Parallel and Distributed Databases

UA.DETI.CBD

José Luis Oliveira / Carlos Costa

Centralized vs Distributed Databases

- Centralized database
  - Data is located in one place (e.g. server)
  - All DBMS functionalities are done by the server
    - Enforcing ACID properties of transactions
    - Concurrency control, recovery mechanisms
    - Answering queries

- Distributed databases
  - Data is stored in multiple places (each is running a DBMS)
  - New notion of distributed transactions
  - DBMS functionalities are now distributed over many machines

Why Distributed Databases?

- ***Scalability***
  - If your **data volume**, **read load** or **write load** grows bigger than a single machine can handle, you can potentially **spread** the load across **multiple machines**.
- ***Fault tolerance / High availability***
  - If your application needs to continue working, even if one machine (or several machines) goes down, you can use multiple machines to give you **redundancy**. When one fails, another one can take over.
- ***Latency***
  - **Applications** are by **nature distributed**. If you have users around the world, you might want to have servers at various locations worldwide, so that **users** can be **served** from a **datacenter** that is geographically **close** to them.

Why Parallel Processing?

- Processing 1 Terabyte
  - at 10MB/s => ~1.2 days to scan
  - 1000 x parallel => 1.5 minute to scan

- **Divide** a **big problem** into **many smaller ones** to be solved in **parallel**

- **Large-scale parallel database** systems increasingly **used** for:
  - storing large volumes of data
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing

Biggest Database Problem

- Large volume of data $\Rightarrow$ use disk and large memory
- Bottlenecks
  - Speed(disk) << speed(RAM) << speed(microprocessor)
- Predictions
  - Moore's law: processor speed growth (with multicore): 50 % per year
  - DRAM capacity growth: 4 × every three years
  - Disk throughput: 2 × in the last ten years
- **Biggest problem: I/O bottleneck**
- **Solution** to increase the I/O bandwidth
  - **data partitioning**
  - **parallel data access**

Parallel Databases

- Parallel databases **improve processing** and **I/O speeds** by using multiple CPUs and disks in parallel
  - data can be partitioned across multiple disks
  - each processor can work independently on its own partition
- Exploit the parallelism in data management in order to deliver **high-performance**, **high-availability** and **extensibility**
  - support very large databases with very high loads
- Different **queries** can be **run** in **parallel**
- Concurrency control takes care of conflicts

Parallel Databases (cont)

- **Critical issues**
  - data placement
  - parallel query processing
  - load balancing

- **Most research** has been done in the **context** of the **relational model** that provides a good basis for data-based parallelism
  - individual relational operations (e.g., sort, join, aggregation) can be executed in parallel

Parallel vs Distributed Databases

Although the **basic principles** of
parallel DBMS are the **same** as in **distributed DBMS**,
the **techniques** for parallel database systems **are** fairly **different**

typically...

Parallel DB

- Fast interconnect
- Homogeneous software
- High performance is goal
- Transparency is goal

Distributed DB

- Geographically distributed
- Data sharing is goal (may run into heterogeneity, autonomy)
- Disconnected operation possible

Parallel vs Distributed Databases

- **Distributed processing usually make use of parallel processing** (not vise versa)
  - can have parallel processing on a single machine

- **Assumptions**
  - **Parallel** Databases
    - Machines are physically close to each other (e.g. same server room)
    - Machines connects with dedicated high-speed LANs and switches
    - Communication cost is assumed to be small
    - Architecture: can be **shared-memory, shared-disk or shared-nothing**
  - **Distributed** Databases
    - Machines can be in distinct geographic locations
    - And connected using public-purpose network, e.g., Internet
    - Communication cost and problems cannot be ignored
    - Architecture: usually **shared-nothing**

Parallel DBMS – Main Goals

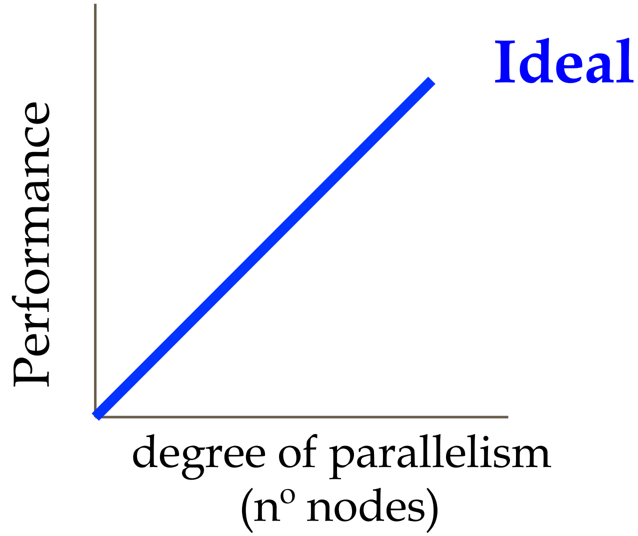
- **High-performance** through **parallelization** of various operations
  - **High throughput** with inter-query parallelism
  - **Low response time** with intra-operation parallelism
  - **Load balancing** is the ability of the system to divide a given workload equally among all processors

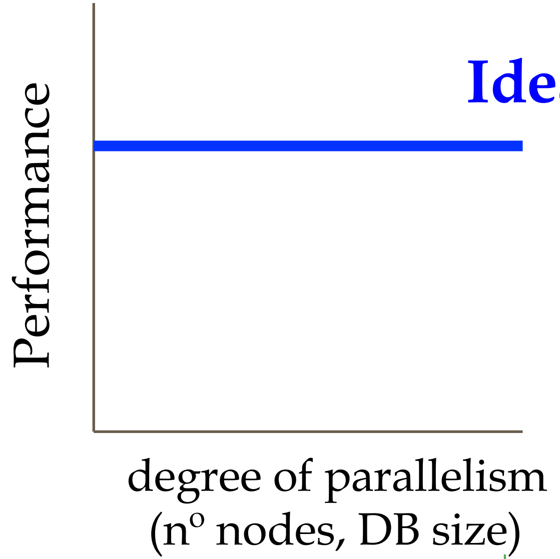
- **High availability** by exploiting **data replication**

- **Extensibility** with the ideal goals
  - Linear speed-up
  - Linear scale-up

Ideal Extensibility Scenario

- **Speed-Up**
  - refers to a **linear increase** in **performance** for a **constant database size** while the **number of nodes** (i.e. processing and storage power) are **increased linearly**
  - more resources means proportionally less time for given amount of data
- **Scale-Up**
  - refers to a **sustained performance** for a **linear increase** in both **database size** and **number of nodes**
  - if resources increased in proportion to increase in data size, time is constant





Barriers to Parallelism

- Startup
  - The time needed to start a parallel operation may dominate the actual computation time
- Interference
  - When accessing shared resources, each new process slows down the others (hot spot problem)
- Skew
  - The response time of a set of parallel processes is the time of the slowest one

