An overview of Map-Reduce & Parallel DBMS

Introduction to Databases

CompSci 316 Spring 2020

DUKE
COMPUTER SCIENCE

So far: One query/update
One machine

Multiple query/updates
One machine

Transactions

One query/update
Multiple machines

Parallel query processing
Map-Reduce, Spark, ..
Distributed query processing

Multiple query/updates, multiple machines:
Distributed transactions, Two-Phase Commit protocol, .. (not covered)

An overview of Map-Reduce

MapReduce: motivation

- Many problems can be processed in this pattern:
  - Given a lot of unsorted data
  - Map: extract something of interest from each record
  - Shuffle: group the intermediate results in some way
  - Reduce: further process (e.g., aggregate, summarize, analyze, transform) each group and write final results

  (Customize map and reduce for problem at hand)

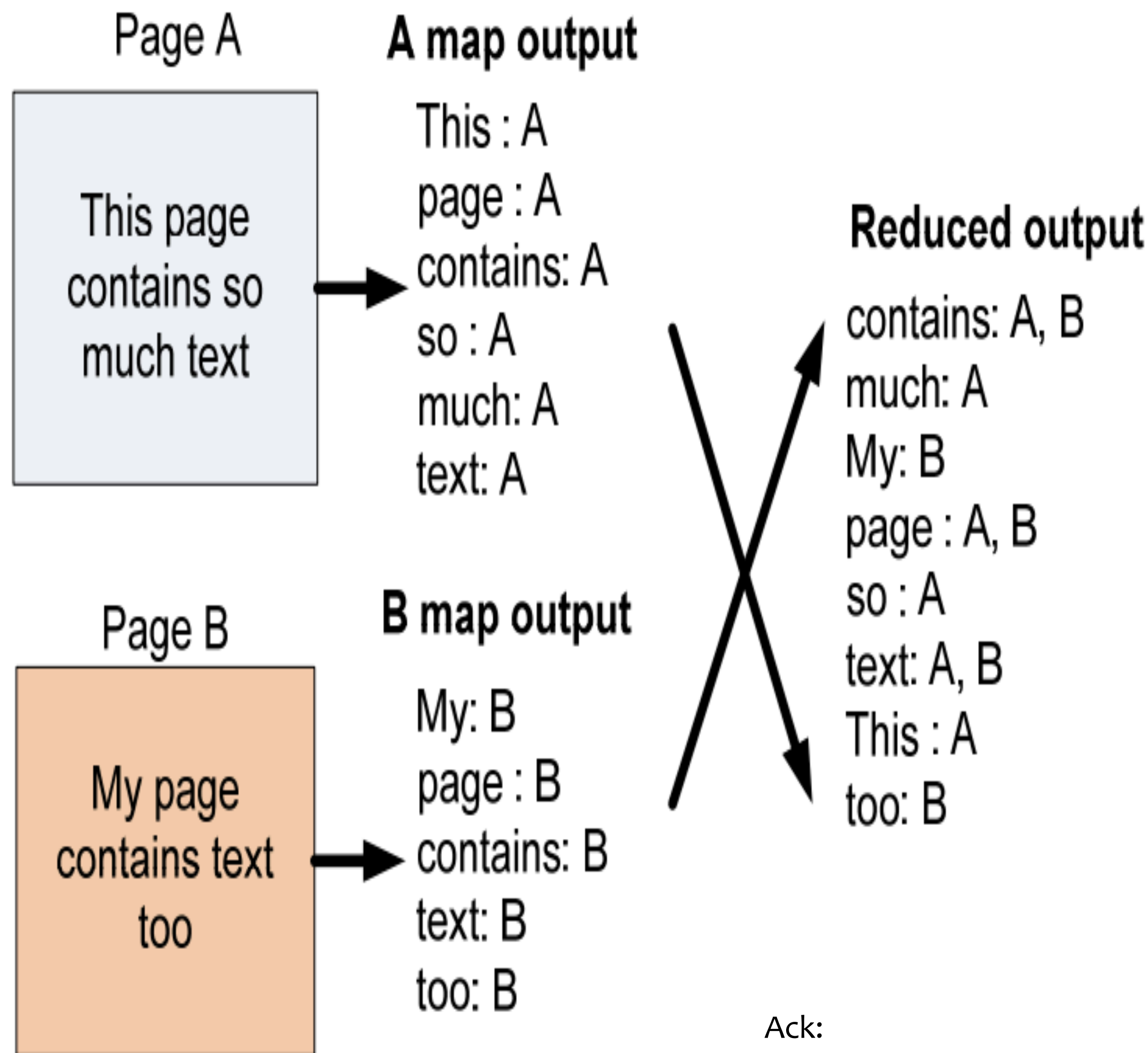
☞ Make this pattern easy to program and efficient to run
  - Original Google paper in *OSDI* 2004
  - Hadoop is most popular open-source implementation
  - Spark still supports it

M/R programming model

- Input/output: each a collection of key/value pairs
- Programmer specifies two functions
  - $\mathrm{map}(k_1, v_1) \rightarrow \mathrm{list}(k_2, v_2)$
    - Processes each input key/value pair, and produces a list of intermediate key/value pairs
  - $\mathrm{reduce}\big(k_2, \mathrm{list}(v_2)\big) \rightarrow \mathrm{list}(v_3)$
    - Processes all intermediate values associated with the same key, and produces a list of result values (usually just one for the key)

