SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (4/6)

- query optimization
  - query rewrite
  - cost-based
- data recovery
  - quick recap of transaction management
  - single instance recovery: ARIES
- transaction handling
  - scheduling of transaction operations
  - concurrency control
  - implementing isolation levels
- parallelization of data and queries
  - horizontal partitioning, vertical partitioning, replication
  - distributed query processing
  - multi-cores
  - map-reduce

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Parallelization of Data and Queries.

Agenda

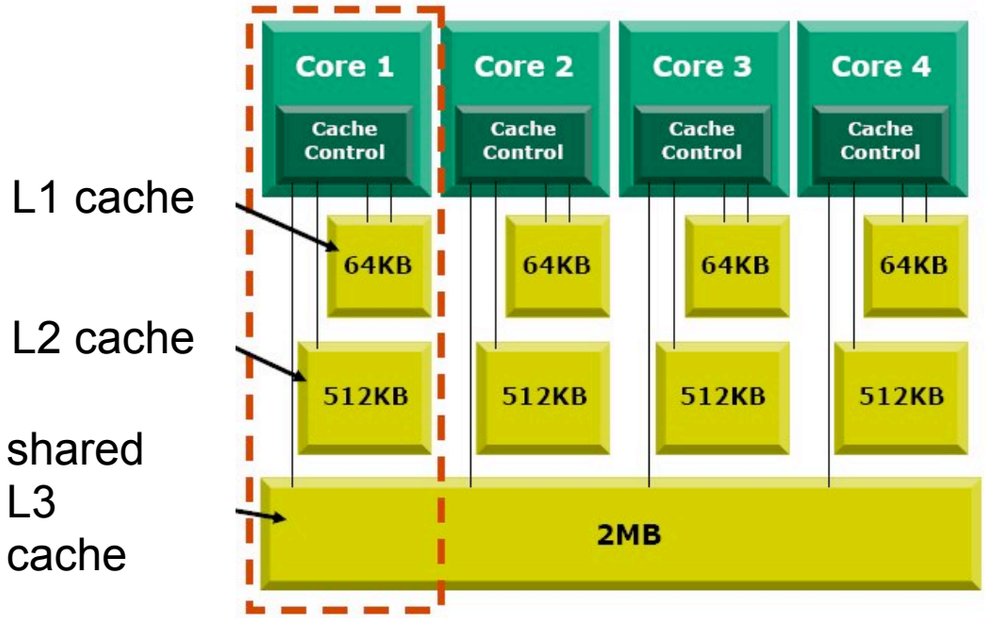
- Motivation
- Architectures
- Partitioning
- Replication
- Parallelization of Query Plans and Operators
- Join Processing (many variants)
- Distributed Query Optimization
- Cost Models
- Transaction Handling
- 2PC

Motivation

- why use only one CPU if we can use many?
- goals:
  - improve throughput (number of queries/updates) handled
  - improve individual queries (time to compute a single query)
  - improve system availability
  - improvement linear to the number of CPUs
- current trends:
  - hardware is getting cheaper
  - example: for a recent study we used
    - 2*Dual Core AMD Opteron 280
      (= 4 CPU cores on each machine running each at 2.4 GHz)
    - 6 GB of main memory
    - about 2K Euros per computing node

Motivation: Multi-Core Systems

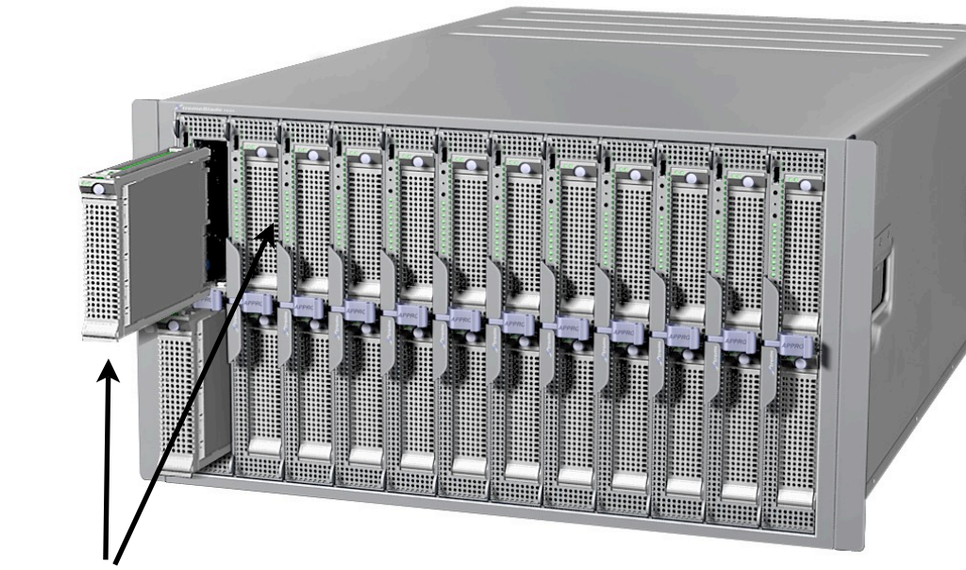
- CPU manufactors hit physical barriers:
  - clock rates may not be increased much further (heat problem)
  - chip structures hard to make smaller (physical barriers)
- solution: improve performance by packing multiple CPUs on the same chip
- current mainstream are dual-cores:
  - Intel Core 2 Duo (e.g., MacBook)
  - AMD Dual-Core Opteron
- high-end server market already sees quad-cores:
  - Intel Xeon (since end of 2006)
  - AMD Quad-Core (September 2007)



AMD Quad-core

Motivation: Multi-Node Systems

- used large number of independent machines
- either
  - standard (desktop) hardware (Google did this)
  - blade servers, i.e,
    - complete computer on a small blade
    - multiple blades in a rack
- relatively cheap
- if used well, may provide tremendous performance boosts
- For instance, assume
  - 16GB of main memory on each blade
  - each blade using at least a Quadcore CPU
  - =96 cores and 384 GB main memory



Motivation: GPU Data Processing

- example: NVIDIA GeForce 9800 GTX
- 128 cores on a single card!!
- cores optimized for graphcs processing
- however useful for other applications as well
- price: 190 Euros
  (as of Jan 1, 2009)

- so why not run the Database on the graphics card?
- several ongoing research projects
- programmable through CUDA:
  - http://en.wikipedia.org/wiki/CUDA
  - http://www.nvidia.com/object/cuda_sdks.html

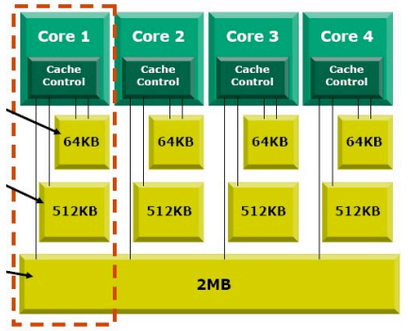
Motivation: GPU Data Processing

- graphic vendors have also started selling general purpose high performance computing chips
- not calling it GPUs anymore
- recent example: NVIDIA Tesla
- 240 cores per processor
- up to 4 processors in **a PC-sized system**
- **=> 960 cores**
- **3.732 Teraflops**
- compare: my MacBook Pro, Intel Core 2 Duo, 2.53 GHz, 6MB L2 Cache is at 20 Gigaflops: **by a factor 187 slower!!**
- **under 10,000 $**
- see video at http://www.youtube.com/nvidiatesla
- how to implement a DB on a 960 core machine?

Basic Techniques.

Architectures

- three main architectures
  1. shared disk
     - independent CPUs/caches/main memories
     - share same disk (or RAID)
  2. shared memory
     - independent CPUs/caches
     - share main memory
     - Example: Dual/Quad Cores (however may share slow caches)
     - many other specialized architectures (all very expensive)
  3. shared nothing
     - independent CPUs/caches/main memories and disks
     - Example: blade rack



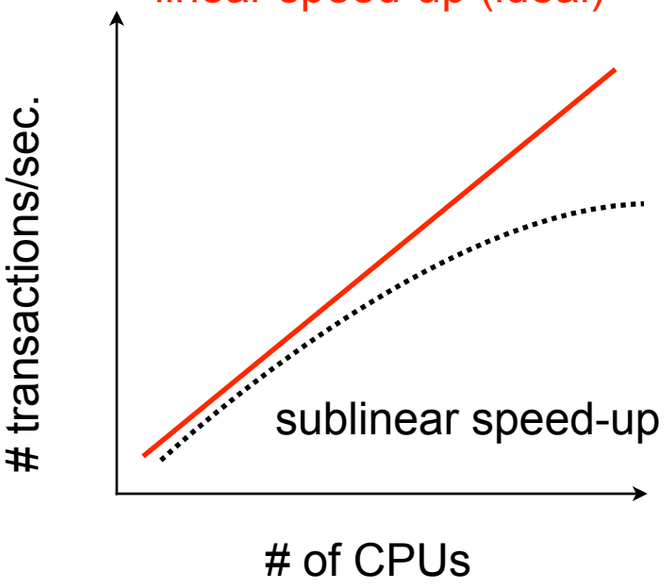
Discussion

- main problem with shared disk and shared memory: **interference**
- contention for memory access
- contention for disk access
- contention severely limits speedup of overall system

- motivated development of shared nothing approaches
  - harder to program (software reorganization)
  - linear **speed-up**: time taken to execute operations decreases in proportion with the number of CPUs and disks added
  - linear **scale-up**: performance is sustained if the number of CPUs and disks are increased in proportion to the amount of data

