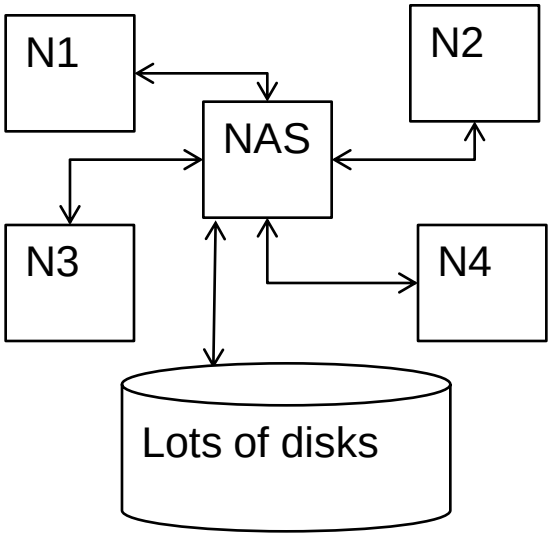
Introduction

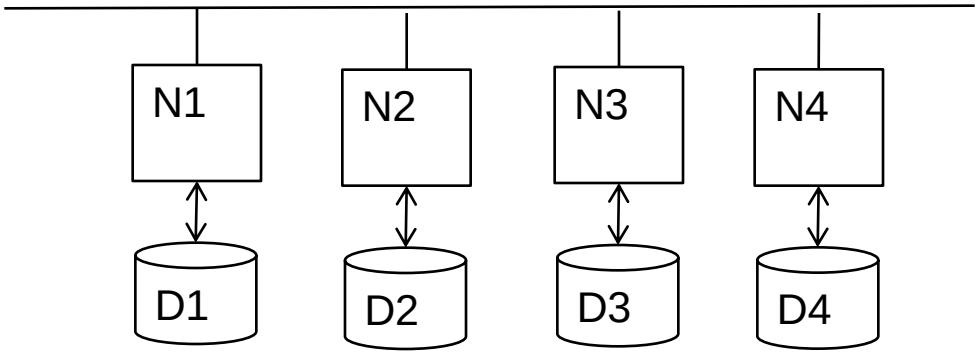
- Parallel machines are becoming quite common and affordable
  - Commodity machines are cheap
  - Multiple processors on a chip
- Databases are growing increasingly large
- Large-scale parallel database systems increasingly used for:
  - processing time-consuming decision-support queries
  - providing high throughput for transaction processing

