

Data Warehousing

FS 2007

Dr. Jens-Peter Dittrich

jens.dittrich@inf

www.inf.ethz.ch/~jensdi

Institute of Information Systems



Query Processing and Indexing (Additional Material)

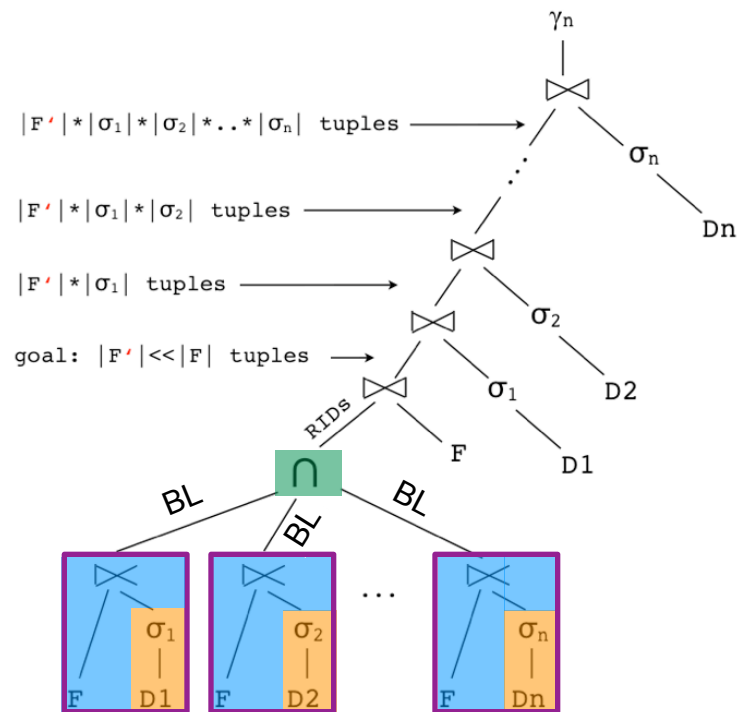


Bloom-Filters in Starjoins

- Question: where to apply bloom-filters in star joins?
- Recall:
 - bloom filter allows to use a smaller bitmap
 - bloom filter will return a **super set** of the required data
 - in general: hash function used for bloom filter is **not** reversible!
- Consequence:
 - from the bitlist returned by the bloom-filter we **cannot** determine which tuples pass the selection
 - therefore bloom-filters can only be applied if
 - we scan one of the inputs anyway
 - we are able to generate all keys that may exist and pass them to the bloom-filter
 - (see slides 29 and 30 from last week)

Bloom-Filters in Starjoins

- One possible problem of semi-join plan using bitmap indexes
 - intersect** requires AND operation on n bitmaps of size $|F|$ each
 - this operation could still be expensive
- Idea: use bloom filters to optimize this intersect operation!
- Algorithm:
 - use a bitlist BL of size $m \ll |F|$ for all join indexes replacing semijoins (Type 2 and Type 3)
 - perform **intersect** on BL (may produce a superset)
 - Note: The overall result of this plan is correct as following joins will eliminate redundant tuples.
 - Effect: saves main memory and processing time

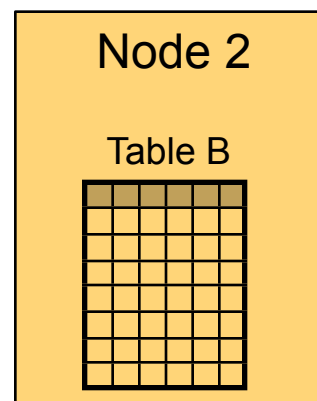
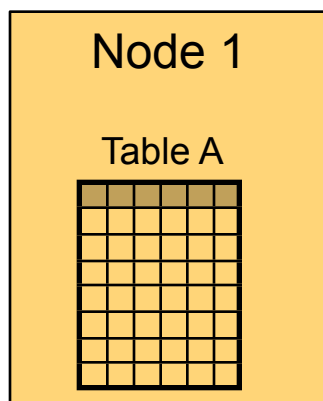


Query Processing and Indexing in Parallel Data Warehouses



Distributed Join Processing in General

- Scenario:

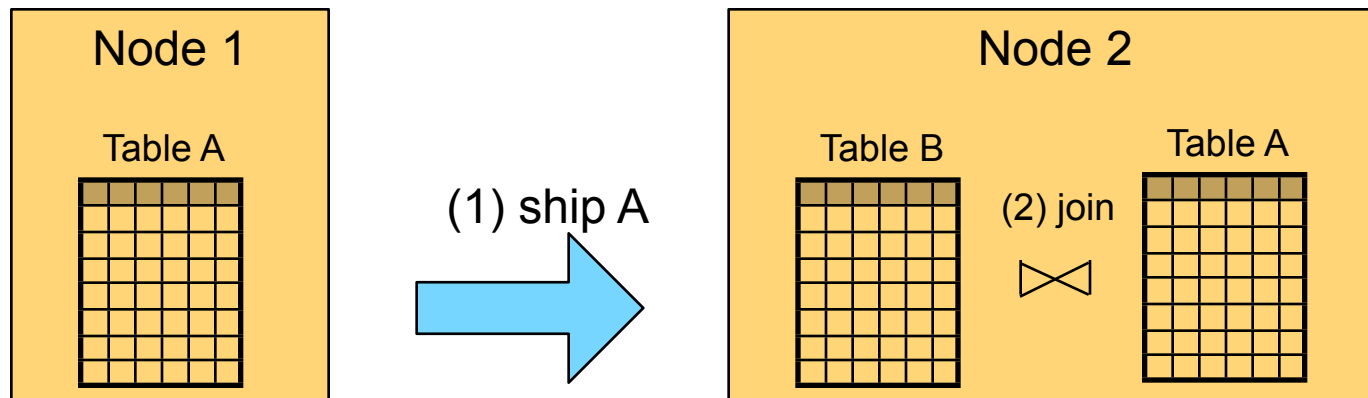


- How to compute $A \bowtie B$?
 - where A resides on machine 1
and B resides on machine 2

Very naïve Approach: ship Data page-wise

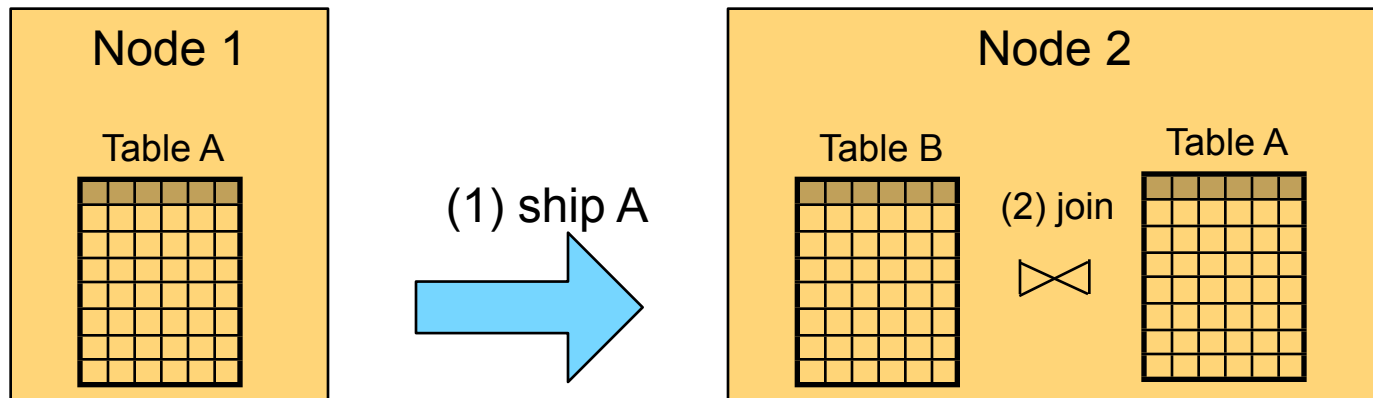
- execute standard join operators on machine 1, e.g., index nested-loops join
- whenever you fetch a page from machine 2, you cache that page on machine 1
- Problem: cost for this may be very high:
 - random access for pages of B on machine 1
 - basically shipping entire relation B to machine 1
- Not 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 at the same node
 - join A and B locally (2)
- (or vice versa)

