Parallel and Distributed Database Systems

Outline

- Architecture
  - Parallel and Distributed Database Systems
- Storage
  - Partitioning vs Replication
  - Indexing
- Query processing
- Transactions and concurrency control

Introduction

- Data storage needs are increasingly growing…
  - multimedia objects like images/videos
  - user data at web-scale
    - 100's of millions of users, petabytes of data
  - any movement is collected and stored for analysis
- Parallel and distributed storage systems help to…
  - store large volumes of data
  - process time-consuming queries
  - provide high throughput for transaction processing
  - enhance **scalability** and **availability**
- Parallel and distributed DBMS build on top of that

Parallel Database Systems

- DBMS with multiple processors and multiple disks connected by a fast interconnection network
  - Single system from the user viewpoint
  - As of 1980s, initially for small scale (10s-100s machines)
- **Motivation**: handle workloads beyond what a single-cpu system can do
  - High performance transaction processing
    - E.g., handling user requests at web-scale
  - Decision support on very large amounts of data
    - E.g., data gathered by large web sites/apps
- Performance measures:
  - **Throughput**: no. tasks that can be done in a specific time
  - **Response time**: time to complete a single task

Degree of Parallelism

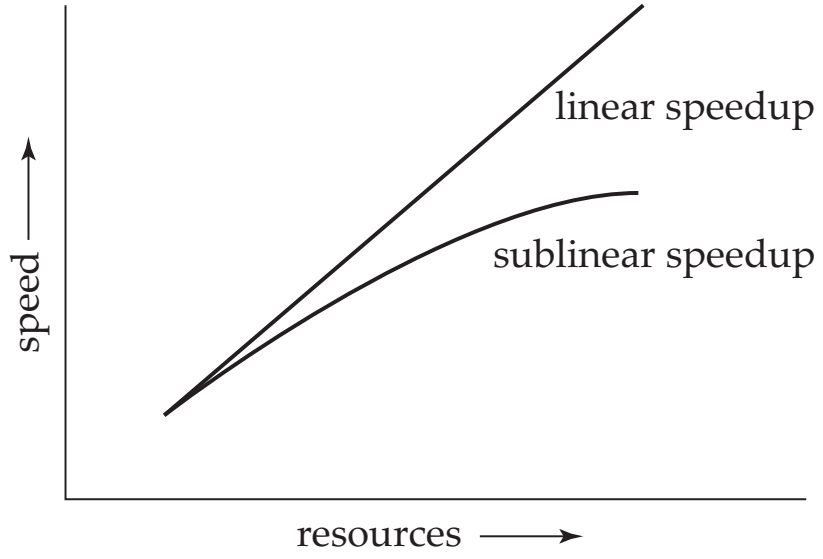
- **Speed-up**: Running a given task in less time by increasing the degree of parallelism

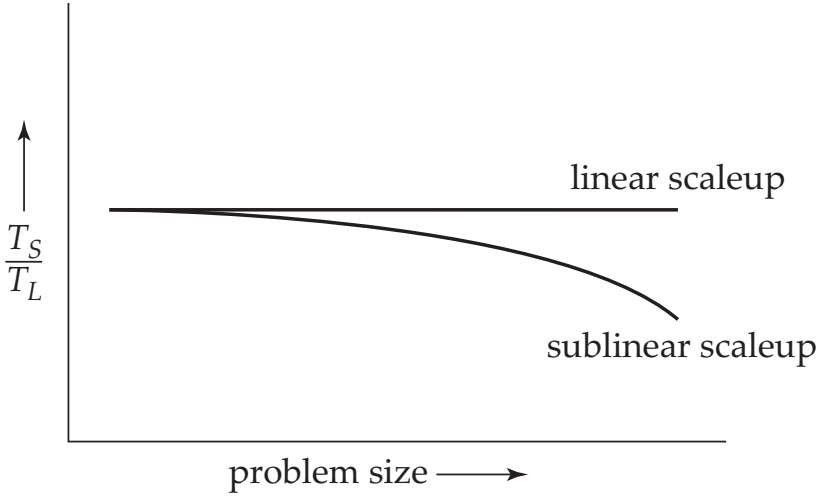
```
speedup = time_in_small_system /
          time_in_large_system
```

- **Scale-up**: Handling larger tasks in the same time by increasing the degree of parallelism
  - N-times larger system to solve an *N*-times larger job

```
scaleup = time_of_small_task_in_small_system /
          time_of_large_task_in_large_system
```

  - Can an *N*-times larger system provide results in the same time if…
    - processing a *N*-times larger problem: **Batch** scale-up
    - *N*-times as many users submit requests: **Transaction** scale-up





Limitations of Parallelism

Speed-up and scale-up are often sublinear due to:

- **Sequential** computation: non-parallelizable parts
  - *Amdahl's law*: if $p \equiv$ fraction of computation that is subject of a speedup and s $\equiv$ speedup of that part, **global speed-up** limited to:

$$1 / [(1-p)+(p/s)]$$

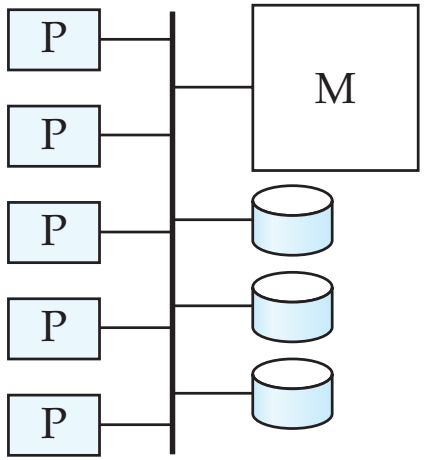
- **Interference**: Concurrent processes waste time competing by shared resources.

- **Skewness**: the amount of work not properly balanced between tasks

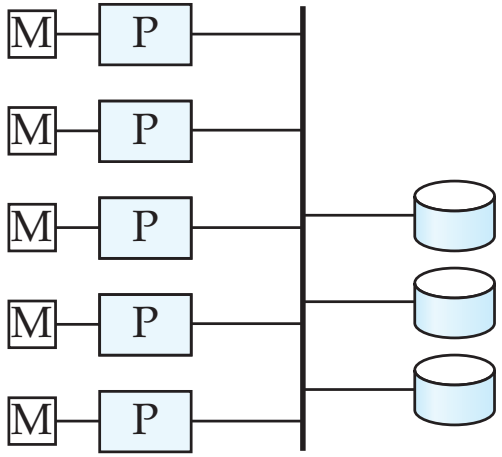
ARCHITECTURE

Parallel Database Architectures

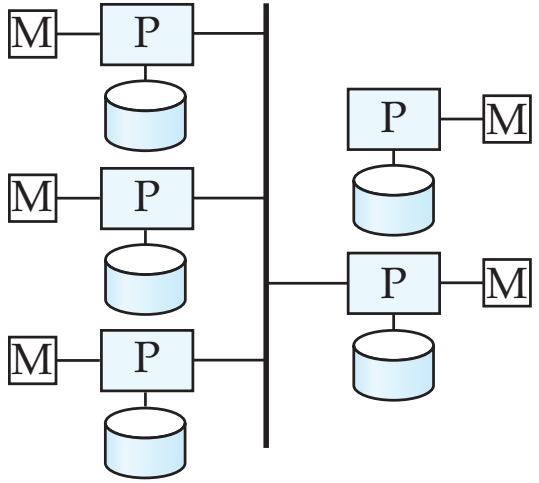
- Types of architectures:



Shared memory

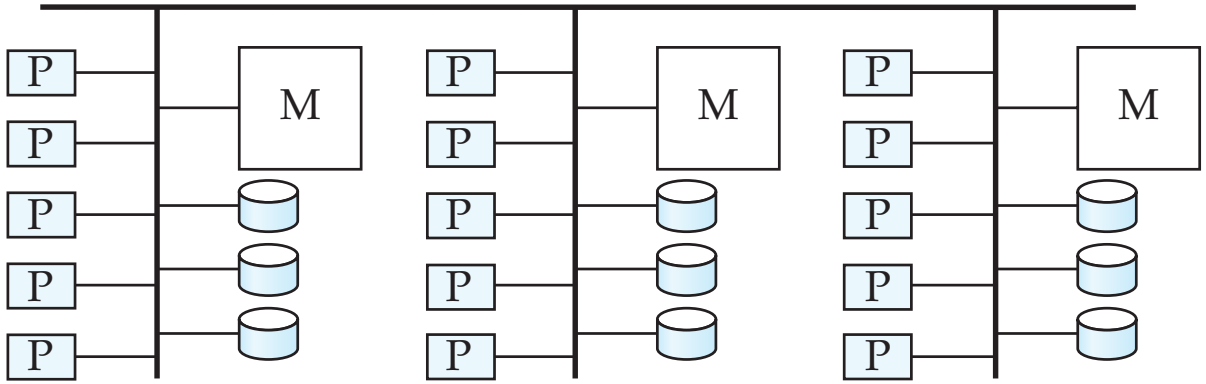


Shared disk



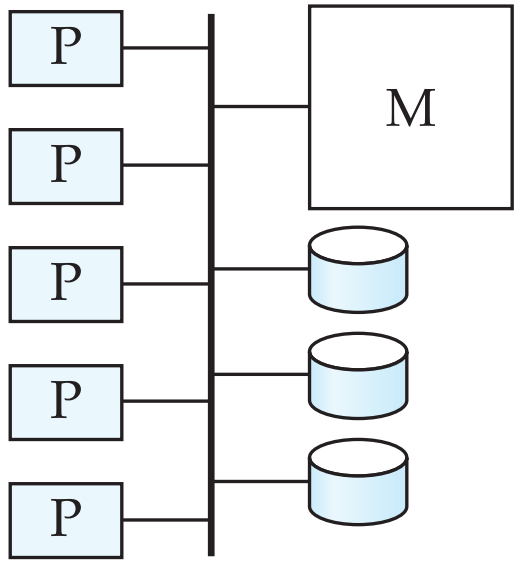
Shared nothing

Hierarchical



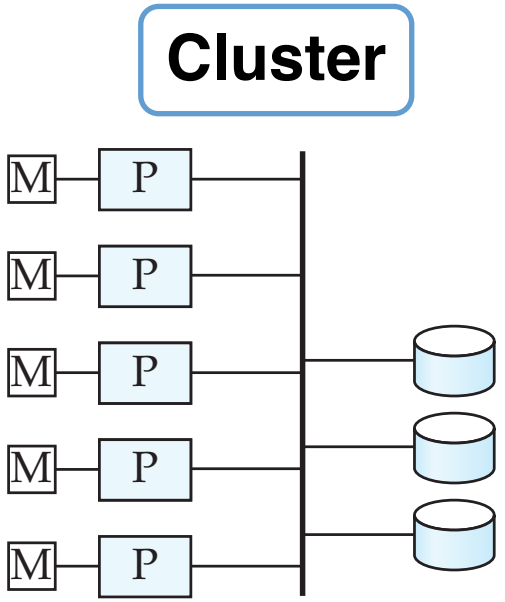
PDB Architectures: Shared Memory

- Processors (and/or cores) have access to a common memory (and disks)
  - Via a single bus in earlier days
    Bottleneck
  - Today, through an interconnection network
- Extremely efficient communication between processors
  - Via writes in memory
- **Not scalable** beyond few hundred processor cores



PDB Architectures: Shared Disk

- Processors with their own memory directly access all disks
  - **Fault-tolerance**: if a processor fails, others can take over its tasks
    - DB is accessible from all processors
  - Scales to large no. processors



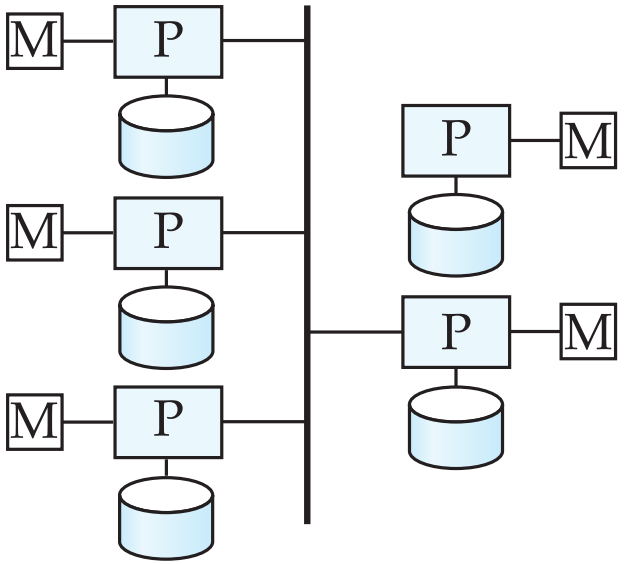
- Disk subsystem can implement RAID architecture
  - Fault tolerant

- **Bandwidth** to access disk subsystem **limits scalability**
  - **Storage Area Networks (SAN)**: High-speed local-area network to connect large banks of disks to nodes

PDB Architectures: Shared Nothing

- Each node has a processor, memory and 1+ disks
  - Communication through a high-speed interconnection network
  - Scales up to thousands of processors

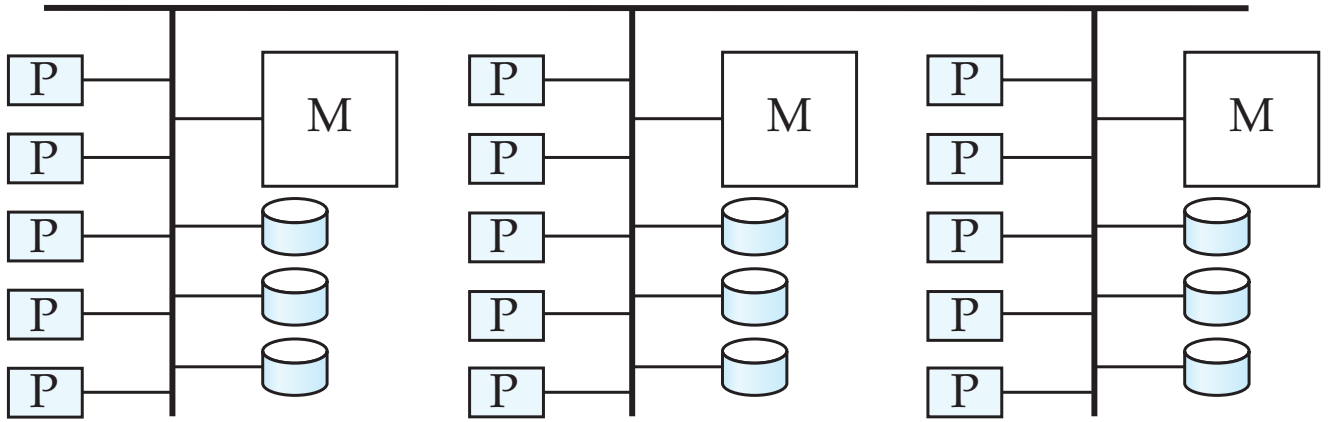
- **Main issue**: large **cost of communication** and **non-local disk access**
  - Sending data involves software interaction at both ends