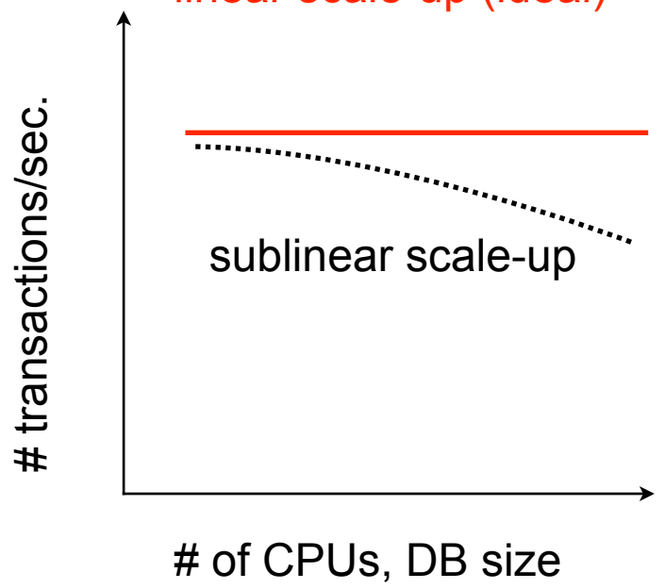
Speed-up vs. Scale-up

Speed-up



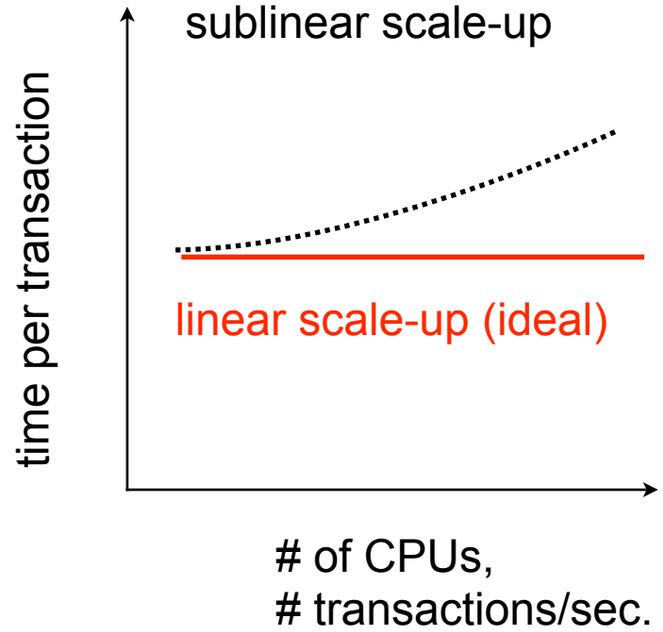
goal: increase throughput

Scale-up with DB size



goal: maintain throughput

Scale-up with #transactions/ sec



goal: maintain time per transaction

scale up: vertical scaling, i.e., adding resources to a single node in a system

scale out: horizontal scaling, i.e., adding nodes to a system

Data Partitioning

- Goal: partition data and place it on different machines in a way such that good speed-up and/or scale-up is achieved
- requires knowledge on query patterns
- techniques:
  - horizontal partitioning
  - vertical partitioning
  - vertical partioning with slight degree of replication
  - mix of horizontal and vertical partitioning
  - data replication
  - mix of replication and partitioning

Horizontal Partitioning

- core idea: distribute data evenly to multiple machines
- given a table R and N machines
- partition R into fragments $R_1,...,R_N$ of equal size
- place one fragment R_i on each machine
- How?
- see discussion for Grace Hash-Join
- really the same idea
  - use hash-function
  - or range partitioning
- also the same problems, e.g., range amenable to skew
- however, range partitioning may work if combined with sampling
- rebalancing may be a challenge

Horizontal Partitioning: Query Processing

- given table pieces $R_1,...,R_N$ of equal size
- given a query Q
- general pattern
  - send Q to all machines
  - execute Q independently on each machine
  - collect N results on a separate machine and merge them
- variant
  - if Q has a where clause referring to the partitioning attribute
  - send Q only to those machines that have data
  - For instance:
    - range partitioning on R.period, one partition per year
    - Q = select from ... where period=Q42003
    - query needs to be sent to one machine only

Vertical Partitioning

- core idea: cluster data that “belongs together“ to the same machine
- Given a table R with attributes att1, ..., att10.
- analyze query workload
- if query workload consists of queries referring to either
  - attributes att1, att3, att5
  - attributes att4, att7, att8
  - attributes att2, att6, att9, att10
- We may split R into three partitions
  - R1=(att1, att3, att5)
  - R2=(att4, att7, att8)
  - R3=(att2, att6, att9, att10)
- Effect: queries may be routed to only one of the machines.
- Compare discussion for DSM in Week 2, Slide 30

Vertical Partitioning with Partial Replication

- same as vertical partitioning, however, use slight degree of replication
- if query workload consists of queries referring to either
  - attributes att1, att2, att3, att5
  - attributes att4, att5, att7, att8
  - attributes att2, att6, att9, att10
- We may split R into three partitions
  - R1=(att1, att2, att3, att5)
  - R2=(att4, att5, att7, att8)
  - R3=(att2, att6, att9, att10)
- att2 and att5 are replicated to multiple partitinos
- increases some redundancy (be careful when updating)
- however, queries may be sent to one machine only

Full Data Replication

- Core idea: replicate data to multiple machines
- good when scale-up not achievable by partitioning
- example
  - huge number of incoming read-only queries
  - replicate DB to multiple machines
  - distribute querys-workload to multiple machines
- be careful when updating (redundancy)
- replications increases availability (if of one of the machines goes down, just fix routing)
- replicas may be synchronous or asynchronous (stale query results)
- replication for locality (caching)
- may be mixed with fragmentation (discussion somwehat similar to different RAID levels for hard disks)

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Parallelizing Query Processing.

Parallelization of Query Plans

- query optimizer has to compute plans that may be efficiently executed on a parallel architecture
- parallelization of individual operators
- important paradigm: split/merge
- parallelization of entire query plans

Split vs. Merge Paradigm

- remember the operator model (pull and push)?
- operators allow us to **stream** data from one operator to another
- we need two additional operators
  - **merge**-operator
    - merges multiple inputs into a single one
    - hides fragmented data
    - example: read multiple fragments of a relation and merge them back to the entire relation
  - **split**-operator
    - splits a stream into multiple streams
    - example: split a relation into multiple fragments
- split and merge should use multiple threads and buffering on inputs and output (alternative: separate exchange operator)
- parallel query plan will consist of standard operators plus merge and split

Parallelization of Individual Operators

- Intra-operator parallelism
- parallel implementation of a single operator (operator becomes a subplan)
- example: sorting (important for bulkloading indexes...)
- given fragments $R_1,...,R_N$ distributed to N nodes
- Algorithm 1
  - on each node i sort fragment R_i
  - send sorted runs to central node to merge (note: this is a bottleneck)
- Algorithm 2
  - on each node repartition data into N ranges (same partitioning function)
  - send data belonging to range i to node i
  - on each node sort data
    (may start sorting as soon as first data items arrive)
  - collect entire sorted relation by visiting nodes in order corresponding to ranges assigned to them

Parallelization of Individual Operators

- inside a single node with multiple CPUs we could also do something like this:
- given relation R and N processing cores
- Algorithm 3
  - partition data in-place into N partitions (using quicksort)
  - sort each partition by a separate thread
  - collect entire sorted relation by visiting nodes in order corresponding to ranges assigned to them
- many other parallel sorting algorithms exists

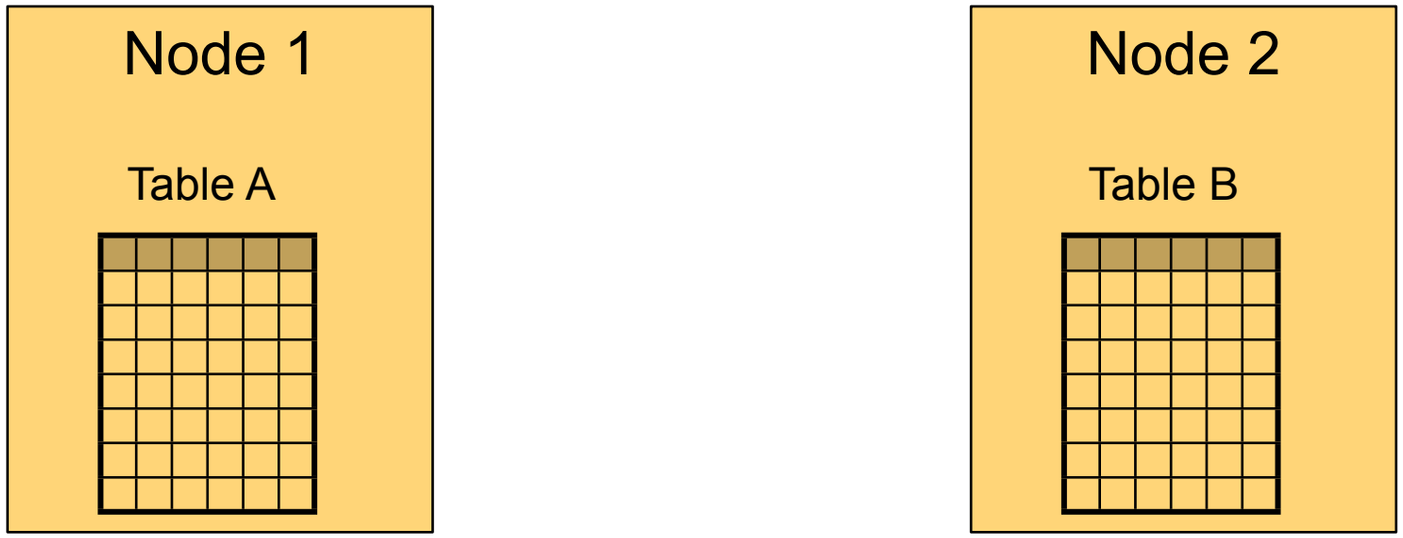
Parallel Join Processing

- parallel join as important as join on single CPU
- however, harder to optimize
- more fragile as distributed systems tend to be less stable
  e.g., varying network bandwidth

- parallel join strategy may have huge impact on overall query performance of a distributed system
- network bandwith may be the killer
- in the following we will explore several different options

Join Processing Scenario

- Scenario:



