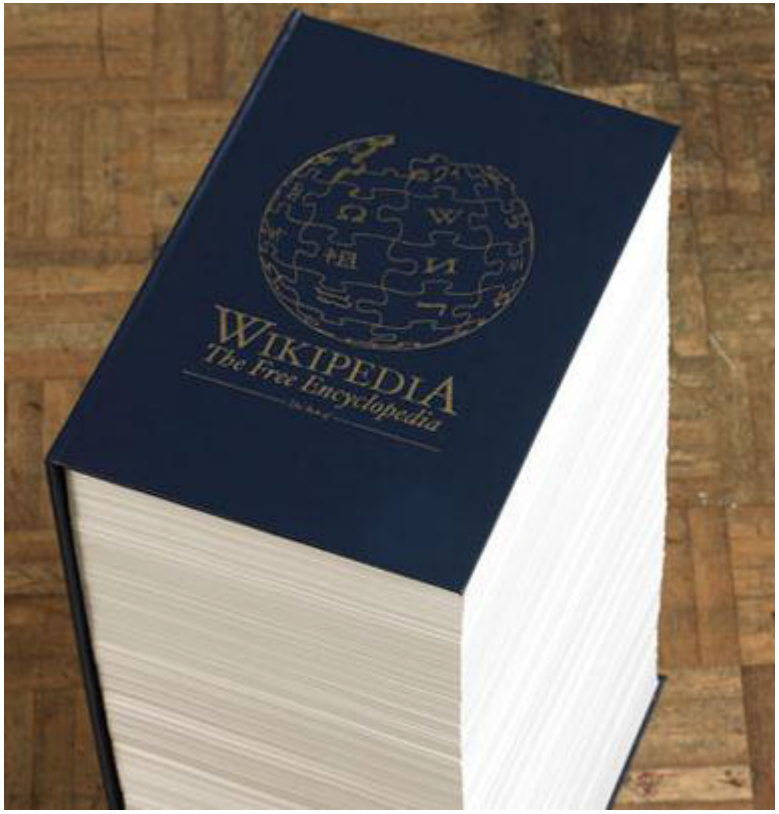
Understanding MapReduce

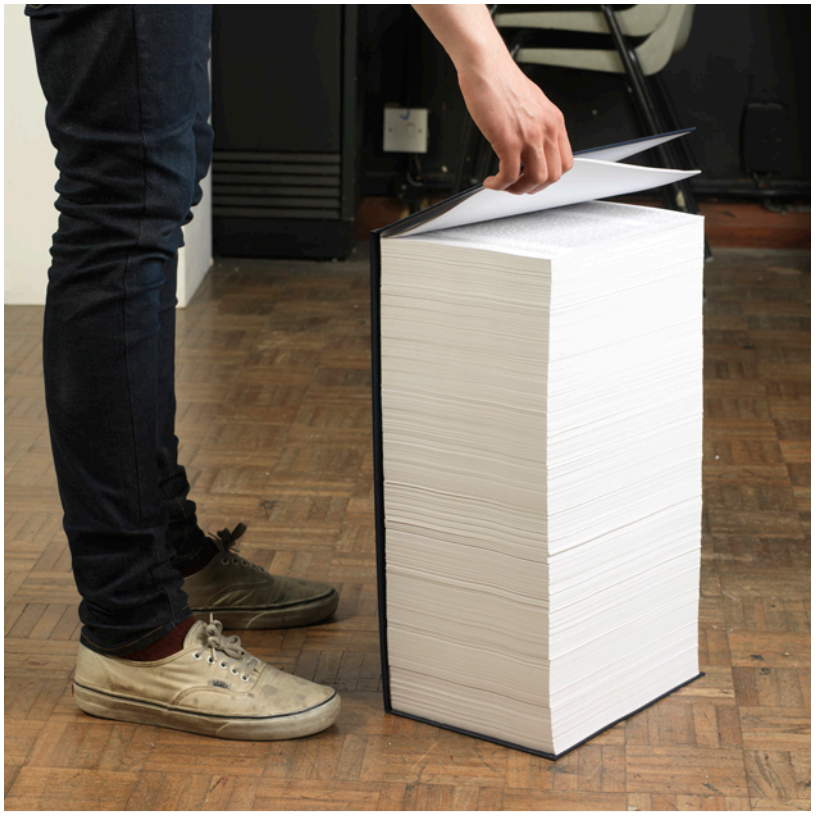
(by example)