Architectures

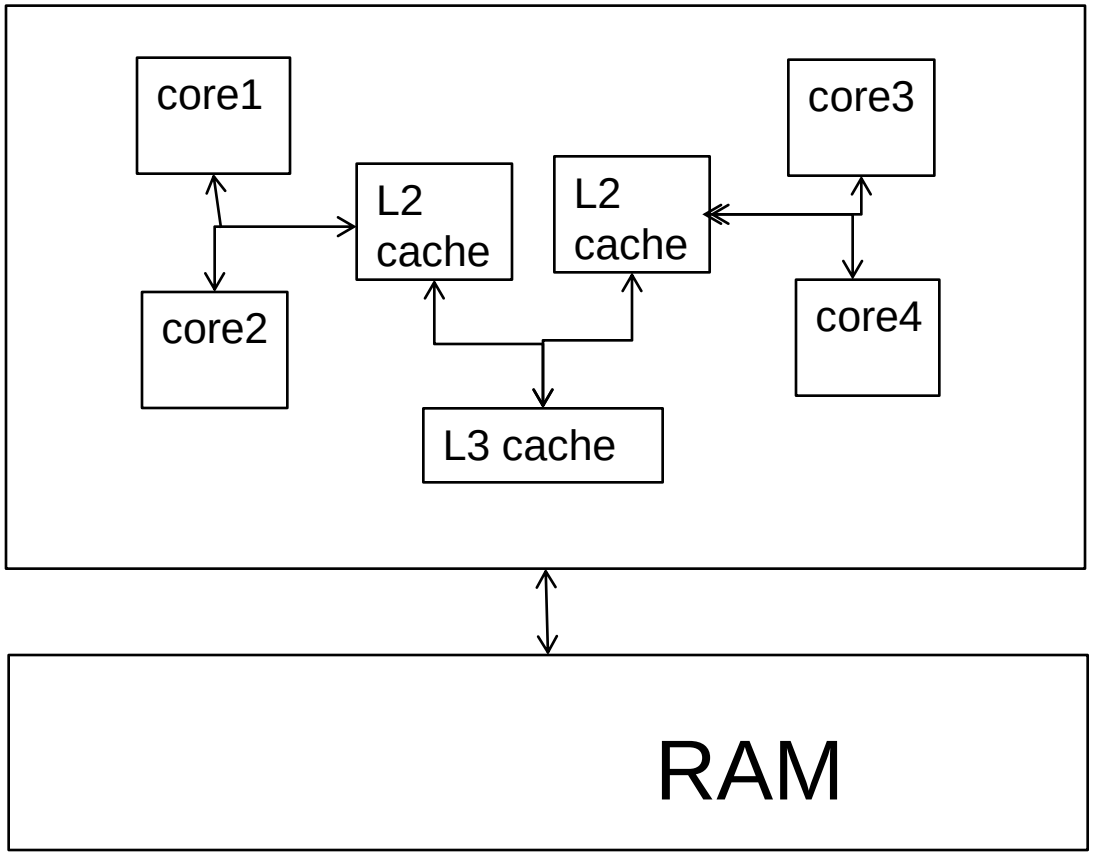
Shared-Disk



Shared Nothing



On a chip



Parallelism in Databases

- Data partitioned across multiple disks → parallel I/O.
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - data can be partitioned and each processor can work independently on its own partition.
  - queries are expressed in high level language (SQL, translated to relational algebra)
    - makes parallelization easier.
- Queries can be run in parallel with each other.
  - Concurrency control takes care of conflicts.
- Thus, databases naturally lend themselves to parallelism.

Partitioning

- Reduce the time to retrieve relations from disk by partitioning the relations on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.
- Partitioning techniques (number of disks = *n*):

  Round-robin:

  Send the *I* th tuple inserted in the relation to disk *i* mod *n*.

  Hash partitioning:

  - Choose one or more attributes as the partitioning attribute(s).
  - Choose hash function *h* with range 0…*n* - 1
  - Let *i* denote result of hash function *h* applied to the partitioning attribute value of a tuple. Send tuple to disk *i*.

Partitioning (Cont.)

- **Range partitioning:**
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector $[v_0, v_1, ..., v_{n-2}]$ is chosen.
  - Let v be the partitioning attribute value of a tuple.
    - Tuples such that $v \leq v_{i+1}$ go to disk $I + 1$.
    - Tuples with $v < v_0$ go to disk 0 and
    - Tuples with $v \geq v_{n-2}$ go to disk n-1.

e.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will go to disk 1, while a tuple with value 20 will go to disk2.

Comparison of Partitioning Techniques

- Evaluate how well partitioning techniques support the following types of data access:
  1. Scanning the entire relation.
  2. Locating a tuple associatively – **point queries**.
     - e.g., $r.A = 25$.
  3. Locating all tuples such that the value of a given attribute lies within a specified range – **range queries**.
     - e.g., $10 \leq r.A < 25$.

Comparison of Partitioning Techniques (Cont.)

Round robin:

- Advantages
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.
- Range queries are difficult to process
  - No clustering -- tuples are scattered across all disks

Comparison of Partitioning Techniques (Cont.)

Hash partitioning:

- Good for scanning a relation.
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
  - Can lookup single disk, leaving others available for answering other queries.
  - Index on partitioning attribute can be local to disk, making lookup and update more efficient
- No clustering, so difficult to answer range queries

Comparison of Partitioning Techniques (Cont.)

Range partitioning:

- Provides data clustering by partitioning attribute value.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
  - Remaining disks are available for other queries.
  - Good if result tuples are from one to a few blocks.
  - If many blocks are to be fetched, they are still fetched from one to a few disks, and potential parallelism in disk access is wasted
    - Example of execution skew.