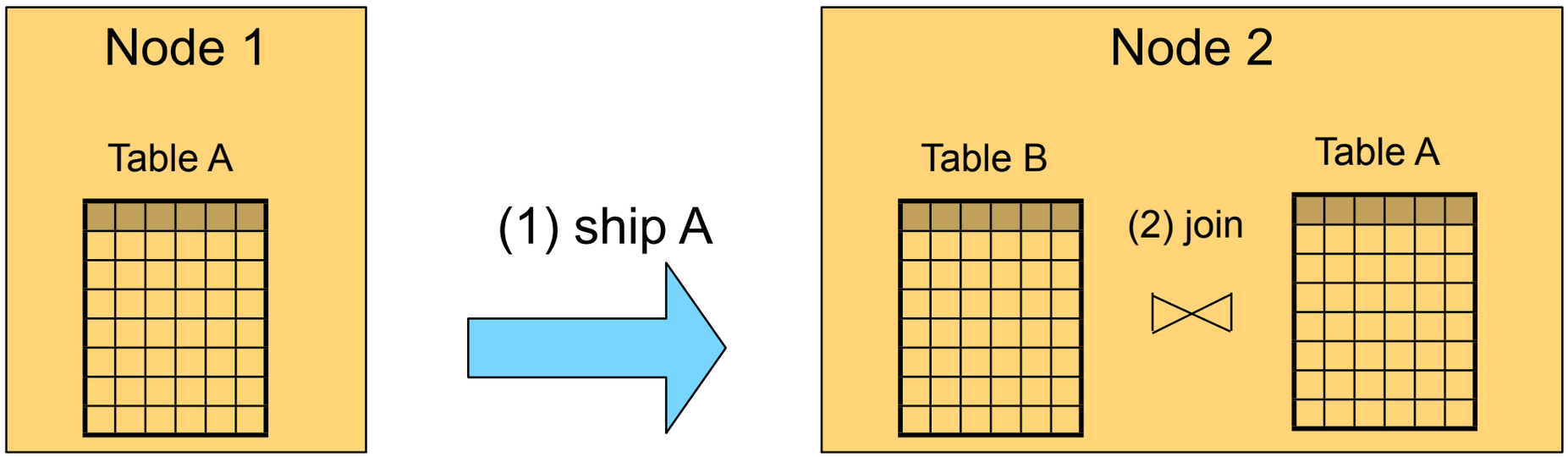
- How to compute A ⋈ B?
  - where A resides on Node 1
    and B resides on Node 2

Very naïve Approach: Ship Data Page-wise

- execute standard join operators on Node 1, e.g., index nested-loops join

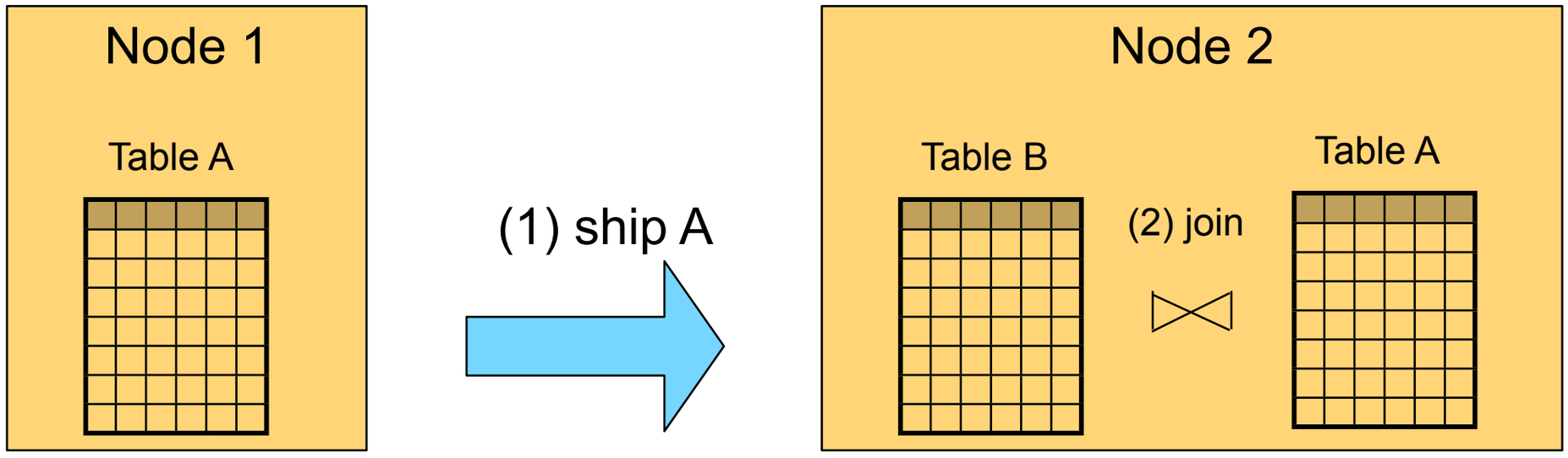
- whenever you fetch a page from Node 2, you cache that page on Node 1
- Problem: cost for this may be very high:
  - random access for pages of table B on Node 2
  - basically shipping entire table B to Node 1
- Not a good idea, only works if B is really small.

Naïve Approach: ship Table to one Side



- Ship one of the tables to the other node
- Example
  - Ship Table A to Node 2 (1)
  - Now both tables A and B are at the same node
  - join A and B locally (2)
- (or vice versa)

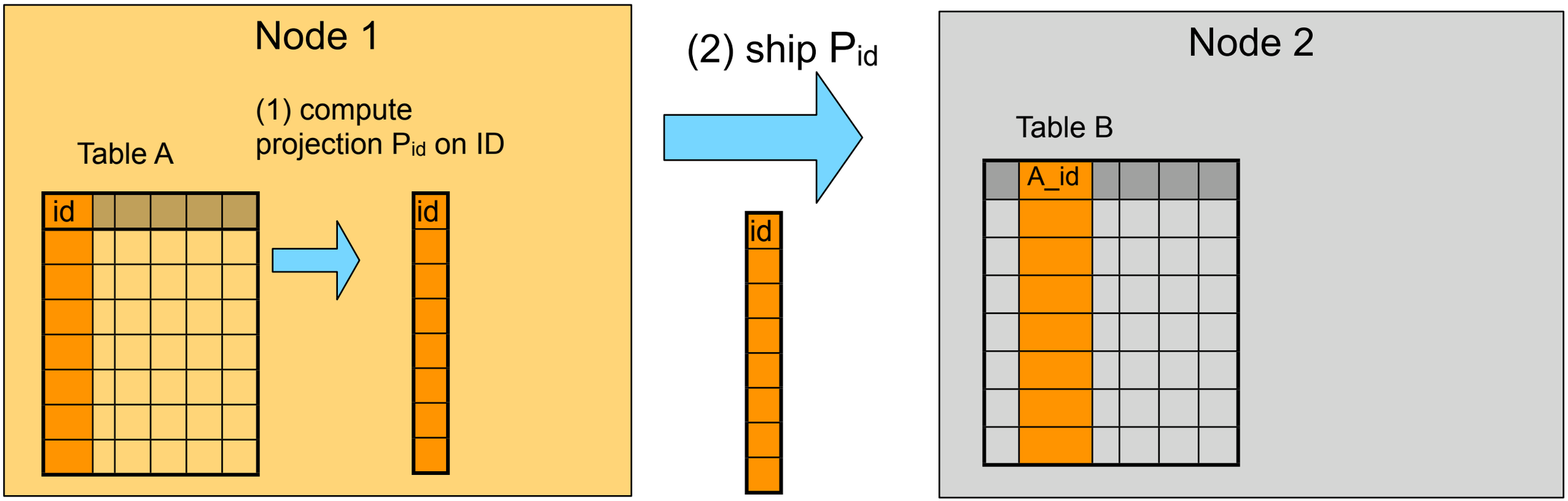
Discussion: Table Shipping



- Problems
  - high cost for shipping entire Table A
  - network cost roughly the same as I/O cost for a hard disk
  - thus: shipping A roughly equivalent to a full table scan of A on Node 1
- Optimizations
  - ship always smaller table to the other side
  - if query contains a selection on A, apply selection before sending A
  - Note: bigger table may become the smaller table (after selection)

Semijoin (1/3)

- Assume A ⋈ B with join predicate A.id==B.a_id

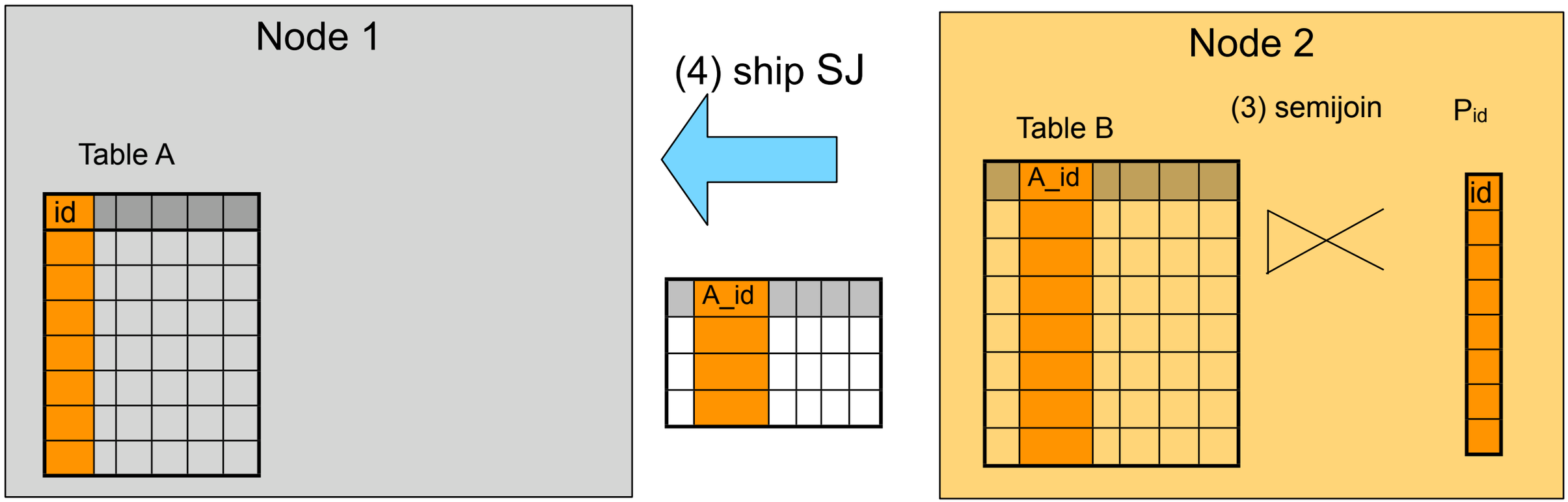


(1) on Node 1: compute the projection P_{id} of Table A on A.id

(2) send P_{id} from Node 1 to Node 2

Semijoin (2/3)

