COURSE 12

Parallel DBMS

Parallel DBMS

- Most DB research focused on specialized hardware
  - *CCD Memory*: Non-volatile memory like, but slower than flash memory
  - *Bubble Memory*: Non-volatile memory like, but slower than flash memory
  - *Head-per-track Disks*: Hard disks with one head per data track
  - *Optical Disks*: Similar to, but higher capacity than, CD-RW's

Parallel DBMS (cont)

- In past, database machines were custom built to be highly parallel:
  - *Processor-per-track*: There is one processor associated with each disk track. Searching and scanning can be done with one disk rotation
  - *Processor-per-head*: There is one processor associated with each disk read/write head. Searching and scanning can be done as fast as the disk can be read
  - *Off-the-disk*: There are multiple processors attached to a large cache where records are read into and processed

Parallel DBMS (cont)

- Hardware did not fulfill promises, so supporting these three parallel machines didn't seem feasible in the future
- Also, disk I/O bandwidth was not increasing fast enough with processor speed to keep them working at these higher speeds
- Prediction was that off-the shelf components would soon be used to implement database servers
- Parallelism was expected to be a thing of the past as higher processing speeds meant that parallelism wasn't necessary

Parallel DBMS (cont)

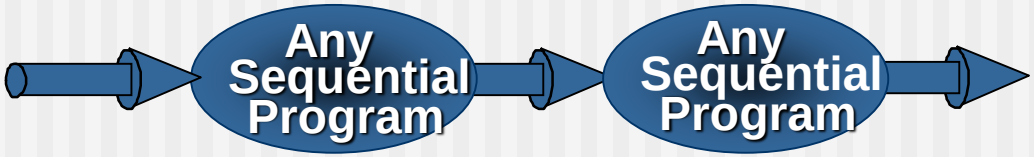
- Prediction was wrong!
- Focus is now on using off-the-shelf components
  - CPU's double in speed every 18 months
  - Disks double in speed much more slowly
  - I/O is still a barrier but is now much better
- But, parallel processing is still a very useful way to speed up operations
  - We can implement parallelism on mass-produced, modular database machines instead of custom designed ones

Parallel DBMS (cont)

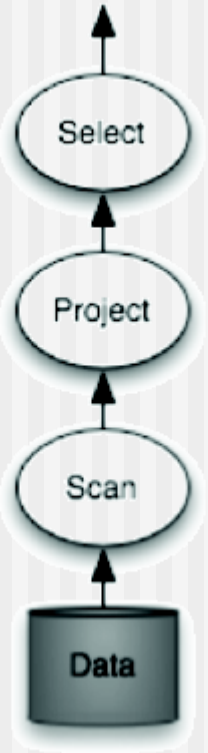
- Why parallel database systems?
  - They are ideally suited to relational data which consists of uniform operations on data streams
  - Output from one operation can be streamed as input next operation
  - Data can be worked on by multiple processors / operations at once
  - Both allow higher throughput. This is good!
  - Both require high-speed interconnection buses between processors/operations. This could be bad!

Definitions

- *Pipeline parallelism:*
  - Many machines each doing one step in a multi-step process.
  - Two operators working in series a data set: output from one operator is piped into another operator to speed up processing



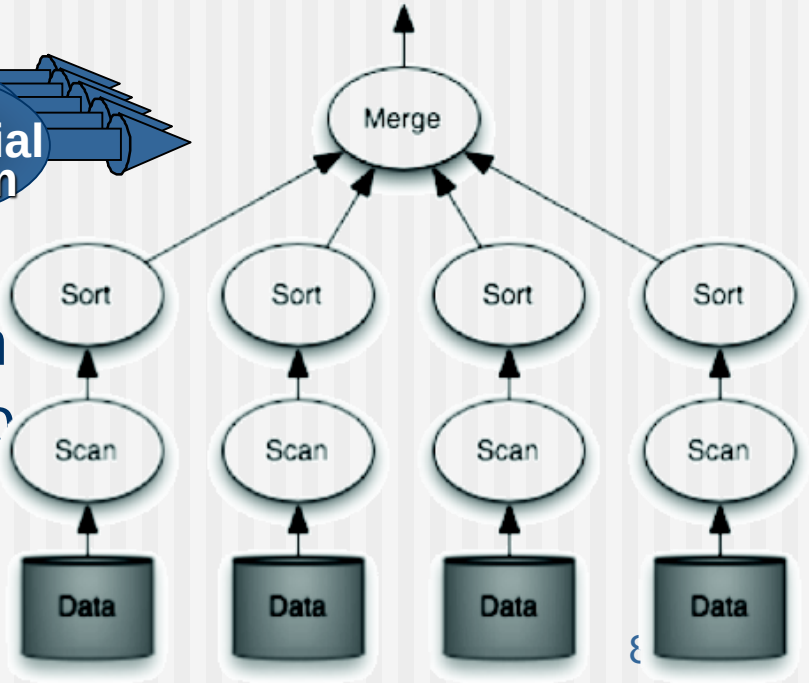
  - Example: Sequential *Scan* fed into a *Projection* fed into a *Select*… etc.