I am a class president

An English teacher asks you:

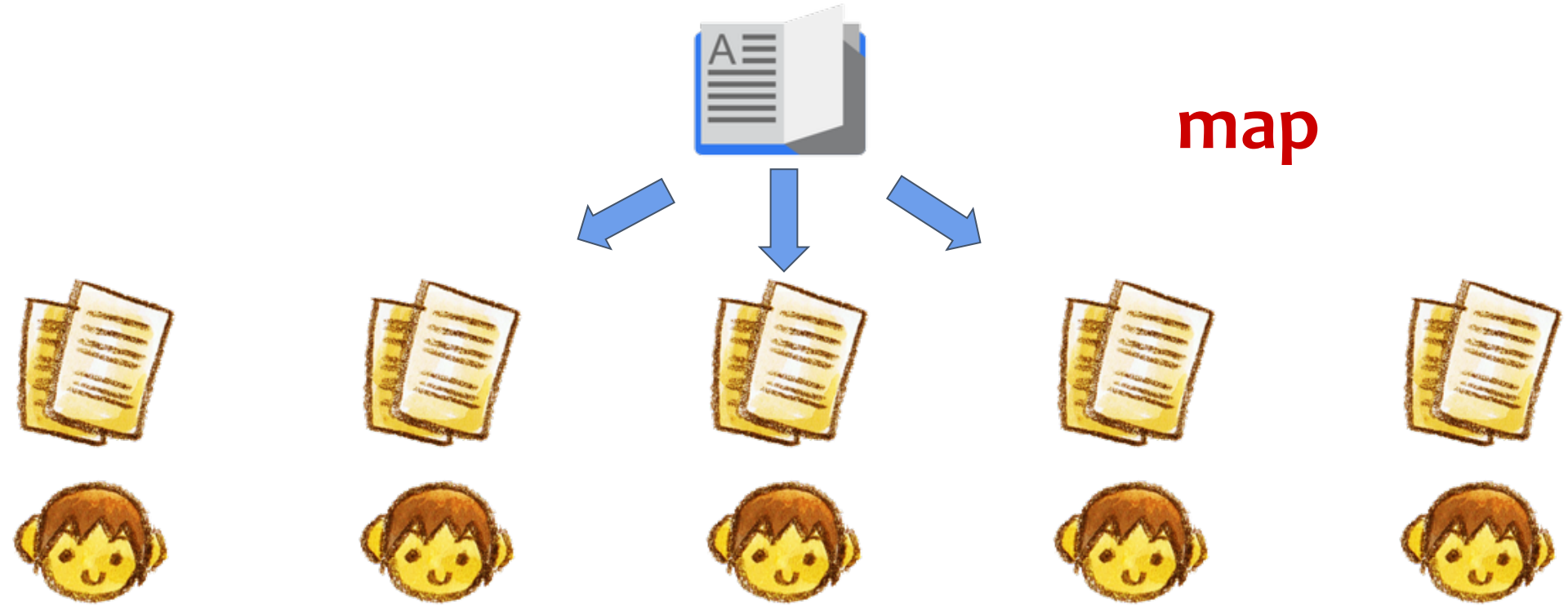
"Could you count the number of occurrences of each word in this book?"





