Parallel Data Processing

Introduction to Databases

CompSci 316 Spring 2019

DUKE COMPUTER SCIENCE

Announcements (Thu., Apr. 18)

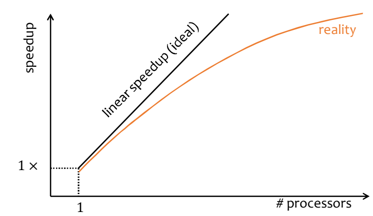
- Final project demo between April 29 (Mon)-May 1 (Wed)
  - If anyone in your group is unavailable during these dates and want to present your demo early please let Sudeepa and Zhengjie know ASAP!
- Homework #4 final due dates
  - Problem 3: today 04/16
  - Problems 4, 5, 6 : next Monday 04/22
  - Problem X1: next Wednesday 04/24

Parallel processing

- Improve performance by executing multiple operations in parallel
- Cheaper to scale than relying on a single increasingly more powerful processor
- Performance metrics
  - Speedup, in terms of completion time
  - Scaleup, in terms of time per unit problem size
  - Cost: completion time × # processors × (cost per processor per unit time)

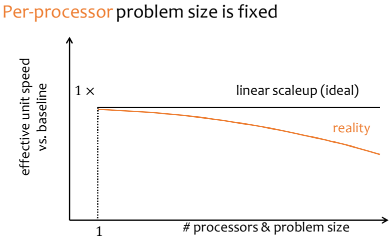
Speedup

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



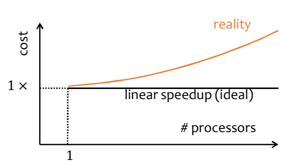
Scaleup

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

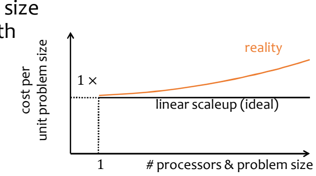


Cost

- Fix problem size



- Increase problem size proportionally with # processors



7

Why linear speedup/scaleup is hard

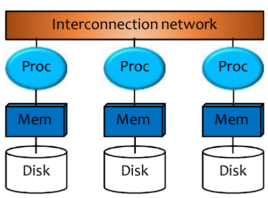
8

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

9

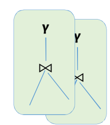
Shared-nothing architecture



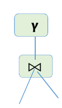
- Most scalable (vs. shared-memory and shared-disk)
  - Minimizes interference by minimizing resource sharing
  - Can use commodity hardware
- Also most difficult to program

10

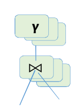
Parallel query evaluation opportunities



- Inter-query parallelism
  - Each query can run on a different processor
- Inter-operator parallelism
  - A query runs on multiple processors
  - Each operator can run on a different processor



- Intra-operator parallelism
  - An operator can run on multiple processors, each working on a different "split" of data/operation
  - ☞Focus of this lecture



11

Parallel DBMS

E.g.: TERADATA

12

Horizontal data partitioning

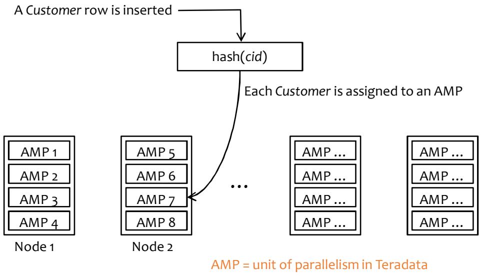
- Split a table R into p chunks, each stored at one of the p processors
- Splitting strategies?

Horizontal data partitioning

- Split a table R into p chunks, each stored at one of the p processors
- Splitting strategies:
  - Round robin assigns the i-th row assigned 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 $r.A$

Teradata: an example parallel DBMS

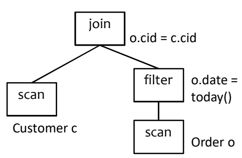
- Hash-based partitioning of *Customer* on *cid*