Simple Example: Map-Reduce

- Word counting
- Inverted indexes

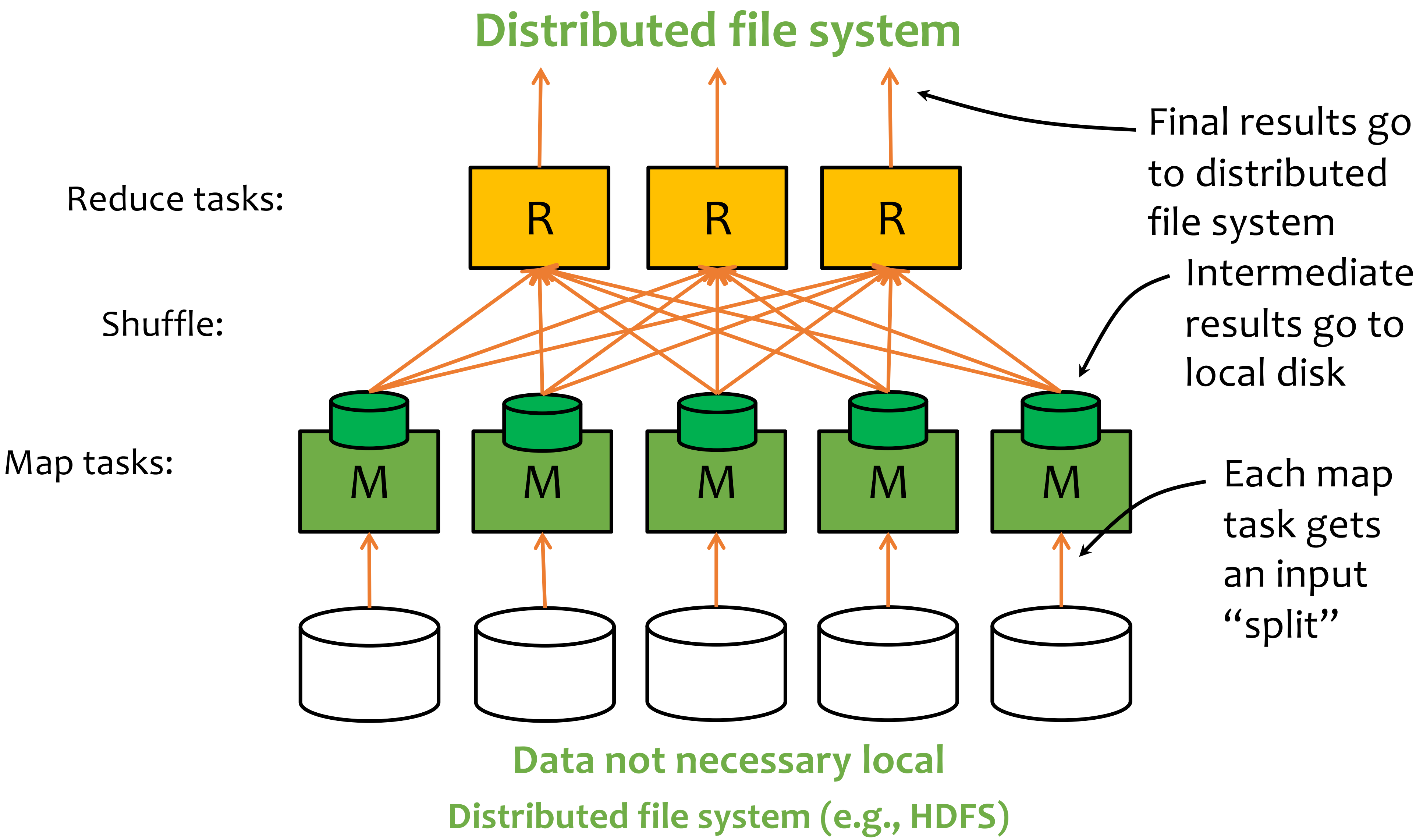


Ack:
Slide by Prof. Shivnath Babu

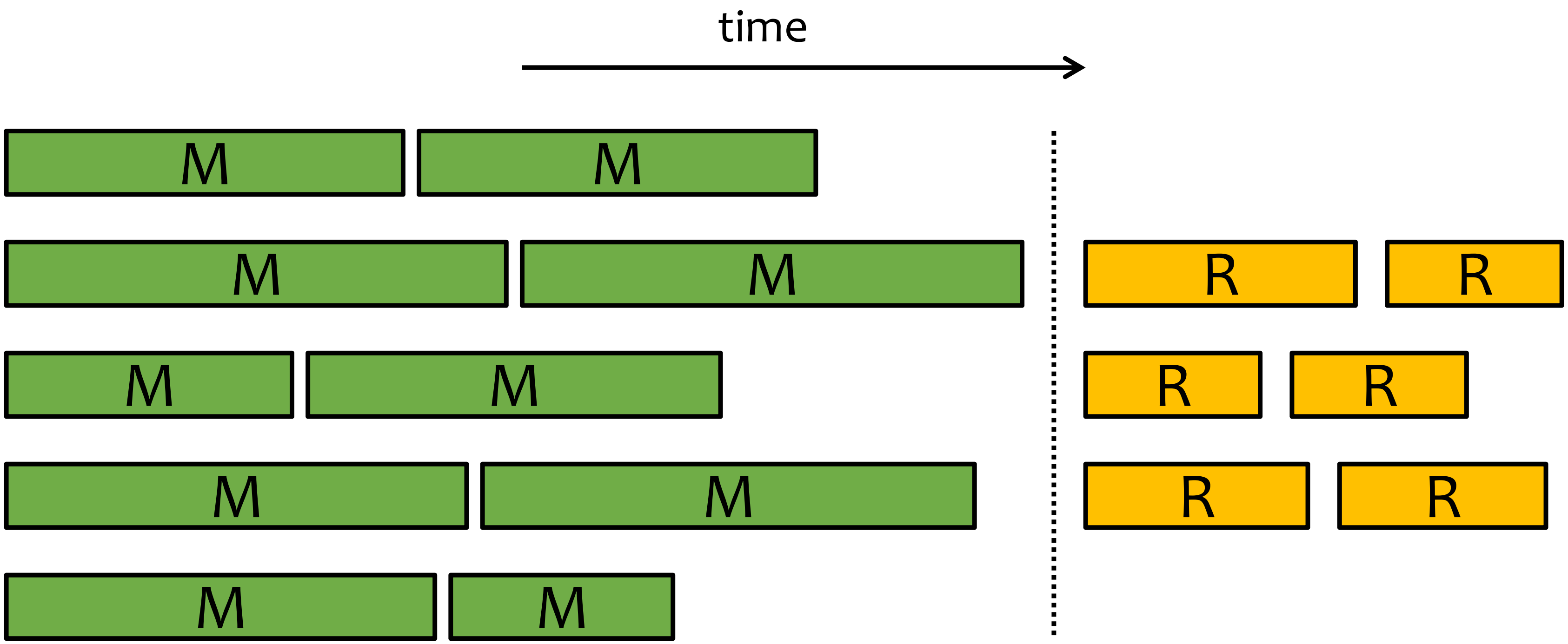
A similar M/R example: word count

- Expected input: a huge file (or collection of many files) with millions of lines of English text
- Expected output: list of (word, count) pairs
- Implementation
  - $\text{map}(\_, \text{line}) \rightarrow \text{list}(\text{word}, \text{count})$
    - Given a line, split it into words, and output $(w, 1)$ for each word w in the line
  - $\text{reduce}\big(\text{word}, \text{list}(\text{count})\big) \rightarrow (\text{word}, \text{count})$
    - Given a word w and list L of counts associated with it, compute $s = \sum_{\text{count} \in L} \text{count}$ and output (w, s)
  - Optimization: before shuffling, map can pre-aggregate word counts locally so there is less data to be shuffled
    - This optimization can be implemented in Hadoop as a "combiner"

M/R execution



M/R execution timeline



- When there are more tasks than workers, tasks execute in "waves"
  - Boundaries between waves are usually blurred
- Reduce tasks can't start until all map tasks are done

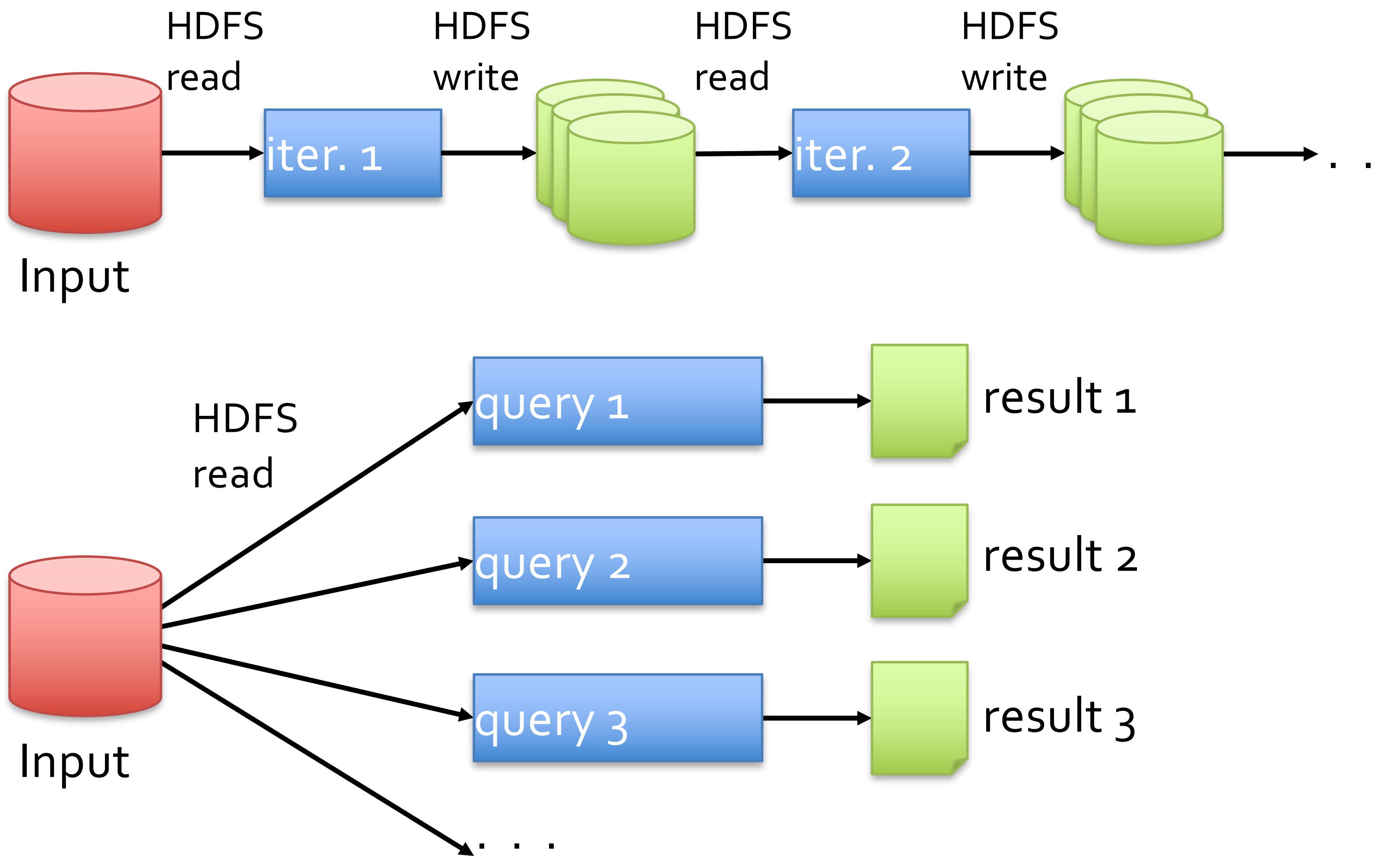
Issues with M/R

- Numbers of map and reduce tasks
  - Larger is better for load balancing
  - But more tasks add overhead and communication
- Worker failure
  - Master pings workers periodically
  - If one is down, reassign its split/region to another worker
- "Straggler": a machine that is exceptionally slow
  - Pre-emptively run the last few remaining tasks redundantly as backup

Why did we need a new programming model "Spark"?

- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - More complex, multi-stage iterative applications (graph algorithms, machine learning)
  - More interactive ad-hoc queries
  - More real-time online processing
- All three of these apps require fast data sharing across parallel jobs

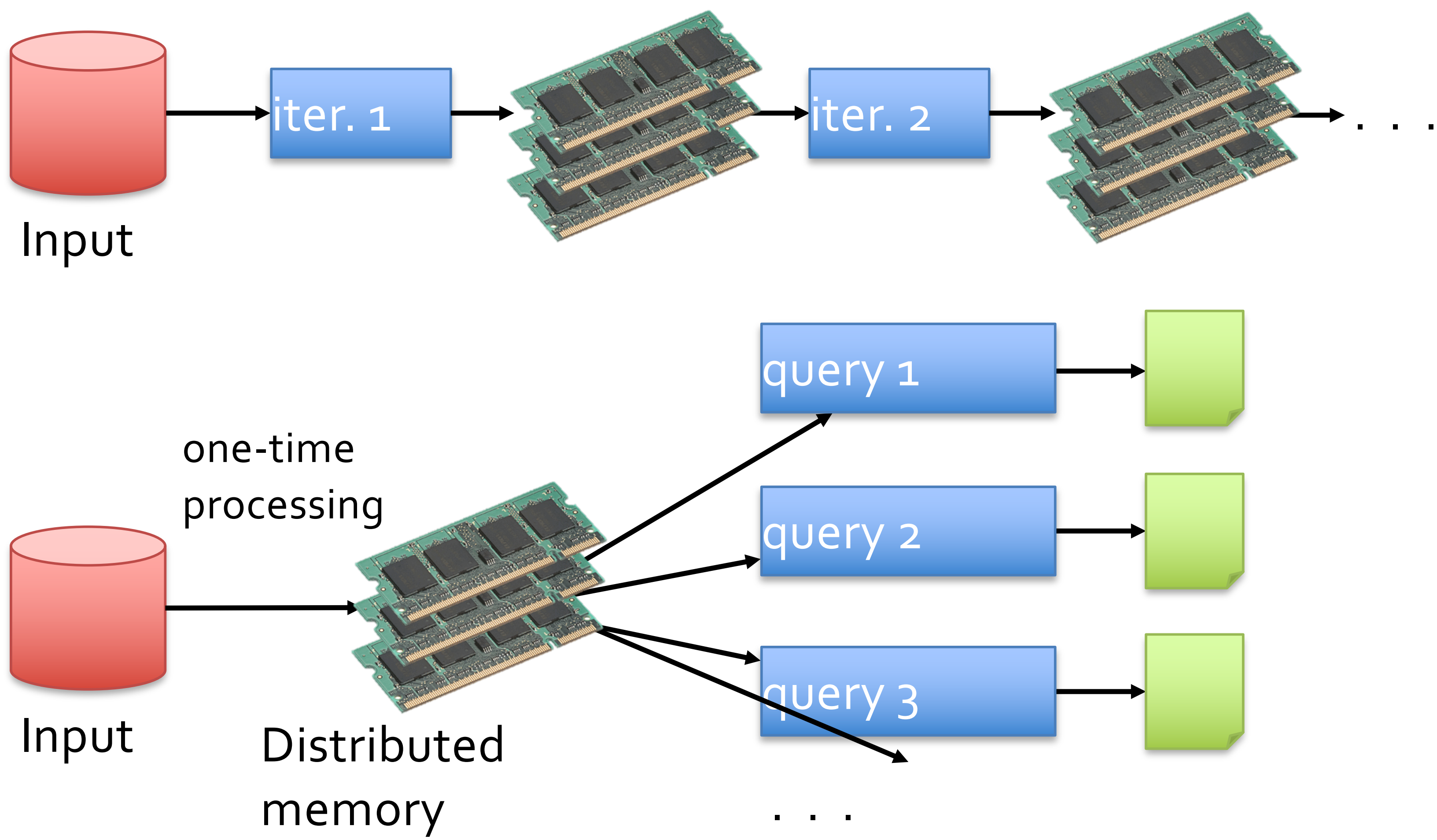
Data Sharing in MapReduce



Slow due to replication, serialization, and disk IO

Data Sharing in Spark

10-100× faster than network and disk



In addition, stores all intermediate results and lineage as
Resilient Distributed Datasets (RDDs) to avoid Recomputation from scratch after crashes

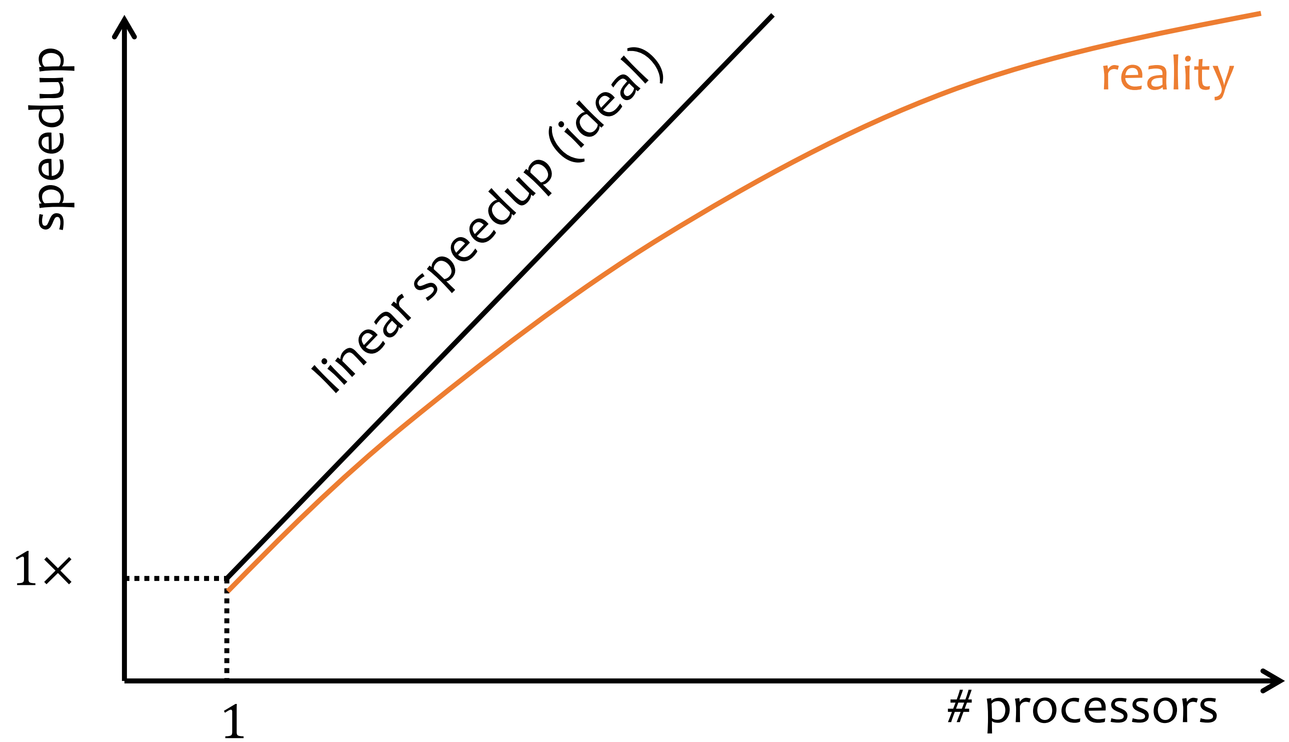
An overview of Parallel Databases

Parallel processing

- Improve performance by executing multiple operations in parallel
- Cheaper to scale than relying on a single increasingly more powerful processor

