Parallel Database Systems

Parallel DBMS

- Uniprocessor technology has reached its limit
  - Difficult to build machines powerful enough to meet the CPU and I/O demands of DBMS serving large number of users
  - At 10 MB/s, 1.2 days to scan 1 TB of data
  - With 1000 nodes, it takes only 1.5 minutes to scan!
- PDBS – a DBMS implemented on a multiprocessor
- Attempts to achieve high performance through parallelism

PDBS vs Distributed DBS

DDBS

- Geographically distributed
- Small number of sites
- Sites are autonomous computers
  - Do not share memory, disks
  - Run under different OS and DBMS

PDBS

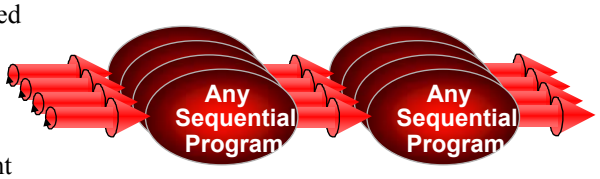
- Processors are tightly coupled using a fast interconnection network
- Higher degree of parallelism
- Processors are not autonomous computers
  - Share memory, disks
  - Controlled by single OS and DBMS

Types of Parallelism

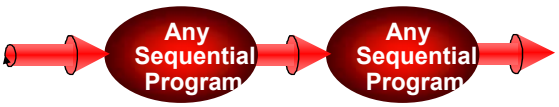
- Intra-Query Parallelism
  - Intra-operator parallelism
    - Multiple nodes working to compute a given operation (e.g., sort, join)
  - Inter-operator parallelism
    - Each operator may run concurrently on a different site
      - Partitioned vs pipelined parallelism
- Inter-Query Parallelism
  - Nodes are divided across different queries

Types of Parallelism

- Partitioned parallelism
  - Input data is partitioned among multiple processors and memories
  - Operator can be split into many independent operators each working on a part of the data

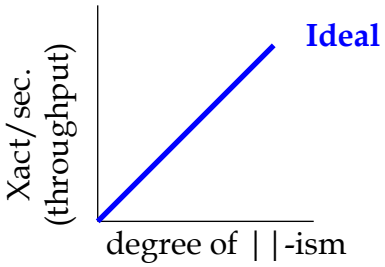


- Pipelined parallelism
  - Output of one operator is consumed as input of another operator



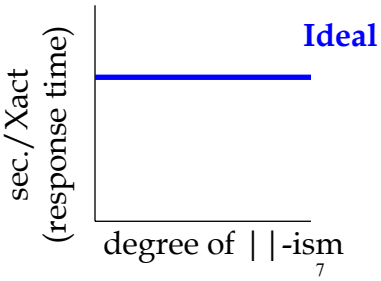
Goals and Metrics

- Speedup
  - More resources means proportionally less time for given amount of data
  - Elapsed time on a single processor/elapsed time on N processors
  - Problem size is constant, and grows the system
  - Linear speedup
    - Twice as much hardware can perform the task in half the elapsed time



Goals and Metrics

- Scaleup
  - If resources increased in proportion to increase in data size, time is constant.
  - Elapsed time of 1 processor on small problem/elapsed time of k processors on problem with k times data size
  - Measures ability of system to grow both the system and the problem
  - Linear scaleup
    - twice as much hardware can perform twice as large a task in the same elapsed time



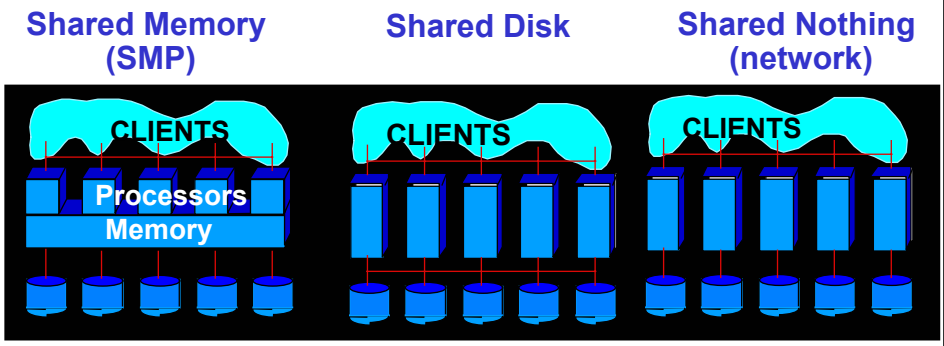
Barriers

- Startup
  - Time needed to start a parallel operation (e.g., creating process, opening files, etc)
  - Can dominate actual execution time with 1000s of processors
- Interference
  - Slowdown each process imposes on others when accessing shared resources
- Skew
  - Service time of a job is the service time of the slowest step of the job

Architecture

- Ideal machine: infinitely fast CPU with an infinite memory with infinite bandwidth
- Challenge – build such a machine out of large number of processors of finite speed
- Simple taxonomy for the spectrum of designs
  - Shared-memory (shared-everything)
  - Shared-nothing
  - Shared-disk

Architecture Issue: Shared What?



Easy to program
Difficult to scaleup

Sequent, SGI, Sun

VMScluster, Sysplex

Hard to program
Easy to scaleup

Tandem, Teradata, SP2

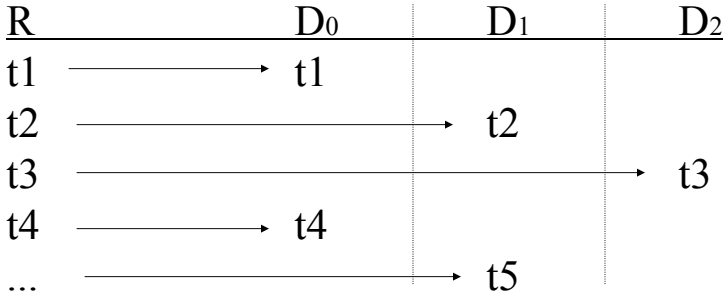
Key Techniques to Parallelism

- Parallelism is an unanticipated benefit of the relational model
- Relational query
  - Relational operators applied to data
- 3 three techniques
  - Data partitioning of relations across multiple disks
  - Pipelining of tuples between operators
  - Partitioned execution of operators across nodes