Handling of Skew

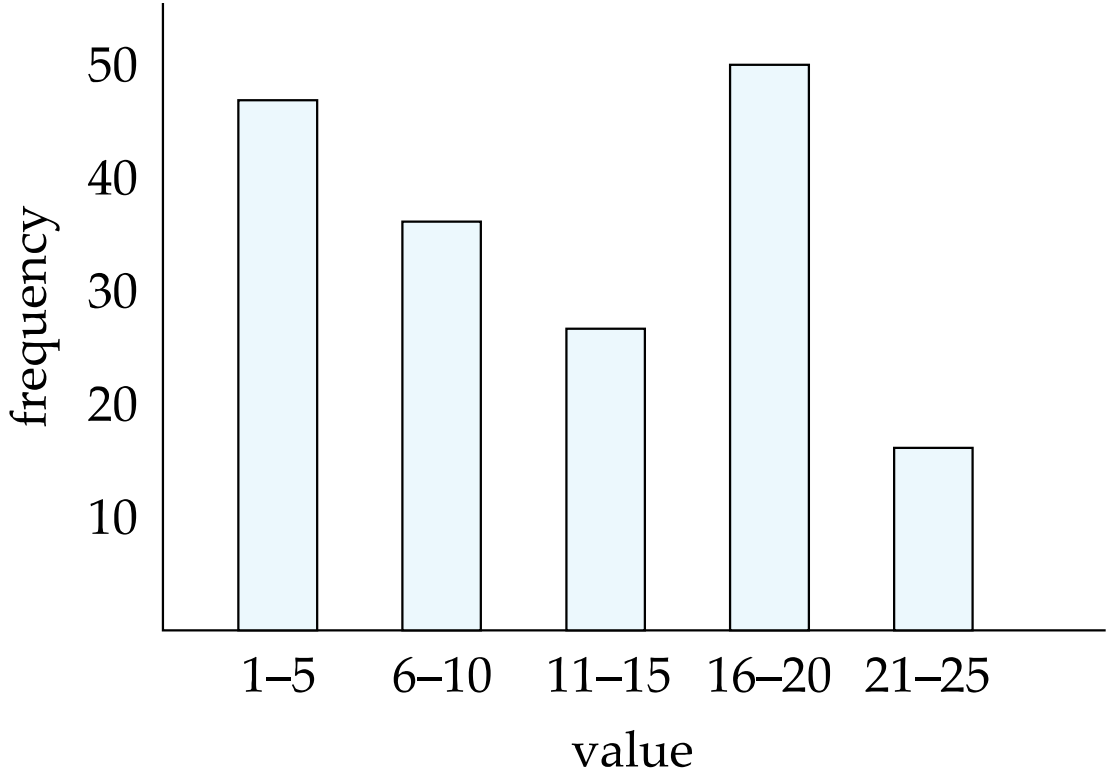
- The distribution of tuples to disks may be **skewed** — that is, some disks have many tuples, while others may have fewer tuples.
- **Attribute-value skew can lead to Partition skew**
  - With range-partitioning, badly chosen partition vector may assign too many tuples to some partitions and too few to others.
  - Less likely with hash-partitioning if a good hash-function is chosen.

Handling Skew in Range-Partitioning

- To create a **balanced partitioning vector** (assuming partitioning attribute forms a key of the relation):
  - Sort the relation on the partitioning attribute.
  - *Let n* denote the number of partitions to be constructed.
  - Construct the partition vector by scanning the relation in sorted order as follows.
    - After every $1/n^{th}$ of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector.
  - Imbalances can result if duplicates are present in partitioning attributes.
- Alternative technique based on **histograms** used in practice

Handling Skew using Histograms

- Balanced partitioning vector can be constructed from histogram in a relatively straightforward fashion
  - Assume uniform distribution within each range of the histogram
- Histogram can be constructed by scanning relation, or sampling (blocks containing) tuples of the relation



Handling Skew Using Virtual Processor Partitioning

- Skew in range partitioning can be handled elegantly using **virtual processor partitioning**:
  - create a large number of partitions (say 10 to 20 times the number of processors)
  - Assign virtual processors to partitions either in round-robin fashion or based on estimated cost of processing each virtual partition
- Basic idea:
  - If any normal partition would have been skewed, it is very likely the skew is spread over a number of virtual partitions
  - Skewed virtual partitions get spread across a number of processors, so work gets distributed evenly!

Interquery Parallelism

- Queries/transactions execute in parallel with one another.
- Increases transaction 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, particularly in a shared-memory parallel database, because even sequential database systems support concurrent processing.
- More complicated to implement on shared-disk or shared-nothing architectures
  - 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.