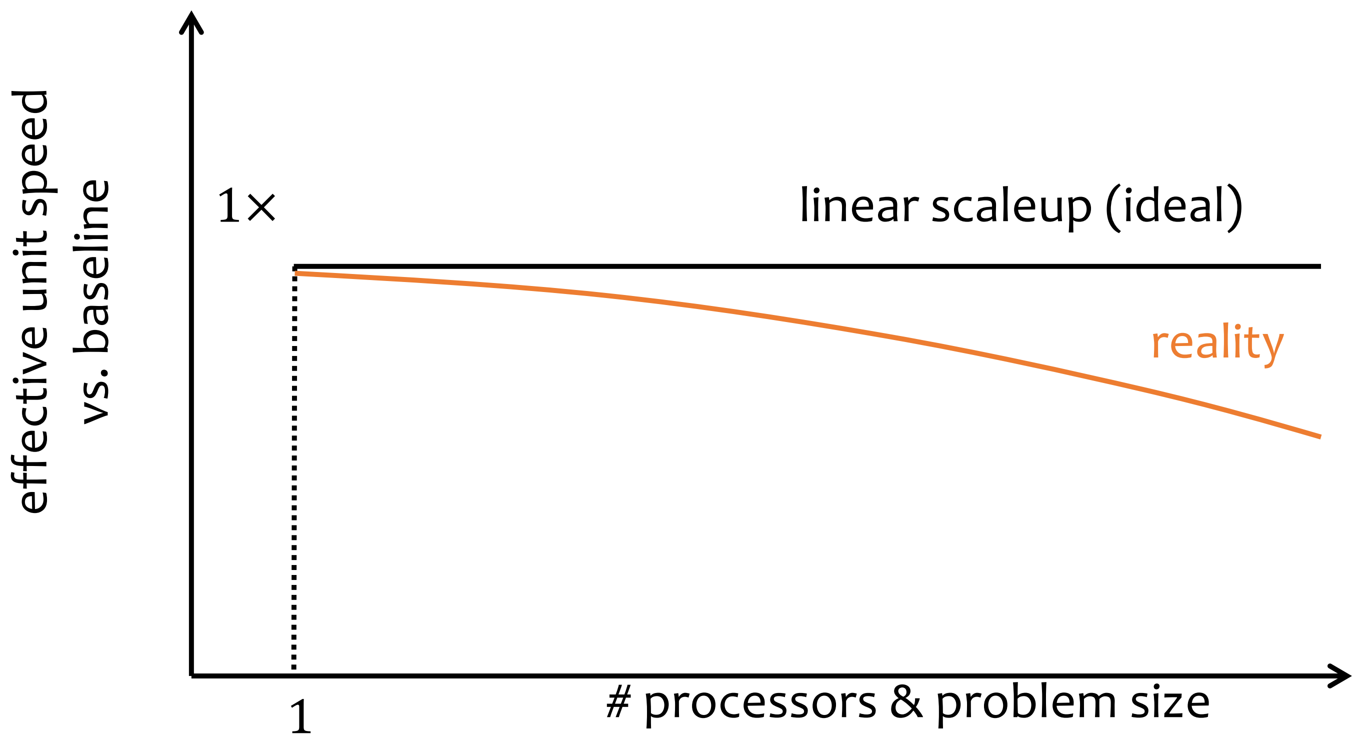
Speedup

- Increase # processors → how much faster can we solve the same problem?
  - Overall problem size is fixed



Scaleup

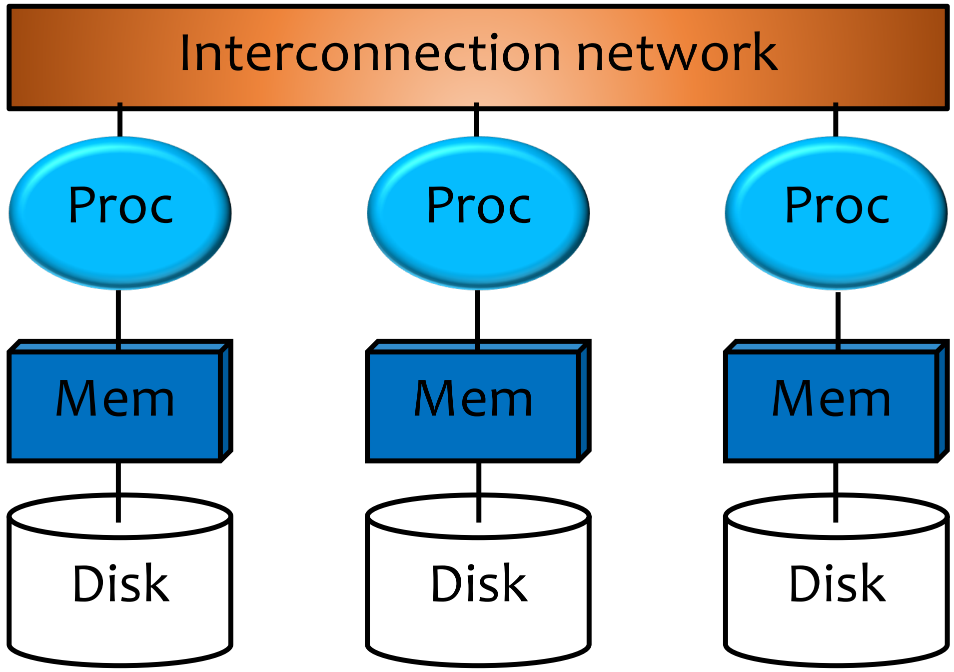
- Increase # processors and problem size proportionally → can we solve bigger problems in the same time?
  - Per-processor problem size is fixed



Why linear speedup/scaleup is hard

- Startup
  - Overhead of starting useful work on many processors
- Communication
  - Cost of exchanging data/information among processors
- Interference
  - Contention for resources among processors
- Skew
  - Slowest processor becomes the bottleneck

Shared-nothing architecture



- Most scalable (vs. shared-memory and shared-disk)
  - Minimizes interference by minimizing resource sharing
  - Can use commodity hardware
- Also most difficult to program

Horizontal data partitioning

- Split a table R into p chunks, each stored at one of the p processors
- Splitting strategies:
  - Round robin or block-partitioning distributes tuples arbitrarily but each processor gets the same amount of data (e.g., can assign the i-th row to chunk $(i \bmod p)$)
  - Hash-based partitioning on attribute A assigns row r to chunk $(h(r.A) \bmod p)$
  - Range-based partitioning on attribute A partitioning the range of $R.A$ values into p ranges, and assigns row r to the chunk whose corresponding range contains $r.A$

Practice Problem: Parallel DBMS

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 <u>block partitioned</u> across machines).