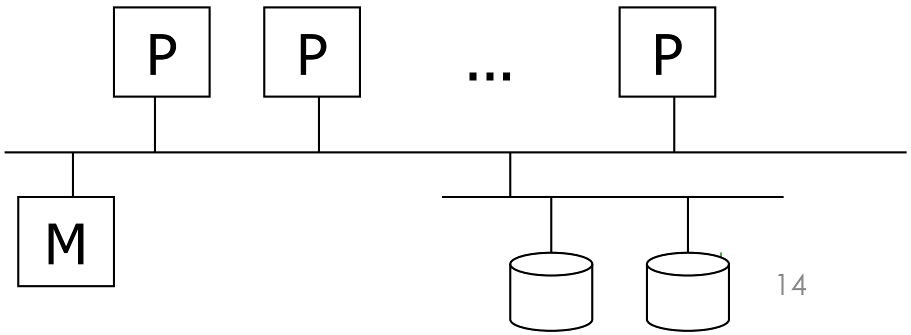
- Parallel data management techniques intend to overcome these barriers

Database Architectures

- Architectures to scale to higher load…
- Multiprocessor architecture
  - Shared memory (SM)
  - Shared disk (SD)
  - Shared nothing (SN)
- Hybrid architectures
  - Non-Uniform Memory Architecture (NUMA)
  - Cluster

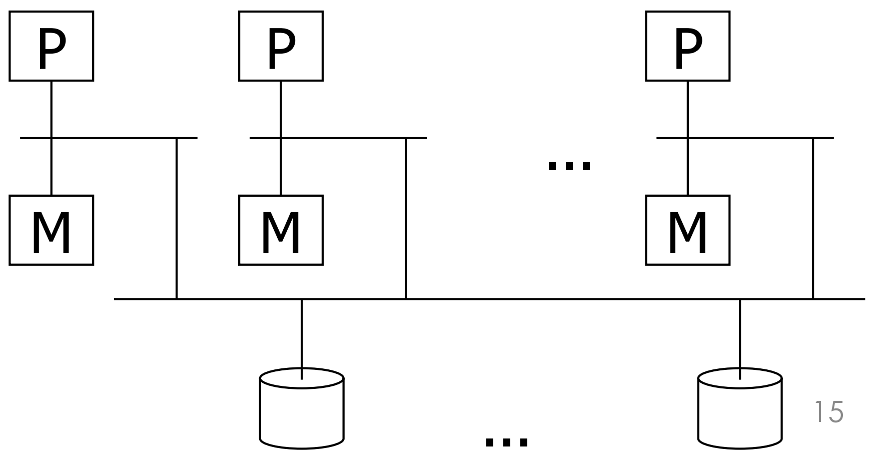
Shared Memory

- Simplest approach - buy a more powerful machine
- Also called ***vertical scaling*** or *scaling up*
- Multiple processors share the main memory (RAM) space but each processor has its own disk (HDD)
  - provide communications among them and avoid redundant copies
- Bottlenecks
  - cost is super-linear: a machine with twice resources (CPU, RAM, disk) typically costs significantly more than twice
  - a machine twice the size cannot necessarily handle twice the load
  - offer limited fault tolerance



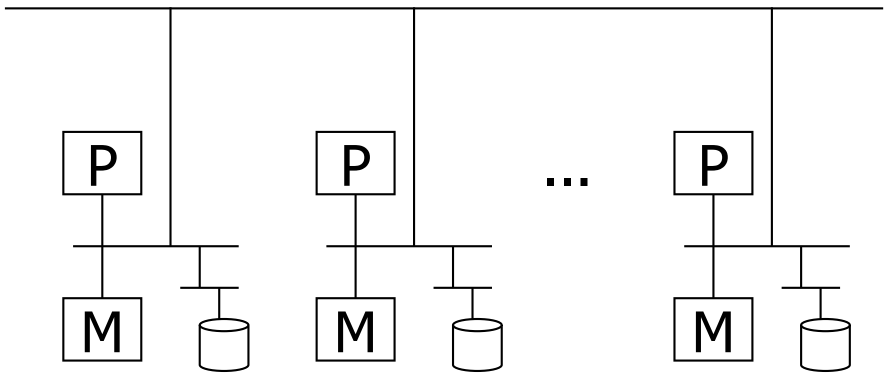
Shared Disk

- Uses several machines with independent CPUs and RAM, but stores data on an **array of disks that is shared between the machines**, connected via a fast network
- Used for some data warehousing workloads
- Advantages over shared memory
  - each processor has its own memory - is not a bottleneck
  - a simple way to provide a degree of fault tolerance.
- Bottleneck
  - limited scalability



Shared Nothing

- Also called ***horizontal scaling*** or *scaling out*
- Each machine or virtual machine running the database software is called a *node that* uses its CPUs, RAM and disks independently
- Any coordination between nodes is done at the software level, using a conventional network
- Most common architecture nowadays
- Advantages:
  - best price/performance ratio
  - extensibility
  - availability
  - reduce latency



- Disadvantages:
  - complexity, difficult load balancing

Hybrid Architectures

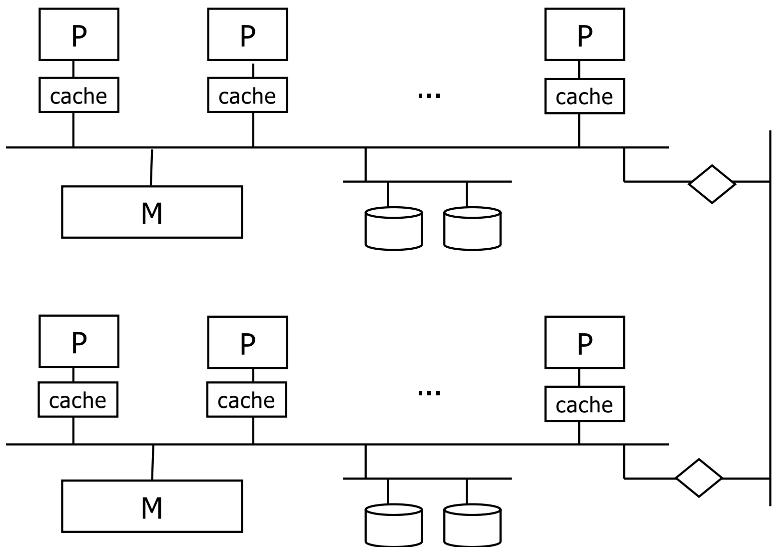
- Various possible **combinations** of the three basic architectures are possible to obtain different **trade-offs** between **cost**, **performance**, **extensibility**, **availability**, etc
- Hybrid architectures try to obtain the **advantages of different architectures**:
  - **efficiency** and **simplicity** of **shared-memory**
  - **extensibility** and **cost** of either **shared disk** or **shared nothing**
- Two main types:
  - NUMA (*non-uniform memory access*)
  - Cluster

NUMA

- Shared Memory vs. Distributed Memory
  - mixes two different aspects:
    - addressing: single address space *and* multiple address spaces
    - physical memory: central *and* distributed
- NUMA uses **single address space on distributed physical memory**
  - eases application portability
  - extensibility
- Cache Coherent NUMA (CC-NUMA)
  - the most successful