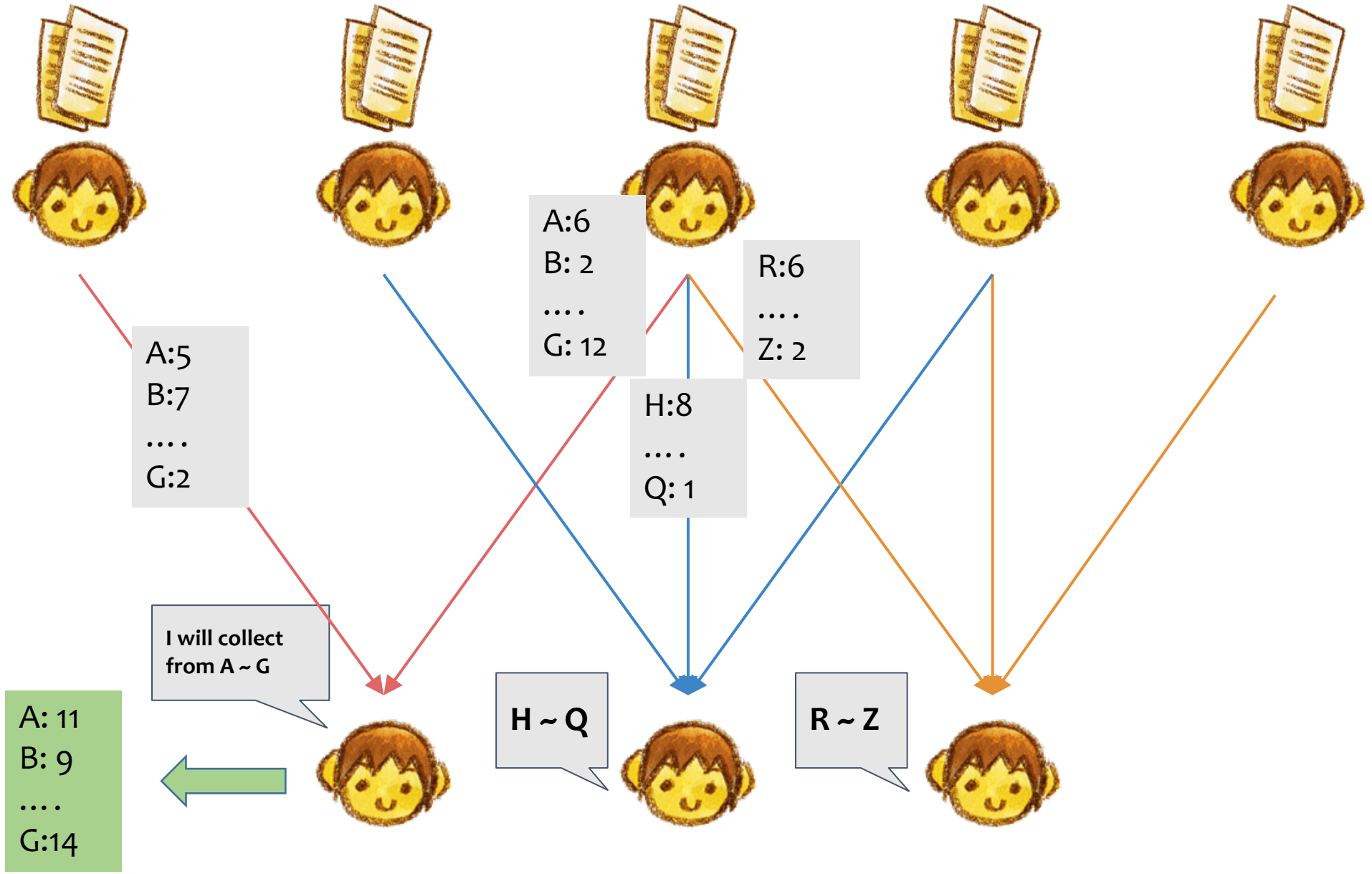
Um… Ok...

Let's divide the workload among classmates.



And let few combine the intermediate results

reduce



Why did MapReduce become so popular?

Is it because Google uses it?

Distributed Computation Before MapReduce

Things to consider:

- how to divide the workload among multiple machines?
- how to distribute data and program to other machines?
- how to schedule tasks?
- what happens if a task fails while running?
- … and … and ...