Data Partitioning

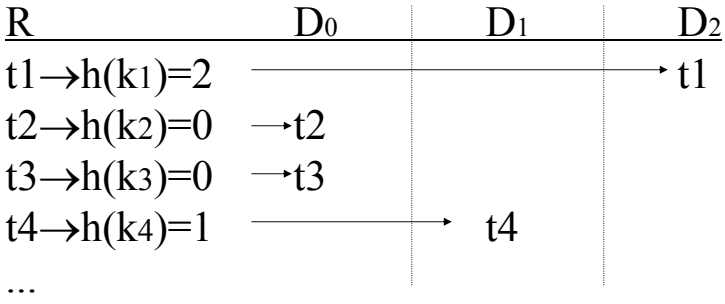
- Partitioning a relation involves distributing its tuples across several disks
- Provides high I/O bandwidth without any specialized hardware
- Three basic (horizontal) partitioning strategies:
  - Round-robin
  - Hash
  - Range

Round robin



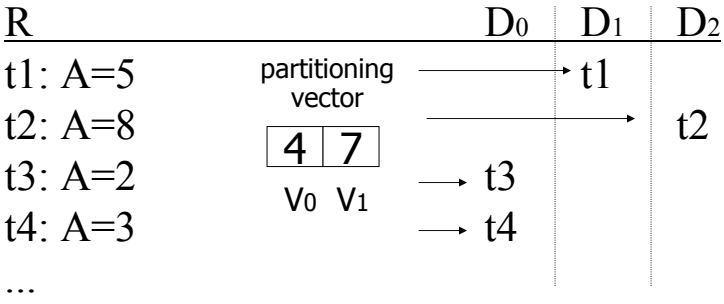
- Evenly distributes data
- Good for scanning full relation
- Not good for *point* or *range* queries

Hash partitioning



- Good for point queries on key; also for *equi-joins*
- Not good for range queries; point queries not on key
- If hash function good, even distribution

Range partitioning



- Good for some range queries on A
- Need to select good vector: else unbalance
  → data skew
  → execution skew

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 speedup and scaleup!

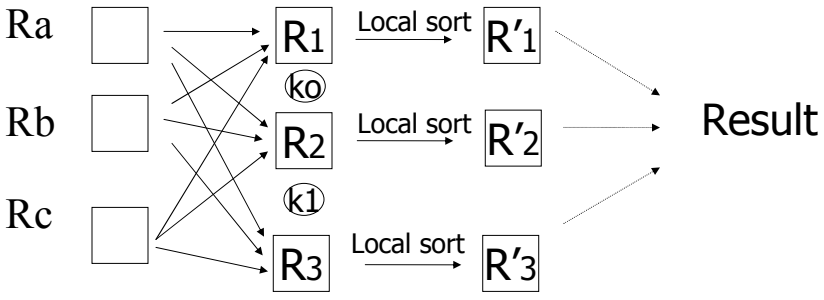
Parallel Sort

Input: R(K, A, ...) partitioned on non-sorting attribute A; Sort on K

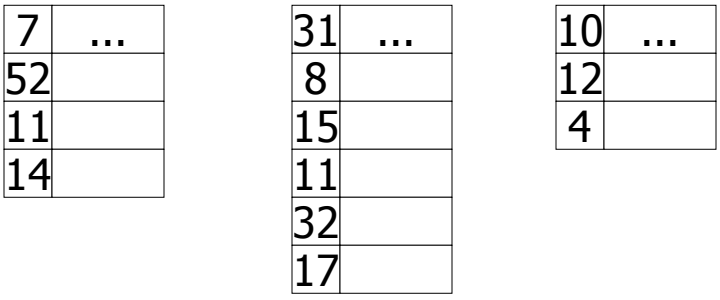
Output: Sorted R partitions on K

Range partitioning sort

- Algorithm:
  (a) Range partition on K
  (b) Local sort



Selecting a good partition vector



R_a R_b R_c

Example

- Each site sends to coordinator:
  - Min sort key
  - Max sort key
  - Number of tuples
- Coordinator computes vector and distributes to sites

  (also decides # of sites for local sorts)

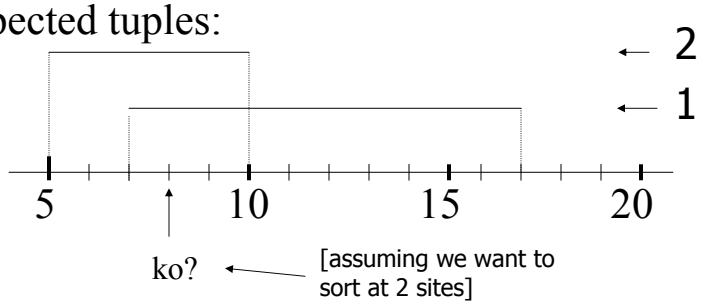
- Sample scenario:

Coordinator receives:

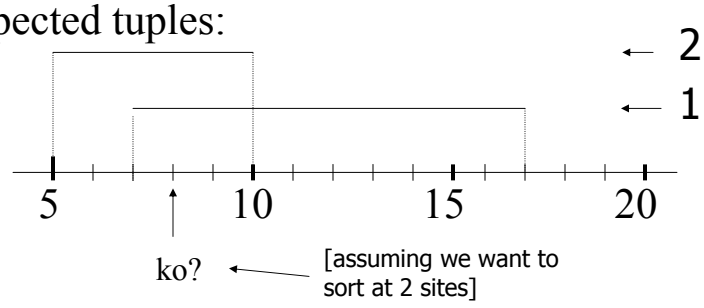
S_A: Min=5 Max=10 # = 10 tuples

S_B: Min=7 Max=17 # = 10 tuples

Expected tuples:



Expected tuples:



Expected tuples = $\frac{\text{Total tuples}}{2}$
__with key < ko

$2(ko - 5) + (ko - 7) = 20/2$

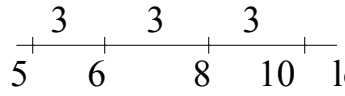
$3ko = 10 + 10 + 7 = 27$

$ko = 9$

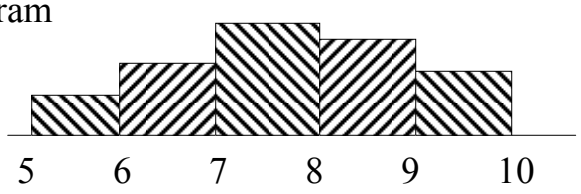
Variations

- Send more info to coordinator
  - Partition vector for local site

Eg. Sa:



- Histogram



➳More than one round

E.g.: (1) Sites send range and # tuples

(2) Coordinator returns "preliminary" vector Vo

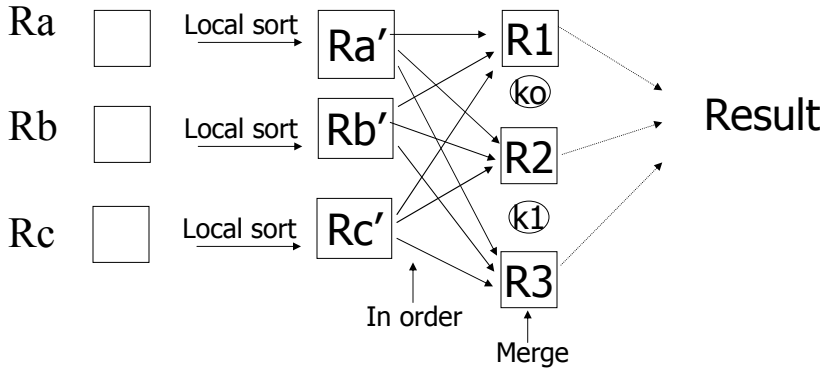
(3) Sites tell coordinator how many tuples in each Vo range

(4) Coordinator computes final vector Vf

- Can you think of any other variation?
- Can you think of a distributed scheme (no coordinator)?

Parallel external sort-merge

- Same as range-partition sort, except sort first



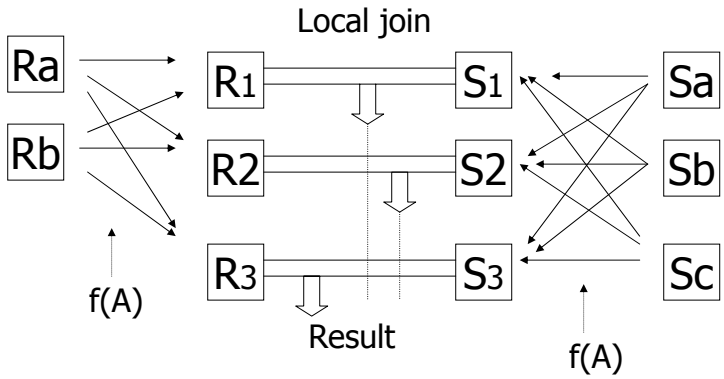
Parallel Join

Input: Relations R, S

Output: R ⋈ S

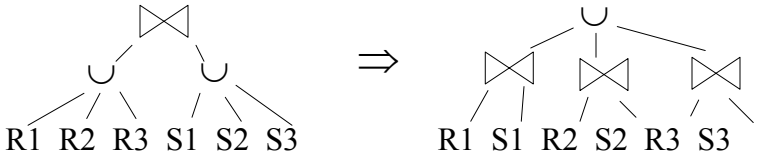
Result partitioned across sites

Partitioned Join (Equi-join)



Notes:

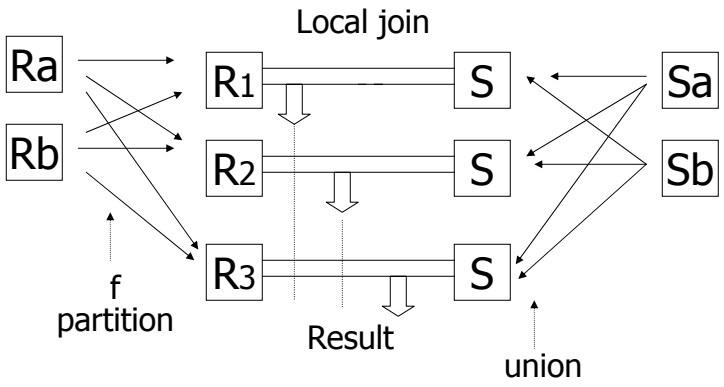
- Same partition function f is used for both R and S (applied to join attribute)
- f can be range or hash partitioning
- Local join can be of any type
  (use any CS3223 optimization)
- We already know why part-join works:



Even more notes:

- Selecting good partition function f very important:
  - Number of partitions
  - Hash function
  - Partition vector
- Good partition vector
  - Goal: $|R_i| + |S_i|$ the same
  - Can use coordinator to select

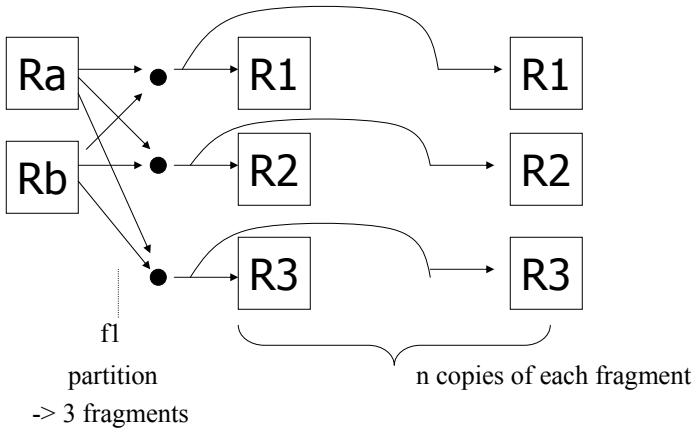
Asymmetric fragment + replicate join



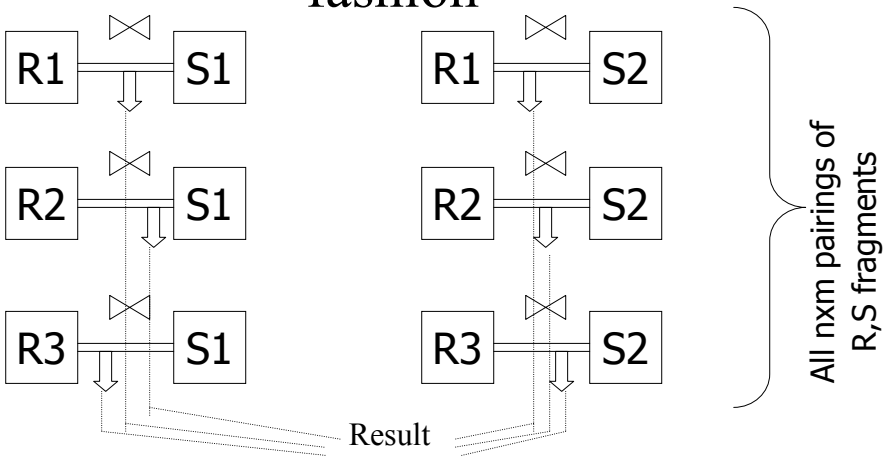
Notes:

- Can use any partition function f for R
  (even round robin)
- Can do any join — not just equi-join
  e.g.: $R \bowtie_{R.A < S.B} S$

General fragment and replicate join



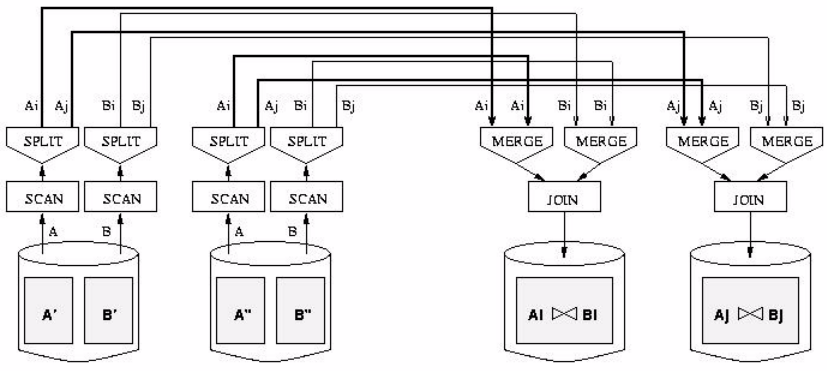
☞ S is partitioned in similar fashion



Notes:

- Asymmetric F+R join is special case of general F+R
- Asymmetric F+R may be good if S small
- Works for non-equi-joins

Dataflow Network for || Join

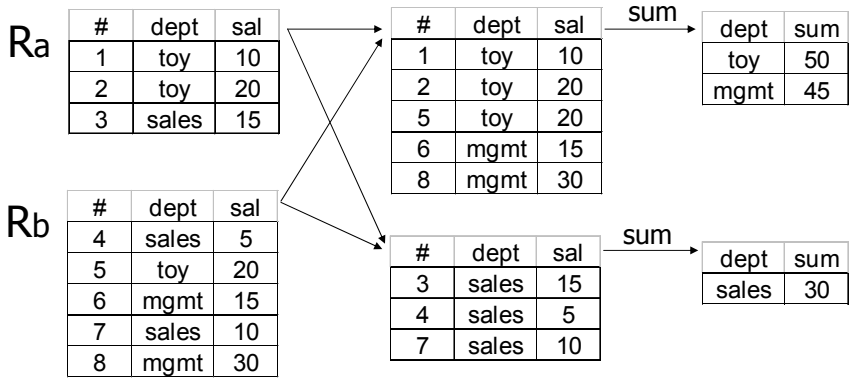


- Good use of split/merge makes it easier to build parallel versions of sequential join code.

Other parallel operations

- Duplicate elimination
  - Sort first (in parallel)
    then eliminate duplicates in result
  - Partition tuples (range or hash)
    and eliminate locally
- Aggregates
  - Partition by grouping attributes;
    compute aggregate locally

Example:



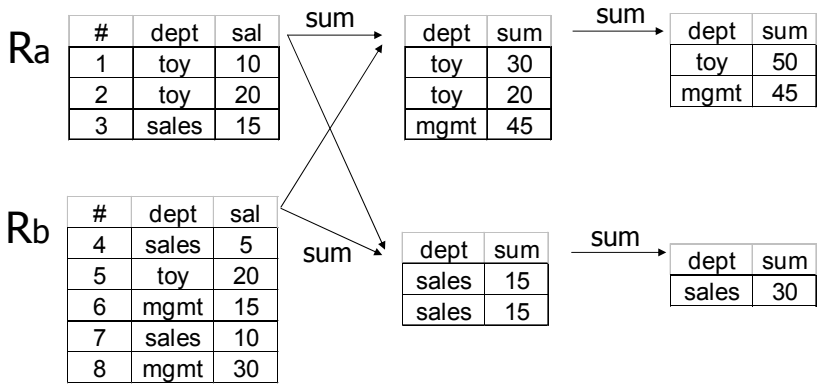
- sum (sal) group by dept

Enhancements for aggregates

- Perform aggregate during partition
  to reduce data transmitted
- Does not work for all aggregate
  functions...

  Which ones?

Example:



- sum (sal) group by dept

Data Skew

- Attribute value skew
  - Some attribute value appear more often than others
- Tuple placement skew
  - Initial distribution of record varies between partitions
- Selectivity skew
  - Selectivity of selection predicates varies between nodes
- Redistribution skew
  - Mismatch between distribution of join key values in a relation and the distribution expected by the hash function
- Join product skew
  - Join selectivity at each node differs

Effectiveness of Parallel Algo

- Load imbalance
- Performance determined by node that completes last
- Need some form of load-balancing mechanism

Taxonomy of Join Algorithms

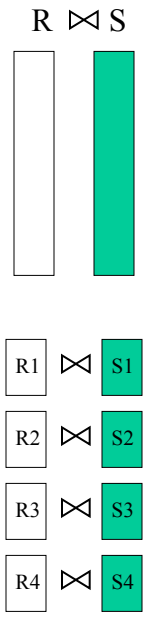
- Based on existing solutions
- Three logical phases
  - Task generation
  - Task allocation/load-balancing
  - Task execution

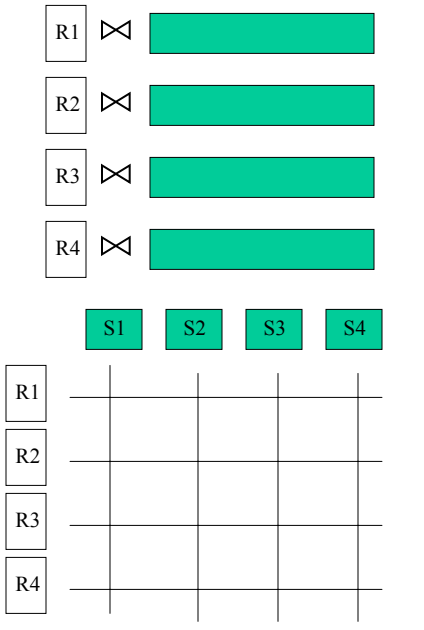
Task Generation

- A join is split into a set of tasks
  - Each task is essentially a join of sub-relations of the base relations
  - Union of the results of all tasks should be the same as the original join
- Four issues
  - Decomposition of base relations
  - Number of tasks to be generated
  - Formation of tasks after decomposition
  - What statistics to maintain