CC-NUMA

- Principle
  - main **memory distributed** as with shared-nothing
  - however, any **processor has access to all other processors**' memories
    - remote memory access very efficient, only a few times (typically between 2 and 3 times) the cost of local access
- Different processors can access the same data in a conflicting update mode
  - a global cache consistency protocols are needed



Parallel & Distributed DBMS Techniques

- **Data placement**
  - Physical placement of the DB onto multiple nodes
  - Static vs. Dynamic
- **Parallel data processing algorithms**
  - Select is easy
  - Join (and all other non-select operations) is more difficult
- **Parallel query optimization**
  - Choice of the best parallel execution plans
  - Automatic parallelization of the queries and load balancing
- **Distributed Transaction management**

Distributed Data Storage

- Two common ways of distribute data across nodes:
- ***Replication***
  - keeping a copy of the same data on several different nodes; potentially in different locations
  - provides redundancy; if some nodes are unavailable, the data can still be served from the remaining nodes
  - can also help improve performance
- ***Partitioning***
  - splitting a big database into smaller subsets called *partitions*
  - different partitions can be assigned to different nodes

- Replication and Partitioning can be combined

Data Transparency

- Definition
  - Degree of (system) user abstraction relatively to the details how and where the data items are stored in a distributed system

- Consider transparency issues in relation to:
  - Replication mechanism
  - Partitioning mechanism
  - Location

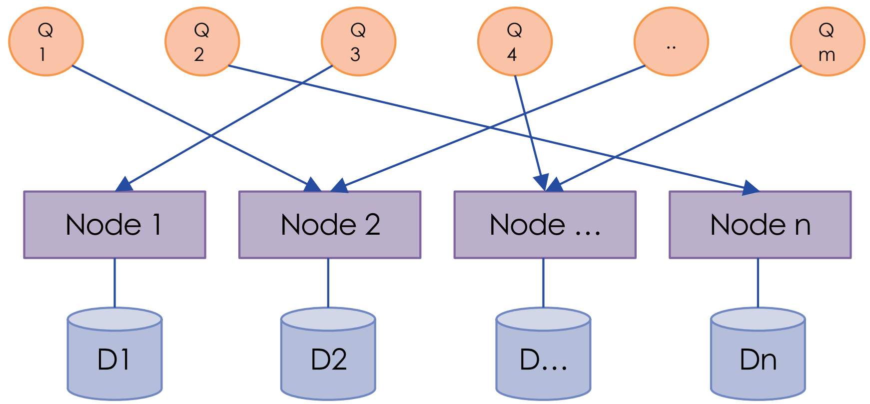
I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning
- Horizontal partitioning – tuples of a relation are divided among many disks
- Partitioning techniques* (number of disks = n)
  - **Round-robin**
    - send the ith tuple inserted in the relation to the disk: i mod n.
  - **Hash partitioning**
    - apply a hash function to one or more attributes that range 0…n - 1
  - **Range partitioning**
    - associates a range of key attribute(s) to every partition

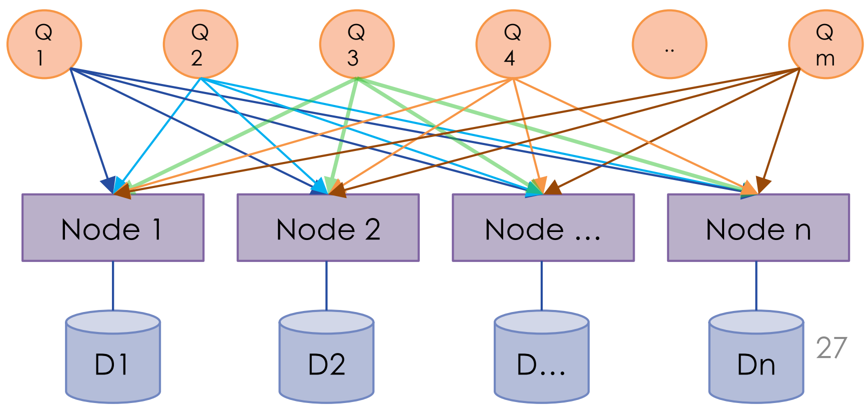
** simplest vision - a more detailed description in the next lessons*

Query Parallelism

- **Interquery** Parallelism
  - Parallel execution of multiple queries generated by concurrent transactions



- **Intraquery** Parallelism
  - Execution of a single query in parallel on multiple processors/disks



Interquery Parallelism

- To **increase** the **transactional throughput**
  - used primarily to scale up a transaction processing system **to support a larger number of transactions per second**
- **Easiest form** of parallelism to support in a **shared-memory** parallel database
- More **complicated** on **shared-disk** or **shared-nothing**
  - locking and logging must be coordinated by passing messages between processors
  - data in a local buffer may have been updated at another processor
  - cache-coherency has to be maintained: reads and writes of data in buffer must find latest version of data

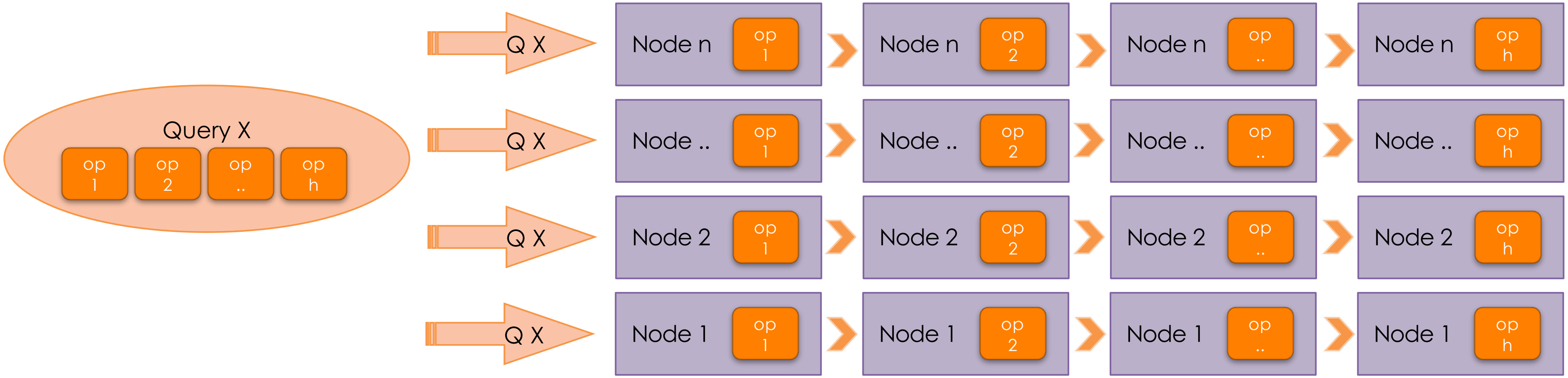
Intraquery Parallelism

- The **same operator** is **executed** by **many processors**, **each** one **working** on a **subset** of the **data**
  - for speeding up long-running queries
- Two complementary forms of intraquery parallelism:
  - **Intra-operation**: parallelize the execution of each individual operation in the query
  - **Inter-operation**: execute the different operations in a query expression in parallel