Distributed Computation After MapReduce

Things to consider:

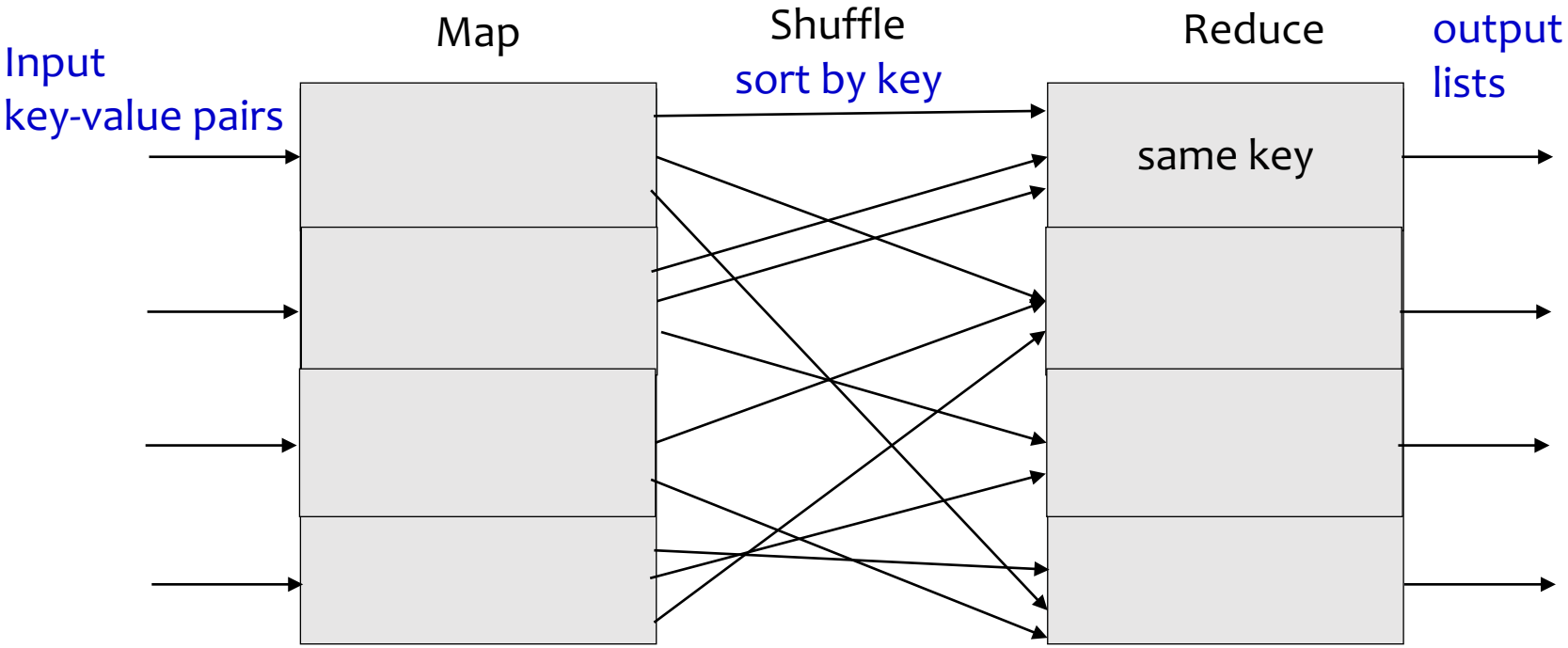
- how to write Map function?
- how to write Reduce function?

MapReduce lowered the knowledge barrier in distributed computation.