Cache Coherency Protocol

- Example of a simple cache coherency protocol for shared disk systems:
  - Before reading/writing to a page, the page must be locked in shared/exclusive mode.
  - On locking a page, the page must be read from disk
  - Before unlocking a page, the page must be written to disk if it was modified.
- More complex protocols with fewer disk reads/writes exist.
- Cache coherency protocols for shared-nothing systems are similar. Each database page is assigned a *home* processor. Requests to fetch the page or write it to disk are sent to the home processor.

Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism:
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel. (e.g., pipelining)

the first form 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.

BUT THERE IS A POTENTIAL COST IN COORDINATION.

Parallel Processing of Relational Operations

- Our discussion of parallel algorithms assumes:
  - *read-only* queries
  - shared-nothing architecture
  - *n* processors, P_0, ..., P_{n-1}, and *n* disks D_0, ..., D_{n-1}, where disk D_i is associated with processor P_i.
- If a processor has multiple disks they can simply simulate a single disk D_i.
- Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.
  - Algorithms for shared-nothing systems can thus be run on shared-memory and shared-disk systems.
  - However, some optimizations may be possible.

Parallel Sort

Range-Partitioning Sort

- Choose processors P_0, ..., P_m, where $m \leq n$ -1 to do sorting.
- Create range-partition vector with m entries, on the sorting attributes

1. Redistribute the relation using range partitioning
   - all tuples that lie in the ith range are sent to processor P_i
   - P_i stores the tuples it received temporarily on disk D_i.
   - This step requires I/O and communication overhead.
2. Each processor P_i sorts its partition of the relation locally.
   - Each processor executes same operation (sort) in parallel with other processors, without any interaction with the others (**data parallelism**).
3. Final merge operation is trivial: range-partitioning ensures that, the key values in processor P_i are all less than the key values in P_j

   for I < j

Parallel Sort (Cont.)

Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks D_0, ..., D_{n-1} (in whatever manner).
- Each processor P_i locally sorts the data on disk D_i.
- The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize the merging of sorted runs as follows:
  1. The sorted partitions at each processor P_i are range-partitioned across the processors P_0, ..., P_{m-1}.
  2. Each processor P_i performs a merge on the streams as they are received, to get a single sorted run.
  3. The sorted runs on processors P_0,..., P_{m-1} are concatenated to get the final result.