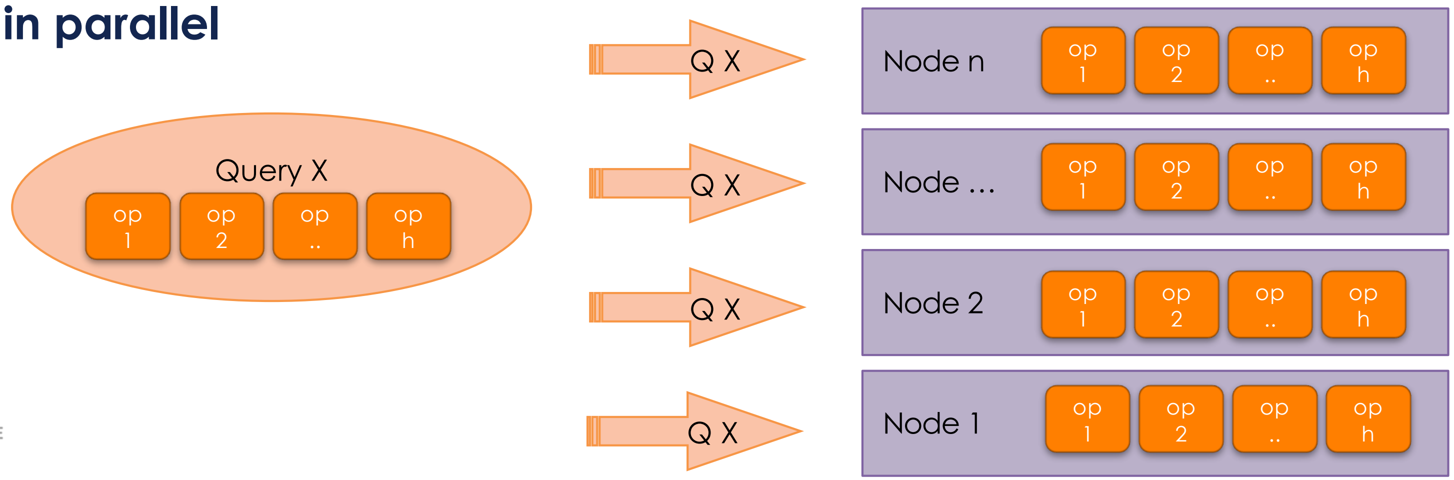
- Intra-operation scales better with increasing parallelism because the number of tuples processed by each operation is typically more than the number of operations in a query

IntraQuery Operator Parallelism

- **Inter-operator** (Pipeline): ordered (or partially ordered) tasks and **different machines** are **performing different tasks**



- **Intra-operator** (Partitioned): a **task divided** over all machines to **run in parallel**



Parallel Data Processing

Assuming:

- Read-only queries
- Shared-nothing architecture
  - shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems
- n processors (P0, ..., Pn-1) and n disks (D0, ..., Dn-1) where disk Di is associated with processor Pi
  - if a processor has multiple disks they can simply simulate a single disk Di

Parallel Algorithms

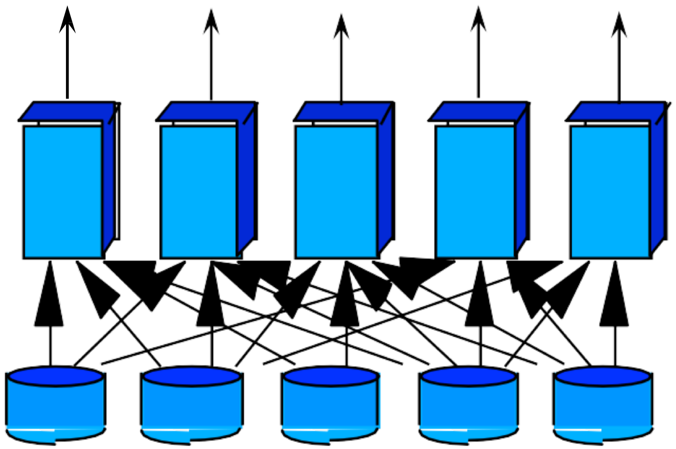
- Use case:
  - **Relational Model**
- **Parallel algorithms** for **relational algebra operators** are the **building blocks** necessary **for parallel query processing**
- Parallel data processing should **exploit intra-operator parallelism**
  - the query is parallelized
  - all nodes perform all operations requested by query
- Focus on **sort**, **select** and **join** operators
  - other binary operators (such as union) can be handled in similar way to join

Parallel Selection - $\sigma_c(R)$

- Relation **R** is **partitioned** over **m machines**
  - each partition of R is around |R|/m tuples
- **Each machine scans** its own **partition** and applies the **Selection** condition c
- Data Partitioning – impact:
  - **round robin** or a **hash function** (over the entire tuple)
    - relation is expected to be well distributed over all nodes
    - **all partitioned will be scanned**
  - **range** or hash-based (on the selection column)
    - relation can be clustered on few nodes
    - **few partitions need to be touched**
- Parallel **Projection** is also straightforward
  - all partitions will be touched
  - not sensitive to how data is partitioned

Parallel Sorting

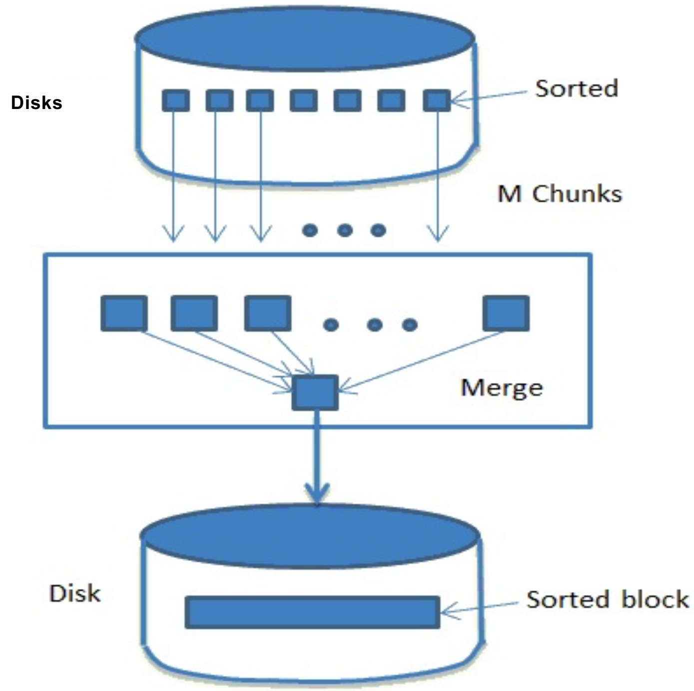
1. **Range-Partitioning Sort**
   - **Choose processors** P0, ..., Pm, where m ≤ n -1 **to do sorting**
   - **Re-partition R based on ranges (on the sorting attributes) into m partitions**
     - this step **requires I/O** and **communication overhead**
   - **Machine i** receives all ith partitions from all machines and **sort that partition**, without any interaction with the others
     - Pi stores the tuples it received temporarily on disk Di
   - **Final merge** operation is trivial: range-partitioning ensures that, for 1 j m, the key values in processor Pi are all less than the key values in Pj
   - Skewed data is an issue
     - ranges can be of different width
     - apply sampling phase first



Parallel Sorting (Cont.)

2. Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks D0, ..., Dn-1 (in whatever manner).
- **Each node sorts its own data**
- **All nodes** start **sending** their **sorted data** (one block at a time) to a **single machine**
- This **machine** applies **merge-sort** technique as data come



Parallel Join

- The join operation **requires pairs** of **tuples** to be **tested** to see if they satisfy the join condition
  - If tuples satisfy the join condition, the pair is added to the join output
- Steps…

1. **Parallel join algorithms** attempt to **split the pairs-testing** over **several processors**
2. **Each processor** then **computes part** of the **join** locally
3. **Results** from each processor are **collected** together to **produce** the **final result**

Join Algorithms

- Three basic parallel join algorithms for partitioned databases
  - **Parallel Nested Loop** (PNL)
  - **Parallel Associative Join** (PAJ)
  - **Parallel Hash Join** (PHJ)
- All previous **algorithms** are **intra-operator parallelism**

- They also apply to other complex operators such as duplicate elimination, union, intersection, etc. with minor adaptation

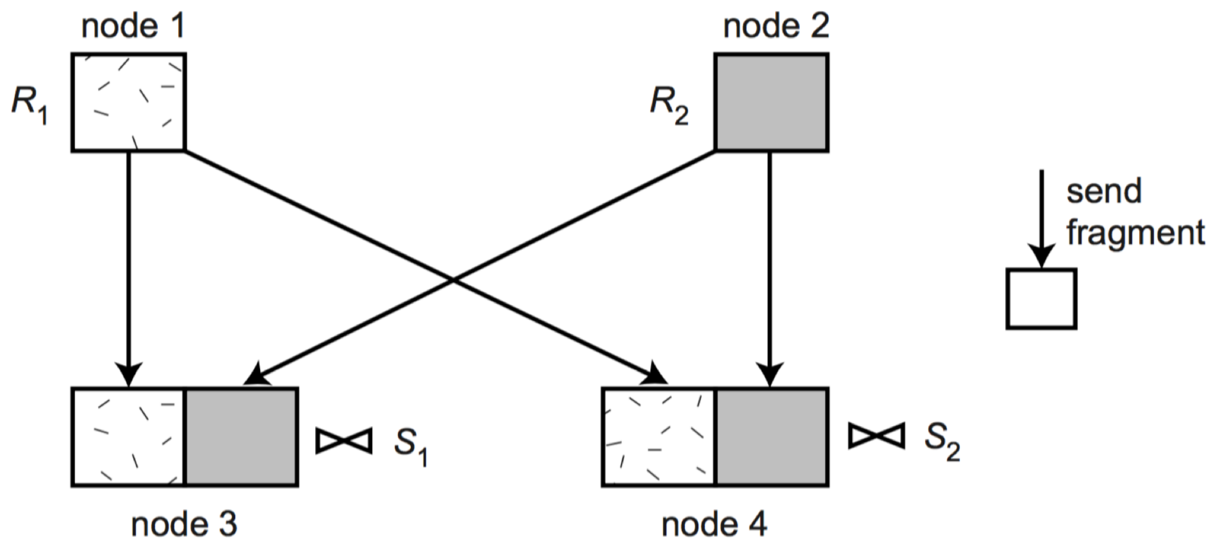
- Next Examples:
  - join of two **relations *R*** and ***S*** that are partitioned over ***m*** and ***n* nodes**, respectively

UNIVERSIDADE DE AVEIRO

Parallel Nested Loop Join

- **Cartesian product** of relations ***R*** and ***S***, in parallel.
- **Simplest** and most **general** method
- Algorithm phases:
  1. **each fragment** (replica) of ***R*** is **sent** to **each node** containing a **fragment** of ***S*** (there are *n* such nodes)
     - this phase is done in parallel by *m* nodes
  2. **each *S*-node** *j* **receives** relation ***R*** **entirely**, and **locally joins** *R* with **fragment** *Sj*.
     - join processing may start as soon as data are received

$$R \bowtie S = \bigcup_{i=1}^{n} (R \bowtie S_i)$$

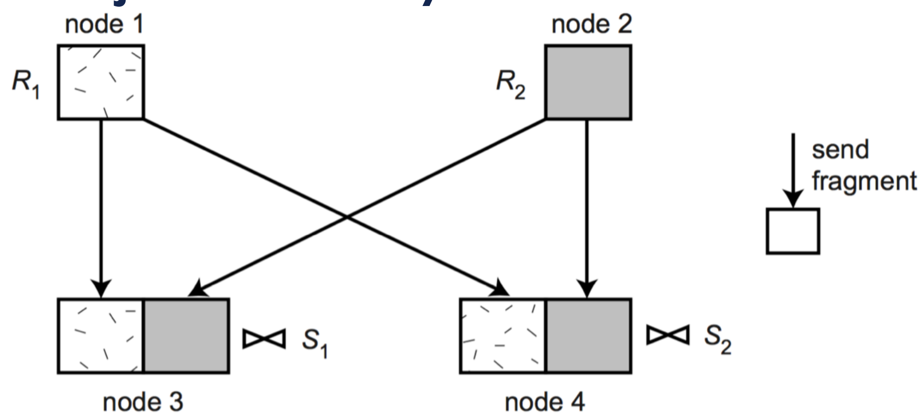


UNIVERSIDADE DE AVEIRO

example m=n=2

Parallel Associative Join

- Applies **only** to **equijoin** with **one** of the **operand relations partitioned** according to the **join attribute**
- Assume
  - **equijoin** predicate is on **attribute *A*** from ***R***, and ***B*** from ***S***
  - ***S*** *is* **partitioned** according to **hash function applied** to attribute ***B***
    - tuples of *S* that have the same *h*(*B*) value are placed at the same node
  - **no knowledge** of how ***R*** is **partitioned**
- Algorithm phases:
  1. relation ***R*** is **sent** associatively to the ***S*-nodes based** on the **hash function** *h* applied to **attribute *A***
  2. **each *S*-node** *j* **receives** in parallel from the different *R*-nodes the relevant subset of *R* (i.e., *Rj*) and **joins** it **locally** with the fragments *Sj*