PDB Architectures: Hierarchical

- Combines characteristics of all other architectures.
  - Top level is a shared-nothing architecture
  - In the middle level, there could be a shared-disk system
  - At bottom level, each node would be a shared-memory system

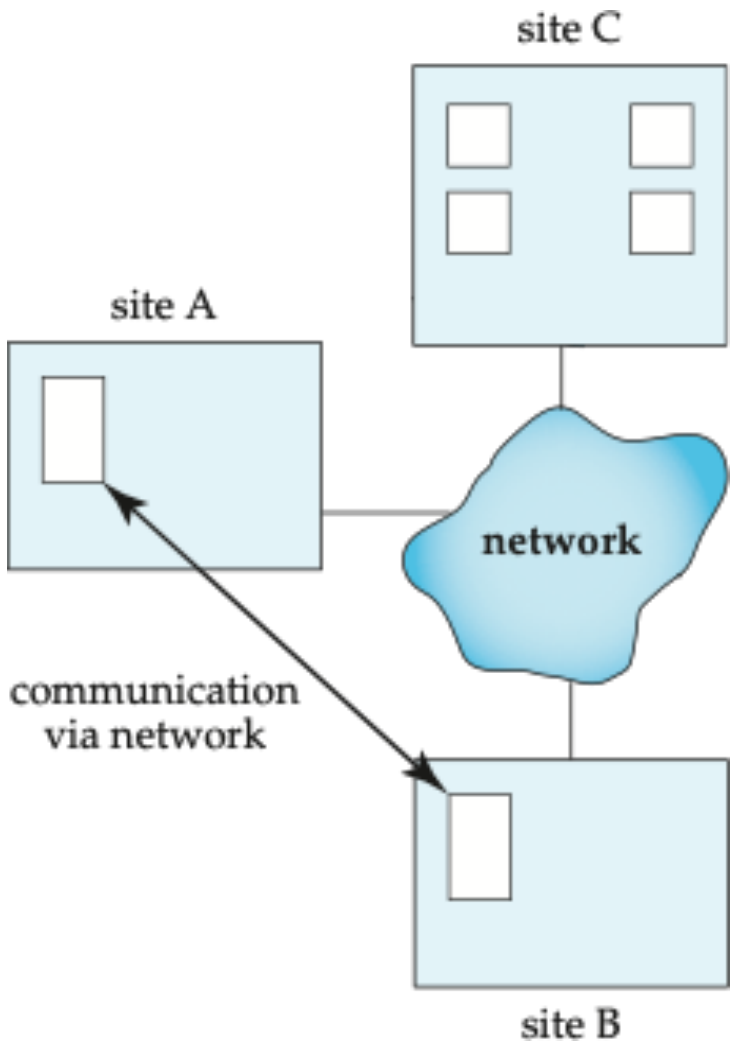
- Typically, **used today** by parallel database systems
  - Each node uses shared-memory parallelism
  - Nodes inter-connected as a shared-nothing system



Distributed Database Systems

- Data stored on machines located at different **sites**
  - Geographically distant
- Communication by private networks or the Internet
  - Latency

- Compared to *shared-nothing PDBs*…
  - Lower bandwidth; higher latency of communications
    - Higher risk of failure
  - Higher availability
    - Tolerant to whole-site failures
      E.g., natural disasters
  - Node size and function, more variability in DDBs



PDB issues: Cache coherence

- **Cache**: Fast but small memory attached to processors

  Time access: few nanoseconds

  - Cache levels within a single multi-core processor:
    - In case of multiple processors, each has its own cache levels

Core 0	Core 1	Core 2	Core 3
L1 Cache	L1 Cache	L1 Cache	L1 Cache
L2 Cache	L2 Cache	L2 Cache	L2 Cache
Shared L3 Cache			

- **Cache coherence**, issue in multi-processor DB systems
  - Local cache values may be outdated w.r.t. changes performed by other cores/cpus
    - Need to be invalidated!
  - Consistency models: strong vs. weak
    - With weak consistency, special care needed to ensure access to up-to-date data

STORAGE

Partitioning

- Partition the relations on (*multiple disks* on) *multiple nodes* to reduce time to retrieve relations
  - **Scalability**

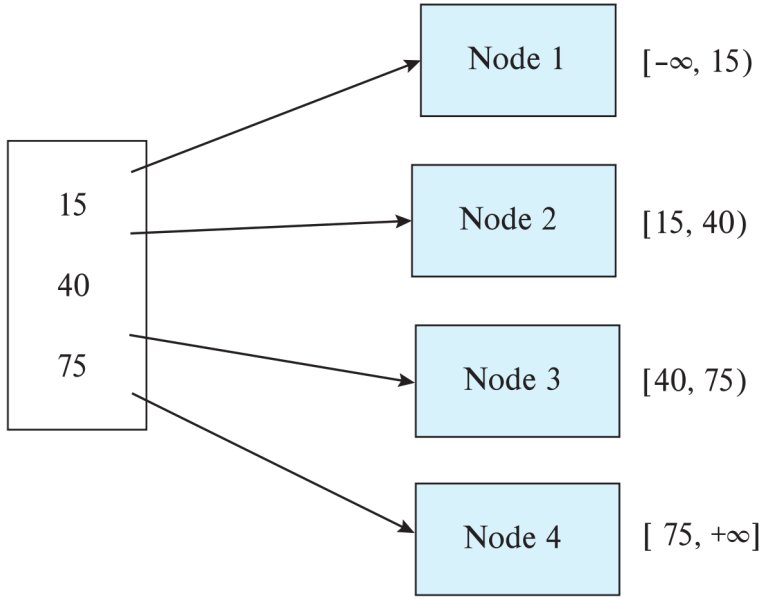
Focus on parallelism across nodes

- **Horizontal partitioning**: tuples of a relation are divided across many nodes.
  - Subsets of tuples reside on different nodes
  - Can be combined with *vertical partitioning*:
    E.g., *r*(*A*, *B*, *C*, *D*) into r1(*A*, *B*) and r2(*A*, *C*, *D*)

Partitioning

- Partitioning techniques:
  - **Round-robin**:
    - For any ordering, i^{th} tuple of the relation is sent to node [i mod n]
  - **Hash**:
    - Given 1+ attributes as partitioning attribute(s), A, and a hash function h:
    - i^{th} tuple of the relation is sent to node $h(tuple_i(A))$
  - **Range**:
    - Given 1+ partitioning attribute(s), A, and a partitioning vector:
    - i^{th} tuple of the relation is sent to node N_j if its value is in j^{th} range, $tuple_i(A) \in range_j$
  - Different effects on CRUD operations, and even in different types of reads (point or range)

$n \equiv$ number of nodes



Partitioning

Main issue: **Skewness**

- **Data-distribution skew**: some nodes end up with a higher no. tuples, due to:
  - Attribute-value skew: some value of the partitioning attribute is more common
    - Can occur in hash and range partitioning.
  - Partition skew: imbalance (even without attribute-value skew)
    - Mainly in range-partitioning, with an inappropriate vector

- **Execution skew**: all the processing is done in the same node, which prevents from taking advantage of parallel computing
  - Even without data distribution skew

Replication

- Replicate data in different nodes to ensure that it is always **available**
  Often, 3 replicas
  - Probability of a system failure increases linearly with no. nodes
    E.g., if a node fails once every 5 years, in a system with 100 nodes there is a failure every 18 days.