Discussion



■ Problems

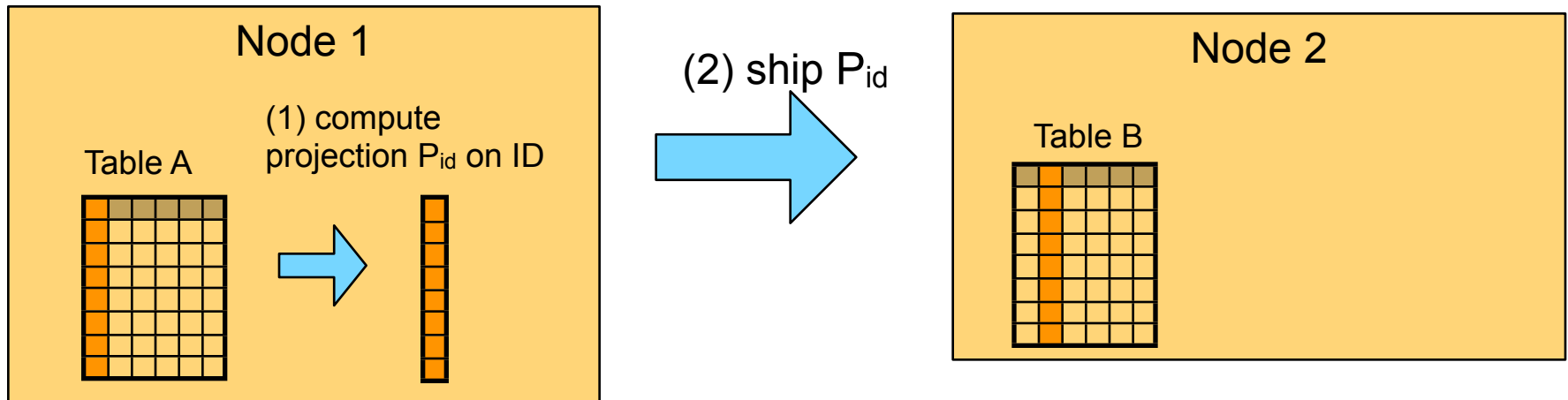
- high cost for shipping entire Table A
- network cost roughly the same as I/O cost for a hard disk
- thus: shipping A roughly equivalent to a full table scan of A on Node 1

■ 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 $A \bowtie B$ with join predicate $A.id == B.a_id$

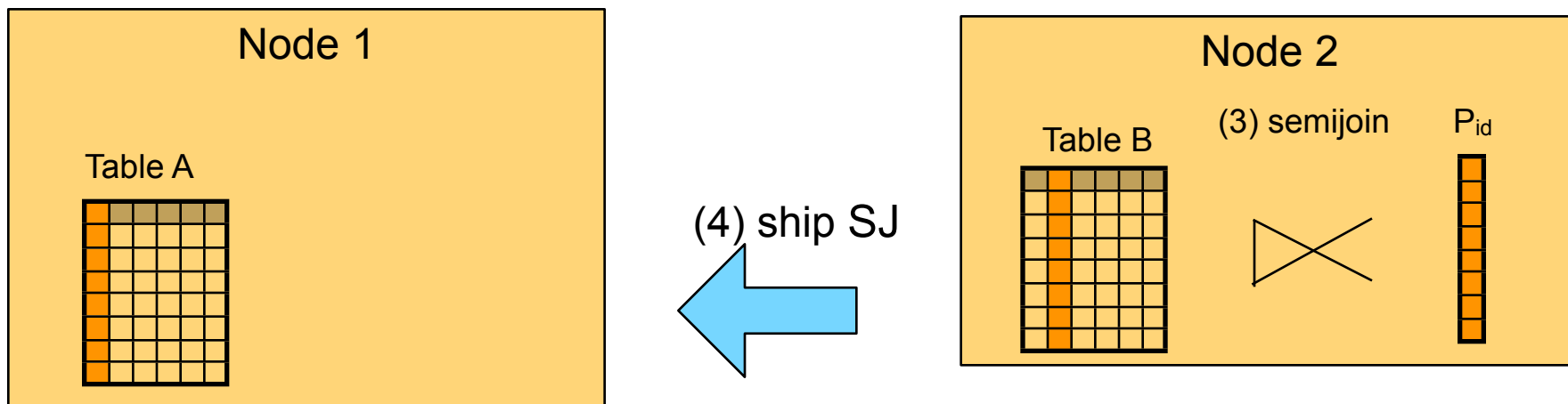


(1) on Node 1 compute the projection P_{id} of Table A on A.id

(2) send P_{id} to Node 2

Semijoin (2/3)