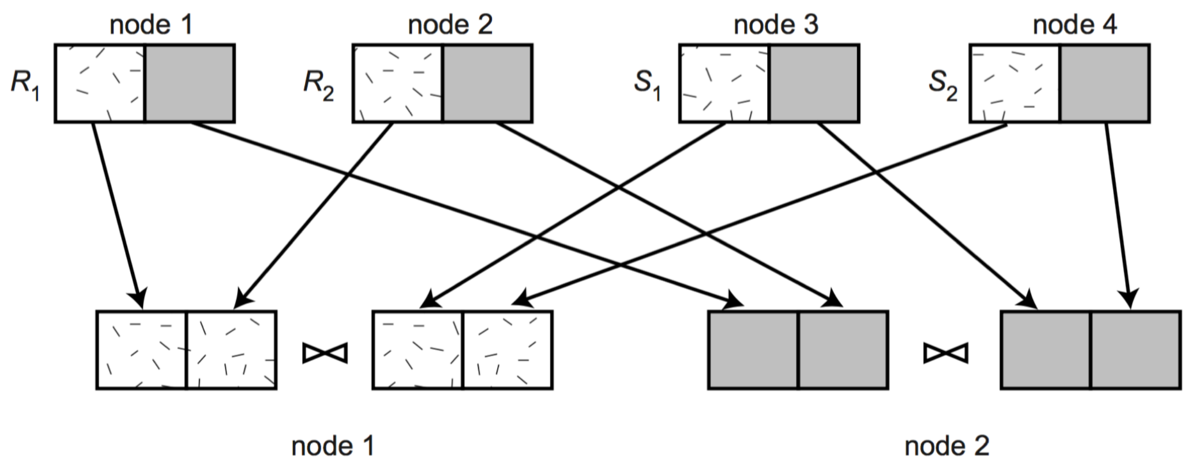
$$R \bowtie S = \bigcup_{i=1}^{n} (R_i \bowtie S_i)$$

UNIVERSIDADE DE AVEIRO

Parallel Hash Join

- Generalization of parallel associative join algorithm
- Also applies to equijoin but does not require any particular partitioning of the operand relations
- **Basic idea**: **partition** of ***R*** and ***S*** into the **same number *p*** of mutually exclusive sets (**fragments**) *R1,R2,…,Rp*, and *S1,S2,…,Sp*,
- The ***p* nodes** may actually be **selected** at **run time** based on the load of the system
- Algorithm phases:
  1. *build*: **hashes *R*** on the **join attribute**, **sends** it to the **target *p* nodes** that build a hash table for the incoming tuples
  2. probe: **sends *S* associatively** to the **target *p* nodes** that probe the hash table for each incoming tuple

$$R \bowtie S = \bigcup_{i=1}^{p} (R_i \bowtie S_i)$$



UNIVERSIDADE DE AVEIRO

Parallel Processing - Costs

- Join processing is achieved with a degree of parallelism of either n or p
- **Each algorithm requires moving at least one of the operand relations**
- **Ideal scenario**: no skew in the partitioning, and no overhead due to the parallel evaluation
  - **expected speed-up: 1/n**
- Take into account **skew** and **overheads**
  - time taken by a parallel operation can be estimated as:

$$\mathbf{Tcost = Tpart + Tasm + max\ (T0,\ T1,\ \ldots,\ Tn\text{-}1)}$$

Tpart - time for partitioning the relations (including communications costs)

Tasm - time for assembling the results

Ti - time taken for the operation at processor Pi. This needs to be estimated taking into account the skew, and the time wasted in contentions

Parallel Query Optimization

- The **objective** is to **select the "best" parallel execution plan** for a query using the following components:
  - Search space
    - Models alternative execution plans as operator trees
    - Left-deep vs. Right-deep vs. Bushy trees
  - Search strategy
    - Dynamic programming for small search space
    - Randomized for large search space
  - Cost model (abstraction of execution system)
    - Physical schema info. (partitioning, indexes, etc.)
    - Statistics and cost functions

- Target: **minimize** the **movement of data** among machines

Best Execution Plan - Example

- Two Machines
  - M1 has the relation R(A,B)
  - M2 has the relation S(C,D)
- Query

  SELECT A, C FROM R join S on B = D;
- Result
  - must be at M2

➢ Options:

1. Copy S to M1
2. Compute the result
3. Send the result to M2

OR

1. Copy R to M2
2. Compute the result

➢ Scenarios:

size R ≃ S

?

size R > S

Best Execution Plan - Example (cont.)

- Even better execution plan:

```
1. On M2 compute
            INSERT INTO TEMP1 SELECT DISTINCT D FROM S;

2. Copy TEMP1 to M1

3. On M1 compute
            INSERT INTO TEMP2
            SELECT A, B FROM R join TEMP1 on B = D;

4. Copy TEMP2 to M2

5. On M2 compute
            INSERT INTO ANSWER
            SELECT A, C FROM TEMP2 join S on B = D;
```

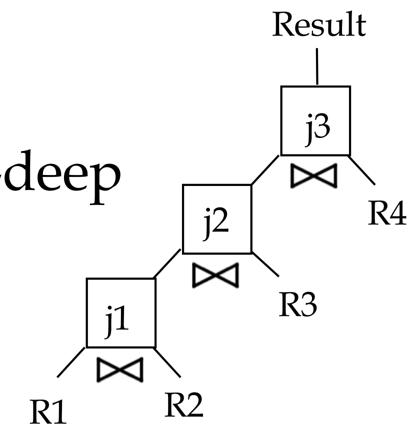
$R \ltimes S$

  – TEMP2 is ***left semijoin*** of R and S

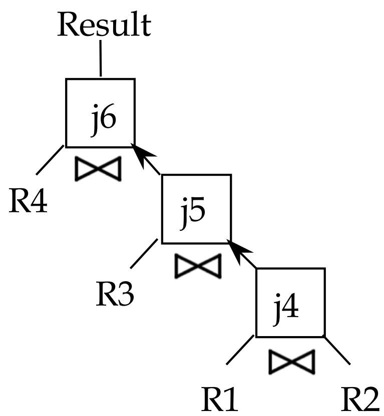
- Very Good if TEMP1 and TEMP2 are relatively small