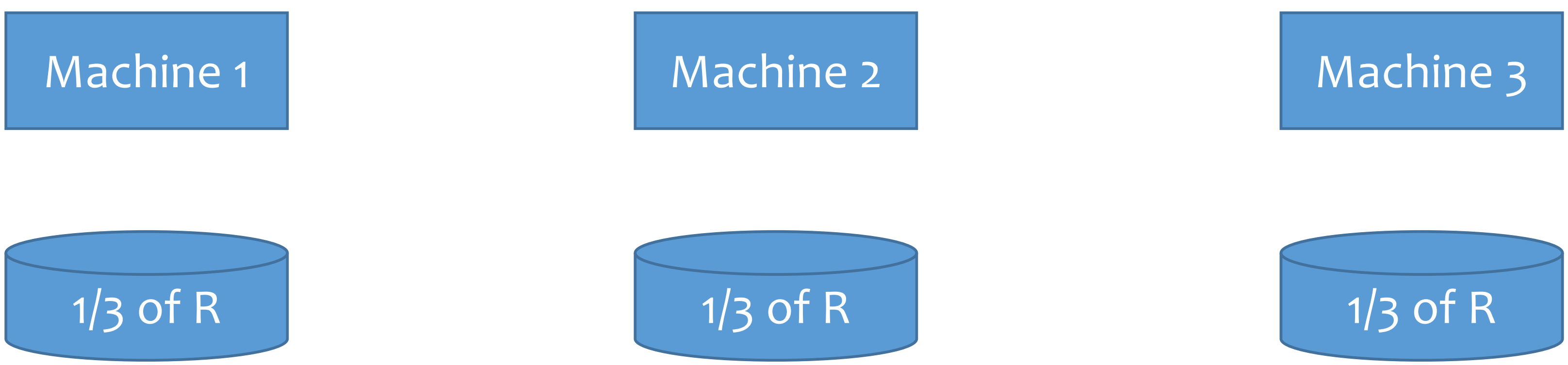
Show a RA plan for this query and how it will be executed across the N = 3 machines.

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)

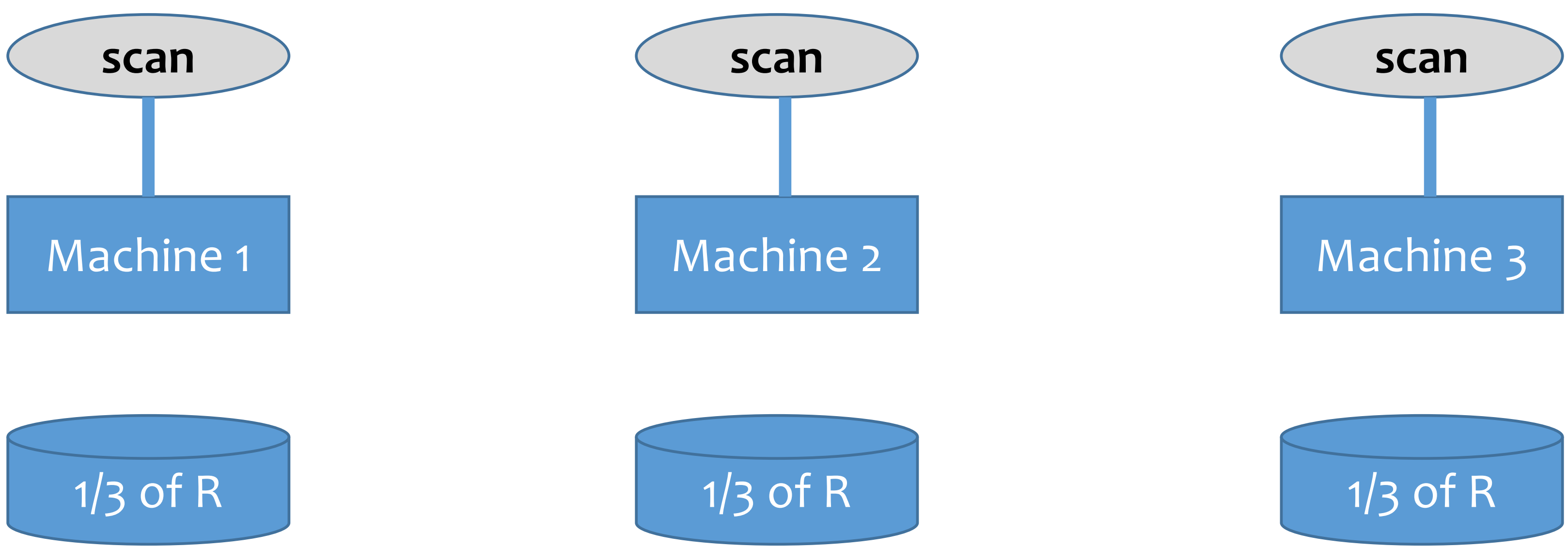
```
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
```



R(a, b)

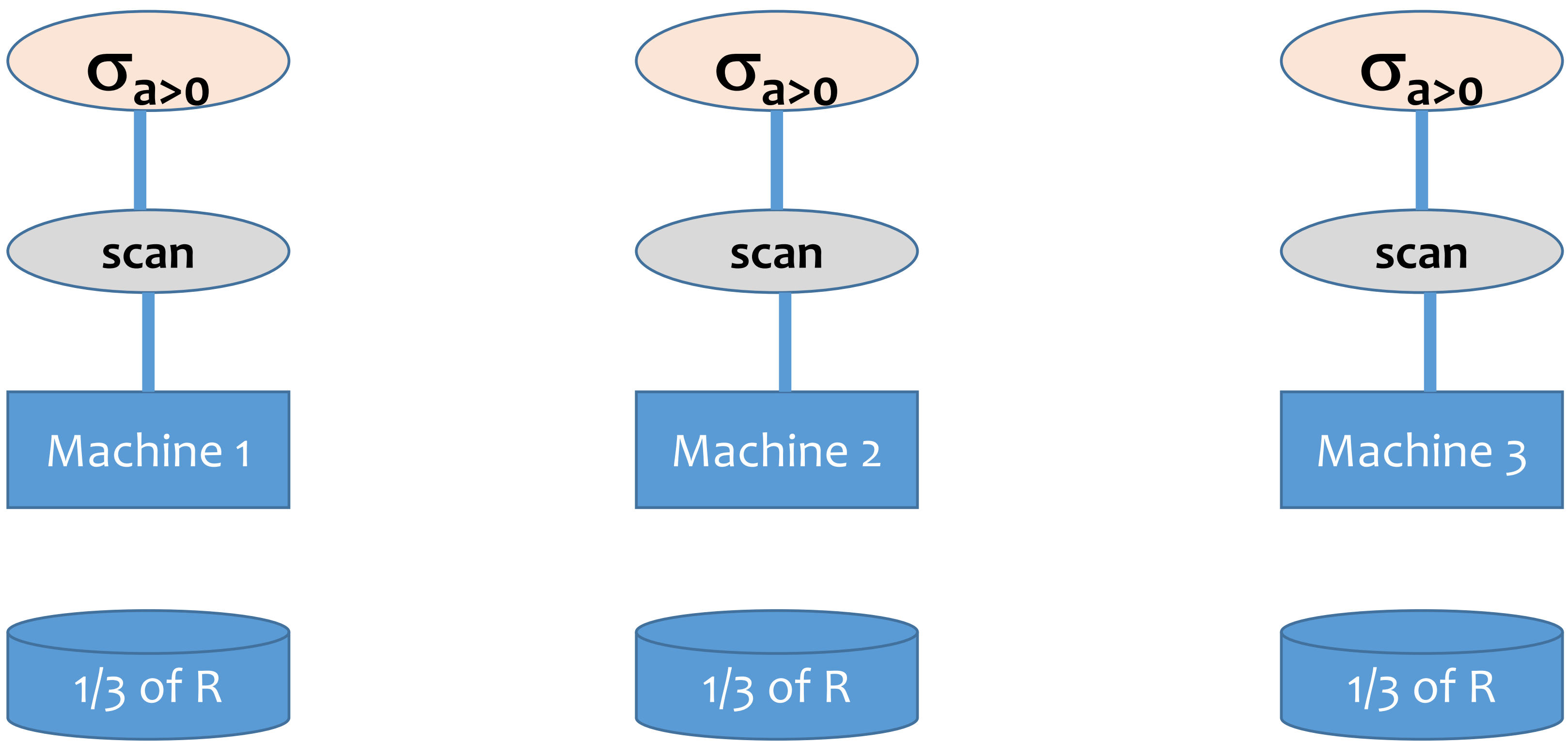
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a