Definitions

- *Partition parallelism:*
  - Many machines doing the same thing to different pieces of data.
  - Many operators working together on one operation; each operator does some of the work on the input data and produces output

  - Example: A table is partitioned over several machines. One scan operator runs on each machine to feed input to a Sort



DBMS: The Parallel Success Story

- DBMSs are the most (only?) successful application of parallelism.
  - Teradata, Tandem vs. Thinking Machines, KSR..
  - Every major DBMS vendor has some parallel server
  - Workstation manufacturers now depend on parallel DB server sales.
- Reasons for success:
  - Bulk-processing (= partition parallelism).
  - Natural pipelining.
  - Inexpensive hardware can do the trick!
  - Users/app-programmers don't need to think in parallel

Mainframes vs. Multiprocessor Machines

Mainframes are unable to provide the high speed processing and interconnect systems	Multiprocessor (Parallel) systems based on inexpensive off-the-shelf components
Mainframes are unable to scale once built. Thus, their total power is defined when they are first constructed	Multiprocessor machines can grow incrementally through the addition of more memory, disks or processors when scale needs to be increased
Mainframes are shared-disk or shared memory systems	Multiprocessor systems are shared-nothing systems

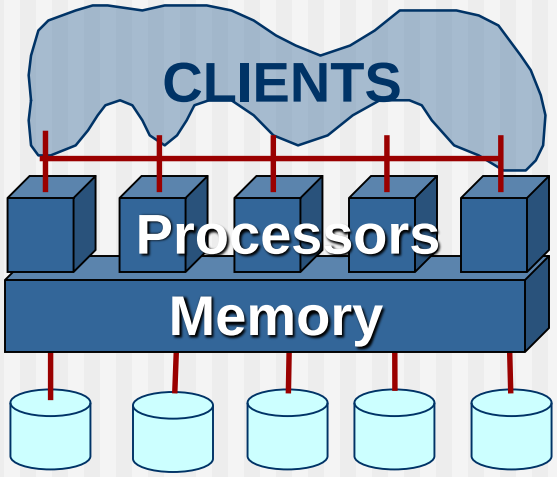
Architecture Issue: Shared What?

Shared Memory (SMP)

- All processors have direct access to a common global memory and all hard disks
- Example: A single workstation

- Characteristics:
  - Easy to program
  - Expensive to build
  - Difficult to scale-up

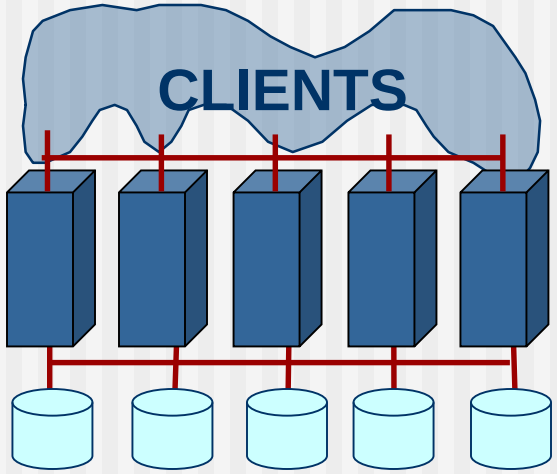
- DBMSs: Sequent, SGI, Sun



Architecture Issue: Shared What? (cont)

Shared Disk

- All processors have access to a private memory and all hard disks
- Example: A network of servers accessing a SAN (Storage Area Network) for storage

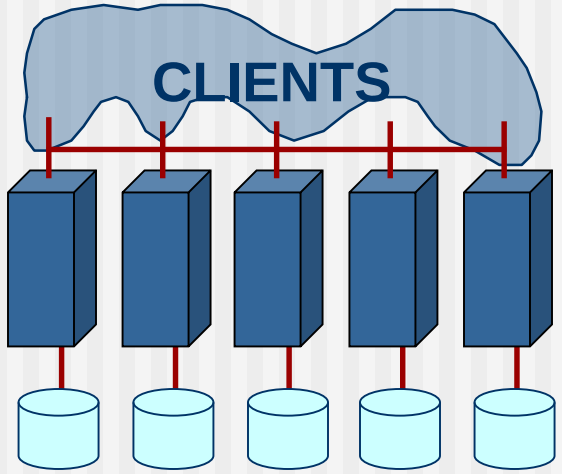


- DBMSs: **VMScluster, Sysplex**

Architecture Issue: Shared What? (cont)

Shared Nothing (network)

- All processors have access to a private memory and disk, and act as a server for the data stored on that disk
- Example: A network of FTP mirrors



- Characteristics:
  - Hard to program
  - Cheap to build
  - Easy to scaleup
- DBMSs: Tandem, Teradata, SP2

What Systems Work This Way

Shared Nothing

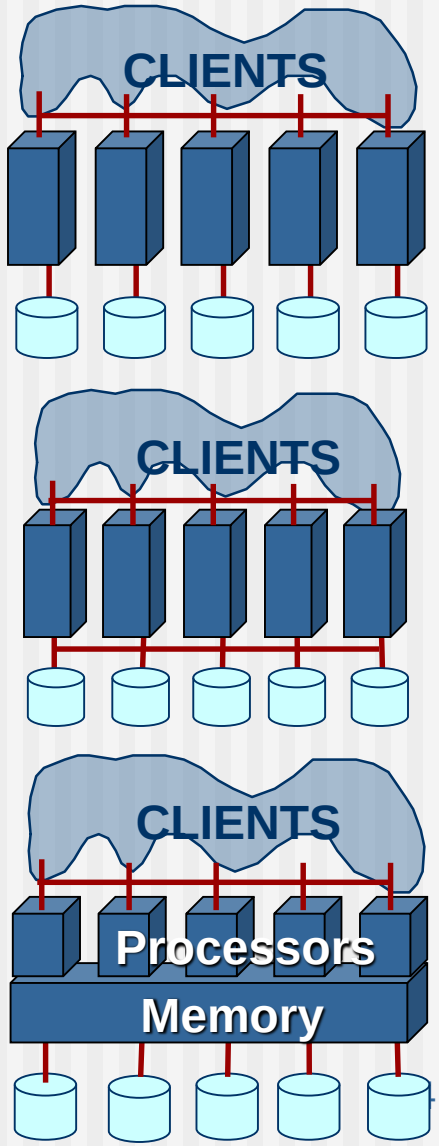
Teradata:	400 nodes
Tandem:	110 nodes
IBM / SP2 / DB2:	128 nodes
Informix/SP2	48 nodes
ATT & Sybase	? nodes