- Assume A ⋈ B with join predicate A.id==B.a_id

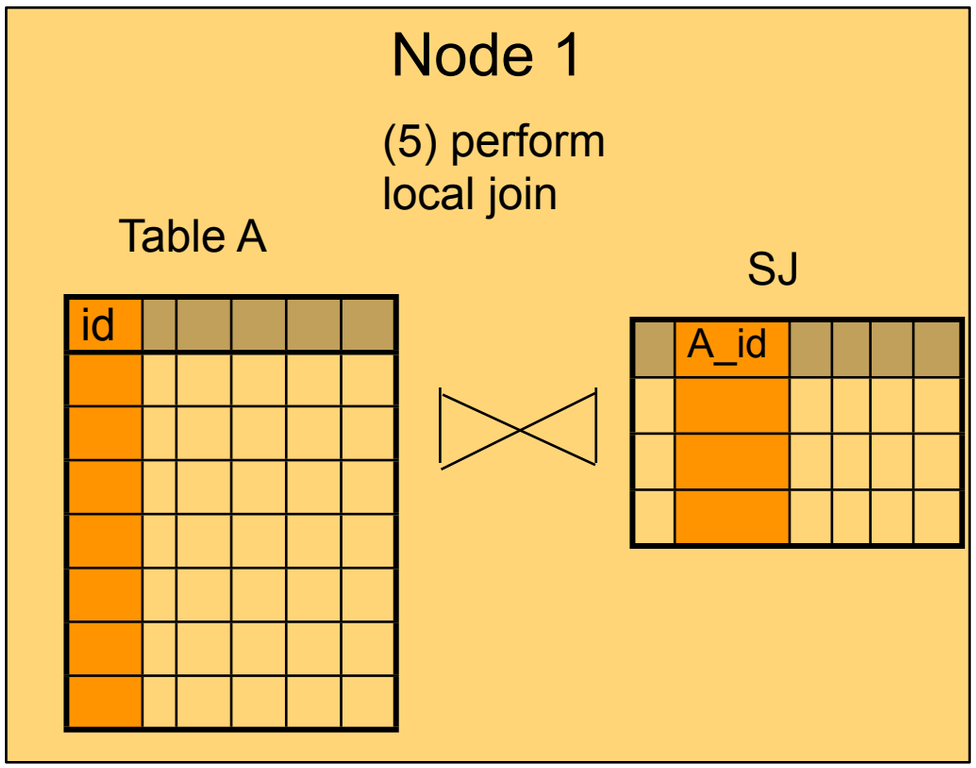


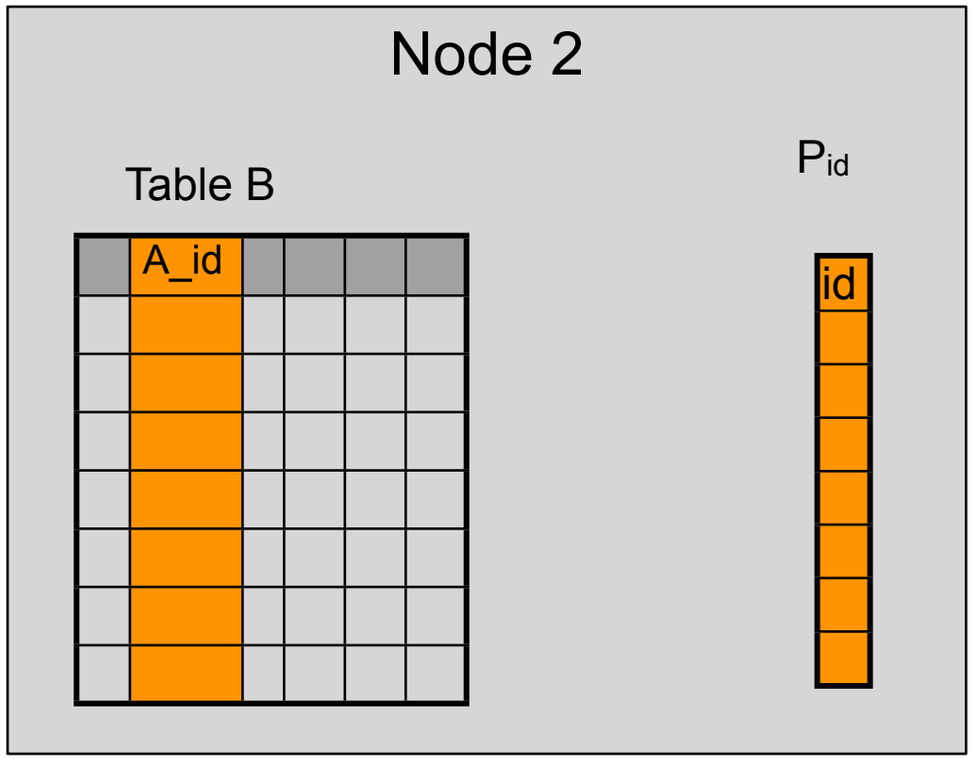
(3) on Node 2: perform the semijoin SJ : = B ⋉ P_{id}, i.e., SJ ⊆ B
(i.e, select all tuples of B that may join with tuples in table A)

(4) Send result of semijoin SJ from Node 2 to Node 1

Semijoin (3/3)

- Assume A ⋈ B with join predicate A.id==B.a_id





(5) Perform local join A ⋈ SJ at Node 1

Semijoin Selectivities

- Works well if selection of semijoins allows us to reduce size of table to be shipped, i.e. ll$\Pi_{ID}(A)$ll + ll B $\ltimes$ P_{id} ll < llBll, here ll . ll denotes the size in bytes
- For example
  - Assume all rows of Table B are needed anyway (none or few of the rows of B can be discarded)
  - Then this approach is more costly than shipping the entire Table B in the first place!
- Consequences:
  - selectivities will impact choice of the right method
  - need a cost-based optimizer to decide whether semijoin makes sense

Left to Right vs. Right to Left

- instead of performing the join from left to right, i.e.,

$$(\Pi_{ID}(A) \ltimes B) \bowtie A$$

  we could als perform it from right to left, i.e.,

$$(\Pi_{A\_ID}(B) \ltimes A) \bowtie B$$

- We should select the join having smaller network-I/O cost.
- Therefore, we need to find out whether

$$||\Pi_{ID}(A)|| + ||\ \Pi_{ID}(A) \ltimes B\ || < ||\Pi_{A\_ID}(B)|| + ||\ \Pi_{A\_ID}(B) \ltimes A\ ||?$$

- If yes: pick left to right, else right to left.
- Consequences:
  - need a cost-based optimizer to make this decision...

Bloomjoin

- core algorithm same as semijoin approach
- optimization lowering network cost (hopefully)
- **idea**: instead of shipping the projection on a key column, ship a compressed bit-vector
- the semijoin is performed using the compressed bit-vector
- use bloom filter technique, i.e, a bitmap having less bitmaps than there are keys
- Problem: superset of tuples will be sent (same problem as in bloom filters for other applications, i.e., differential files)
- net gain of this method very sensitive to choice of a good hash function

Repartitioning Joins

- suppose we have N computing nodes (machines)
- tables A and B initially distributed across N nodes
- Core idea:
  - partition A and B into N partitions $A_1,...,A_N$ and $B_1,...,B_N$
  - use same partitioning function for both tables (hash or range)
  - each node sends partitions A_i and B_i to node i
  - on each node i: perform local join $A_i \bowtie B_i$
- Note:
  - really the same idea as in grace hash join
  - same issues with possible skew for range partitioning
  - however: each partition pair gets assigned to one computing node
  - all local joins $A_i \bowtie B_i$ may be performed in parallel!

Discussion

- as local join methods we may use any join
- depending on how to distribute join results, we will have to join results from individual nodes by merge operations
- if range partitioning is used, the merged results may be obtained by visiting nodes in range order

Improved Repartitioning Join

- problem with repartitioning join:
  each local join is just using the locally available main memory
- core idea:
  - executed joins $A_i \bowtie B_i$ one after the other
  - each join is executed in parallel using **all** processors
- Effect:
  - each local join uses all available memory on all machines

Improved Repartitioning Join Algorithm

1. at each node
   - use a hash-function h1 to partition A and B into k partitions $A_1,...,A_k$ and $B_1,...,B_k$, Let A be the smaller relation.
   - choose k such that each partition A_i of A fits into the **combined main memory** of all N nodes
2. For i=1,..,k:
   - at **each** node in parallel:
     - First thread:
       For each tuple t in A_i:
           send tuple t to node h2(t)%N //i.e., use a second hash function!
     - Second thread:
       Collect all incoming tuples for partition A_i in a hash table (build input)
   - **After all A_i tuples have been distributed:**
   - First thread:
     For each tuple t in B_i:
         send tuple t to node h2(t)%N
   - Second thread:
     Join all incoming tuples for partition B_i to existing hash table of tuples from A_i (probe input)

Discussion

- second hash function h2 ensures that tuples are (hopefully) uniformly distributed across all N nodes
- reduces cost for individual joins and therefore overall join cost
- network usage better: network I/O distributed over the entire join time
- however: local I/O cost for partitioning at each node

Data Partitioning in the Presence of Joins

- Key idea
  - partition data in the first place such that most joins will be local
  - hash on key columns (either single or multiple column)
  - decluster A and B tables
- Example
  - A.id==B.a_id
  - hash(id) = id%N
  - partition into N groups $(A_0,B_0),\ldots,(A_N,B_N)$
    where A_i contains all elements of A for which hash(id) == i
    where B_i contains all elements of B for which hash(id) == i
- Effect
  - All joins become local!!
  - basically we are performing the repartitiong phase at indexing time