Decomposition

- Basic criterion
  - Union of results of subtasks = result of original join
- 3 methods
  - Full fragmentation (N tasks)
  - Fragment and Replicate (N tasks)
  - Full replication (N * M tasks)



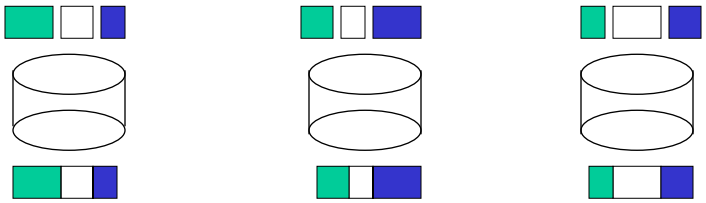


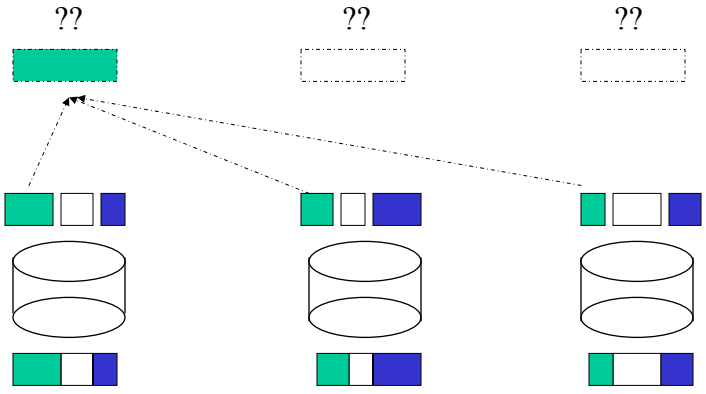
Number of tasks

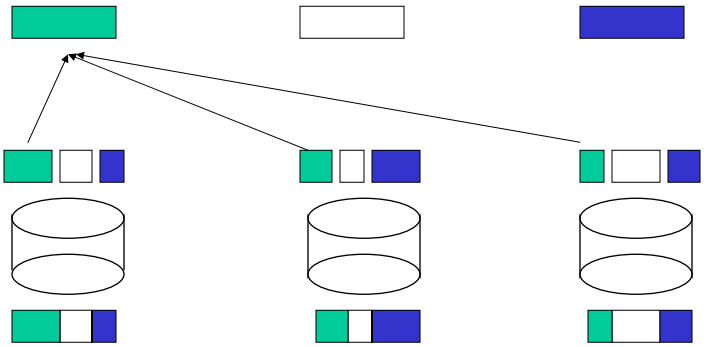
- What is the optimal number of tasks?
- Too many vs too few
- Given p processors
  - Generate p tasks – one per node
  - Generate more than p tasks, allocate to balance the load based on some criterion
    - Split larger tasks to smaller ones, then allocate to balance some criterion
    - Merge smaller ones, then allocate

Task Formation

- Whether the tasks are *physical* or *logical*
  - Physical tasks exists before execution
  - Logical tasks to be formed during execution
    - More useful for load balancing
    - Lower cost (transmission and I/O)

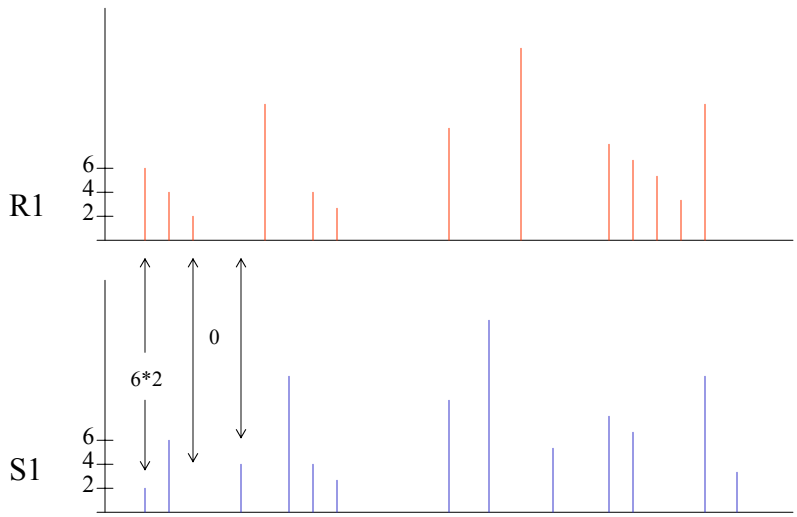


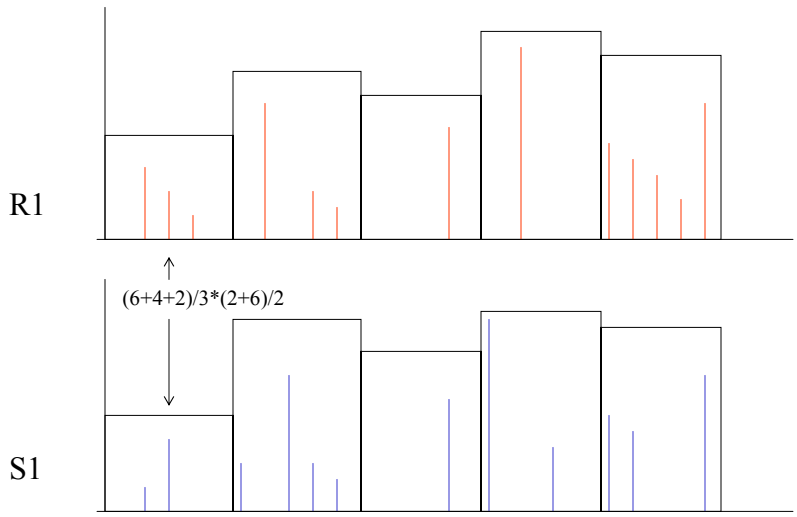




Statistics Maintained

- What statistics to maintain during task generation?
- Used by task allocation/load-balancing phase to determine how tasks should be allocated
- Perfect information
  - Relation size, distribution, result size
- Group records into equivalence classes
  - Maintain number of distinct classes
  - For each class, the number of distinct attributes and records from the relations
  - Can be used to estimate result size and execution time