Shared Disk

Oracle	170 nodes
DEC Rdb	24 nodes

Shared Memory

Informix	9 nodes
RedBrick	? nodes



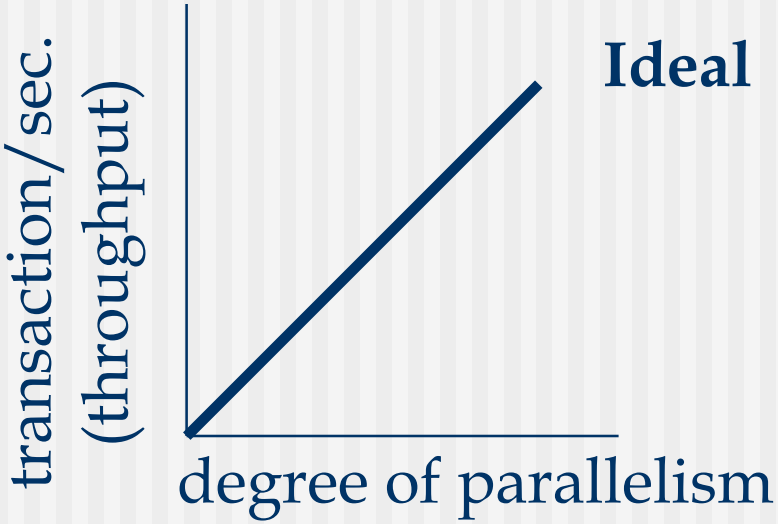
Architecture Issue

- Trend in parallel database systems is towards shared nothing designs:
  - Minimal interference from minimal resource sharing
  - Don't require an overly powerful interconnect
  - Don't move large amounts of data through an interconnect - only questions and answers
  - Traffic is minimized by reducing data at one processor before sending over interconnect
  - Can scale to hundreds and thousands of processors without increasing interference (may slow network performance)

Some Terminology

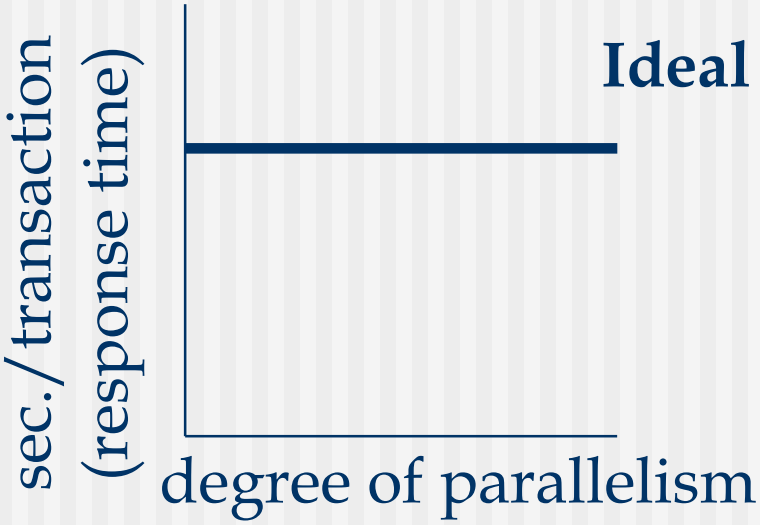
Speed-Up

More resources means proportionally less time for given amount of data.



Scale-Up

If resources increased in proportion to increase in data size, time is constant.



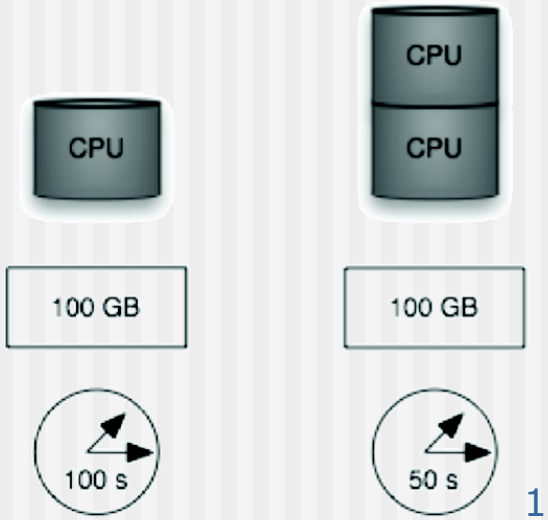
Ideally, a parallel system would experience *Linear Speed-up* and *Linear Scale-up*

Speed-Up

$$\text{Speed-Up} = \frac{\text{small system processing time}}{\text{large system processing time}}$$

- Speed-up is linear if an N-times larger system executes a task N-times faster than a smaller system running the same task (i.e. above equation evaluates to N)
- Hold the problem size constant and grow the system

- Example
  - Speed-up = 100/50
  - System size increased by factor of 2
    $\Rightarrow$ linear speed-up



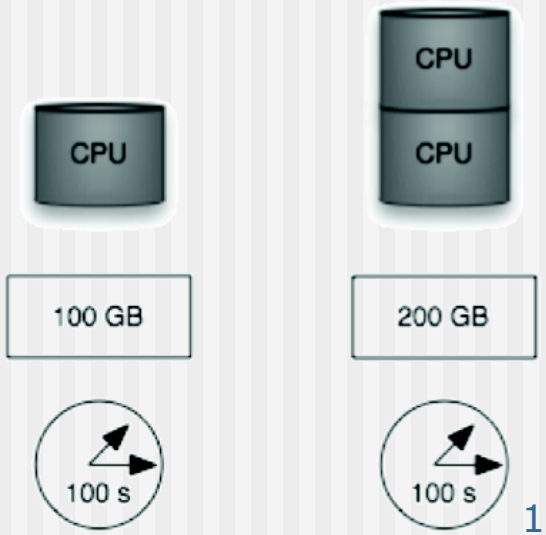
Scale-Up

$$\text{Scale-Up} = \frac{\text{small system processing time on small problem}}{\text{large system processing time on big problem}}$$