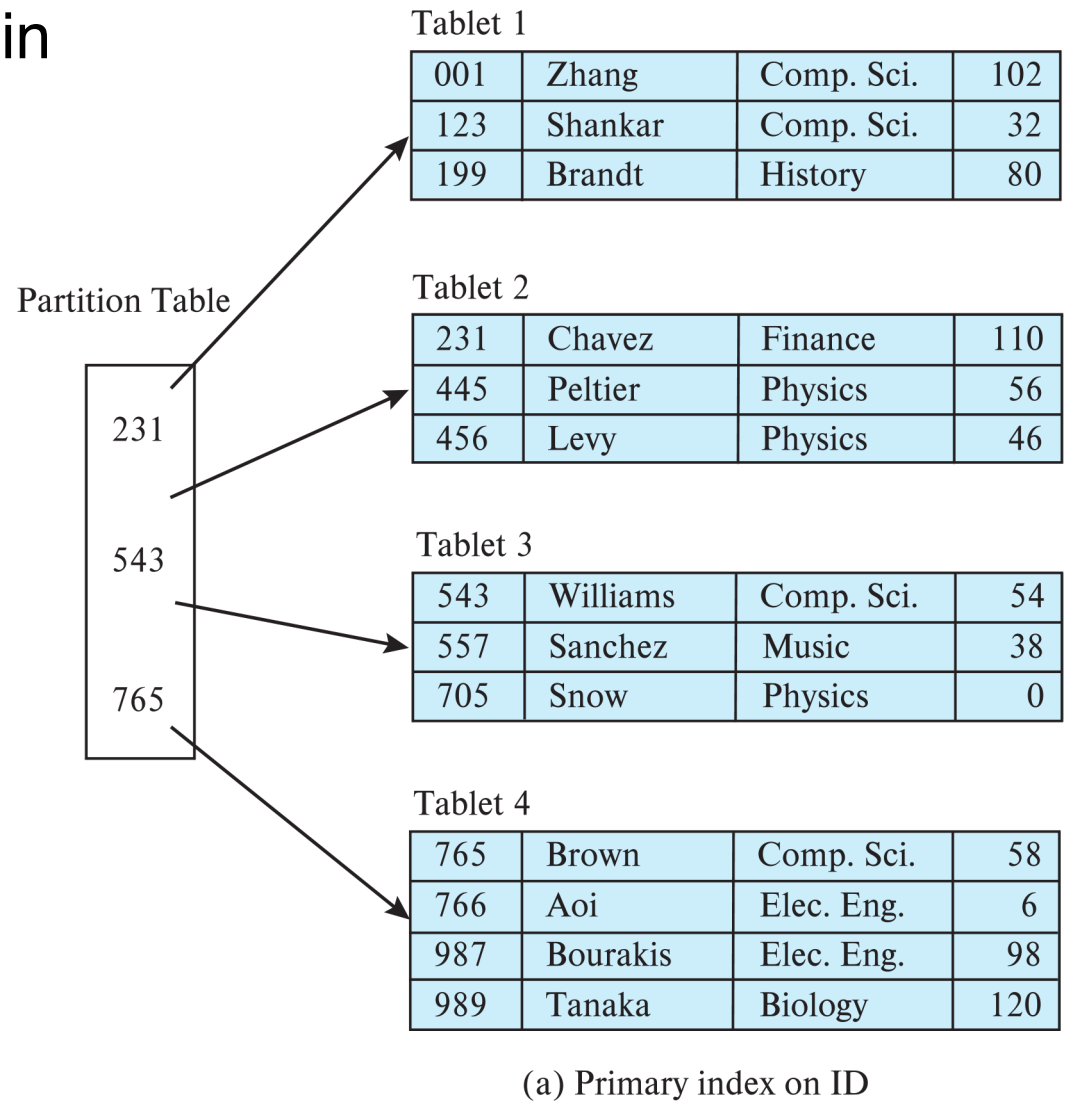
- Main issue: **consistency**
  - Reads should get the latest value
  - Need special concurrency control mechanisms

- Possible solutions:
  - Using a **master replica**
    - Each partition has a master among its replicas
    - Read & update at master, which spreads it to other replicas
  - Using **update protocols**:
    - *Two-phase commit*
    - *Persistent messaging* (Eventual consistency)
    - *Consensus protocols* (Replicas' agreement, no master)

Parallel Indexing

- **Local index**: built on local data stored in a single node
  - Saved in the same node

- **Global index**: built on all the data, regardless of where it is stored
  - Saved partitioned across nodes
  - **Global primary index**: defined on a partitioning attribute
    - Local index
  - **Global secondary index**: defined on an attribute which is not a partitioning one
    - Notice the complexity!!



(a) Primary index on ID

QUERY PROCESSING

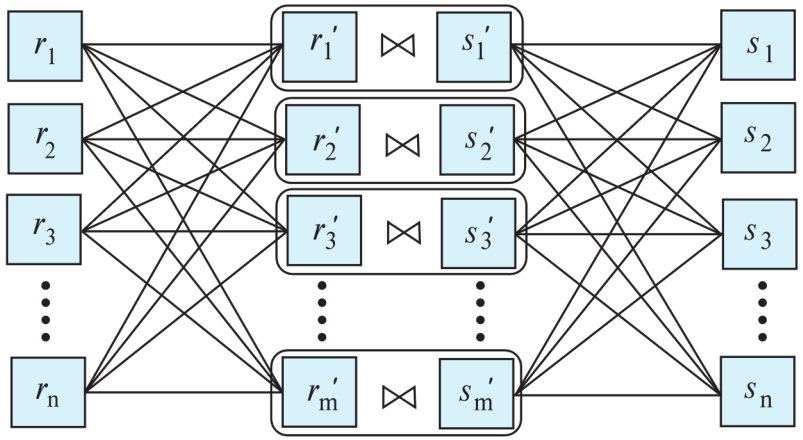
Parallel Query Processing

Parallelism in DBMS:

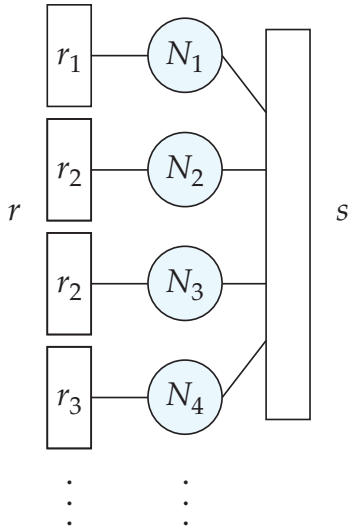
- **Interquery parallelism**: Different queries run in parallel (in different nodes)
  - To scale up transaction processing systems
- **Intraquery parallelism**: A single query run in parallel (different parts of a query, in different nodes)
  - To speed up long-running queries
  - Might be:
    - **Intraoperation parallelism**: A single operation, run in parallel
      Enables high degree of parallelism
    - **Interoperation parallelism**: Different operations of a query, run in parallel
      Limited degree of parallelism