R1
6
4
2
6*2
0
S1
6
4
2


R1
(6+4+2)/3*(2+6)/2
S1

Estimate Execution Time

- Use cost models
- E.g. for hash join, I/O cost = 3*|R| + 3*|S| where |R| and |S| are number of pages of relations R and S

Task Allocation

- At the end of task generation phase, we have a pool of tasks to be allocated to nodes for concurrent processing
- Two strategies
  - Allocate without load-balancing
    - When tasks are generated, they are sent to the appropriate nodes
    - Bad for skew data
  - With load-balancing
    - Tasks are allocated so that load across all nodes are approximately the same
      - Tasks are sorted according to some criterion
      - Tasks are allocated in descending order in a greedy manner (the next task to be allocated is sent to the node that is expected to finish first)

Task Allocation (Cont)

- Load-Balanced Schemes
  - Load metrics
  - Allocation strategies

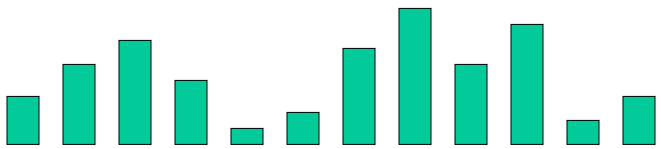
Task Allocation (Cont)

- Load metrics
  - What criterion?
    - Cardinality
    - Estimated results size
    - Estimated execution time

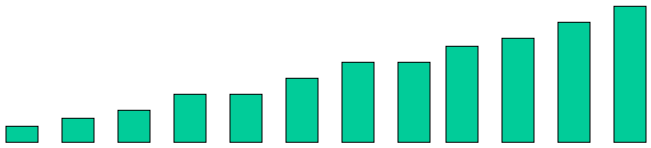
Task Allocation (Cont)

- Allocation strategy
  - How each task is to be allocated
    - Statically
      - Allocated prior to the execution of the tasks
      - Each node knows exactly how many and which tasks to process
      - Most widely used Adaptively
    - Adaptive
      - Demand-driven
      - Each node assigned and process a task a time
      - Acquire the next task only when the current tasks is completed

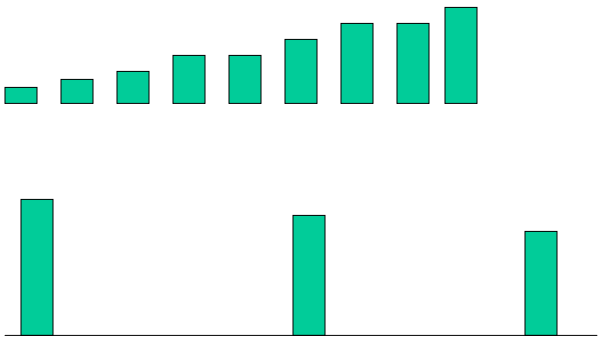
Static Task Allocation (12 tasks, 3 processors)



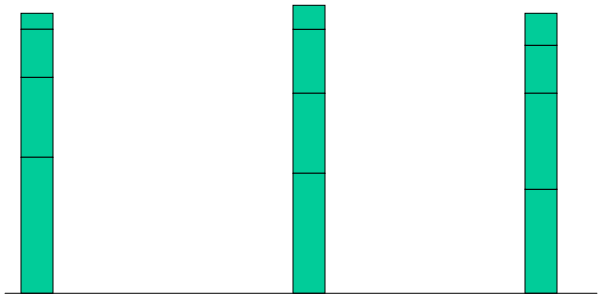
Static Task Allocation
(12 tasks, 3 processors)



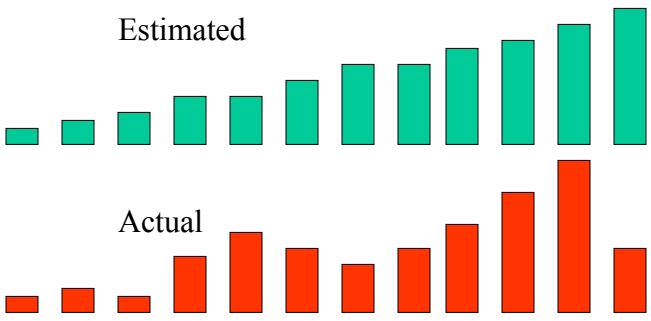
Static Task Allocation
(12 tasks, 3 processors)



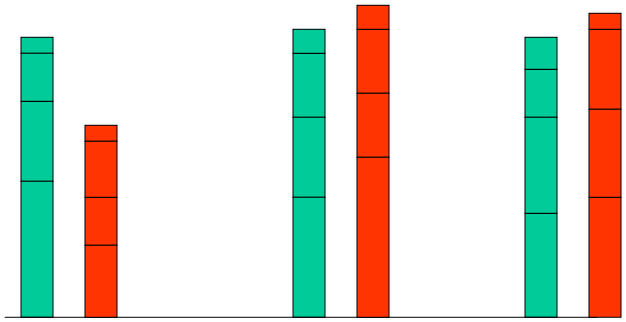
Static Task Allocation
(12 tasks, 3 processors)



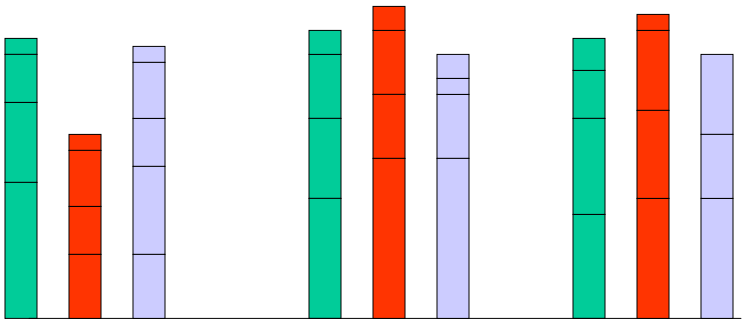
Static Task Allocation
(12 tasks, 3 processors)



Static Task Allocation
(12 tasks, 3 processors)



Adaptive Task Allocation
(12 tasks, 3 processors)



Task Execution

- Each node independently perform the join of tasks allocated to it
- Two concerns
  - Workload
    - No redistribution once execution begins
    - Redistribution if system is unbalanced
      - Need a task redistribution phase to move tasks (sub-tasks) from overloaded node to underloaded nodes
      - Must figure out who is the donor and who is idling
      - Work with static load balancing strategy
  - Local join methods
    - Nested-loops, sort-merge, hash join