Example: Data Partitioning in the Presence of Joins

Table A

ID
1
2
3
4
5
6
7
8
9
10
11
12

A.id==B.a_id

Table B

a_ID
5
7
4
12
9
11
10
2

Partition

N=4

Node 1

Table A_0

ID
4
8
12

Table B_0

a_ID
4
12

Node 2

Table A_1

ID
1
5
9

Table B_1

a_ID
5
9

Node 3

Table A_2

ID
2
6
10

Table B_2

a_ID
2
10

Node 4

Table A_3

ID
3
7
11

Table B_3

a_ID
7
11

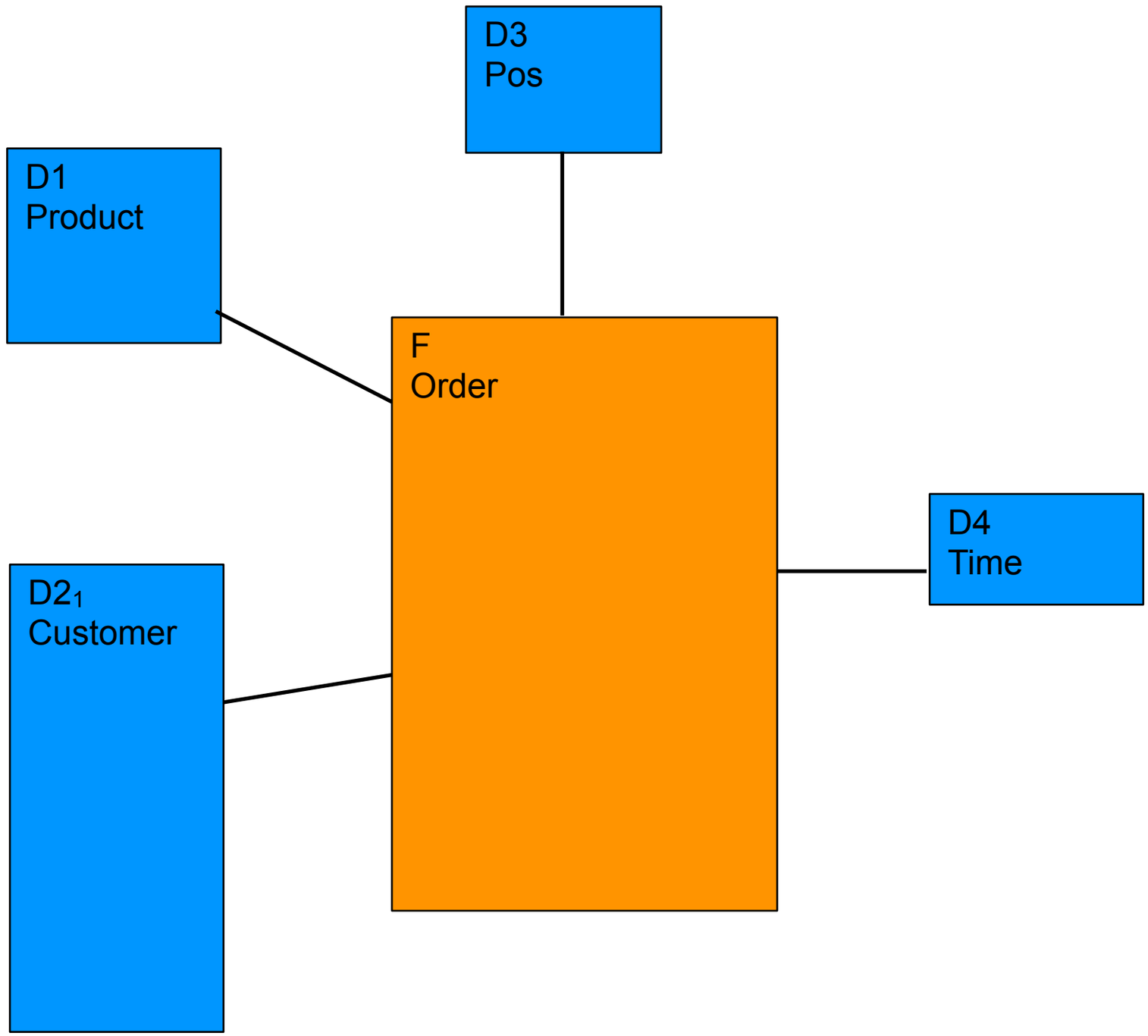
Variants of Data Partitioning

- **co-located join** (A and B both partitioned on join key, previous slides):
  best option
- **directed join** (one of the tables partitioned at runtime, then shipped):
  OK
- **repartitioned join** (none of the tables partitioned, repartition both, distribute to machines):
  avoid this
- same analogies to hash joins on single instances

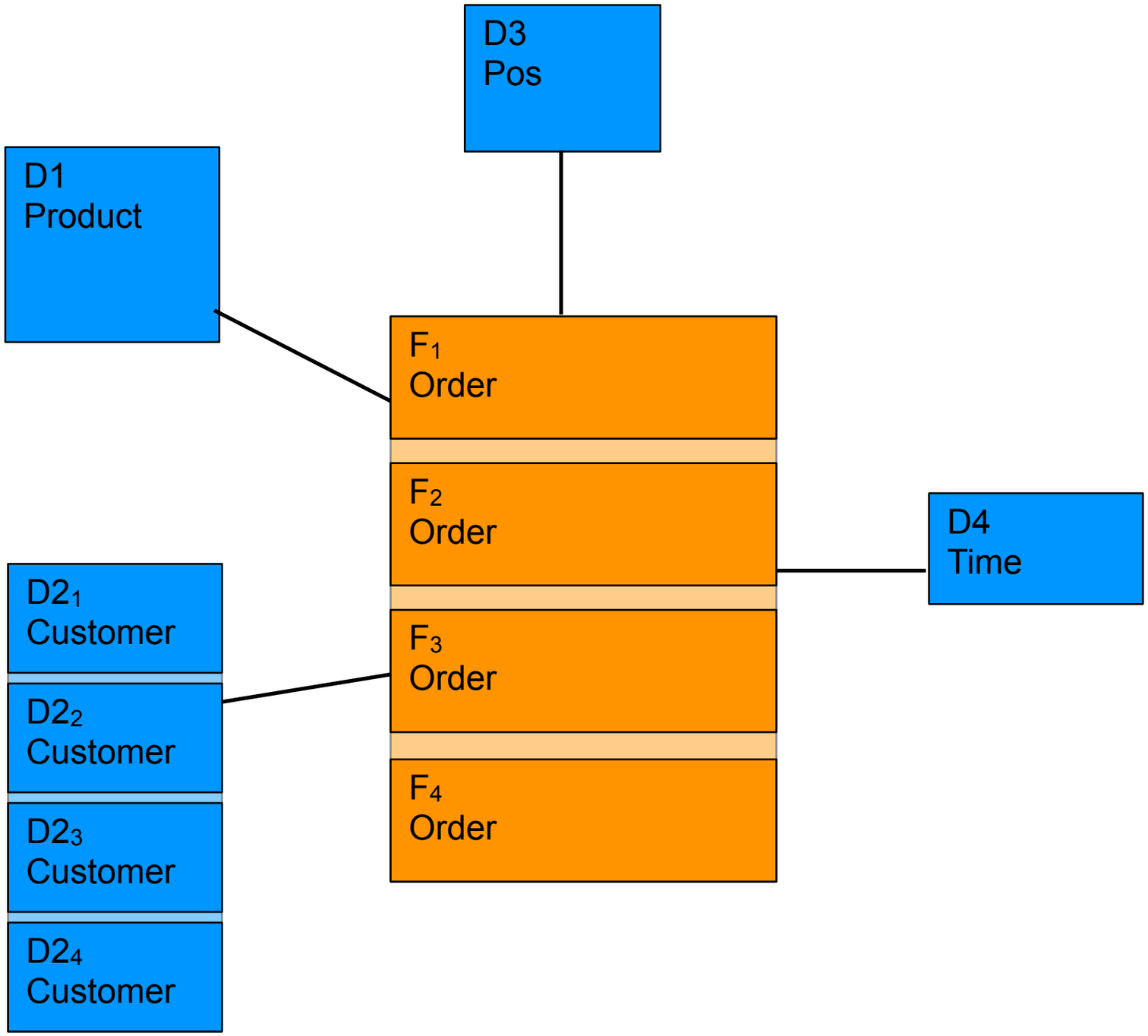
Data Replication in the Presence of Joins

- partitioning does not help much for smaller relations
- key idea
  - allow for some degree of replication
  - replicate smaller tables on multiple sides to allow for parallel query execution
- good for small tables
- in practice: use combination of partitioning and replication