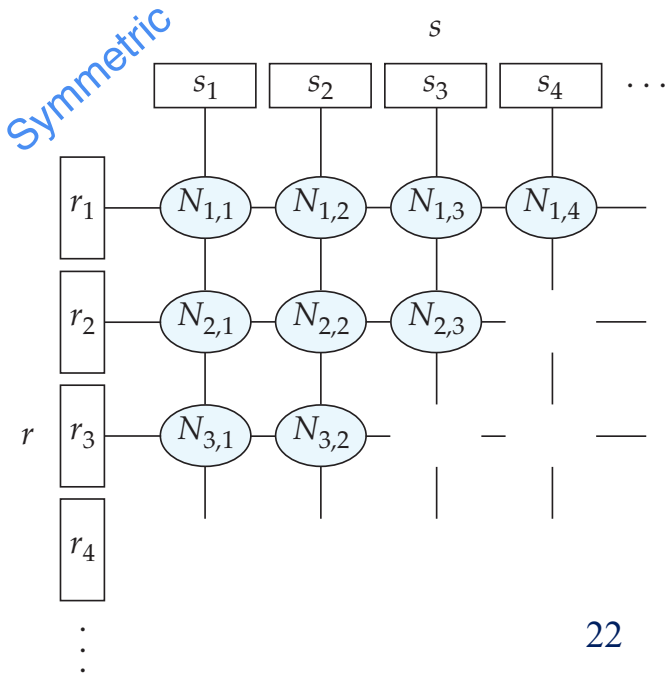
Example: Parallelazing a Join

- **Partitioned join**
  1. Partition *r* and *s* similarly on their join attributes
     - *Range* or *hash partitioning*
  2. Compute the join locally
     - Use locally any join algorithm
  - Natural and equi-joins



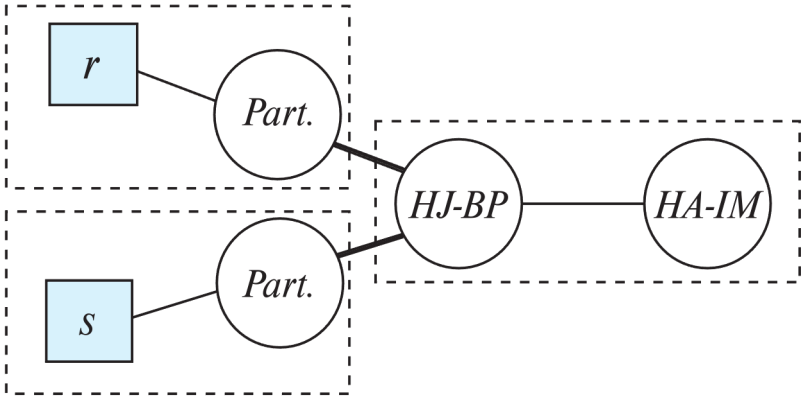
- **Fragment and replicate join**
  1. Partition one or both relations
  2. Compute join
  - (Almost) any join type





Interoperation Parallelism

- **Pipelined parallelism**
  - Each operation executed in parallel in a different core/node

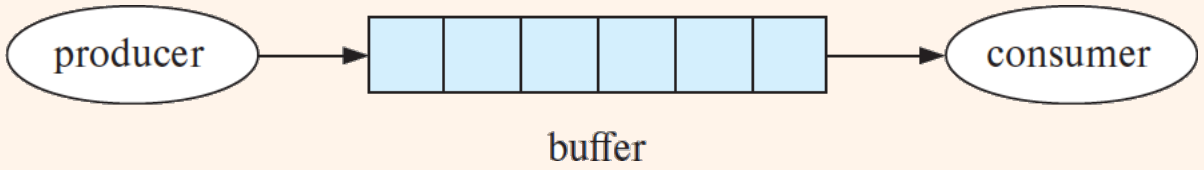


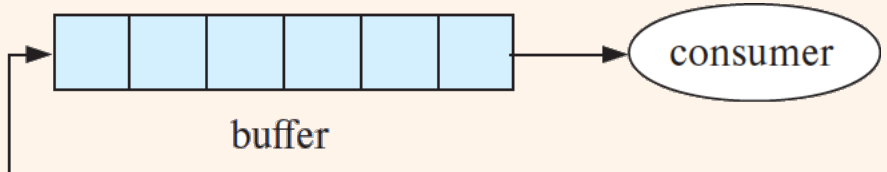
- Limitations
  - Does not provide a high degree of parallelism
  - What happen with blocking operators?
  - Usually, small speedup as operators' execution cost is skewed
- Utility: it avoids writing intermediate results to disk

Exercise: pipeline type

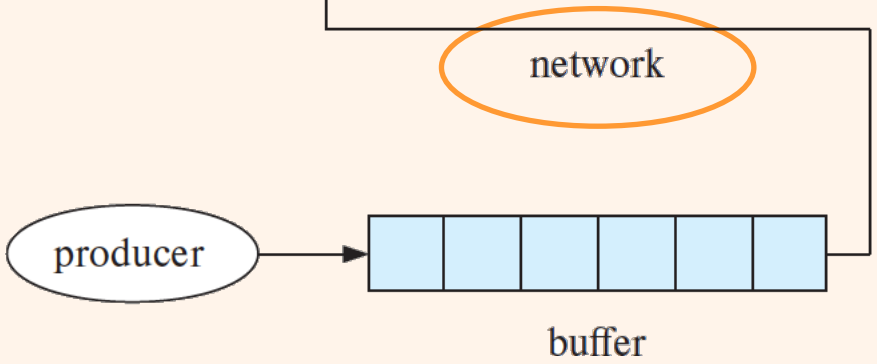
- We know that different operators can be pipelined via an **eager push** or a **lazy pull** procedure…
- Which is more appropriate for parallel pipelines?

Shared memory





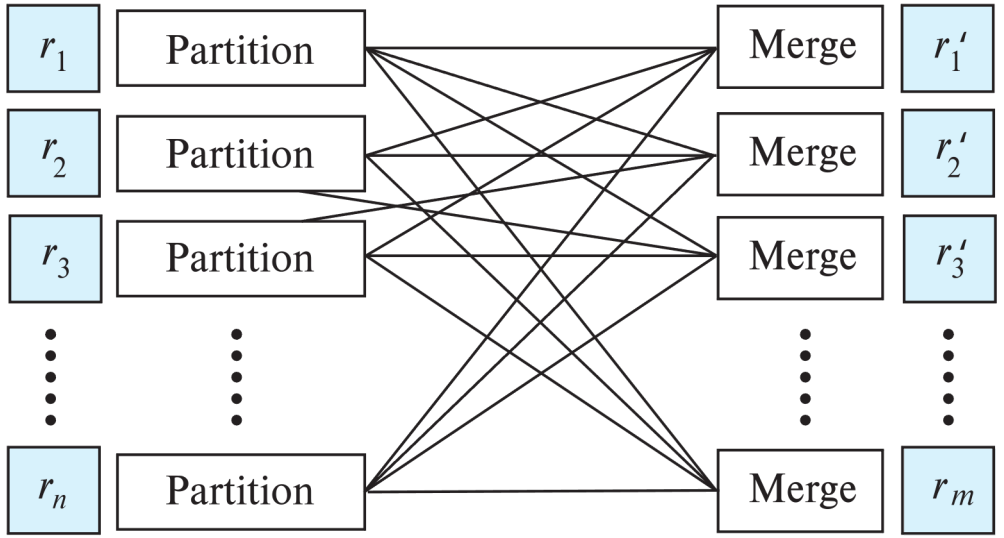
Shared nothing



Interoperation Parallelism

Widely used in parallel DBs

- Repartition using the **exchange operator**, then execute operations only in **local** data. Two steps:
  1. **Partition** step (hash or range)
     E.g.:
     - Send all data to single node
     - Replicate data in all nodes
       - **Broadcasting**
     - Hash/Range partitioning
  2. **Merge** step (random or ordered)