- Scale-up is linear if an N-times larger system executes a task N-times larger in the same time as a smaller system running a smaller task (i.e. above equation evaluates to 1)
- Scale-up grows the problem and system size

- Example
  - Scale-up = 100/100
  - System and problem size increased by factor of 2 and scale-up is 1

    ⇒ we have linear scale-up



Barriers to Linear Speed-up/Scale-up

- *Startup*: Time needed to start-up the operation (i.e. launch the processes)

- *Interference*: Slowdowns imposed on processes through the access of shared resources

- *Skew*: Number of parallel operations increase but the variance in operation size increases too (job is only as fast as the slowest operation!)

Architectures and Speed-up/Scale-up

- *Shared nothing* architectures experience near-linear speedups and scaleups
  - Some time must be spent coordinating the processes and sending requests and replies across a network
- *Shared memory* systems do not scale well
  - Operations interfere with each other when accessing memory or disks
- *Shared disk* systems do not scale well
  - Need to deal with the overhead of locking, concurrent updates, and cache consistency

Architectures and Speed-up/Scale-up

- The shortcomings of shared memory and shared disk systems makes them unattractive next to shared nothing systems
- Why are shared nothing systems only now becoming popular?
  - Modular components have only recently become available
  - Most database software is written for uni-processor systems and must be converted to receive any speed boost

Remember Parallelism?

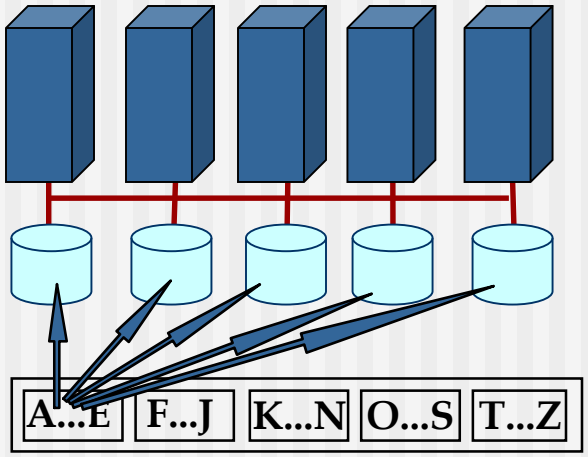
- Factors limiting pipelined parallelism
  - Relational pipelines are rarely very long (ie: chains past length 10 are rare)
  - Some operations do not provide output until they have consumed all input (i.e. sorts and aggregation)
  - Execution cost of some operations is greater than others (i.e. *skew*)
- Thus, pipelined parallelism often will not provide a large speed-up!

Remember Parallelism?

- Partitioned execution offers better speed-up and scale-up
  - Data can be partitioned so inputs and outputs can be used and produced in a *divide-and-conquer* manner
- Data partitioning is done by placing data fragments on several different disks/sites
  - Disks are read and written in parallel
  - Provides high I/O bandwidth without any specialized hardware

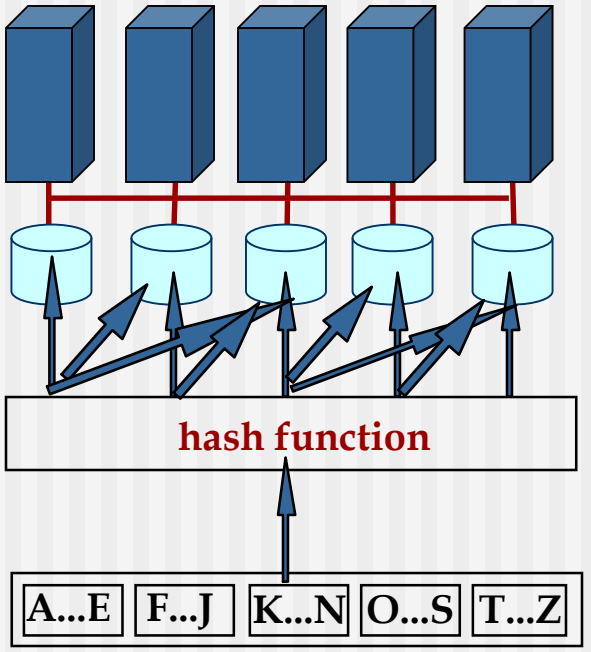
Automatic Data Partitioning

- Round-robin Partitioning
  - Data is fragmented in a round-robin fashion (i.e. The first tuple goes to site A, then site B, then site C, then site A, etc.)
  - Ideal if a table is to be sequentially scanned (spread load)
  - Poor performance if a query wants to associatively access data (i.e. find all students of age 23)



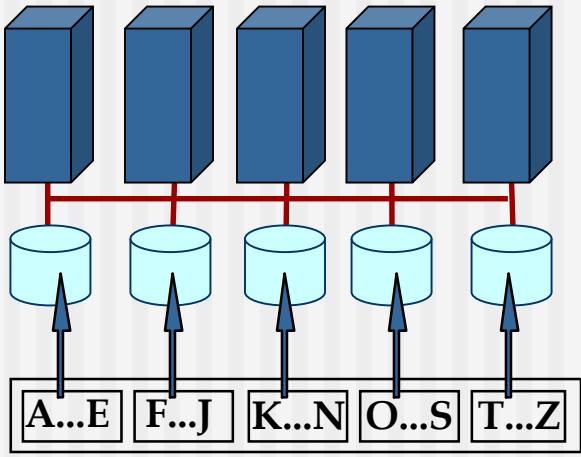
Automatic Data Partitioning (cont)