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



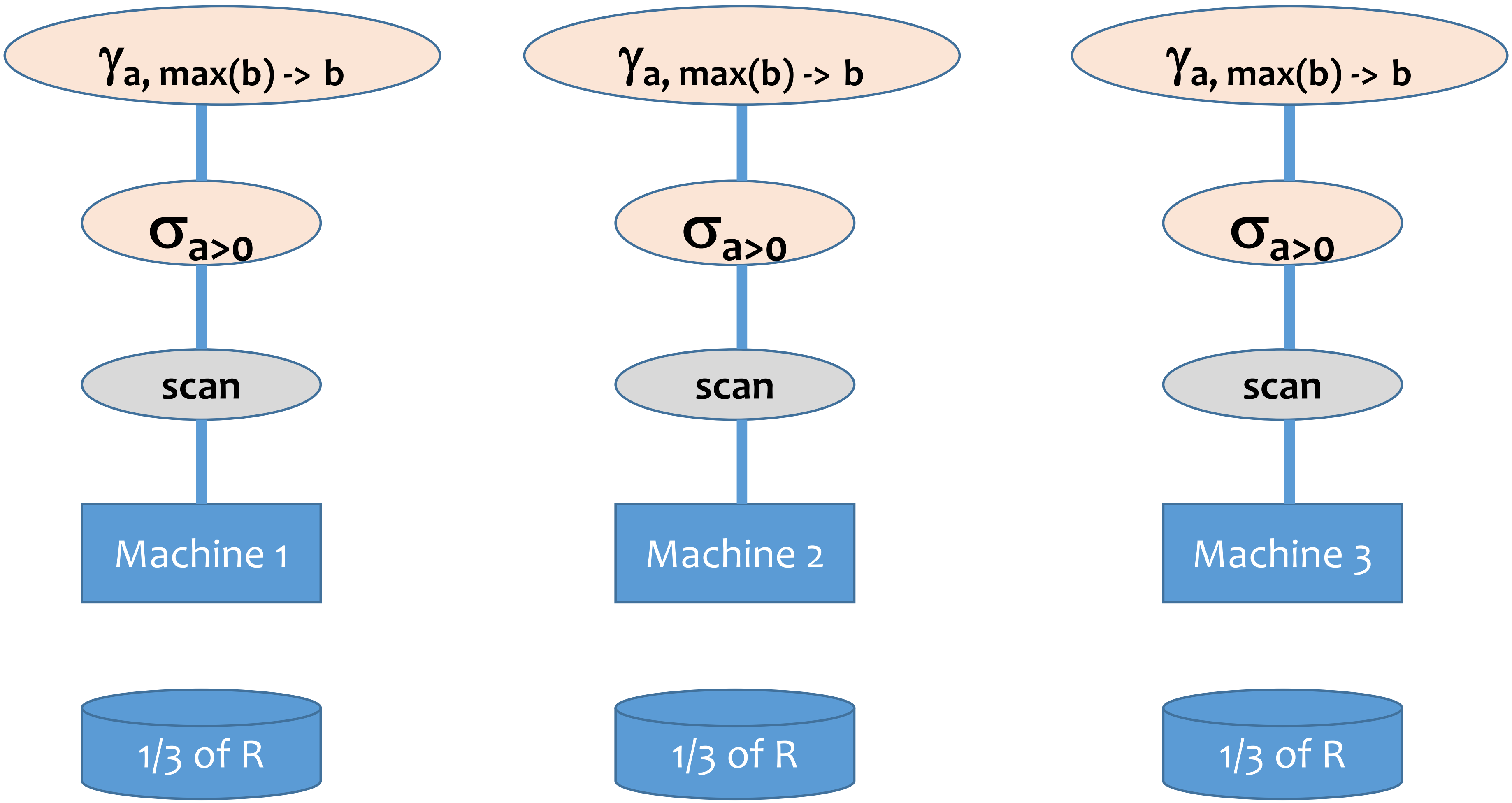
R(a, b)

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



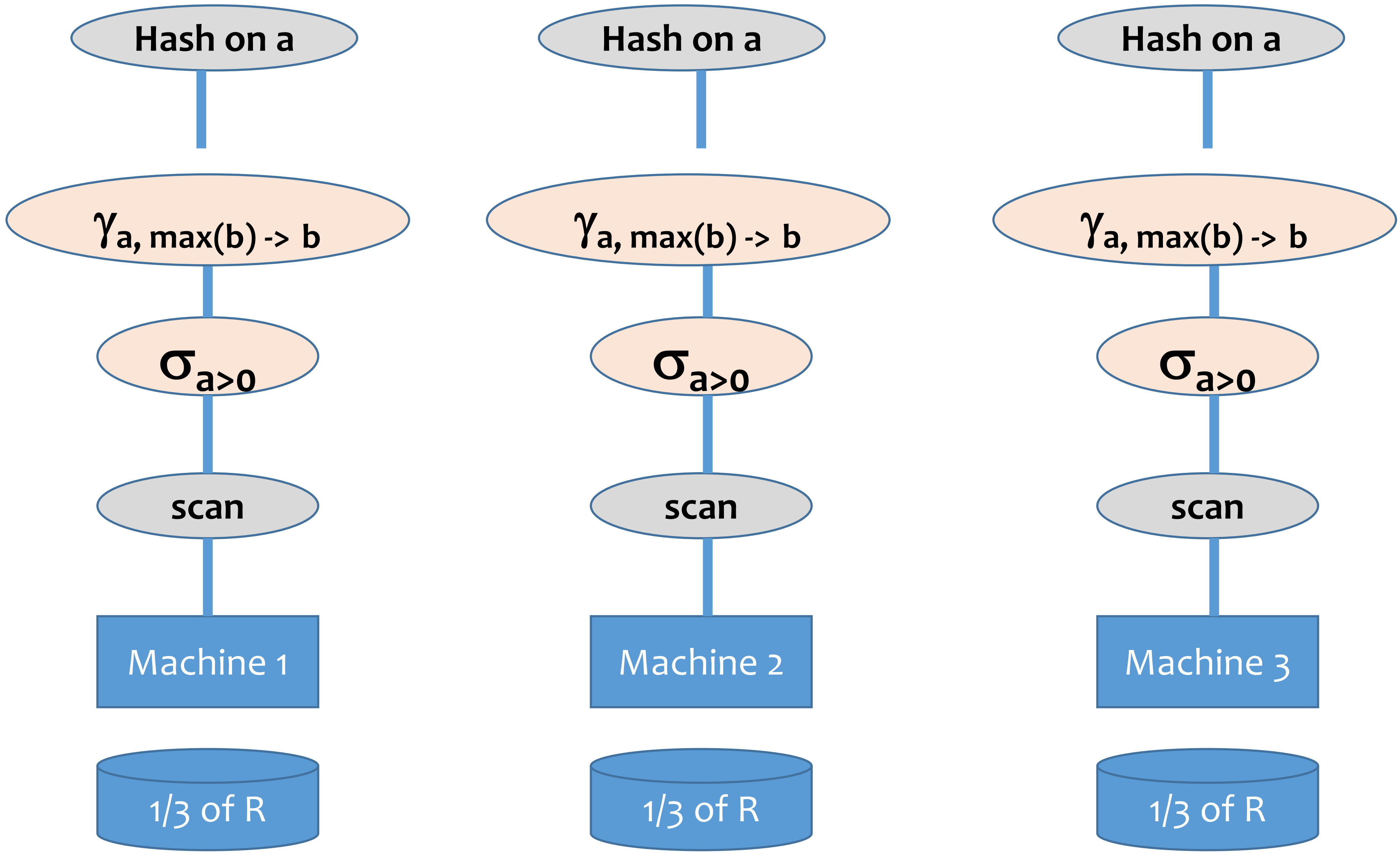
R(a, b)