Developers needed
before MapReduce

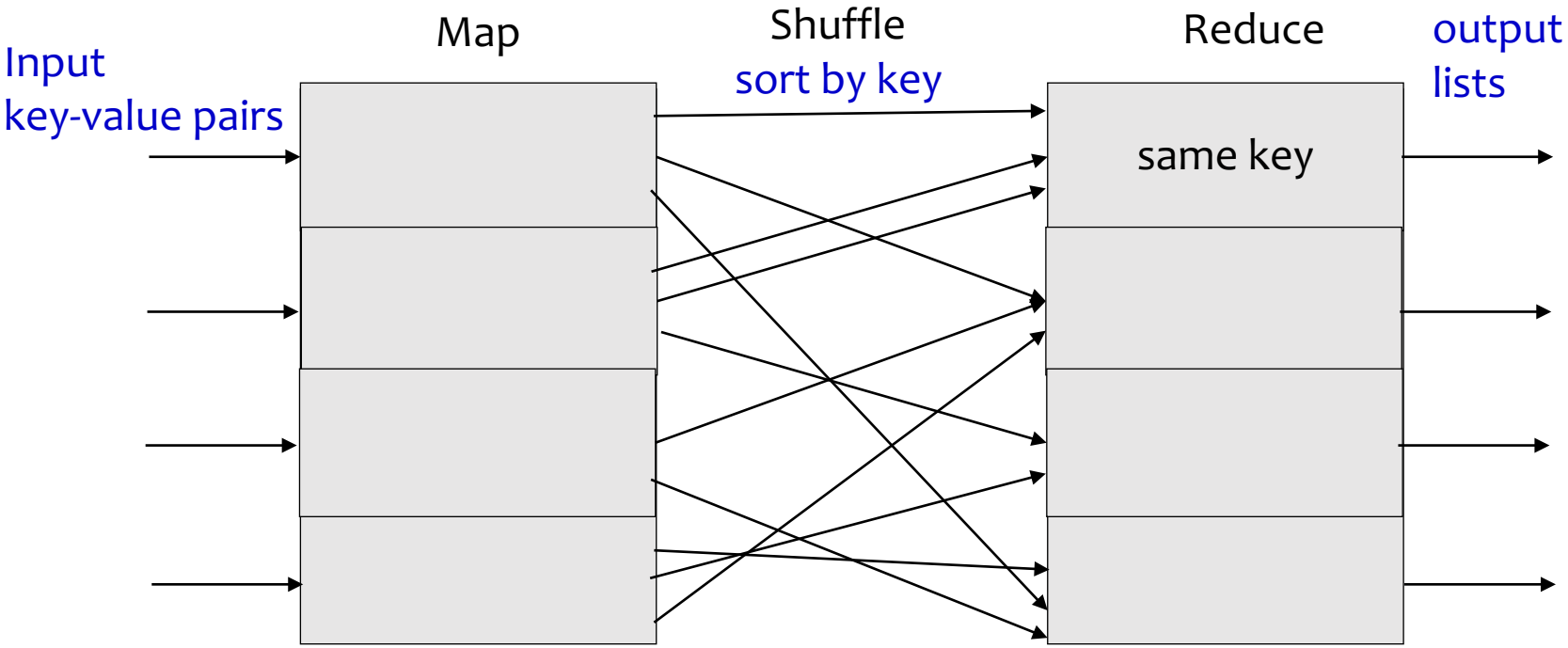
Developers needed
after MapReduce

Map-Reduce Steps



- Input is typically (key, value) pairs
  - but could be objects of any type
- Map and Reduce are performed by a number of processes
  - physically located in some processors