- Hash Partitioning
  - Data is fragmented by applying a hash function to an attribute of each row
  - The function specifies placement of the row on a specific disk
  - Ideal if a table is to be sequentially or associatively accessed (i.e. all students of age 23 are on the same disk - saves the overhead of starting a scan at each disk)



Automatic Data Partitioning (cont)

- Range Partitioning
  - Tuples are clustered together on disk
  - Ideal for sequential and associative access. Also ideal for clustered access (again, saves the overhead of beginning scans on other disks; also does not require a sort!)
  - Risk of data skew (i.e. all data ends up in one partition) and execution skew (i.e. all the work is done by one disk)
    - Need a good partitioning scheme to avoid this



Partitioning Conclusions

- Partitioning raises several new design issues
  - Databases must now have a fragmentation strategy
  - Databases must now have a partitioning strategy
- Partitioning helps us in several ways
  - Increased partitioning decreases response time (usually, and to a certain point)
  - Increased partitioning increases throughput
  - Sequential scans are faster as more processors and disks are working on the problem in parallel
  - Associative scans are faster as fewer tuples are stored at each node meaning a smaller index is searched

Partitioning Conclusions (cont)

- Partitioning can actually increase response time
  - Occurs when starting a scan at a site becomes a major part of the actual execution time. This is bad!
- Data partitioning is only half the story
  - We've seen how partitioning data can speed up processing
  - How about partitioning relational operators?

Different Types of DBMS parallelism

- Intra-operator parallelism
  - get all machines working to compute a given operation (scan, sort, join)
- Inter-operator parallelism
  - each operator may run concurrently on a different site (exploits pipelining)
- Inter-query parallelism
  - different queries run on different sites
- We'll focus on intra-operator parallelism

Parallelizing Relational Operators

- Objective
  - Use existing operator implementations without modifications
- Only 3 mechanisms are needed
  - Operator *replication*
  - *Merge* operator
  - *Split* operator
- Result is a parallel DBMS capable of providing linear speed-up and scale-up!

Intra-operator parallelism

- If we want to parallelize operators, there are two options:

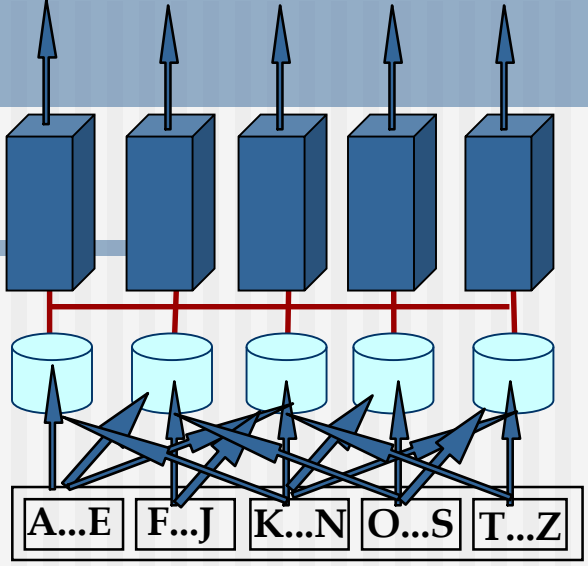
1) Rewrite the operator so that it can make use of multiple processors and disks:
   - Could take a long time and a large amount of work
   - Could be system specific programming

2) Use multiple copies of unmodified operators with partitioned data
   - Very little code needs to be modified
   - Should be moveable to any new hardware setup

Parallel Scans

- Scan in parallel, and merge.
- Selection may not require all sites for range or hash partitioning.
- Indexes can be built at each partition.
- Question: How do indexes differ in the different schemes?
  - Think about both lookups and inserts!
  - What about unique indexes?

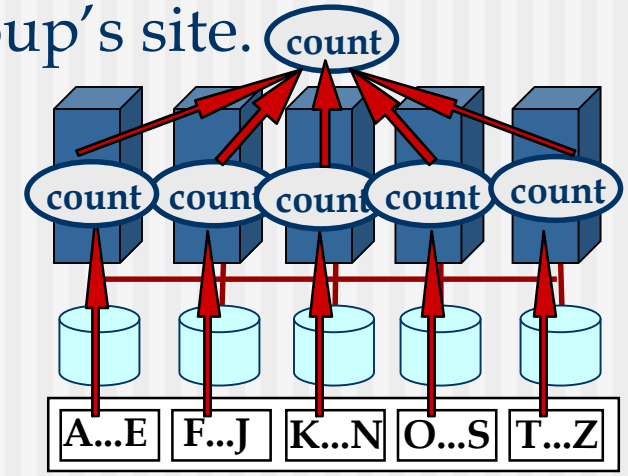
Parallel Sorting



- Current records:
  - 8.5 Gb/minute, shared-nothing; Datamation benchmark in 2.41 secs (http://now.cs.berkeley.edu/NowSort/)
- Idea:
  - Scan in parallel, and range-partition as you go.
  - As tuples come in, begin "local" sorting on each
  - Resulting data is sorted, and range-partitioned.
  - Problem: ***skew!***
  - Solution: "sample" the data at start to determine partition points.

Parallel Aggregates

- For each aggregate function, need a decomposition:
  - **count**(S) = **Σ count**(s(i)), ditto for **sum**()
  - **avg**(S) = (**Σ sum**(s(i))) / **Σ count**(s(i))
  - and so on...
- For groups:
  - Sub-aggregate groups close to the source.
  - Pass each sub-aggregate to its group's site.
    - Chosen via a hash fn.