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



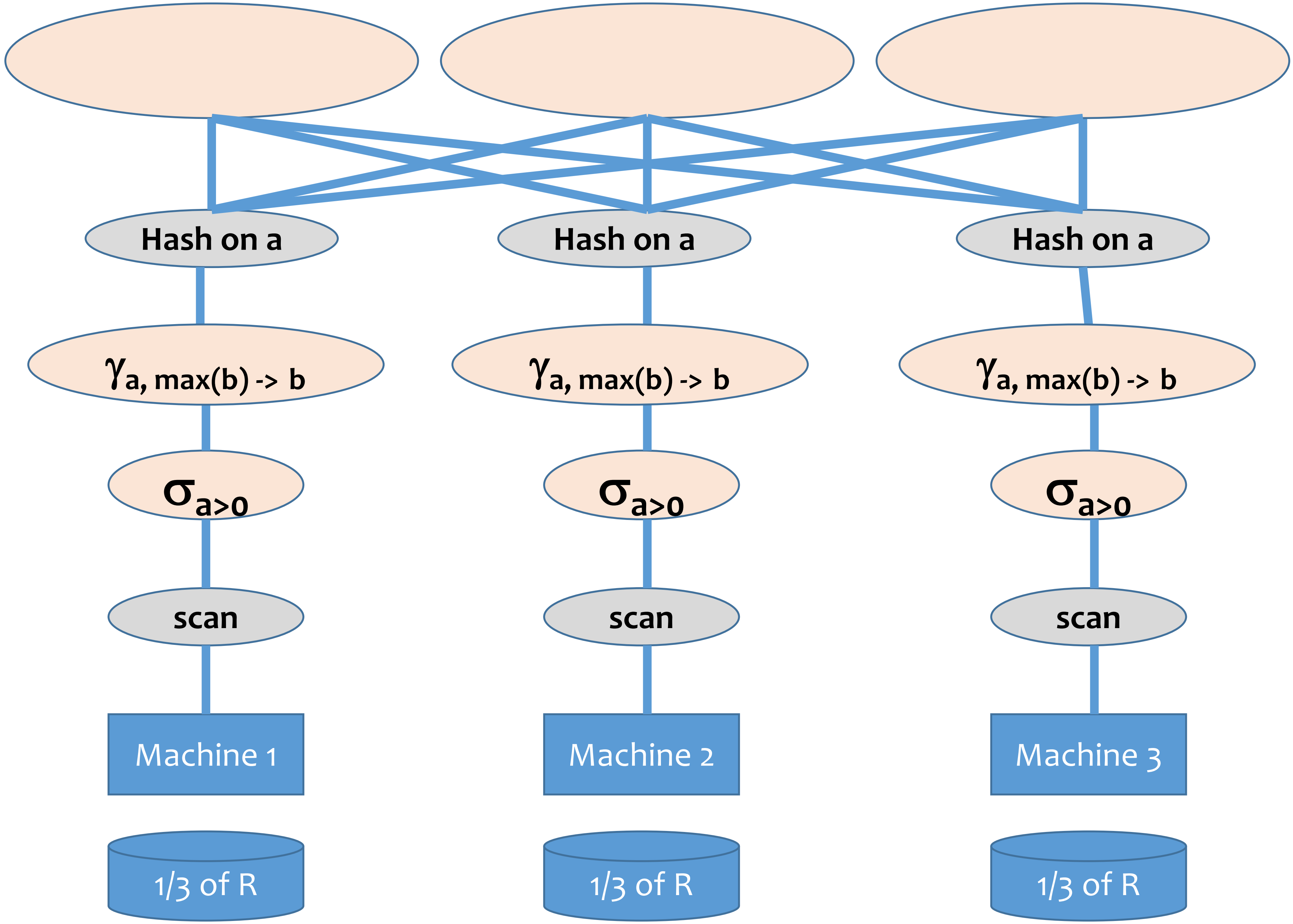
R(a, b)

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



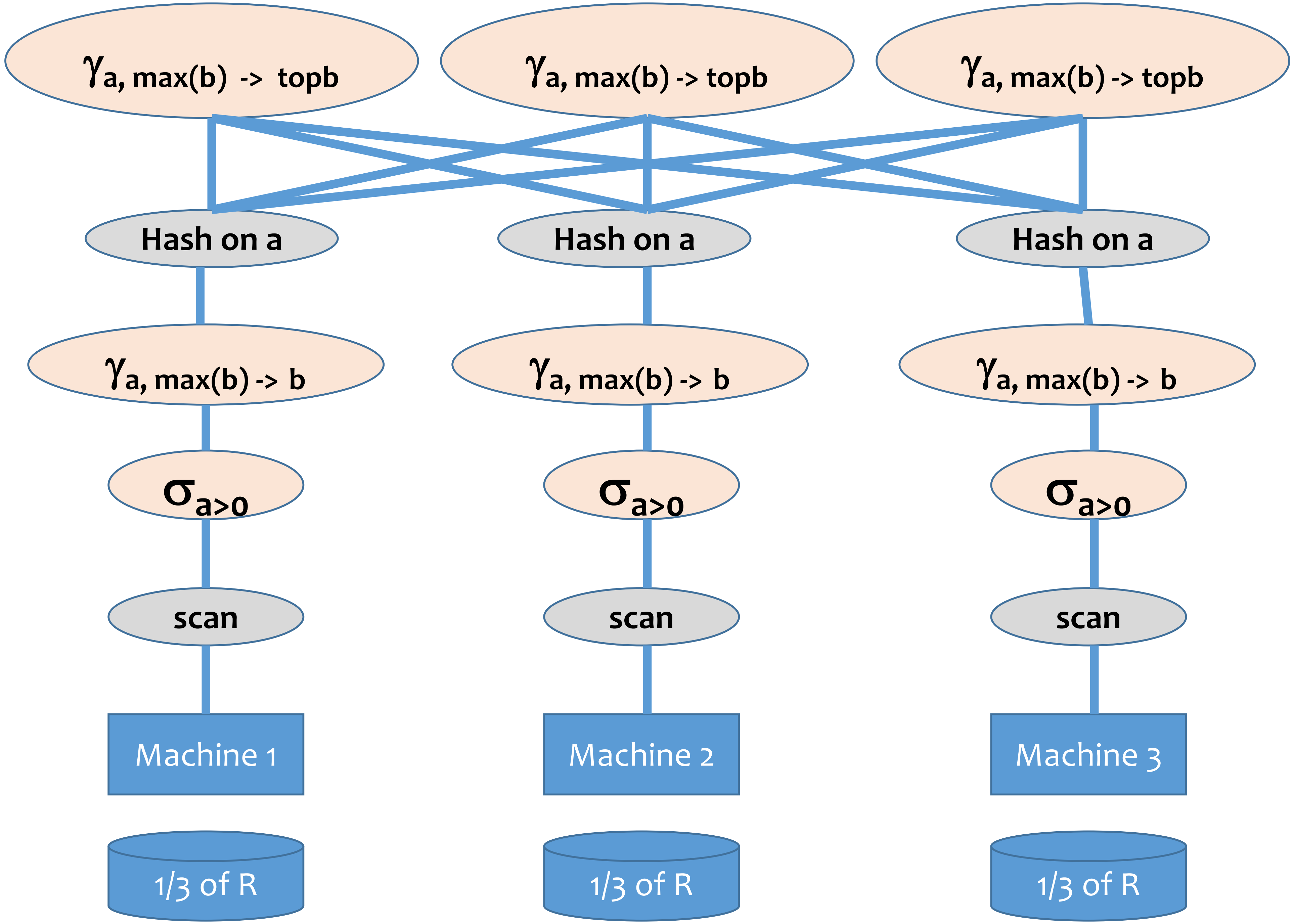
R(a, b)