Static Load-Balanced Algo

- Balance the load of nodes when tasks are allocated
- Once a task is initiated for execution, no migration of task (sub-task) from one node to another

An Example

- Consider join of R and S
- Parallel system has p nodes
- Assume that R and S are declustered across all nodes (using RR scheme)

Example (Cont)

- Generate k *physical* tasks (k > p)
  - Full fragmentation
- During partitioning, each node also collects statistics (size of partitions) for the partitions assigned to it
- A node designated as coordinator collects all the information
  - Estimate the result size, and execution time
  - Estimate the average completion time and inform all nodes about it

Example (Cont)

- Each node finds the smallest number of tasks such that the completion time of these tasks will not exceed the average completion time
- A node is overloaded if it still has tasks remaining
- Each node reports to coordinator the difference between the load and the average load, and the excess tasks
- Coordinator reallocates the excess tasks to underloaded nodes
- After redistribution of workload, each processor independently processes its tasks

Example: Static Scheme $R \bowtie_{R.A=T.A} T$

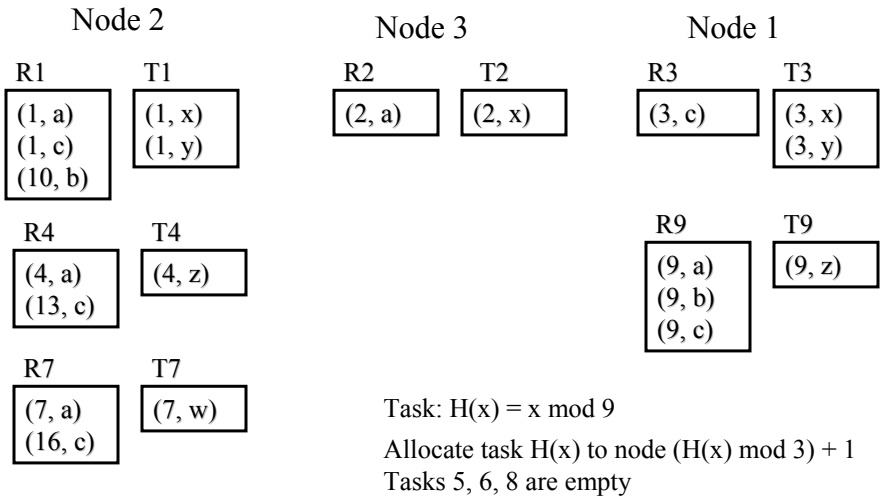
Initial (fragment) Placement (Round Robin)

Rf1	Tf1	Rf2	Tf2	Rf3	Tf3
(1, a) (3, c) (9, a) (13, c)	(1, x) (4, z) (9, z)	(1, c) (10, b) (9, b) (7, a)	(2, x) (3, y) (7, w)	(2, a) (4, a) (9, c) (16, c)	(1, y) (3, x)

Example (Cont)

$R \bowtie_{R.A=T.A} T$

Redistribute to form 9 physical tasks



Task: H(x) = x mod 9

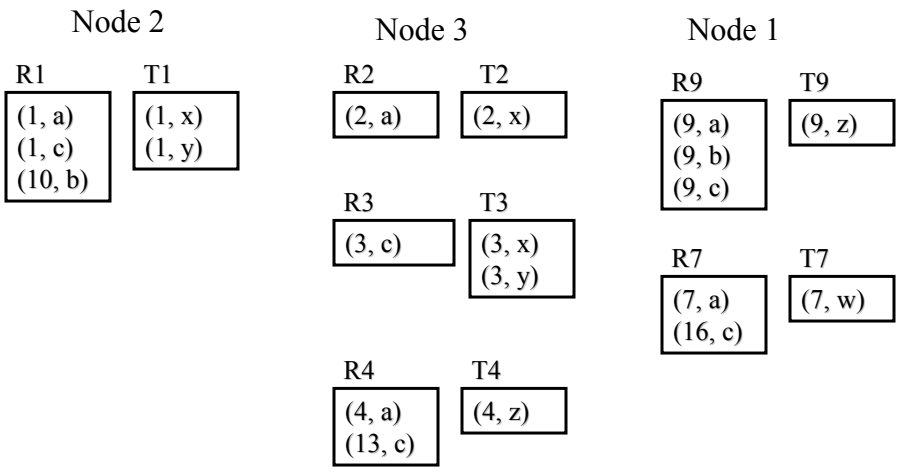
Allocate task H(x) to node (H(x) mod 3) + 1

Tasks 5, 6, 8 are empty

Example (Cont)

$R \bowtie_{R.A=T.A} T$

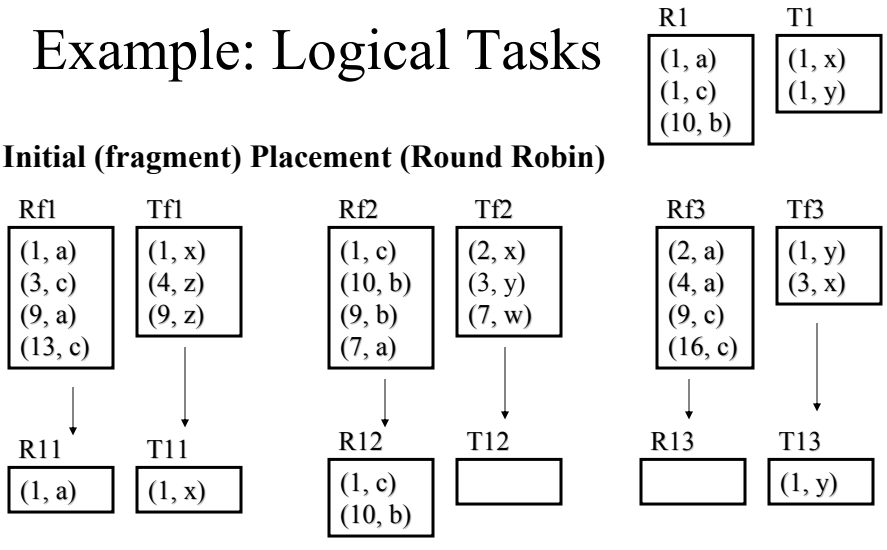
After load balancing: Assume balance result size