Parallel Join

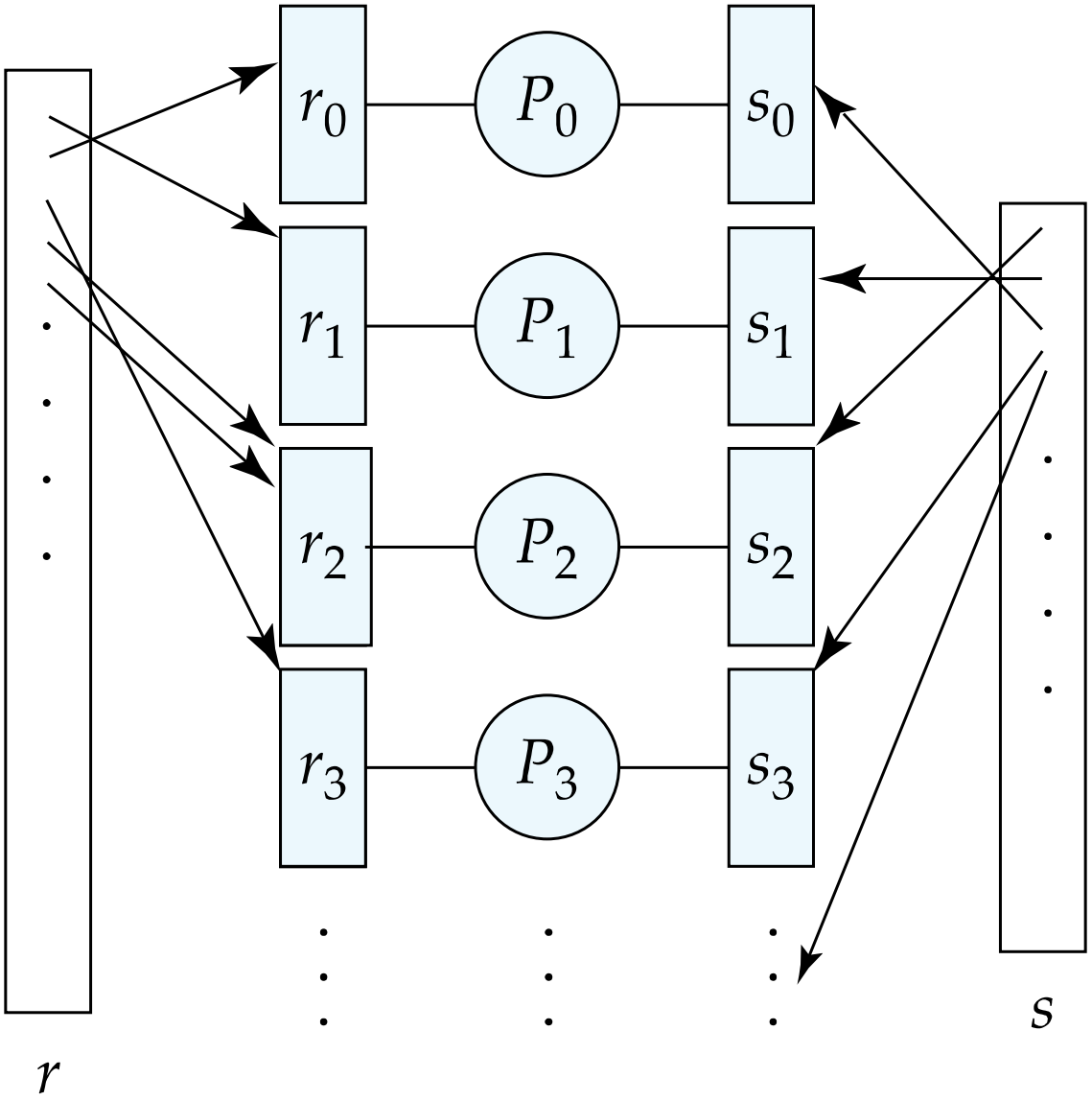
- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result.

Partitioned Join

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and compute the join locally at each processor.
- Let *r* and *s* be the input relations, and we want to compute $r \bowtie_{r.A=s.B} s$.

1. *r* and *s* each are partitioned locally into *n* partitions, denoted r_0, r_1, ..., r_{n-1} and s_0, s_1, ..., s_{n-1}.
   - *r* and *s* must be partitioned on their join attributes *r*.A and *s*.B), using the same range-partitioning vector or hash function.
   - Can use either *range partitioning* or *hash partitioning*.
2. Partitions r_i and s_i are sent to processor P_i,
3. Each processor P_i locally computes $r_i \bowtie_{ri.A=si.B} s_i$.
   - Any of the standard join methods can be used.

Partitioned Join (Cont.)



Other Relational Operations

Selection $\sigma_\theta(r)$

- If θ is of the form $a_i = v$, where a_i is an attribute and v a value.
  - If r is partitioned on a_i the selection is performed at a single processor.
- If θ is of the form $l <= a_i <= u$ (i.e., θ is a range selection) and the relation has been range-partitioned on a_i
  - Selection is performed at each processor whose partition overlaps with the specified range of values.
- In all other cases: the selection is performed in parallel at all the processors.

Other Relational Operations (Cont.)

- Duplicate elimination
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash-partitioning) and perform duplicate elimination locally at each processor.
- Projection
  - Projection without duplicate elimination can be performed as tuples are read in from disk in parallel.
  - If duplicate elimination is required, any of the above duplicate elimination techniques can be used.

Grouping/Aggregation

- Partition the relation on the grouping attributes
- Compute the aggregate values locally at each processor.
- Can reduce cost of transferring tuples during partitioning by partly computing aggregate values before partitioning.
- Consider the **sum** aggregation operation:
  - Perform aggregation operation at each processor P_i on those tuples stored on disk D_i
    - results in tuples with partial sums at each processor.
  - Result of the local aggregation is partitioned on the grouping attributes, and the aggregation performed again at each processor P_i to get the final result.
- Fewer tuples need to be sent to other processors during partitioning.