Search space - Operator Trees

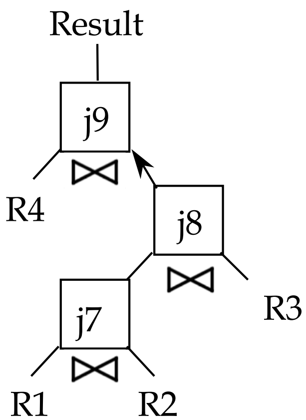
Left-deep



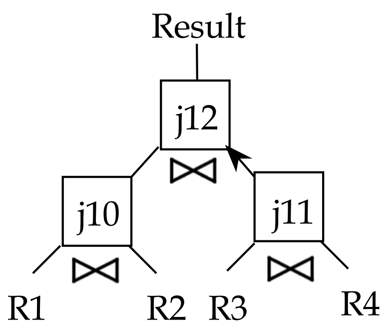
Right-deep



Zig-zag



Bushy

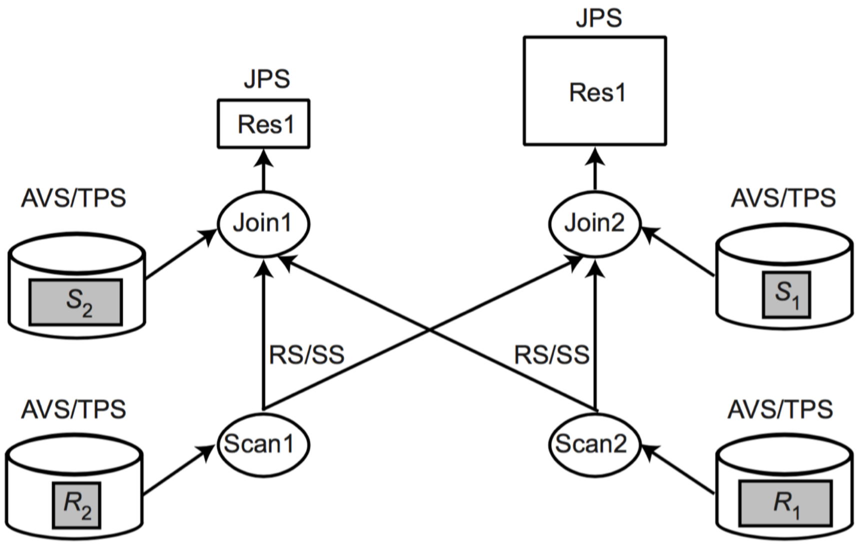


Load Balancing

- **Balancing** the **load** of **different transactions** and **queries among different nodes** is essential to **maximize throughput**
- **Problems** arise for **intra-operator parallelism** with ***skewed* data distributions**
  - attribute data skew (AVS)
    - inherent to dataset (e.g., there are more citizens in Paris than in Aveiro).
  - tuple placement skew (TPS)
    - introduced when the data are initially partitioned (e.g., with range partitioning)
  - selectivity skew (SS)
    - introduced when there is variation in the selectivity of select predicates on each node
  - redistribution skew (RS)
    - occurs in the redistribution step between two operators (similar to TPS)
  - join product skew (JPS)
    - occurs because the join selectivity may vary between nodes
- **Solutions**
  - sophisticated parallel algorithms that deal with skew
  - dynamic processor allocation (at execution time)

Data Skew - Example

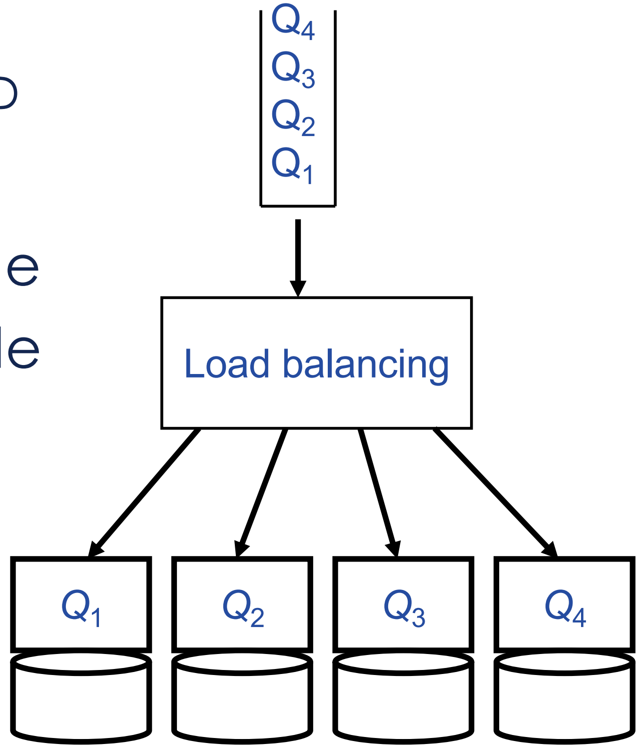
- A **query** over two relations ***R*** and ***S*** that are **poorly partitioned**
  - due to either data (AVS) or the partitioning function (TPS)
  - processing times of instances (scan1 and scan2) are not equal
- **Join** operator case is **worse**
  - the number of tuples received is different due to the poor redistribution of the partitions (RS) or variable selectivity according to the partition of *R* processed (SS)
  - uneven size of *S* partitions (AVS/TPS) yields different processing times for tuples sent by scan operator. The result size is different from one partition to the other due to join selectivity (JPS)



boxes are proportional to the size of partitions

Load Balancing in a DB Cluster

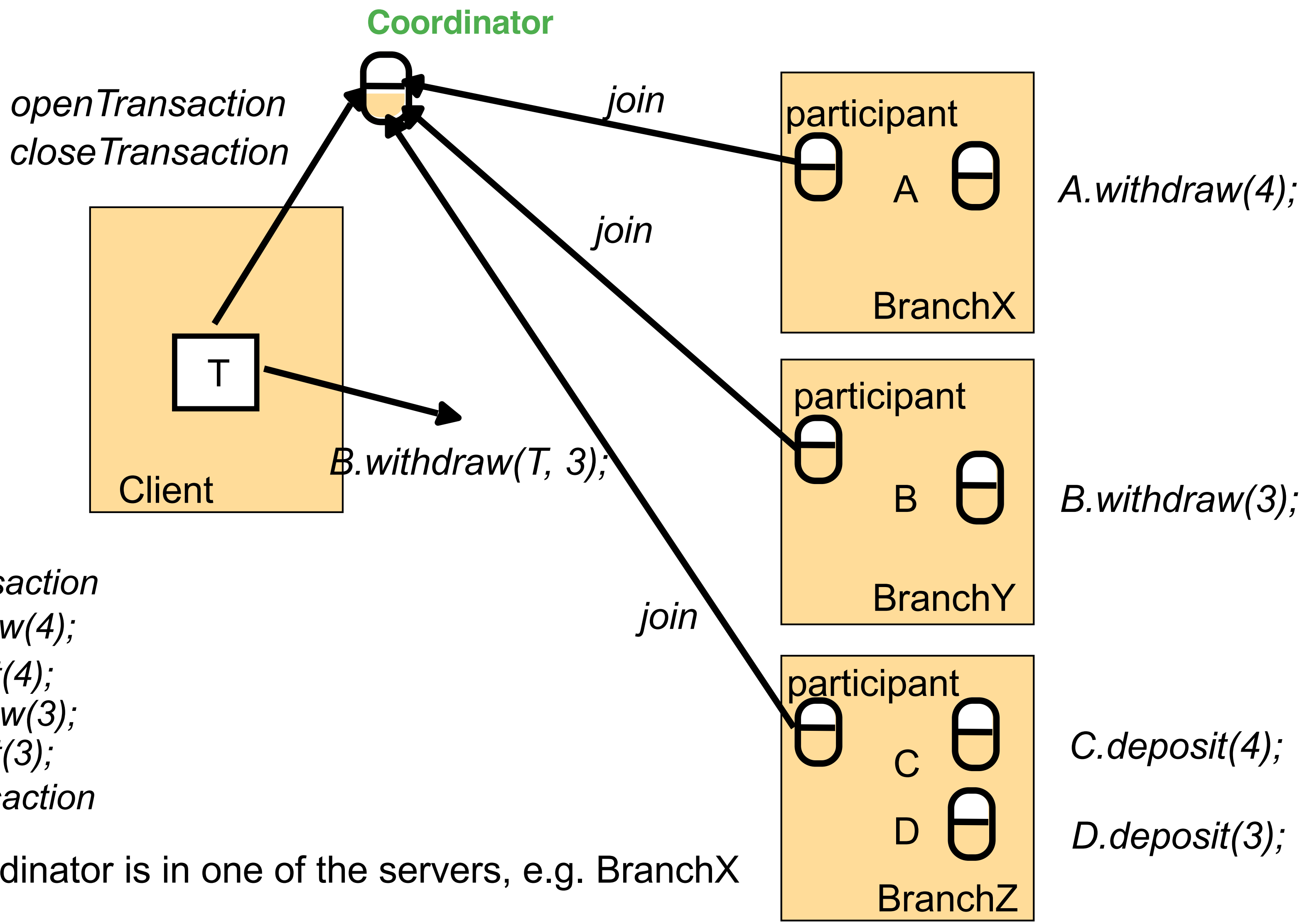
- Choose the node to execute Q
  - round robin
  - the least loaded
    - Need to get load information
- Failover
  - In case a node N fails, N's queries are taken over by another node
    - requires a copy of N's data or SD
- In case of interference
  - data of an overloaded node are replicated to another node