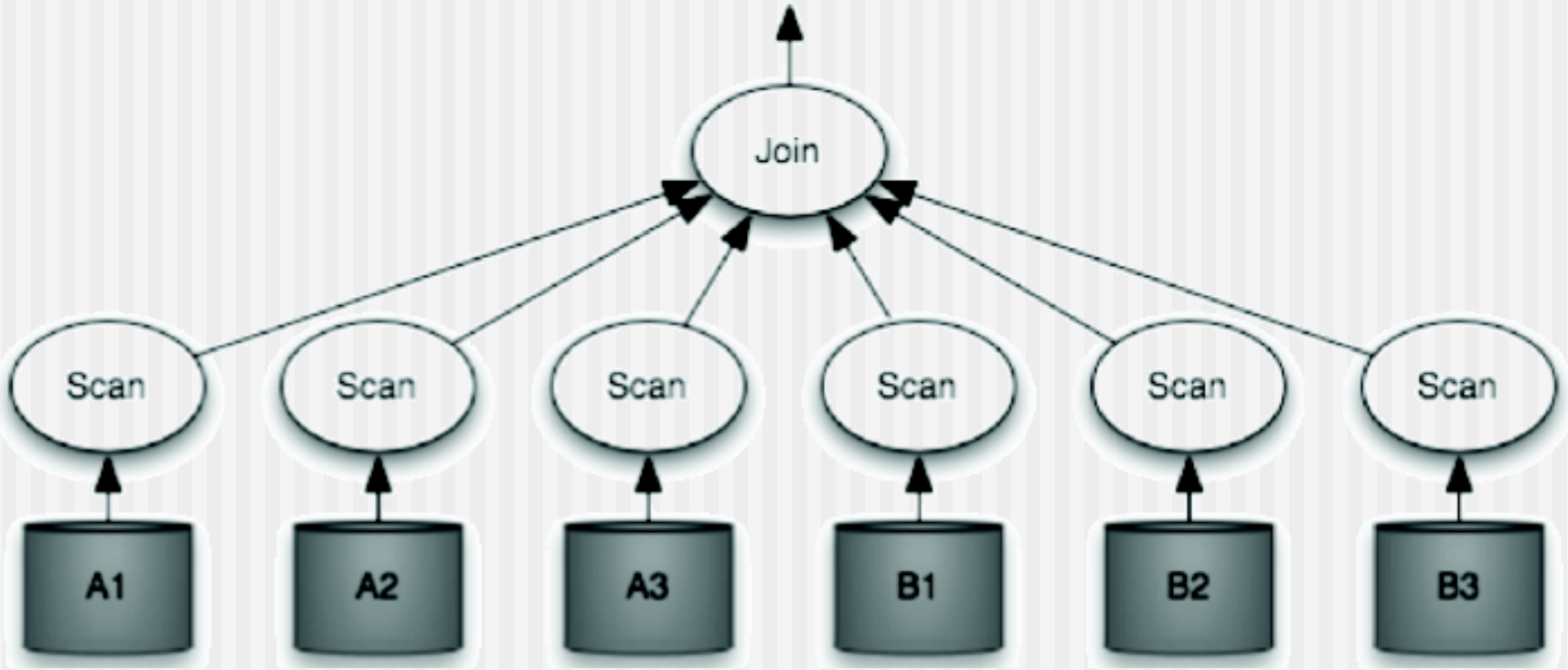
Intra-operator parallelism (cont)

- Suppose we have two range-partitioned data sets, A and B, partitioned as follows
  - A1, B1 = A-H tuples
  - A2, B2 = I-Q tuples
  - A3, B3 = R-Z tuples

- Suppose we want to join relation A and B. How can we do this?

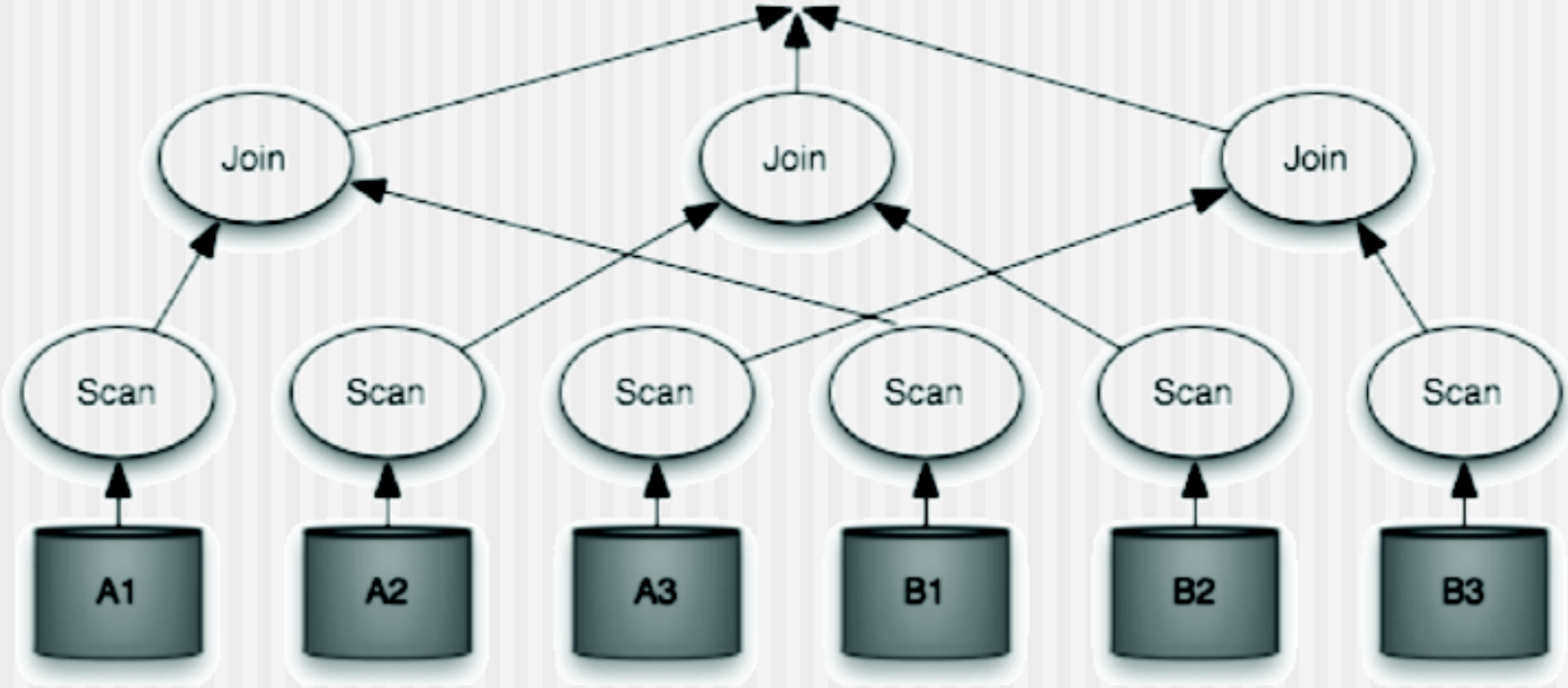
Intra-operator parallelism (cont)