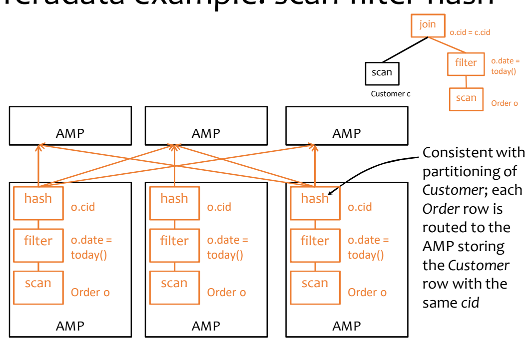
Example query in Teradata

- Find all orders today, along with the customer info

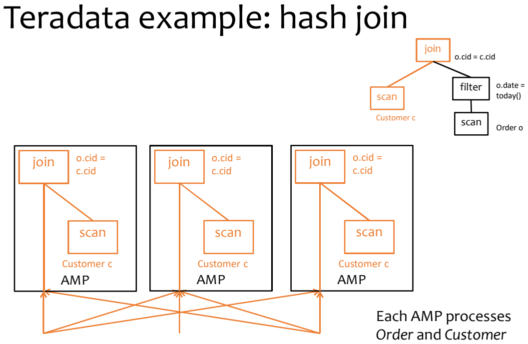
SELECT *
FROM Order o, Customer c
WHERE o.cid = c.cid
AND o.date = today();



Teradata example: scan-filter-hash



Teradata example: hash join



Each AMP processes *Order* and *Customer* rows with the same *cid* hash

Parallel DBMS vs. MapReduce?

Parallel DBMS vs. MapReduce

- Parallel DBMS
  - Schema + intelligent indexing/partitioning
  - Can stream data from one operator to the next
  - SQL + automatic optimization
- MapReduce
  - No schema, no indexing
  - Higher scalability and elasticity
    - Just throw new machines in!
  - Better handling of failures and stragglers
  - Black-box map/reduce functions → hand optimization

A brief tour 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
- "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
- "BD" today: Spark
  - Compared to MapReduce: smarter memory usage, recovery, and optimization
  - Higher-level DB-like abstractions (but still no updates)

Summary

- "DB": parallel DBMS
  - Standard relational operators
  - Automatic optimization
  - Transactions
- "BD" 10 years go: MapReduce
  - User-defined map and reduce functions
  - Mostly manual optimization
  - No updates/transactions
- "BD" today: Spark
  - Still supporting user-defined functions, but more standard relational operators than older "BD" systems
  - More automatic optimization than older "BD" systems
  - No updates/transactions

Practice Problem:

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 (in no particular order).

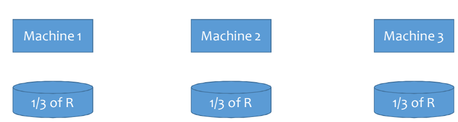
Show a RA plan for this query and how it will be executed across the N = 3 machines.

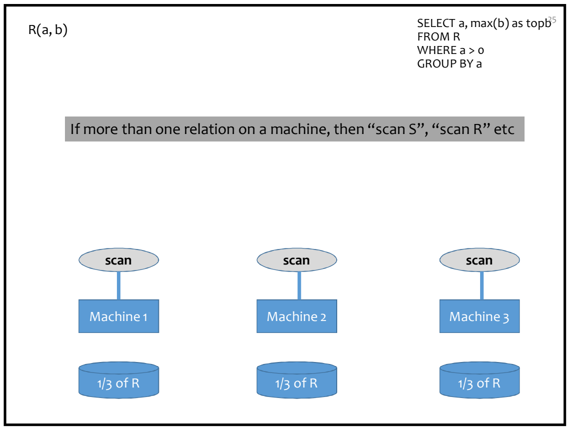
Pick an efficient plan that leverages the parallelism as much as possible.

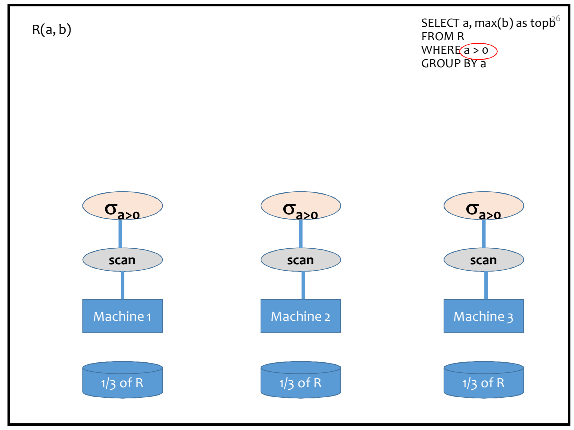
- **SELECT a, max(b) as topb**
- **FROM R**
- **WHERE a > 0**
- **GROUP BY a**