Interoperator Parallelism

- **Pipelined parallelism**
  - Consider a join of four relations
    - $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
  - Set up a pipeline that computes the three joins in parallel
    - Let P1 be assigned the computation of
      temp1 = $r_1 \bowtie r_2$
    - And P2 be assigned the computation of temp2 = temp1 $\bowtie r_3$
    - And P3 be assigned the computation of temp2 $\bowtie r_4$
  - Each of these operations can execute in parallel, sending result tuples it computes to the next operation even as it is computing further results
    - Provided a pipelineable join evaluation algorithm (e.g., indexed nested loops join) is used

Factors Limiting Utility of Pipeline Parallelism

- Pipeline parallelism is useful since it avoids writing intermediate results to disk
- Useful with small number of processors, but does not scale up well with more processors. One reason is that pipeline chains do not attain sufficient length.
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g., aggregate and sort)
- Little speedup is obtained for the frequent cases of skew in which one operator's execution cost is much higher than the others.

Independent Parallelism

- **Independent parallelism**
  - Consider a join of four relations

    $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$
    - Let P_1 be assigned the computation of temp1 = $r_1 \bowtie r_2$
    - And P_2 be assigned the computation of temp2 = $r_3 \bowtie r_4$
    - And P_3 be assigned the computation of temp1 $\bowtie$ $temp_2$
    - P_1 and P_2 can work **independently in parallel**
    - P_3 has to wait for input from P_1 and P_2
      - Can pipeline output of P_1 and P_2 to P_3, combining independent parallelism and pipelined parallelism
  - Does not provide a high degree of parallelism
    - useful with a lower degree of parallelism.
    - less useful in a highly parallel system.

Query Optimization

- Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.
- Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.
- When **scheduling** execution tree in parallel system, must decide:
  - How to parallelize each operation and how many processors to use for it.
  - What operations to pipeline, what operations to execute independently in parallel, and what operations to execute sequentially, one after the other.
- Determining the amount of resources to allocate for each operation is a problem.
  - e.g., allocating more processors than optimal can result in high communication overhead.
- Long pipelines should be avoided as the final operation may wait a lot for inputs, while holding precious resources

Query Optimization (Cont.)

- The number of parallel evaluation plans from which to choose from is much larger than the number of sequential evaluation plans.
  - Therefore heuristics are needed while optimization
- Two alternative heuristics for choosing parallel plans:
  - No pipelining and inter-operation pipelining; just parallelize every operation across all processors.
    - Finding best plan is now much easier --- use standard optimization technique, but with new cost model
  - First choose most efficient sequential plan and then choose how best to parallelize the operations in that plan.
    - Can explore pipelined parallelism as an option
- Choosing a good physical organization (partitioning technique) is important to speed up queries.

Design of Parallel Systems

Some issues in the design of parallel systems:

- Parallel loading of data from external sources is needed in order to handle large volumes of incoming data.
- Resilience to failure of some processors or disks.
  - Probability of some disk or processor failing is higher in a parallel system.
  - Operation (perhaps with degraded performance) should be possible in spite of failure.
  - Redundancy achieved by storing extra copy of every data item at another processor.