Option 1: Start 6 scan operations and feed them, in partitioned order, into one central join operation



Intra-operator parallelism (cont)

Option 2: Start 6 scan operations and feed them into three join operations

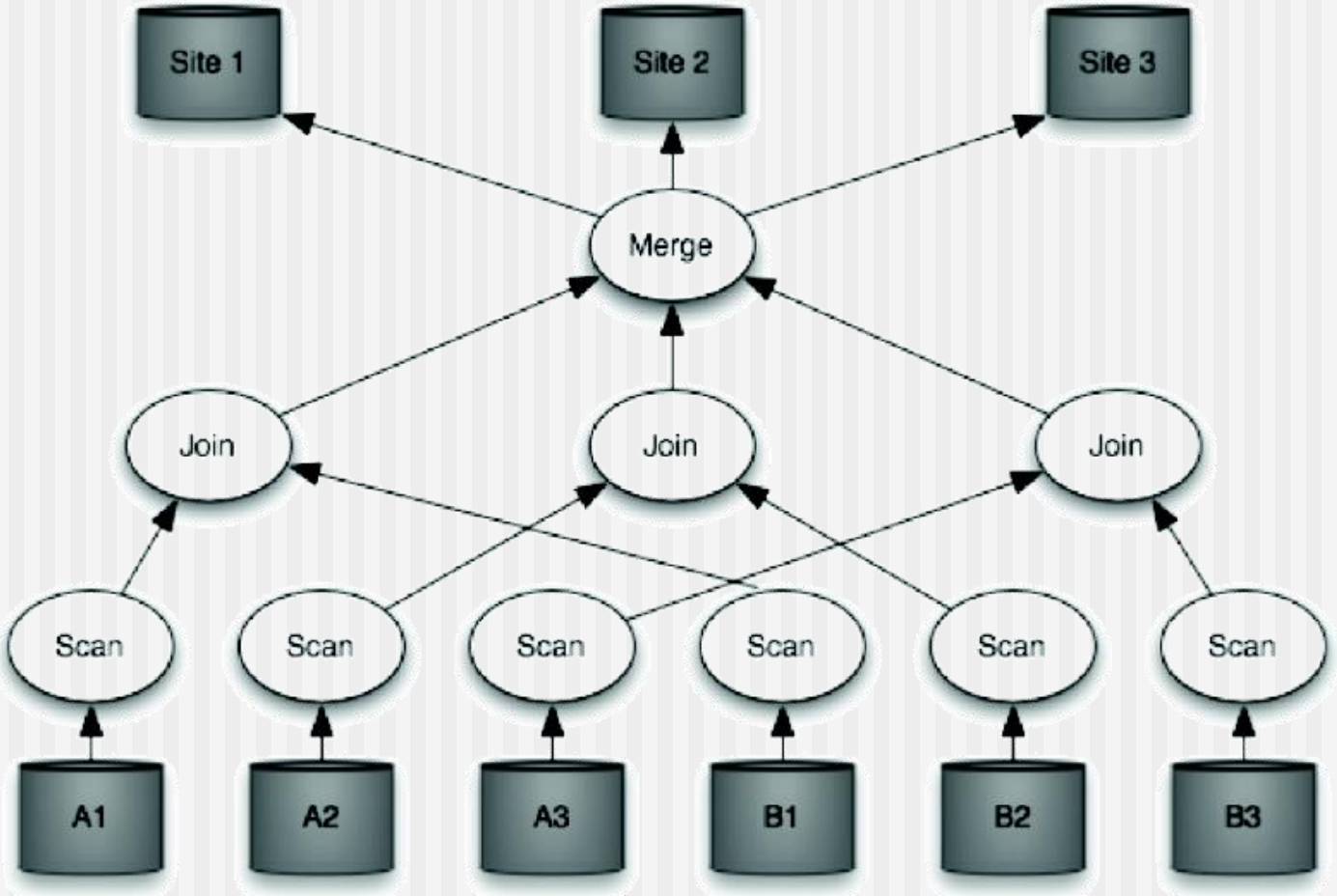


Intra-operator parallelism (cont)

- Option 1: Exploits partitioned data but only partial parallel execution
  - I.e.: Scan operations are done in parallel, but one central join is a bottleneck
- Option 2: Exploits partitioned data and full parallel execution
  - I.e.: Scan and join operations are done in parallel
- What if we were joining this data and replicating the result table across several sites?
  - Can we further leverage parallel processing?