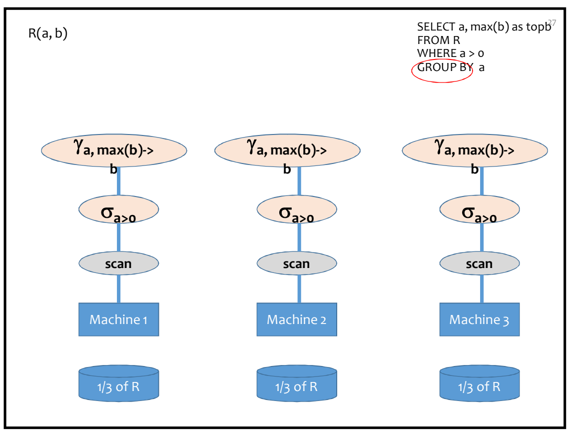
R(a, b)

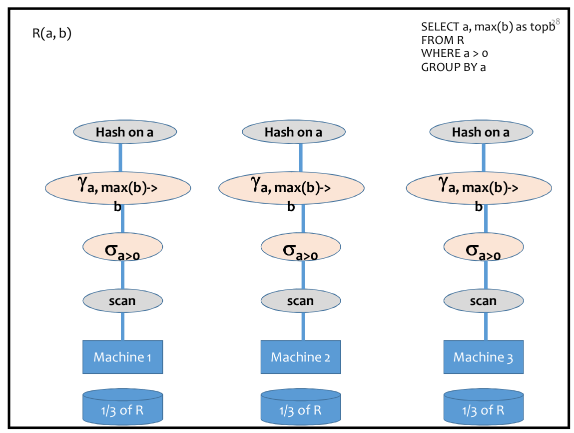
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a

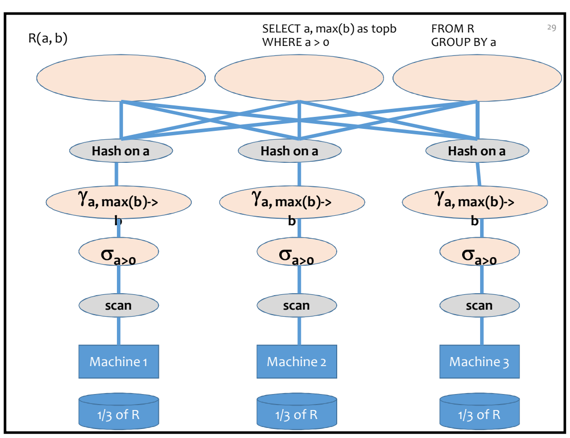


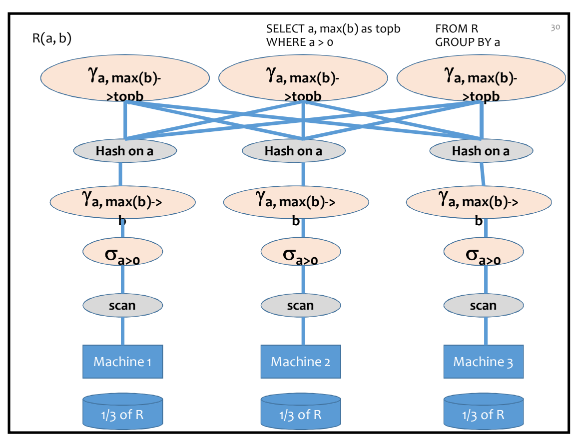

R(a, b)
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
25
If more than one relation on a machine, then "scan S", "scan R" etc
scan
scan
scan
Machine 1
Machine 2
Machine 3
1/3 of R
1/3 of R
1/3 of R


R(a, b)
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
26
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


R(a, b)
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
27
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


R(a, b)
SELECT a, max(b) as topb
FROM R
WHERE a > 0
GROUP BY a
28
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


R(a, b)
SELECT a, max(b) as topb
WHERE a > 0
FROM R
GROUP BY a
29
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


R(a, b)
SELECT a, max(b) as topb
WHERE a > 0
FROM R
GROUP BY a
30
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

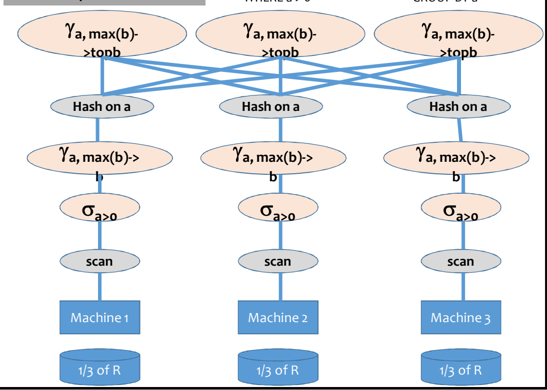
Benefit of hash-partitioning

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

- 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
WHERE a > 0
FROM R
GROUP BY a