Example for a Simple Star Schema

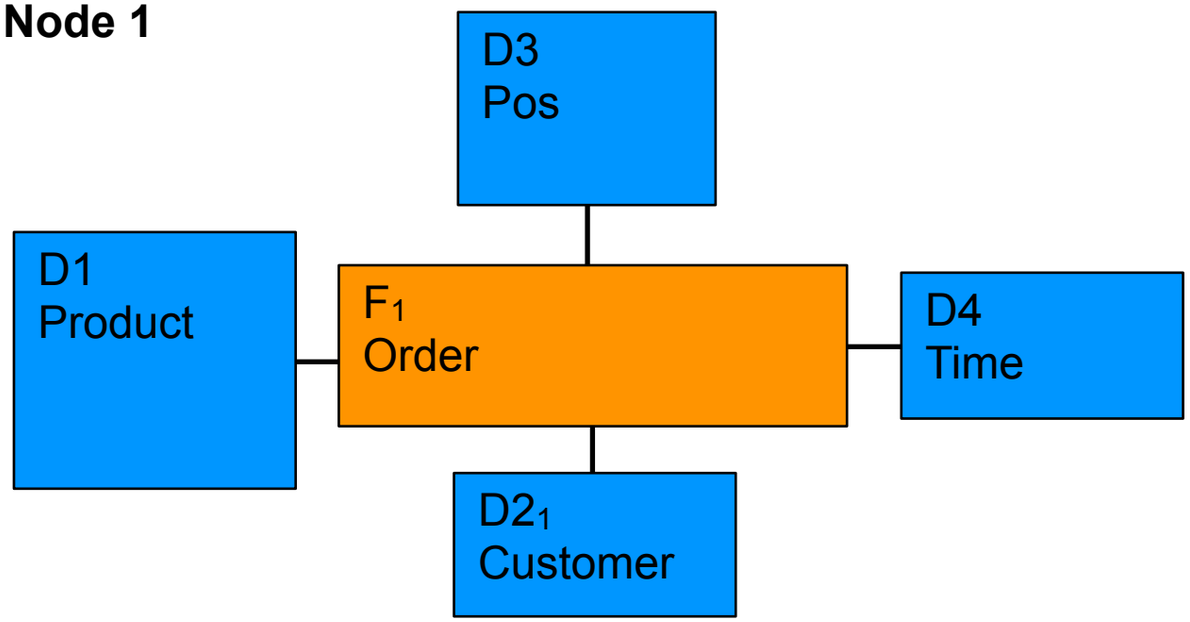


Step1: Partition fact table F and dimension table D2

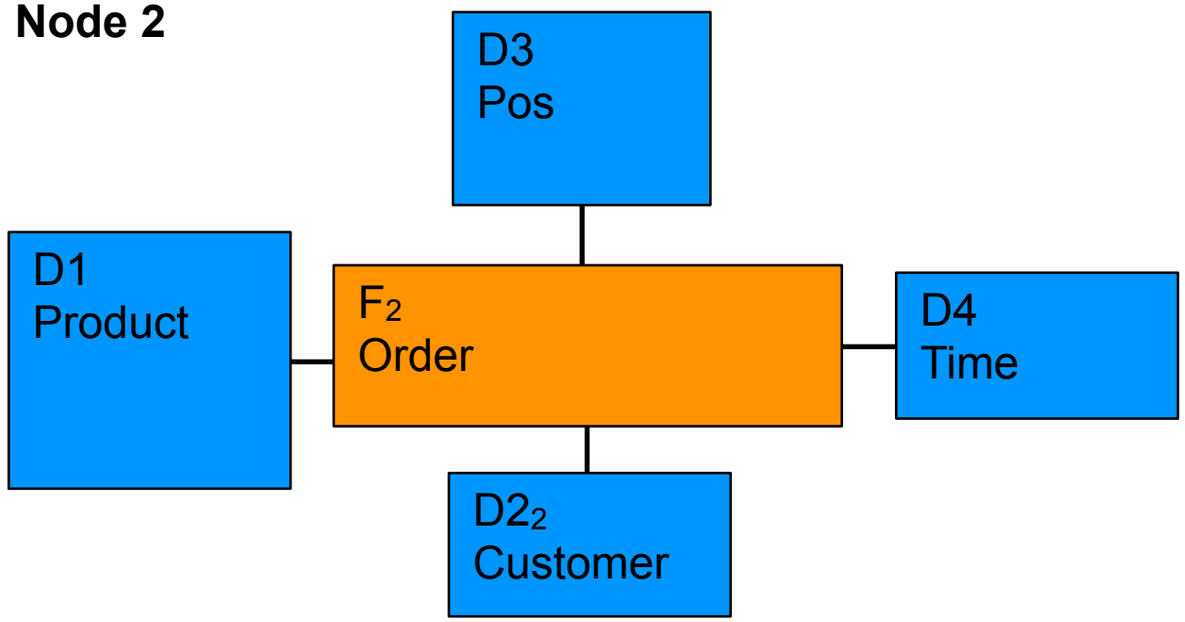


Step2: Replicate D1, D3 and D4

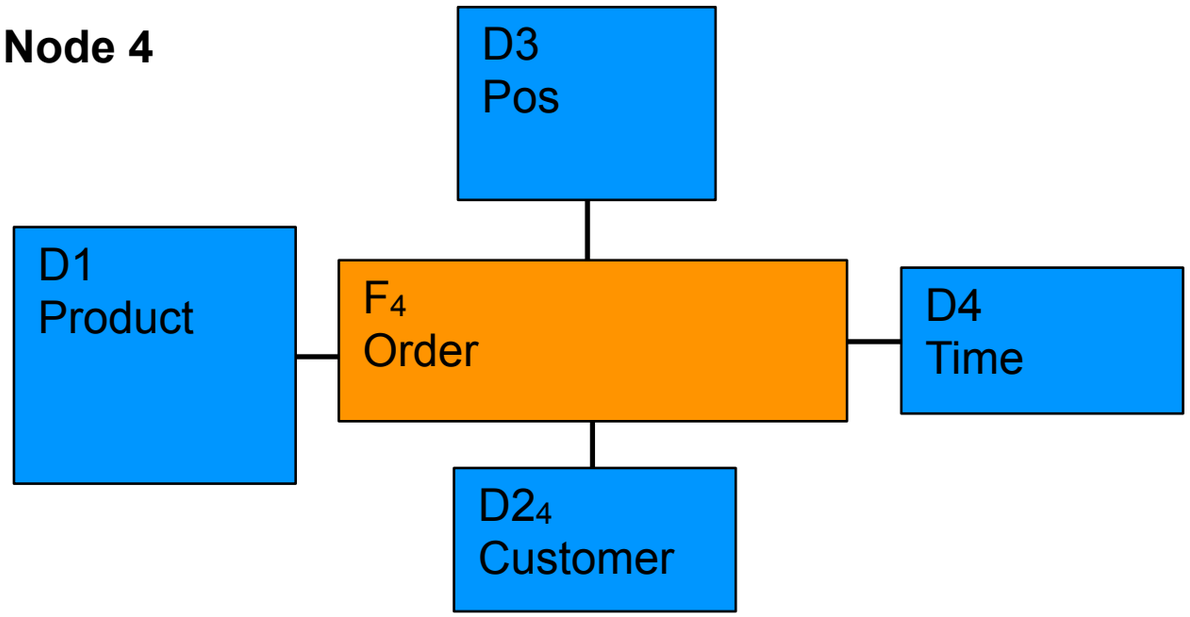
Node 1



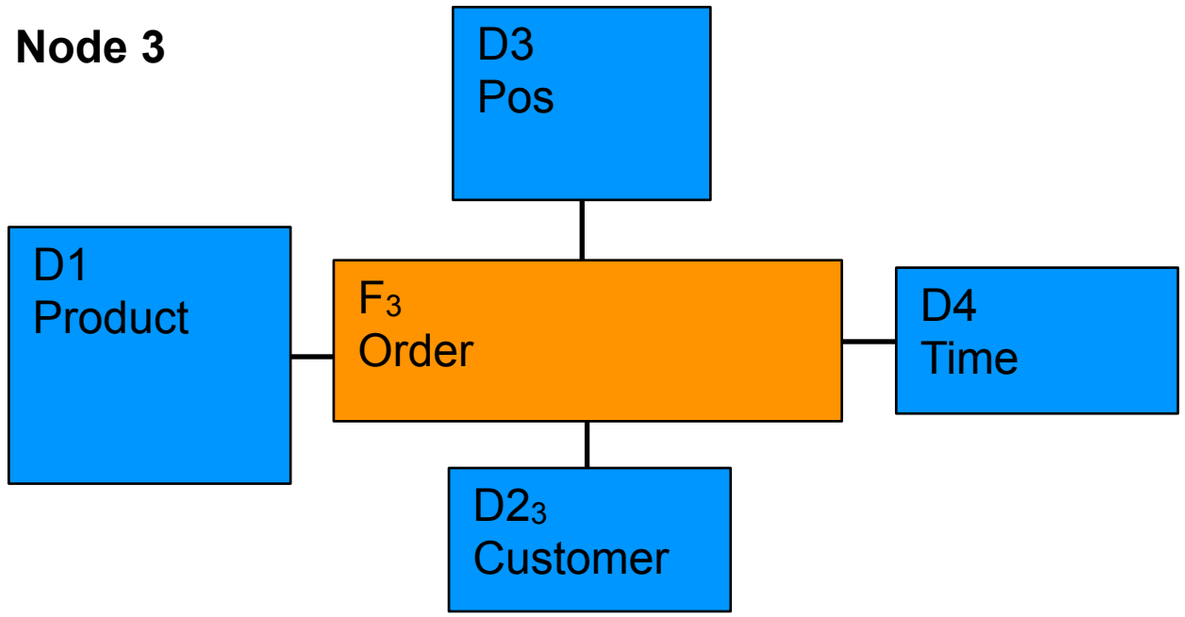
Node 2



Node 4



Node 3



Distributed Query Optimization

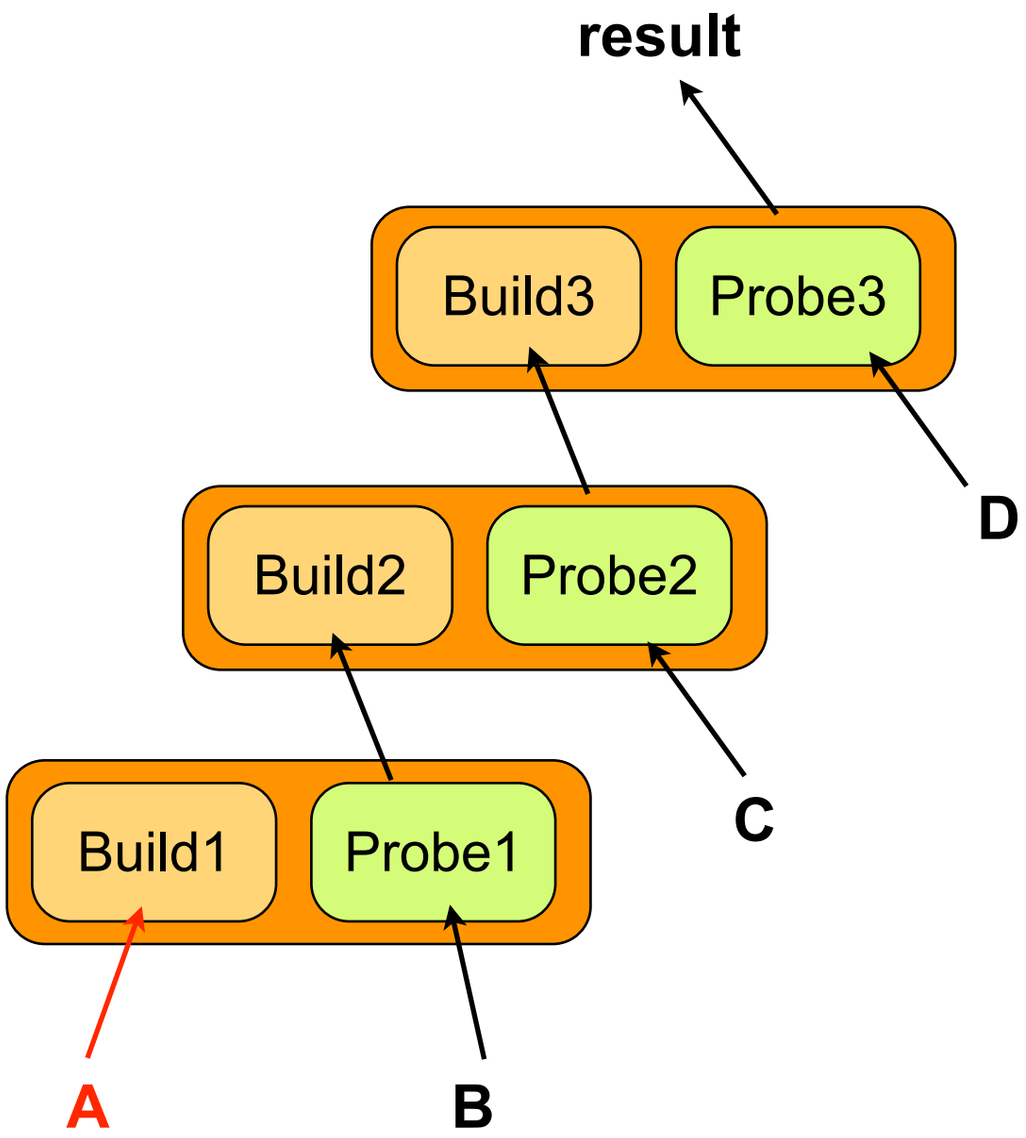
- more complicated than the single-threaded case
- need to extend cost model
  - model network cost, i.e., transfer cost for a given plan
  - understand degree/cost improvement of parallelization for
    - single operators
    - subplans
- different join orders allow DB different degrees of parallelization
  - left-deep vs. right-deep plans
  - bushy plans

Cost Models

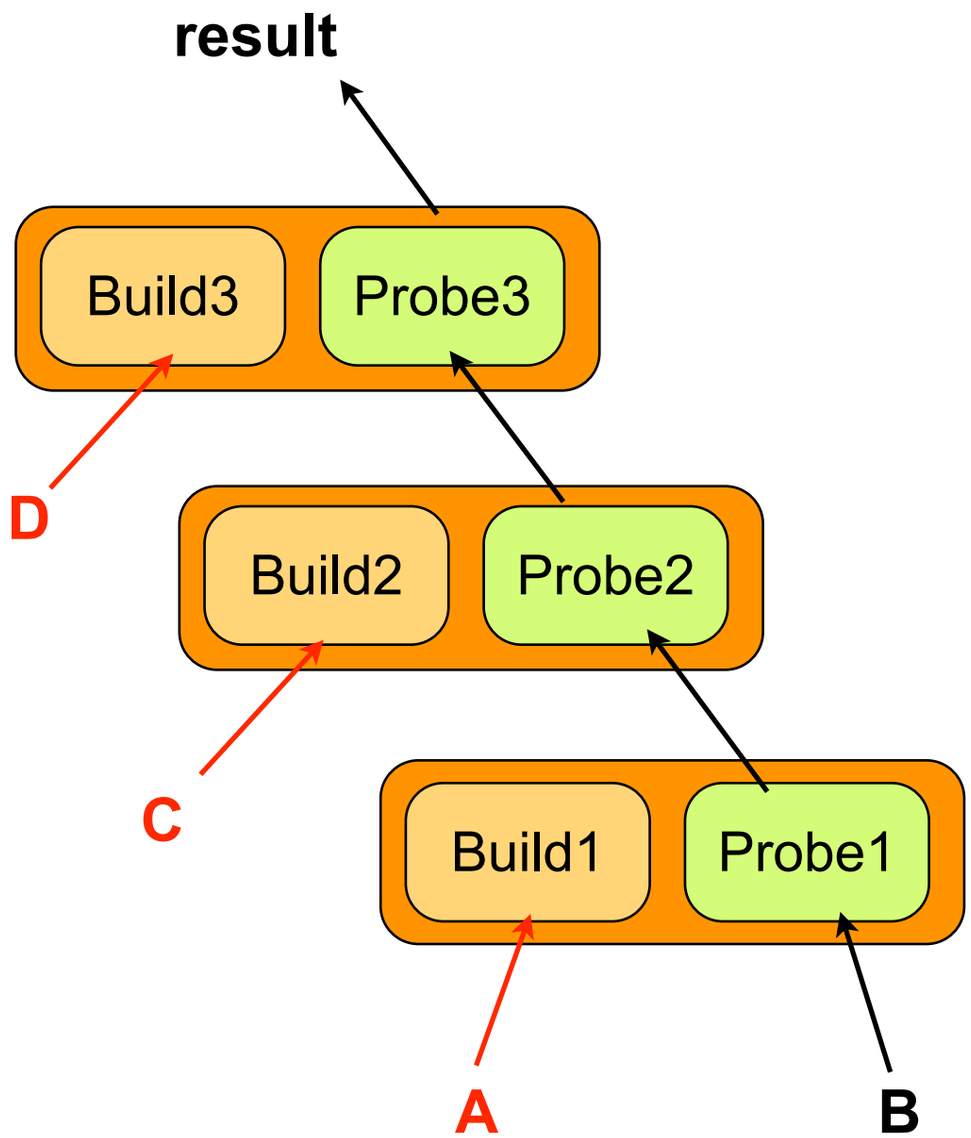
- extend local I/O-cost or CPU-cost include network cost
  - cost for sending/receiving data over the wire
  - network latency
- similarly to costmodels on a single instance, cost model should be calibrated for the parallel DB, i.e.,
  - come up with a network cost model