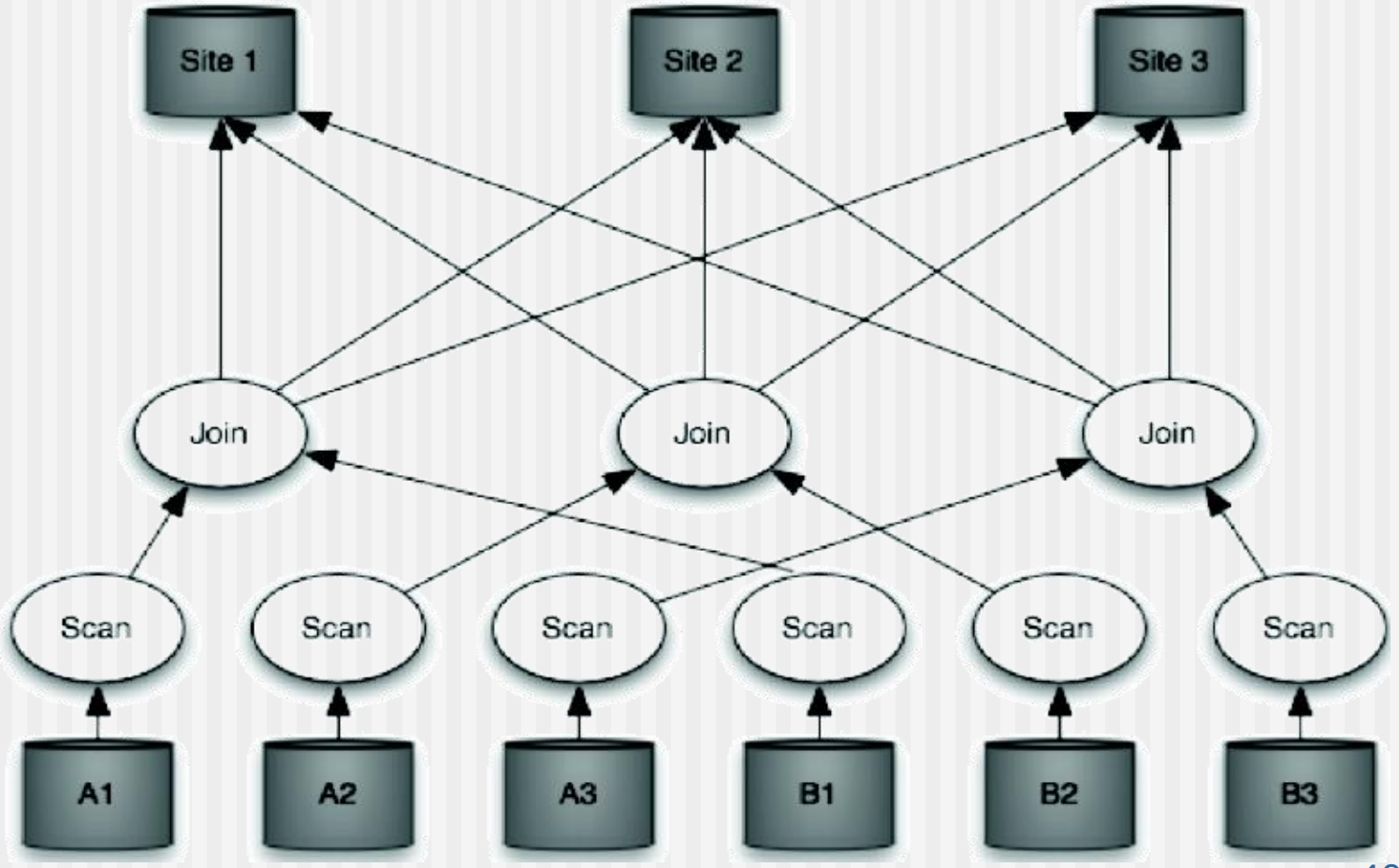
Intra-operator parallelism (cont)

Option 1: After the join operation, merge the data into one table at one site and transmit the data to the other replication sites



Intra-operator parallelism (cont)

Option 2: After the join operation, split he output and send each tuple to each replication site



Intra-operator parallelism (cont)

- Option 1: Exploits parallel processing but we have a bottleneck at the merge site
  - Could also have high network overhead to transfer all the tuples to one site, then transfer them again to the replication sites
- Option 2: Exploits parallel processing and splits output
  - Each site is given each tuple as it is produced

Intra-operator parallelism (cont)

- If each process runs on an independent processor with an independent disk, we will see very little interference
- What happens if a producing operator gets too far ahead of consuming operator? (i.e.: the scan works faster than the join)
  - Operators make use of independent buffers and flow control mechanisms to make sure that data sites aren't overwhelmed by each other
- Obviously, some operations will be better suited to parallel operation than others

Grosch's Law

- Grosch's law (1960) stipulates that more expensive computers are better than less expensive computers
- Mainframes cost:
  - $25 000/MIPS (Million Instructions Per Second)
  - $1 000/MB RAM
- Now, commodity parts cost:
  - $250/MIPS
  - $100/MB RAM (remember those days?!)

$\Rightarrow$ Grosch's law has been broken:

- Combining several hundred or thousand of these shared nothing systems can create a much more powerful machine than even the most expensive mainframe!

Future Research

- *Parallel Query Optimization*: take network traffic and processor load balancing into account.
- *Application Program Parallelism*: programs need to be written to take advantage of parallel operators
- *Physical Database Design*: design tools must be created to help database administrators decide on table indexing and data partitioning schemes to be used
- *Data Reorganization Utilities*: must be written to take into account parallelism and the factors that affect parallel performance

Conclusions

- Database systems want cheap, fast hardware systems
  - This means commodity parts, not custom built systems
- A shared nothing architecture has been shown to be ideal at performing parallel operations cheaply and quickly
  - They experience good speedup and scaleup