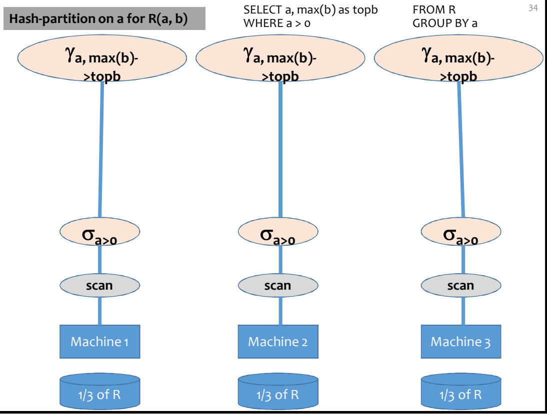
Hash-partition on a for R(a, b)

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
WHERE a > 0
FROM R
GROUP BY a



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

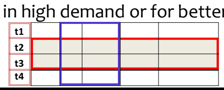
Distributed Data Processing

- Distributed replication & updates
- Distributed join (Semijoin)
- Distributed Recovery (2-phase commit)

37

1. Distributed replication and updates

- Relations are stored across several sites
  - Accessing data at a remote site incurs message-passing costs
- A single relation may be divided into smaller fragments and/or replicated
  - Fragmented - typically at sites where they are most often accessed
    - Horizontal partition: E.g. SELECT on city to store employees in the same city locally
    - Vertical partition: store some columns along with id (lossless?)
  - Replicated – when the relation is in high demand or for better fault tolerance



38

Updating Distributed Data

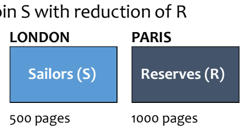
- Synchronous Replication: All copies of a modified relation must be updated before the modifying transaction commits
  - Voting: write a majority of copies, read enough
    - E.g. 10 copies, write any 7, read any 4 (why 4? Why read < write?)
    - Read any write all : read any copy, write all
  - Expensive remote lock requests, expensive commit protocol
- Asynchronous Replication: Copies of a modified relation are only periodically updated; different copies may get out of sync in the meantime
  - Users must be aware of data distribution
  - More efficient – many current products follow this approach
  - E.g. Have one primary copy (updateable), multiple secondary copies(not updateable, changes propagate eventually)

39

2. Distributed join -- Semijoin

- Suppose want to ship R to London and then do join with S at London. May require unnecessary shipping.
- Instead,

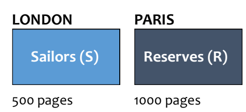
1. At London, project S onto join columns and ship this to Paris
   - Here foreign keys, but could be arbitrary join
2. At Paris, join S-projection with R
   - Result is called reduction of Reserves w.r.t. Sailors (only these tuples are needed)
3. Ship reduction of R to back to London
4. At London, join S with reduction of R



40

Semijoin – contd.

- Tradeoff the cost of computing and shipping projection for cost of shipping full R relation
- Especially useful if there is a selection on Sailors, and answer desired at London



41

3. Distributed Recovery (details skipped)

- Two new issues:
  - New kinds of failure, e.g., links and remote sites
  - If "sub-transactions" of a transaction execute at different sites, all or none must commit
  - Need a commit protocol to achieve this
  - Most widely used: Two Phase Commit (2PC)
- A log is maintained at each site
  - as in a centralized DBMS
  - commit protocol actions are additionally logged
  - One coordinator and rest subordinates for each transaction
  - Transaction can commit only if *all* sites vote to commit

42

Parallel vs. Distributed DBMS?

43

Parallel vs. Distributed DBMS

Parallel DBMS	Distributed DBMS

• Parallelization of various operations
  • e.g. loading data, building indexes, evaluating queries
 	
• Data may or may not be distributed initially
 	
• Distribution is governed by performance consideration
	• Data is physically stored across different sites
	  – Each site is typically managed by an independent DBMS
	
	• Location of data and autonomy of sites have an impact on Query opt., Conc. Control and recovery
	
	• Also governed by other factors:
	  – increased availability for system crash
	  – local ownership and access