  - measure network cost on a real system
  - use measurements as a parameter (constants) in your model
- note: local databases may even have different cost models (important for heterogeneous databases)
- then we need to translate different local costs into global cost

Left-Deep versus Right-Deep Joins

- Assume a join of four relations A, B, C, and D using the simple (main memory) hash join



Left-Deep Plan

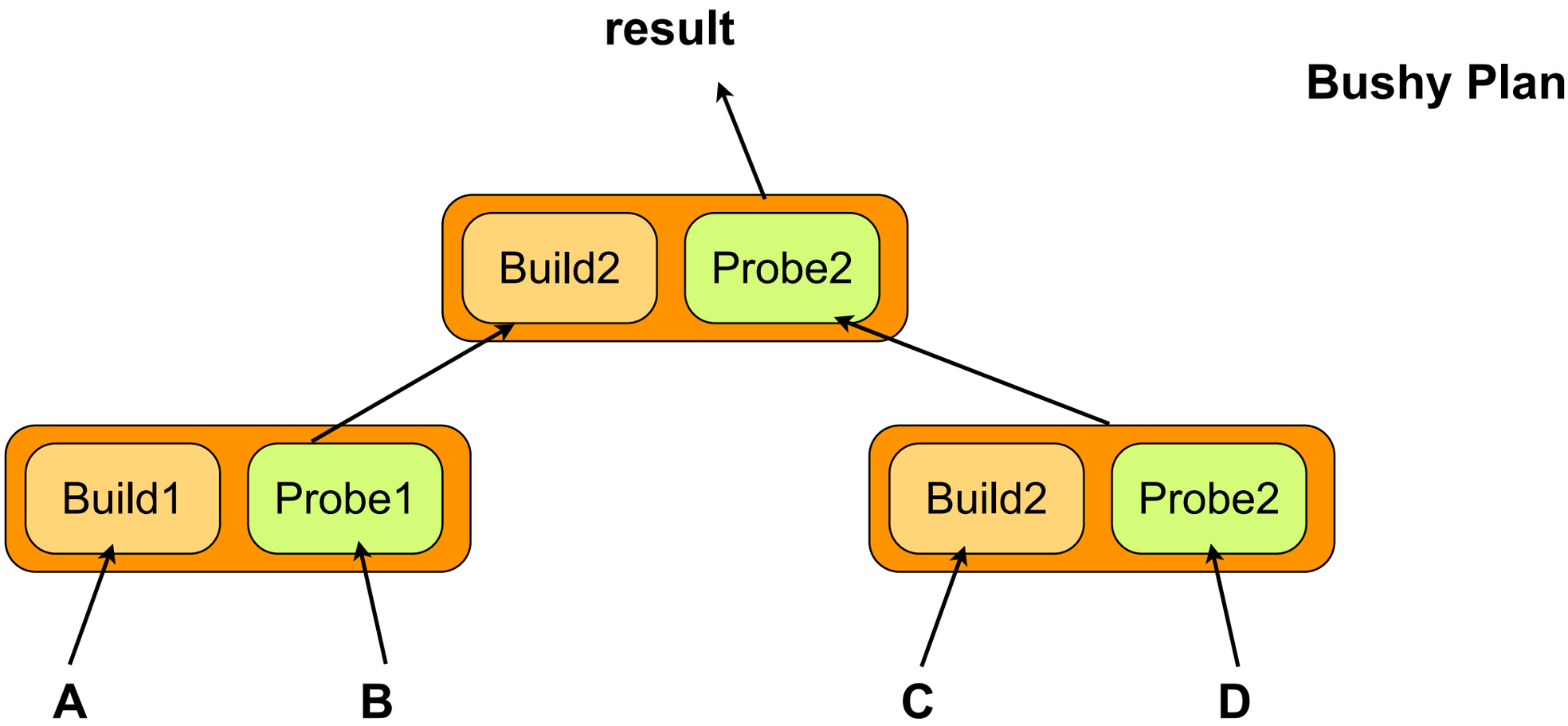


Right-Deep Plan

Left-Deep versus Right-Deep Joins

- left-deep plan hard to parallelize
- right-deep plan, however, may build hash tables (Build1, Build2, Build3) in parallel if enough main memory is available
  - what if C or D are huge?
- left-deep tree express full materialization of intermediate results
  (materialized in hash tables)
- right-deep tree may be more efficient due to parallelism but needs more main memory
- again: cost-based optimization needed

Bushy Plans



- A ⋈ B and C ⋈ D could be executed in parallel on separate nodes
- in general: bushy plans allow for higher degree of parallelism

Distributed Transaction Handling.