- E.g.:
  - Partitioned join
    1. Exchange op.: hash/range partitioning
    2. Local join
  - Asymmetric fragment and replicate join
    1. Exchange op.: broadcast replication
    2. Local join

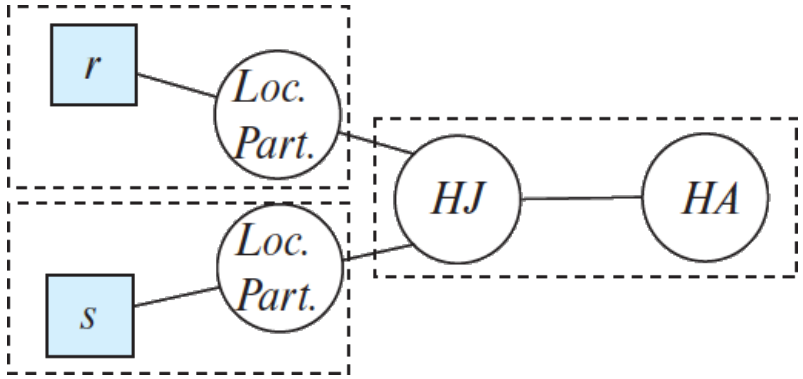
- Major benefit: standard DB query engines can be used at each node, with little adaptation

Query Optimization for Parallel Plans

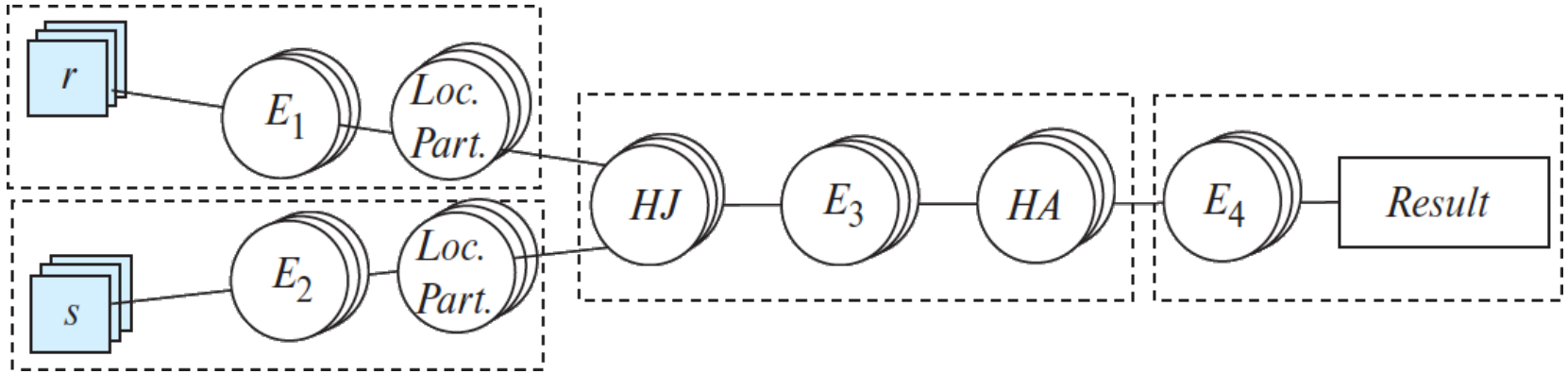
- Query optimization is **significantly more complex** in parallel DBs
  - Cost models are more complicated (e.g., extra partitioning costs)
- A **parallel query evaluation plan** must specify
  - How to parallelize each operation (algorithm, partitions, etc.)
  - How to *schedule* the plan
    - Which nodes perform each operation
    - Which operations are pipelined in same node; Which are executed sequentially or in parallel
- How to choose it? Two alternatives:
  A. (1) Choose most efficient **sequential** plan; (2) parallelize the operations in that plan
     - Heuristic: best sequential plan may not lead to best parallel plan
  B. Use standard query optimizer with an extended cost model and parallelize every operation across all nodes
     - Use **exchange operator** to perform (re)partitioning among nodes

Query Optimization for Parallel Plans

- Query optimization is **significantly more complex** in parallel DBs
  - Cost models are more complicated (e.g., extra partitioning costs)
- A **parallel query evaluation plan** must specify
  - How to parallelize each operation (algorithm, partitions, etc.)
  - How to *schedule* the plan
    - Which nodes perform each operation
    - Which operations are pipelined in same node; Which are executed sequentially or in parallel



Sequential



Parallel

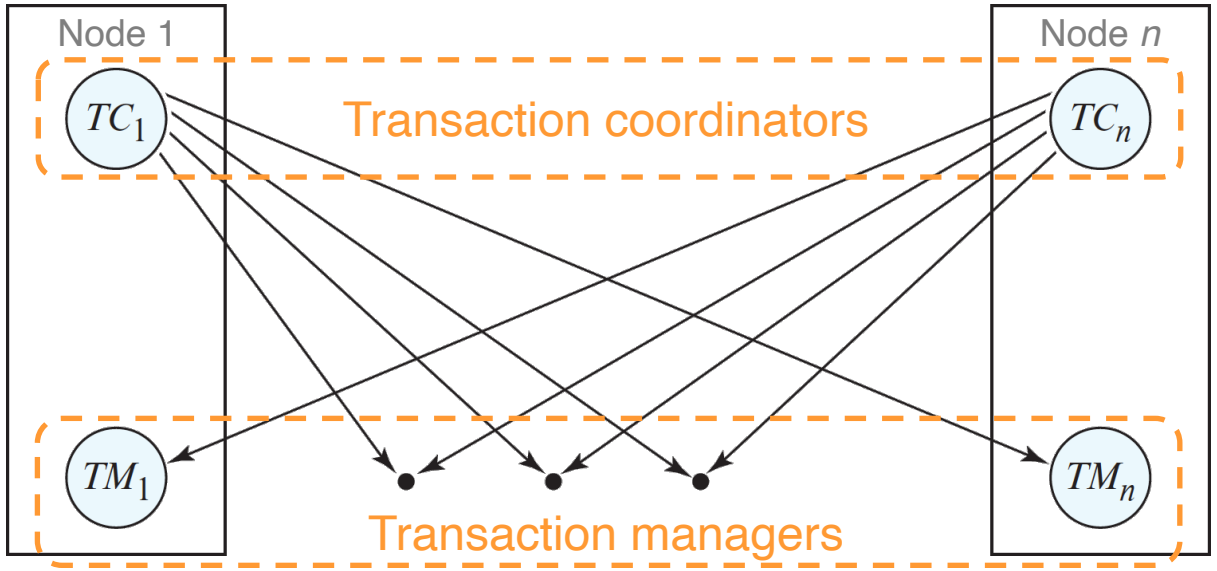
TRANSACTIONS AND CONCURRENCY CONTROL

Distributed Transactions

- **Local transactions**: Access/update data at only one node
- **Global transactions**: Access/update data at more than one node
  - How to ensure ACID properties with global transactions spanning multiple nodes?
- Additional failure types:
  - Failure of a node
  - Loss of messages
    - Handled by network protocols
  - Failure of a communication link
    - Handled by network protocols
  - Network partition