Static Techniques

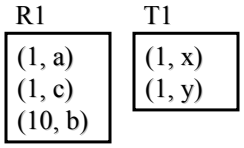
- Problems
  - Extra communication and I/O costs
  - Granularity of load-balancing is an entire tasks
  - Bad for highly skewed data
- Solutions
  - Form logical tasks
  - Split expensive tasks to smaller sub-tasks

Example: Logical Tasks

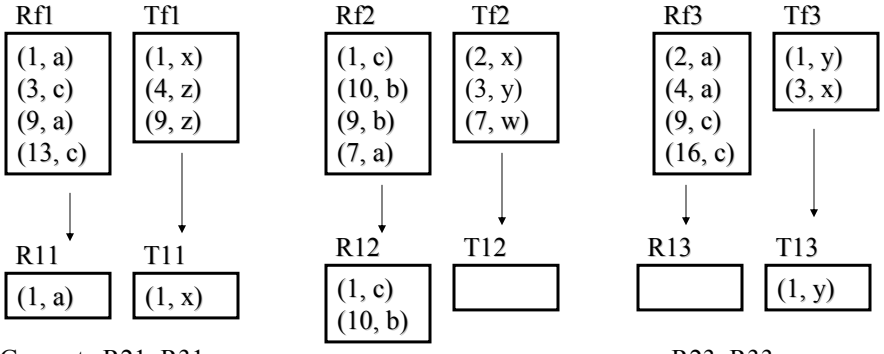


$R1 = R11 \cup R12 \cup R13$

Example: Logical Tasks



Initial (fragment) Placement (Round Robin)



Generate R21, R31, …
Generate T21, T31, …

Generate R22, R32, …
Generate T22, T32, …

R23, R33, …
T23, T33, …

Dynamic Load-Balancing

- Useful when estimation is wrong
- Redistribute tasks which may be partially executed from one node to another
- Task generation and allocation phases are the same as that for static load-balancing techniques
- Task execution phase is different

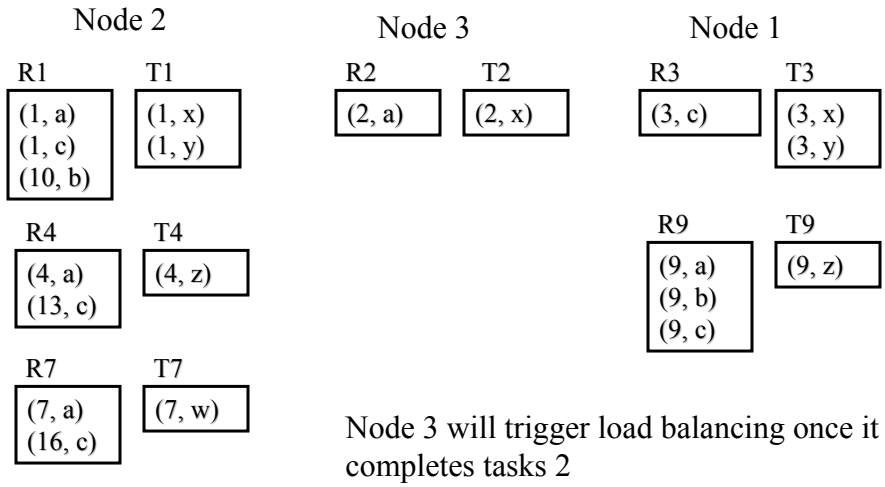
Dynamic Load-balancing (Cont)

- Task execution phase
  - Each node maintains additional information about the task that is being processed
    - Size of data remaining
    - Size of result generated so far
  - When a node finishes ALL the tasks allocated to it, it will *steal* from other nodes
    - The overloaded node and the amount of load to be transferred are then determined, and the transfer is realized by shipping data from donor to idle nodes

Example

$R \bowtie_{R.A=T.A} T$

After task allocation



Node 3 will trigger load balancing once it completes tasks 2

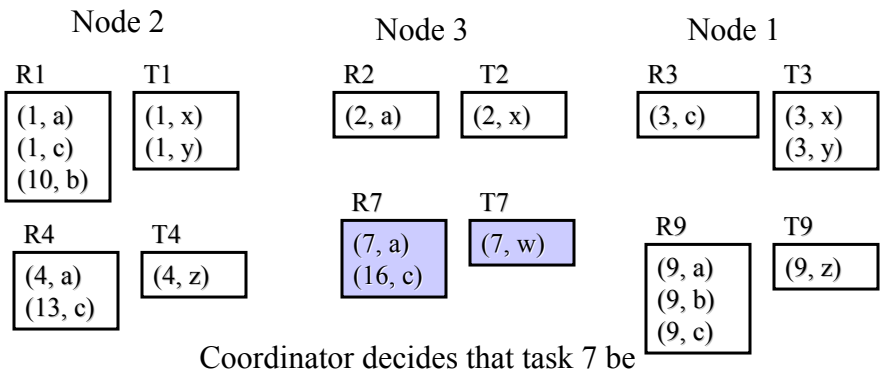
Dynamic Load-Balancing (Cont)

- Process of transferring load
  - Idle node sends a load-request message to coordinator to ask for more work
  - At coordinator, requests are queued in a FCFS basis. Coordinator broadcasts a load-info message to all nodes
  - Each node computes its current load and informs the coordinator
  - Coordinator determines the donor and sends a transfer-load message to donor, after which it proceeds to serve the next request
  - Donor determines the load to be transferred and sends the load to the idle node
  - Once the load is determined, donor can proceed to compute its new load for another request
- Process of transferring loads between a busy and an idle node is repeated until the minimum time has been achieved (or no more tasks to transmit)

Example

$R \bowtie_{R.A=T.A} T$

After task allocation



Coordinator decides that task 7 be transferred from Node 2 to Node 3
Next round, may decide to transfer task 4 from Node 2 to Node 2

Task Donor?

- Any busy node can be donor, but the one with the heaviest load preferred
- Use the estimated completion time of a node (ECT)
  - ECT = estimated time of unprocessed task + estimated completion time of current task

Amount?

- Heuristics to determine amount
  - Transfer unprocessed task first, if any
  - Amount of load transferred should be large enough to provide a gain in the completion time of the join operation
  - Completion time of donor should still be larger than that of idle node after the transfer

Current Trends

- NOWs
- P2P
- Active Disks
- Computational/Data Grid