SELECT a, max(b) as topb
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



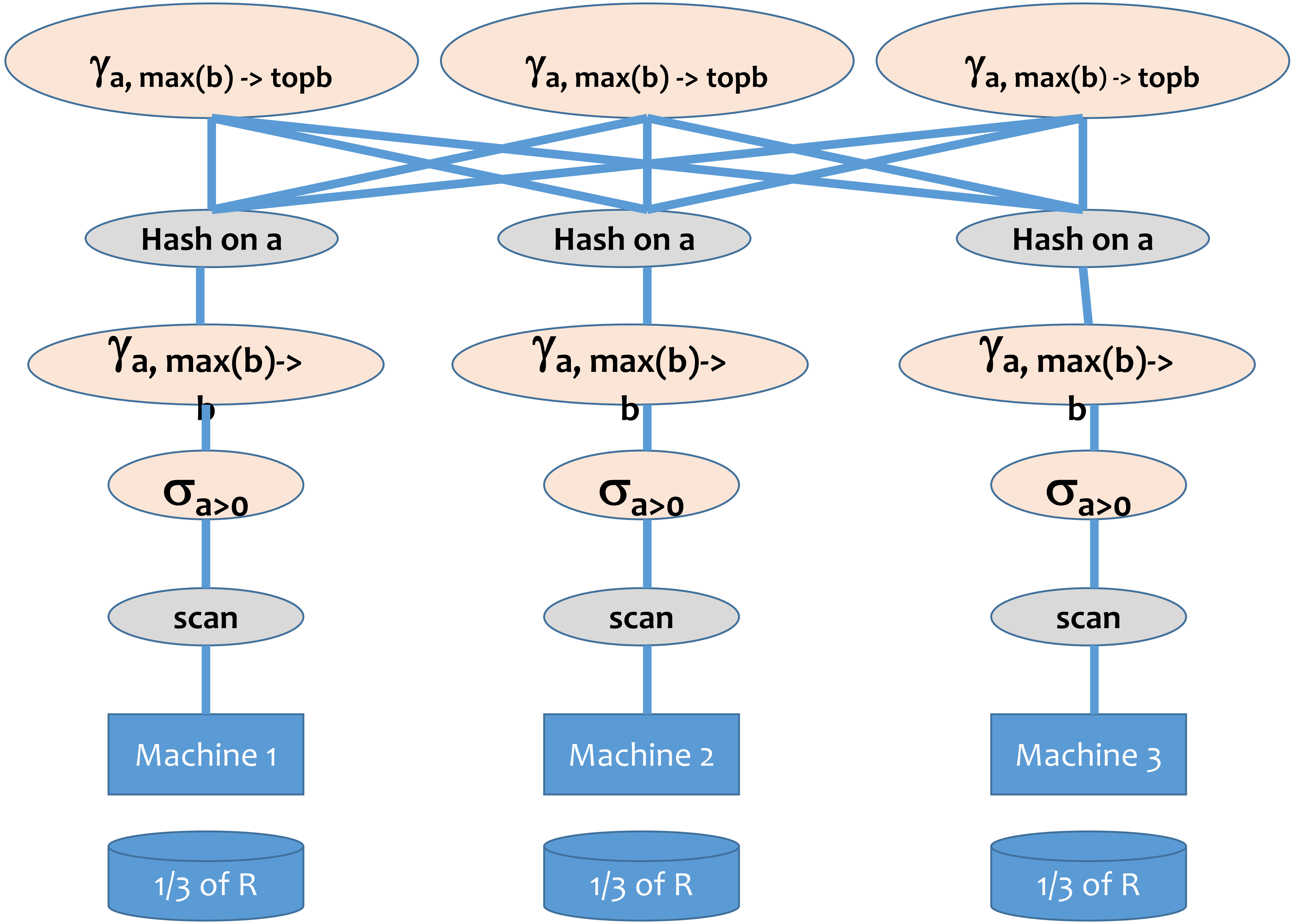
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a

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 and MR

Prev: block-partition

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



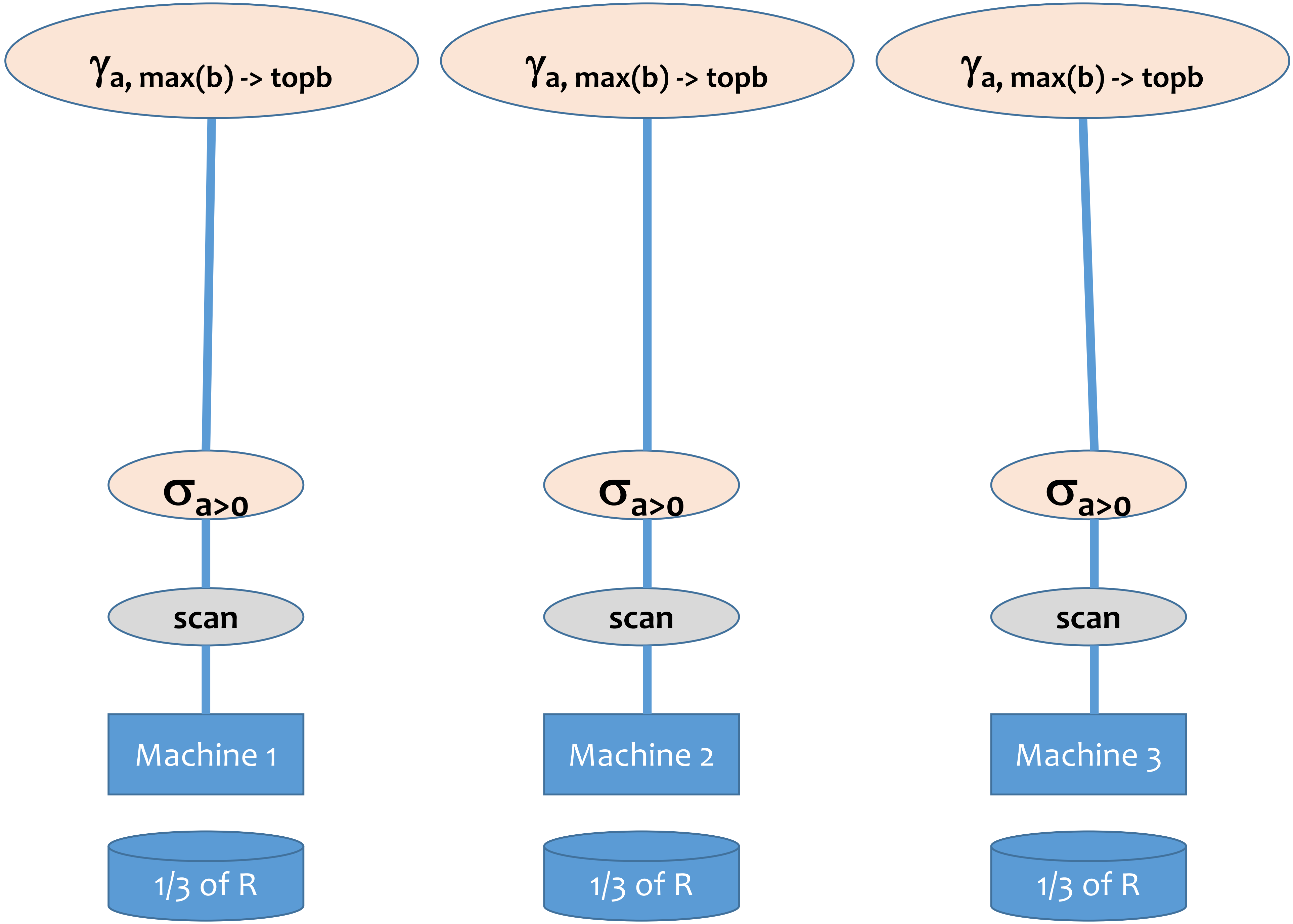
Hash-partition on a for R(a, b)

```
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
FROM R
WHERE a > 0
GROUP BY a



A brief summary of three approaches

- "DB": parallel DBMS, e.g., Teradata
  - Same abstractions (relational data model, SQL, transactions) as a regular DBMS
  - Parallelization handled behind the scene, automatic optimizations
  - Transactions supported
- "BD (Big Data)" 10 years go: MapReduce, e.g., Hadoop
  - Easy scaling out (e.g., adding lots of commodity servers) and failure handling
  - Input/output in files, not tables
  - Parallelism exposed to programmers
  - Mostly manual optimization
  - No transactions/updates
- "BD" today: Spark
  - Compared to MapReduce: smarter memory usage, recovery, and optimization
  - Higher-level DB-like abstractions (but still no updates/transactions)

What are the "NOSQL" systems?

They have the ability to

- horizontally scale "simple read/write operations" throughput over many servers (e.g., joins are expensive or not supported)
- replicate and to distribute (partition) data over many servers
- a weaker concurrency model than ACID (BASE – Basically Available, Soft state, Eventually consistent)
- Efficiently use distributed indexes and RAM for data storage
- dynamically add new attributes to data records (like JSON)

- Example: MongoDB, CouchDB, Dynamo, MemBase…

Conclusions

- We discussed using a database system (queries), designing a database, database internals, and approaches to handling big data
- There are many more traditional and new DB topics that we could not cover
  - Recursion in SQL
  - Data mining and exploration
  - Query optimization
  - Distributed DBMS
  - NOSQL and new database systems
  - Data cleaning and uncertainty in data
  - …..
- If you are interested in database research or projects, we would be happy to discuss with you!
- Read carefully final exam rules and policy (and all announcements on sakai/piazza) - Final exam will be comprehensive -- all lectures are included
- Projects are due on 04/24 (Friday) including report, code, and video
- Please fill out course evals if you have not done that already!
- Good luck!