Distributed Transactions

- Each node incorporates two subsystems:
  - Each **transaction coordinator**:
    - Coordinates the execution of all the transactions (local/global) initiated at that node
      - Starts the execution of transactions assigned to the node
      - Breaks up the transaction and distributes subtransactions among the appropriate nodes for execution
      - Coordinates the termination of the transactions (abort/commit)
  - Each **transaction manager** (*standard*):
    - Manages the execution of (sub)transactions that access data stored in the node
      - Both global and local transactions
    - Maintains a log for recovery
    - Carries out concurrency control



Commit

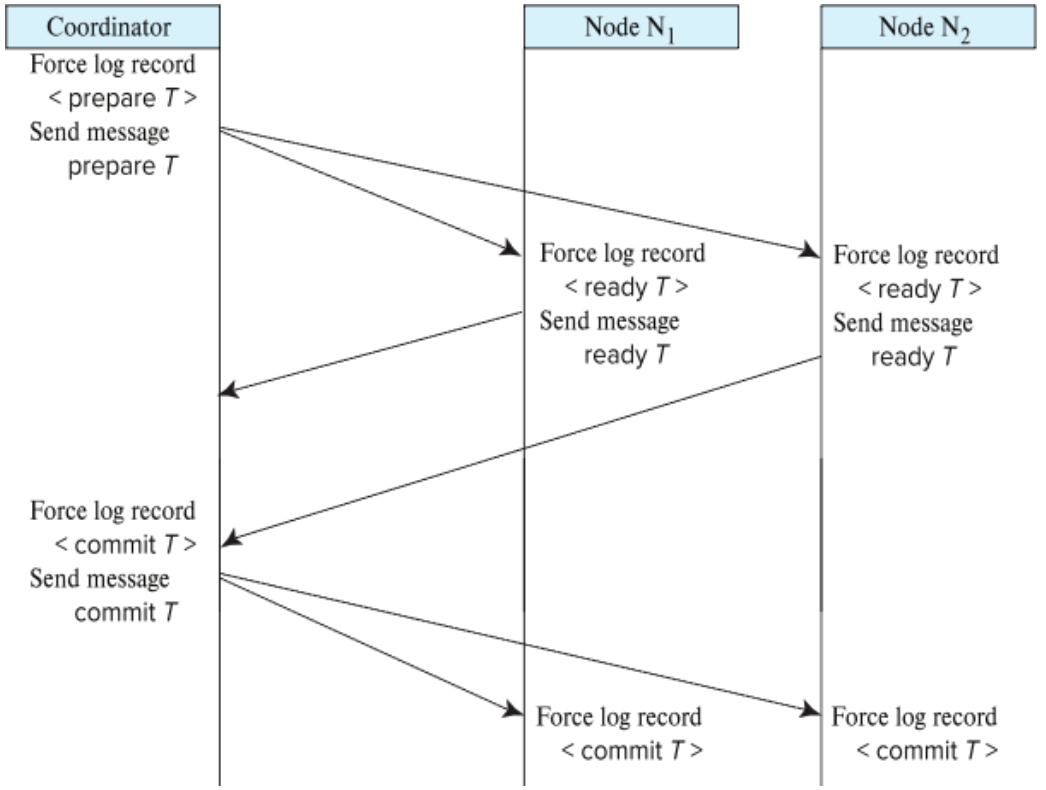
- **Atomicity**: a transaction must commit or abort at all the nodes.
- **Two-phase Commit** (2PC) protocol
  - Starts when all the nodes at which a transaction T has been executed inform the coordinator that T has completed.

Two phases:

1. Making a decision
   - Coordinator C_i asks all nodes to *prepare* to commit transaction T
   - Manager M_i at each node answers if it can commit or not transaction T

No answer? node aborts

2. Implementing the decision
   - Coordinator sends a **commit** T or **abort** T message to each node, according to the previous answers.



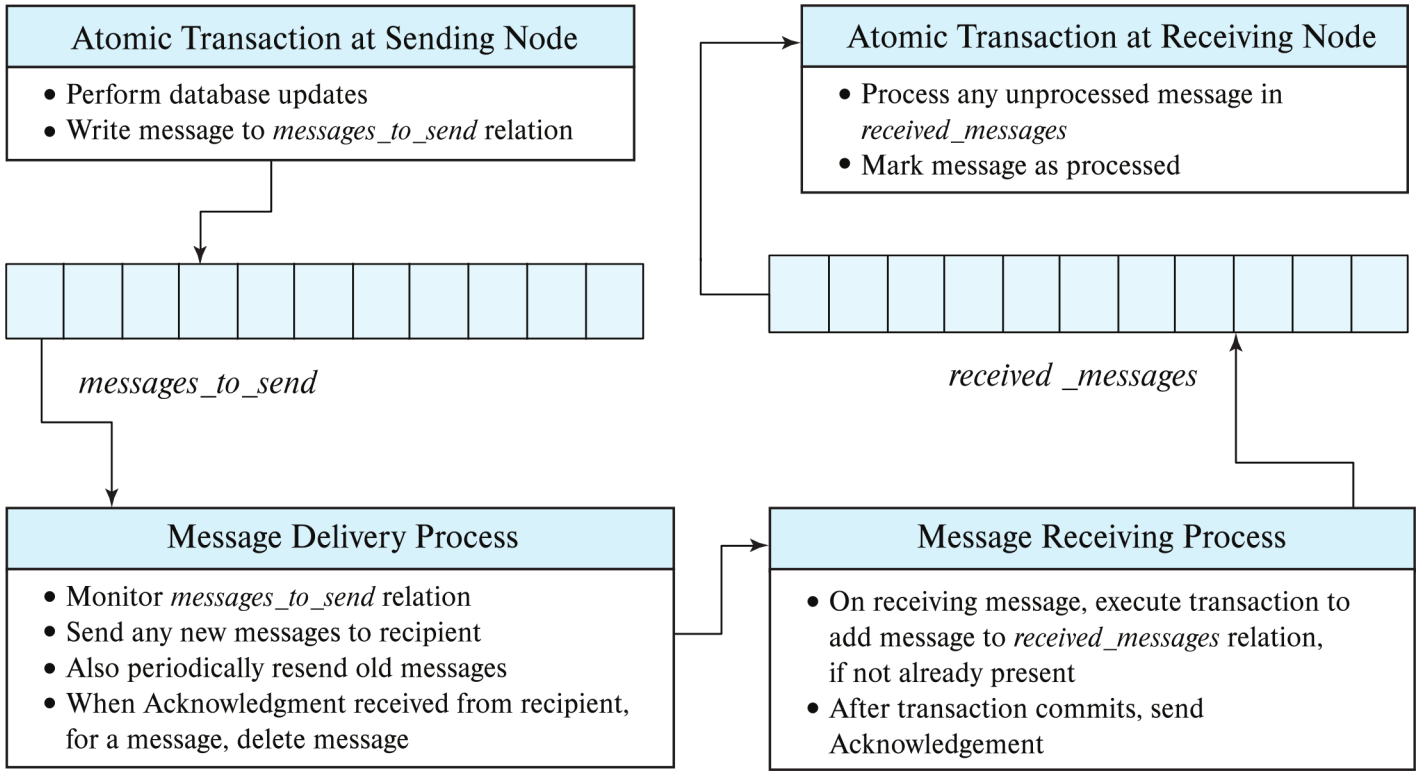
Commit

- Possible failures:
  - Node failure
  - Coordinator failure
    - It might result in a **blocking** situation: resources locked in nodes waiting for the coordinator to recover
  - Network partitioning

- Inter-system coordination:
  - What happens if we need to implement transactions about two DBs from two separate organizations?
    E.g.: Fund transfer between two banks
    - Might first bank's system be blocked by another one's system??!!