Distributed Transaction Handling

- assume again a money transfer:
  transfer 50 CHF from account A on Node 1 to account B on Node 2
- we assume that each node locally cares about recovery
  - redo: actions of winner TAs have to be redone whether they committed locally or “globally“
  - undo: changes done by TAs that did not commit either locally or globally have to be removed
- main challenge: TAs span multiple machines
- it may never happen that one node aborts the operations and the other commits: either all nodes abort or all nodes commit
- => we need some sort of centralized TA management

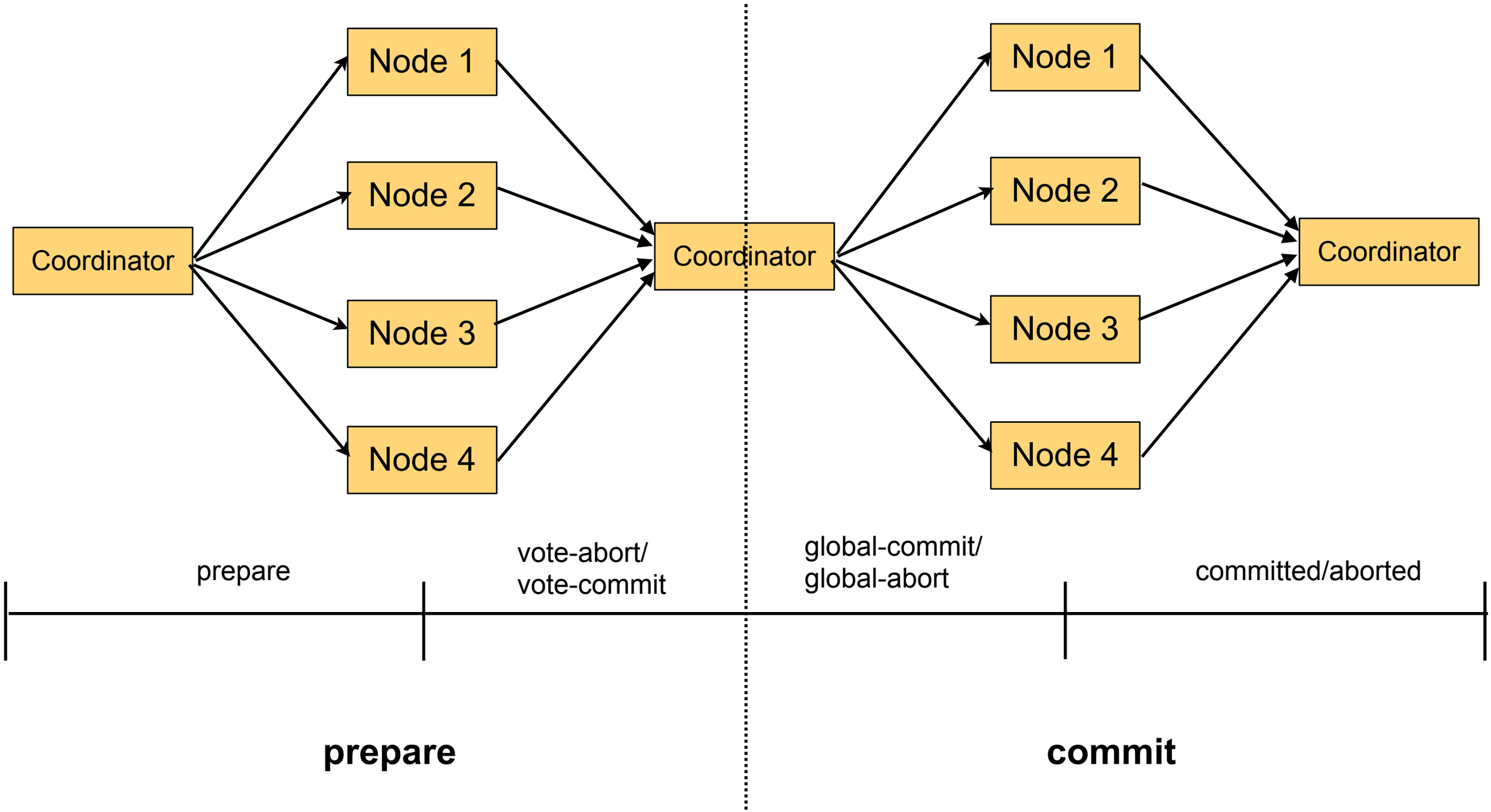
2PC: Two-Phase Commit Protocol

- goal: provide atomicity of TAs in a distributed environment
- most important protocol for parallel/distributed DBs
- assume a **coordinator C** (for simplicity let‘s assume a separate node)
- assume **N processing nodes** participating in the TA

2PC: Two-Phase Commit Protocol

1. coordinator C sends a PREPARE message to all N processing nodes
2. each processing node receives the PREPARE message and either
   - sends back a READY message to C if it is ready to commit
   - sends back a FAILED if it is not able to commit
3. if coordinator C has received a READY from all nodes N
   (a) C sends a COMMIT to all nodes N
   (b) each node N will commit the local changes
   (c) each node N will send an ACK (acknowledgement) to C
4. else, i.e., at least one of the nodes returned a FAILED or did not return a response within a given time interval in Step 2
   - coordinator C sends an ABORT to all nodes

2PC Graphically



Details

- coordinator C will log alls its actions w.r.t. 2PC, e.g.,
  - writes a list of TAs to the local log before sending PREPARE
- processing nodes will log alls its actions w.r.t. 2PC, e.g.,
  - writes a log entry to the local log before sending READY
  - in Step 4 write log entry COMMIT or ABORT before sending COMMIT or ABORT
- if processing nodes sends a READY, it has to make sure that it is able to commit in all cases

Crash of Coordinator or Nodes

- 2PC has to be able to survive crashes of the coordinator and of nodes
- discussion omitted here
- also extensions of 2PC to 3PC
- details: see book by Tamer Özsu, Patrick Valduriez: “Principles of Distributed Database Systems“ Chapter 12.6.1 and 12.6.2

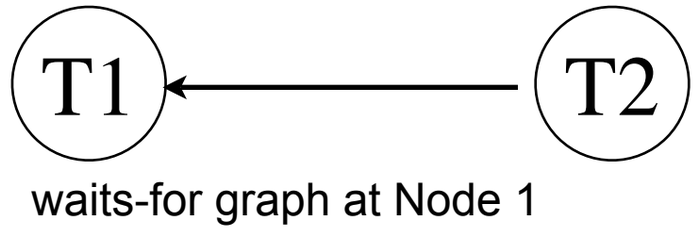
Concurrency Control

- how to lock data in a parallel database?
- three main approaches
  - centralized locking:
    - single node responsible for locking and unlocking of all objects
    - vulnerable to failure of this node
  - primary copy:
    - one copy of each object is desgnated as the primary copy
    - node that stores primary copy handles locking for this object
    - requires communictaion with two nodes: primary copy and copy
  - fully distributed
    - locking and unlocking done independently for all copies
    - communication only with a local DB
    - however, when writing locks must be set on all sites storing copies

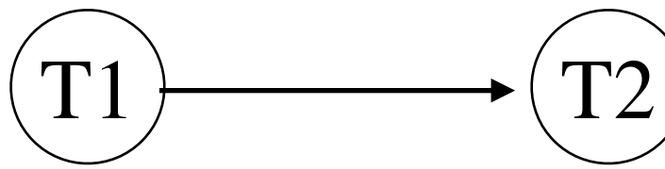
Distributed Deadlock Detection

- when using primary copy or fully distributed locking, distributed deadlocks may occur
- similar to a centralized DBMS, deadlocks must be detected and resolved (by aborting some deadlocked TAs)
- there may be distributed deadlocks even if the local waits-for graphs does not have a cycle!

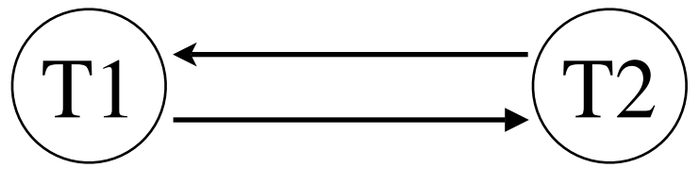
- Example:
  - assume that nodes 1 and node 2 contain copies of objects O1 and O2
  - T1 obtains S-lock on O1 and X-lock on O2 at Node 1
  - T2 obtains S-lock on O2 and X-lock on O1 at Node 2
  - T1 requests X-lock on O2 at Node 2
  - T2 requests X-lock on O1 at Node 1
  - => Deadlock!



waits-for graph at Node 1



waits-for graph at Node 2



global waits-for graph

Algorithms

- Algorithm 1:
  - periodically send local waits-for graph to a central site
  - central cite creates global waits-for graph to detect deadlocks
- Algorithm 2:
  - group nodes into a hierarchy
  - same as Algorithm 1, create waits-for graph for a subtree
  - better locality than a single centralized node
    (for a possibly huge amount of local DBMSs)
- Algorithm 3:
  - timeout
  - very simple
  - may be the only option if local waits-for graphs may not be combined
    (e.g., local site may not offer interface to get waits-for graph)

Literature

- M. Tamer Özsu/Patrick Valduriez: Principles of Distributed Database Systems, Prentice Hall
- Kemper/Eickler: Datenbanksysteme
- Raghu Ragakrishnan/Johannes Gehrke: Database Management Systems, Mc Graw Hill
- Donald Kossmann: The State of the Art in Distributed Query Processing, ACM Computing Surveys 2000

Next Topic: Other Project-Specific Techniques.