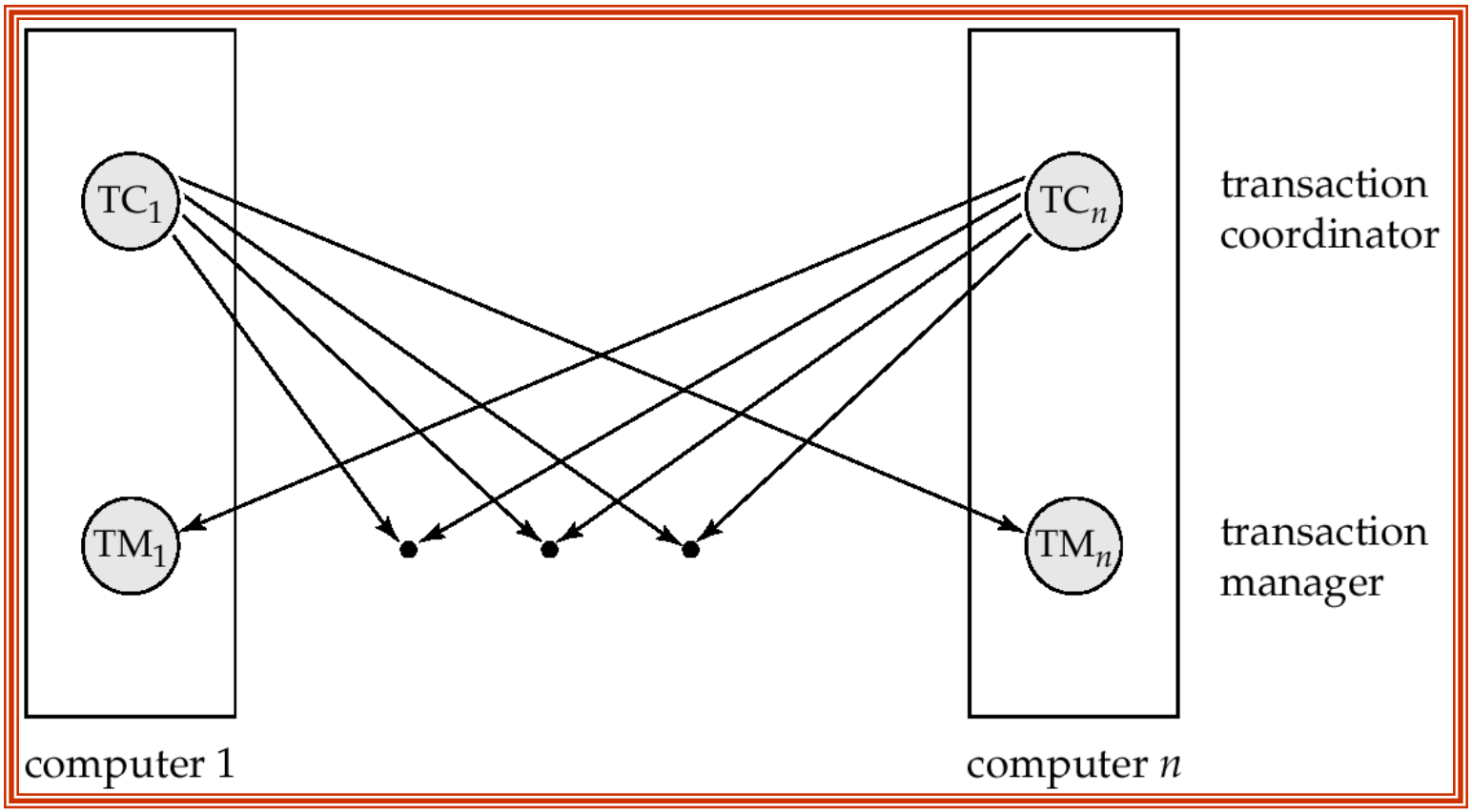
Design of Parallel Systems (Cont.)

- On-line reorganization of data and schema changes must be supported.
  - For example, index construction on terabyte databases can take hours or days even on a parallel system.
    - Need to allow other processing (insertions/deletions/updates) to be performed on relation even as index is being constructed.
  - Basic idea: index construction tracks changes and "catches up" on changes at the end.
- Also need support for on-line repartitioning and schema changes (executed concurrently with other processing).

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.

Transaction System Architecture



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of messages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2 *PC*) protocol is widely used
- The *three-phase commit* (3 *PC*) protocol is more complicated and more expensive, but avoids some drawbacks of two-phase commit protocol.

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
  - force *all records* for *T* to stable storage
  - add the record <**ready** *T*> to the log
  - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- *T* can be committed if C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record reaches stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: site executes **redo** (*T*)
- Log contains <**abort** *T*> record: site executes **undo** (*T*)
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*)
  - If *T* aborted, **undo** (*T*)
- The log contains no control records concerning *T* implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response C_1 must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for *T* is executing then participating sites must decide on *T*'s fate:
  1. If an active site contains a <**commit** *T*> record in its log, then *T* must be committed.
  2. If an active site contains an <**abort** *T*> record in its log, then *T* must be aborted.
  3. If some active participating site does not contain a <**ready** *T*> record in its log, then the failed coordinator C_i cannot have decided to commit *T*. Can therefore abort *T*.
  4. If none of the above cases holds, then all active sites must have a <**ready** *T*> record in their logs, but no additional control records (such as <**abort** *T*> of <**commit** *T*>). In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem** : active sites may have to wait for failed coordinator to recover.