Commit

- **Persistent messages**:
  - Once a transaction that sent a message is committed, the message **must** be delivered, and **only once**



Concurrency Control

- **Single lock-manager** running on a *single* site, S_i, where all lock requests are sent
  - Pros:
    - Simple implementation
    - Simple deadlock handling
  - Cons:
    - Central site becomes a bottleneck
    - Vulnerable to central site failure

- **Distributed lock-manager**: locking is implemented by lock managers at each site
  - Lock managers control access to local data items
  - Pros:
    - Work is distributed
    - Robust to failures
  - Cons:
    - Deadlock detection and handling is really a **big issue**

Concurrency Control

- Your system uses with multiple replicas for data?
- **Majority** protocol
  - Transaction requests locks at multiple/all replicas
  - Lock is successfully acquired on the data item only if it obtains the lock at a majority of replicas (>n/2)
  - Benefit:
    - Can deal with node failures
    - Decentralized
  - Drawback:
    - High cost due to multiple messages
    - Possibility of deadlock even when locking a single item

Concurrency Control

- Your system uses with multiple replicas for data?
- **Read one, write all** protocol
  - All copies are updated within writes
    - Locking at **all** the sites
  - You could read from any replica
    - Locking at the **local** site
  - Assumes all copies are available
    - Blocking if **any** of the sites is unavailable!!
- Possible modification: Read one, write *all-available*
  - Unavailable sites are just ignored in writes
  - But, careful! An unreached site may come back up unaware that it was disconnected (connection failure)
    - It has old values, and a read from it would return an old value

An old situation:

Concurrency
vs.
consistency

Parallel and Distributed Database Systems

RAID

- **Redundant Arrays of Independent Disks (RAID)**: disk organization techniques that manage many disks and provide a *single-disk* view
  - **high performance** by using multiple disks in parallel,
  - **high reliability** by storing data redundantly,

Techniques to avoid data loss are **critical** with large numbers of disks

- Chance of disk failure is *much higher* in a collection of disks than in a single disk

  E.g., in a system with 100 disks, each with MTTF of 100,000 hours (~11 years), system MTTF is of 1,000 hours (~41 days)

Boosting Reliability via Redundancy

- **Redundancy**: store extra information, usable to rebuild information in case of disk failure
  - Simplest strategy: **mirroring** (or **shadowing**): duplicate every disk.
    Logical disk consists of two physical disks.
    - Writes on both disks, reads from either disk
    - Data loss only if both disks fail in a time interval not enough for reparation
- Mean time to **data loss** (MTDL) depends on:
  - mean time to failure (MTTF)
  - **mean time to repair** (MTTR)
    With MTTF=100,000 hours and MTTP=10 hours, MTDL= $100{,}000^2/(2*10)$ hours (~57,000 years) (assuming **independent failures**)
    - **Dependent failures** usually due to natural disasters, power failures, …

Boosting Performance via Parallelism

- **Two goals** of disk-system parallelism:
  1. Load-balance of multiple small accesses
  2. Parallelize large accesses

- Improve transfer rate by striping data across disks
  - **Block-level striping**:
    - 1 logical disk mapped to *n* physical disks
    - i^{th} logical block = $\lfloor i/n \rfloor$-th block of physical disk number $(i \bmod n)+1$
    - High data-transfer rate for large reads
      Requests for different blocks run in parallel on the different disks
    - A single block request in this system is as costly as in a physical disk
      But n-1 disks are free to answer other requests in parallel!
    - No redundancy!! (high performance, low reliability)

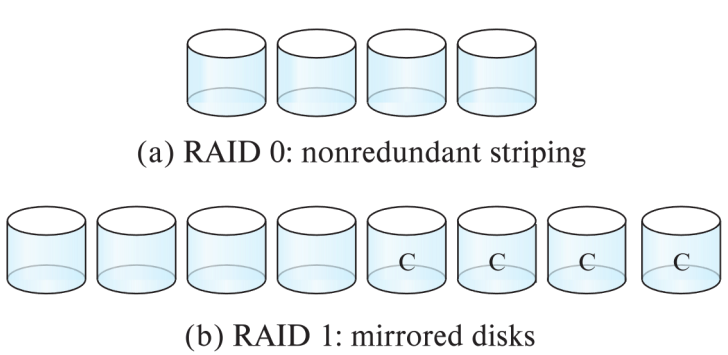
Parity Blocks

- **Parity blocks**: Help to recover data blocks
  - Blocks are grouped in sets. Each set has a *parity block*.
  - Stores in j^{th} bit the XOR of the j^{th} bits of all the set blocks
- You can **recover** a block by computing the XOR of all other set's blocks and their parity block

- When writing to a block, its parity block must be updated. 2 options:
  - Take old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
  - Take all the blocks of the set and recompute the parity block
    More efficient for writing large amounts of data sequentially

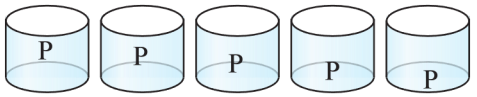
RAID Levels

- Provide redundancy at low cost by using disk striping and parity bits
- Each RAID level has differing cost, performance and reliability:

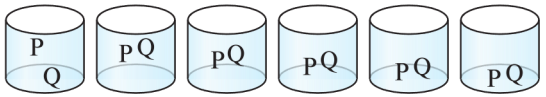


(a) RAID 0: nonredundant striping

(b) RAID 1: mirrored disks



(c) RAID 5: block-interleaved distributed parity



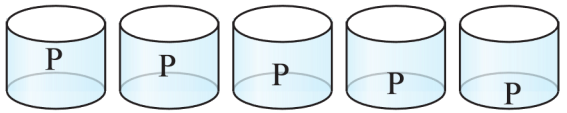
(d) RAID 6: P + Q redundancy

** P and Q: error correcting bits; C: second copy

- **RAID Level 0**: Block striping; non-redundant.
  Used in high-performance applications where data loss is not critical.
- **RAID Level 1**: Mirrored disks with block striping.
  Best write performance. Popular in applications for storing log files.

RAID Levels

- **RAID Level 5**: Block-interleaved distributed parity
  - Splits data and parity blocks among all disks.
    (opposed to level 4, which stored data in *N* disks and parity in a separate disk)
  - With D disks, parity block for n^{th} set of logical blocks is stored on disk (*n mod* D), with the data blocks stored on the other (D-1) disks.



(c) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

  - Data blocks and parity blocks cannot be stored in the same disk for reliability issues

Choice of RAID Level

- Factors to consider:
  - Monetary cost
  - Performance: Number of I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disk
- Guidelines:
  - Level 0 is used only when data safety is not critical
  - Level 1 better write performance and higher storage cost than level 5. It is preferred for applications where:
    - many random/small updates are carried out
  - Level 5 is preferred for applications where:
    - writes are sequential and large (many blocks, parity update at once)
    - writes are uncommon (mostly reads)