Map-Reduce Steps

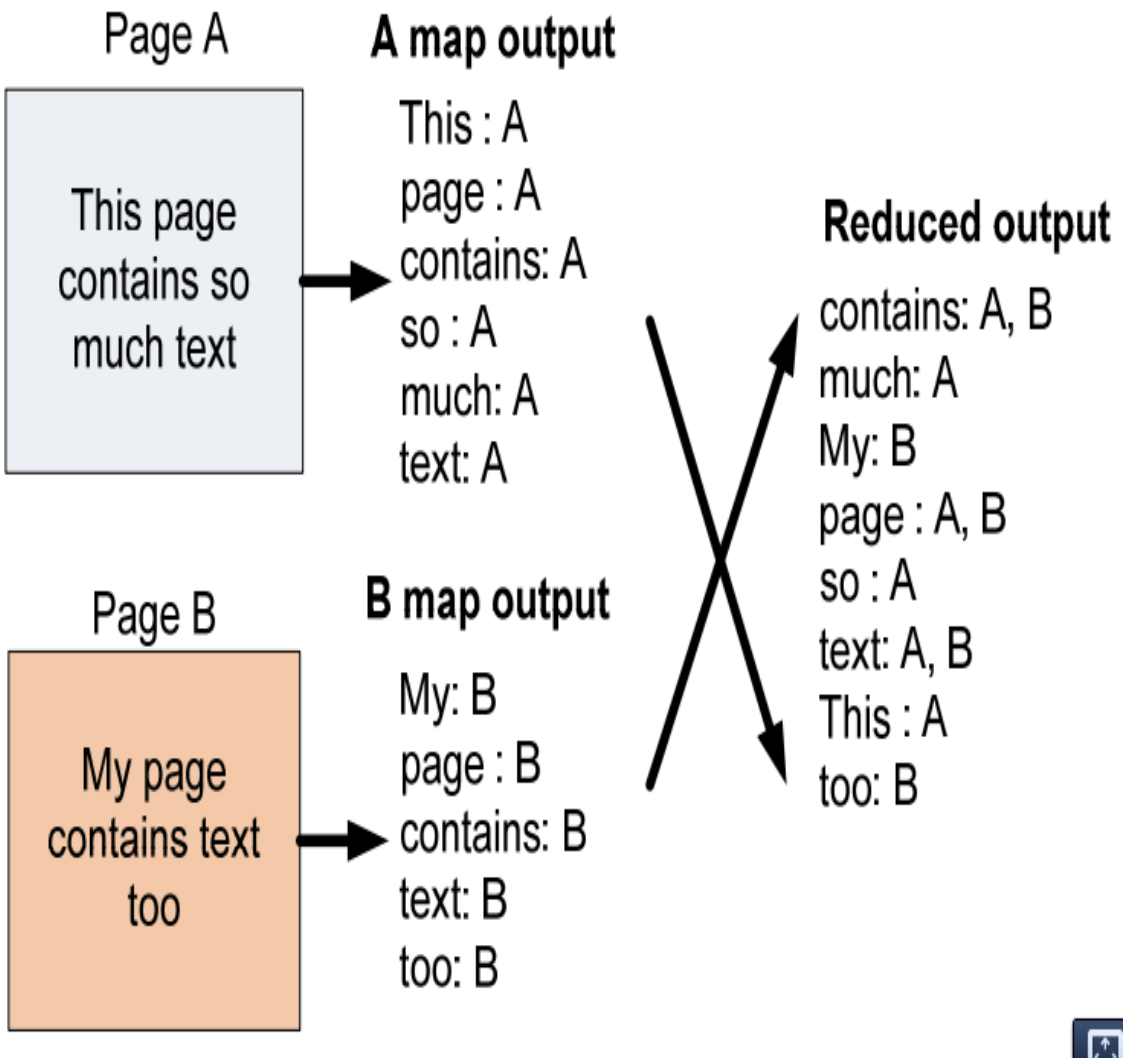


1. Read Data
2. Map – extract some info of interest in (key, value) form
3. Shuffle and sort
   - send same keys to the same reduce process

4. Reduce
   - operate on the values of the same key
   - e.g. transform, aggregate, summarize, filter
5. Output the results (key, final-result)