Distributed Transactions

- Transaction may access data at several sites
- Each site has a **local transaction manager** responsible for:
  - maintaining a log for recovery purposes
  - participating in coordinating the concurrent execution of the transactions executing at that site
- Each site has a **transaction coordinator**, which is responsible for:
  - **starting** the execution of transactions that originate at the site
  - **distributing** subtransactions at appropriate sites for execution
  - coordinating the **termination** of each transaction that originates at the site, which may result in the transaction being committed at all sites or aborted at all sites

Distributed Banking Transaction

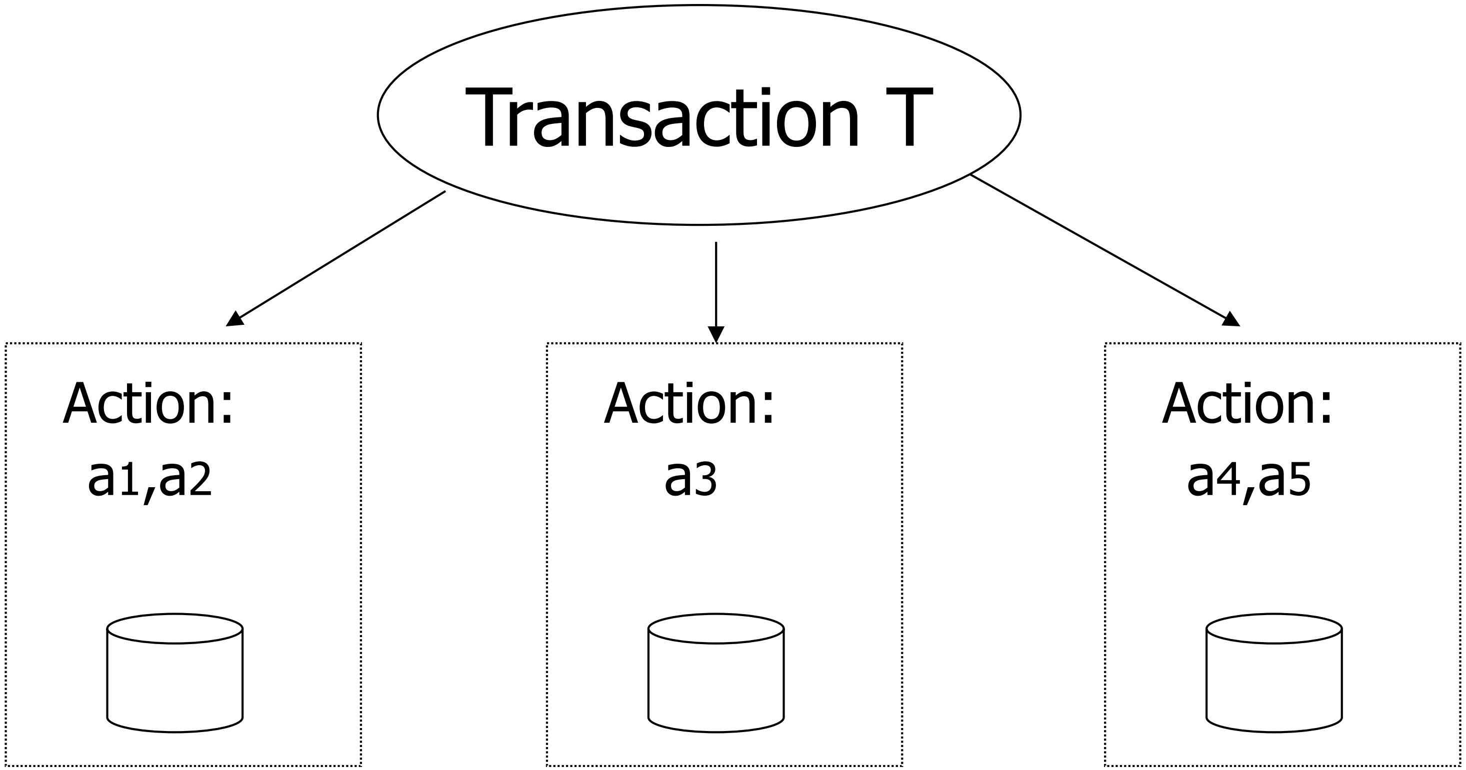


T = openTransaction
A.withdraw(4);
C.deposit(4);
B.withdraw(3);
D.deposit(3);
closeTransaction

Note: the coordinator is in one of the servers, e.g. BranchX

Distributed commit problem

- **Commit** must be **atomic**…
- How a distributed transaction that has components at several sites can execute atomically?
- **Solution**: Two-phase commit (2PC), Centralized 2PC, Distributed 2PC, Linear 2PC, etc



Two-phase commit protocol

- **First phase - coordinator collecting a vote** (commit or abort) **from each participant**
  - Participant stores partial results in permanent storage before voting
- **Second phase - coordinator** makes a **decision**
  - **if** all participants want to commit and no one has crashed, coordinator multicasts "commit" message
    - everyone commits
    - if participant fails, then on recovery, can get commit msg from coordinator
  - **else** if any participant has crashed or aborted, coordinator multicasts "abort" message to all participants
    - everyone aborts

Resources

- Martin Kleppmann, ***Designing Data-Intensive Applications***, O'Reilly Media, Inc., 2017.

- M. Tamer Ozsu, Patrick Valduriez, **Principles Of Distributed Database Systems** – 3rd ed, Springer, 2011.

- Abraham Silberschatz, Henry F. Korth, S. Sudarshan, **Database System Concepts –** 6th ed, McGraw-Hill, 2010.

Summary

- Centralized vs Distributed vs Parallelized Systems
- Parallel Databases
  - Concept / Objectives
  - Architectures
  - Types of Parallelism
  - DBMS Techniques
    - Data Placement
    - Processing Algorithms
    - Query Optimization
    - Transaction Management