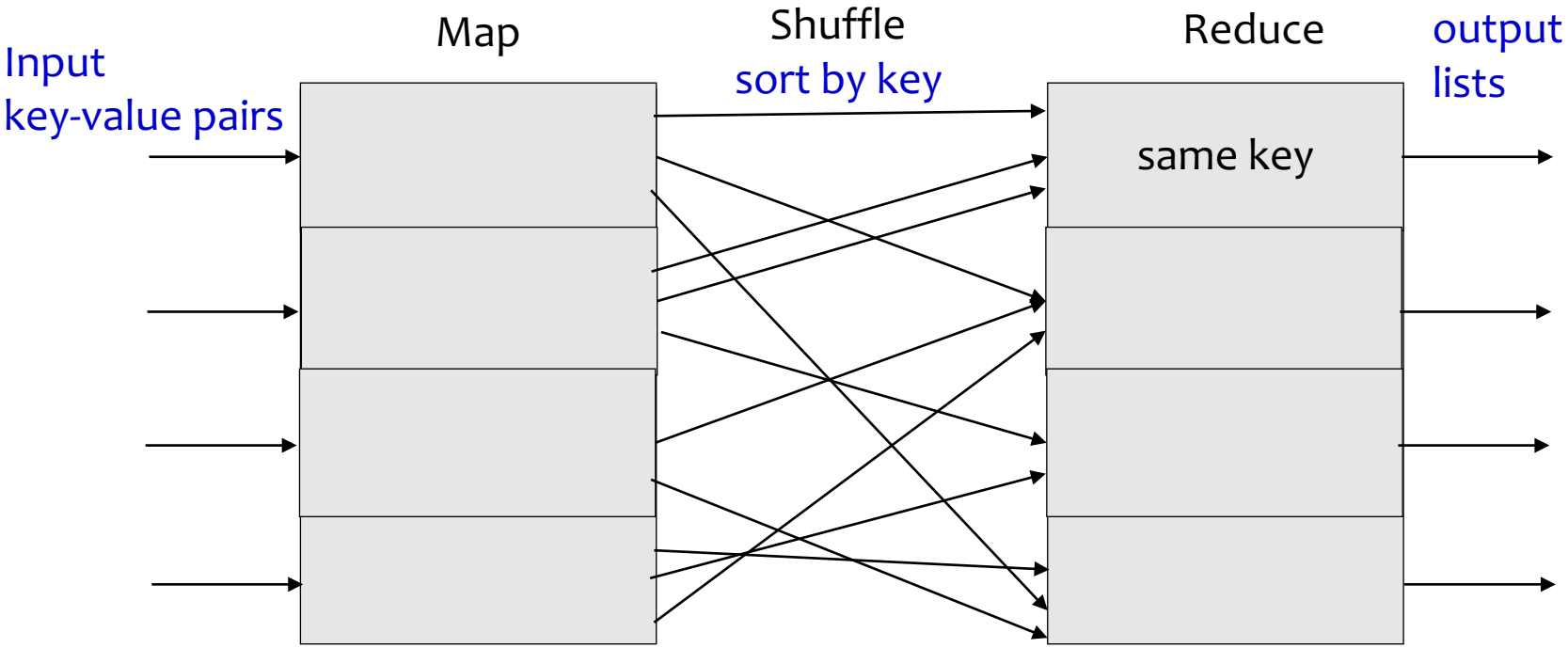
Simple Example: Map-Reduce

- Word counting
- Inverted indexes



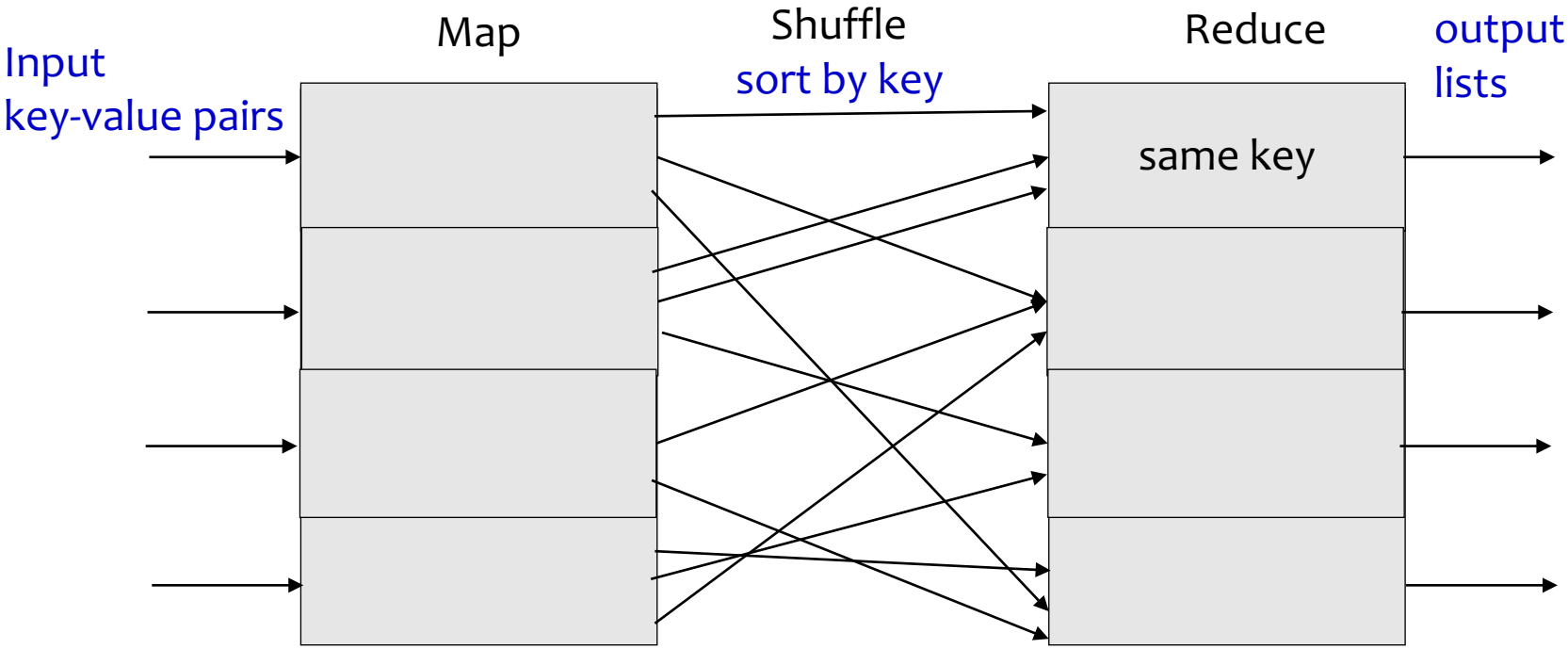
Ack:
Slide by Prof. Shivnath Babu

Map Function



- Each map process works on a chunk of data -- master selects available workers
- Input: (input-key, value)
- Output: (intermediate-key, value) -- may not be the same as input key value
- Example: list all doc ids containing a word
  - output of map (word, docid) – emits each such pair
  - word is key, docid is value
  - duplicate elimination can be done at the reduce phase

Reduce Function



- Input: (intermediate-key, list-of-values-for-this-key) – list can include duplicates
  - each map process can leave its output in the local disk, reduce process can retrieve its portion
  - master selects idle workers for reduce
- Output: (output-key, final-value)
- Example: list all doc ids containing a word
  - output will be a list of (word, [doc-id1, doc-id5, ….])
  - if the count is needed, reduce counts #docs, output will be a list of (word, count)

Key Players in MapReduce

- A Map-Reduce "Job"
  - e.g. count the words in all docs
  - complex queries can have multiple MR jobs
- Map or Reduce "Tasks"
  - A group of map or reduce "functions"
  - scheduled on a single "worker"
- Worker
  - a process that executes one task at a time
  - one per processor, so 4-8 per machine
  - Follows whatever the Master asks to do
- One Master
  - coordinates many workers.
  - assigns a task to each worker

Fault Tolerance

Although the probability of a machine failure is low, the probability of a machine failing among thousands of machines is common.

How does MapReduce handle machine failures?

Worker Failure

- The master sends heartbeat to each worker node.
- If a worker node fails, the master reschedules the tasks handled by the worker.

Master Failure

- The whole MapReduce job gets restarted through a different master.

End of Lecture 24

Same Example Problem: Map Reduce

Explain how the query will be executed in MapReduce

- **SELECT a, max(b) as topb**
- **FROM R**
- **WHERE a > 0**
- **GROUP BY a**

Specify the computation performed in the map and the reduce functions

Map

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- Each map task
  - Scans a block of R
  - Calls the map function for each tuple
  - The map function applies the selection predicate to the tuple
  - For each tuple satisfying the selection, it outputs a record with key = a and value = b

•When each map task scans multiple relations, it needs to output something like
key = a and value = (‘R’, b)
which has the relation name ‘R’

Shuffle

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- The MapReduce engine reshuffles the output of the map phase and groups it on the intermediate key, i.e. the attribute a

•Note that the programmer has to write only the map and reduce functions, the shuffle phase is done by the MapReduce engine (although the programmer can rewrite the partition function)

Reduce

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- Each reduce task
  - computes the aggregate value **max(b) = topb** for each group (i.e. ***a***) assigned to it (by calling the reduce function)
  - outputs the final results: **(a, topb)**

A local combiner can be used to compute local max before data gets reshuffled (in the map tasks)

- Multiple aggregates can be output by the reduce phase like **key = a and value = (sum(b), min(b))** etc.

- Sometimes a second (third etc) level of Map-Reduce phase might be needed

Any benefit of hash-partitioning for Map-Reduce?

```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```

- **For MapReduce**
  - Logically, MR won't know that the data is hash-partitioned
  - MR treats map and reduce functions as black-boxes and does not perform any optimizations on them

- But, if a local combiner is used
  - Saves communication cost:
    - fewer tuples will be emitted by the map tasks
  - Saves computation cost in the reducers:
    - the reducers would have to do anything