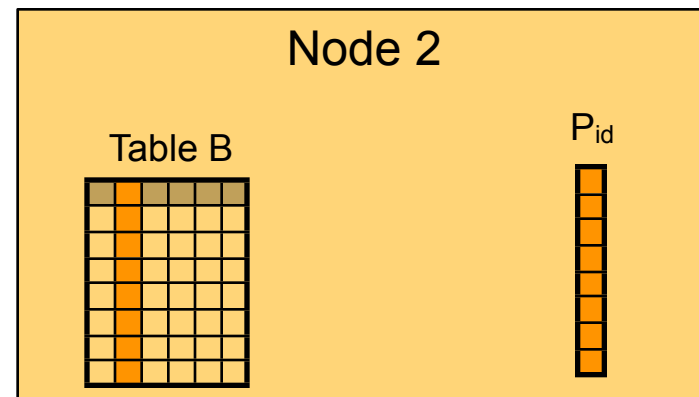
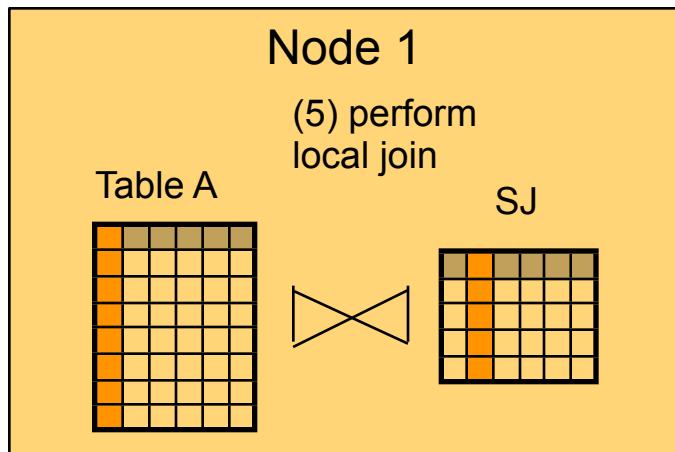
- Assume $A \bowtie B$ with join predicate $A.id == B.a_id$



- Perform the semijoin $SJ := B \bowtie P_{id}$
(i.e, select all Tuples of B that may join with tuples in Table A)
- Send result of semijoin SJ to Node 1

Semijoin (3/3)

- Assume $A \bowtie B$ with join predicate $A.id == B.a_id$



(5) Perform local join $A \bowtie (SJ)$ at Node 1



Discussion

- Works well if selection of semijoins allows to reduce size of table to be shipped.
- 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!
- Consequence: need to decide whether this method makes sense based on semi-join selectivity. (cost-based optimization)

Bloomjoin

- algorithm same as semijoin approach
- but: ship a bit-vector instead of (foreign) key column
- use bloom filter technique (hash function, same as for value bitmaps)
- goal: only send a small bit list (reduce network I/O)
- i.e., bit list is shorter than number of keys that need to be sent
- Problem: superset of tuples will be sent (same problem as in bloom filter for value bitmaps)
- net gain of this method very sensitive to choice of a good hash function

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 F and D tables
- Example
 - $A.id == B.a_id$
 - $\text{hash}(id) = id \% 4$
 - partition into 4 groups $(A_0, B_0), \dots, (A_3, B_3)$
 - where A_i contains all elements of A for which $\text{hash}(id) == i$
 - where B_i contains all elements of B for which $\text{hash}(id) == i$
- Effect
 - All joins become local!!

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

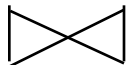
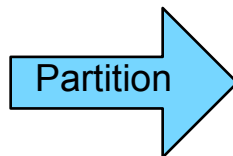


Table B

a_ID			
5			
7			
4			
12			
9			
11			
10			
2			

Partition

Table A₀

ID			
4			
8			
12			

Node 1

Table B₀

a_ID			
4			
12			

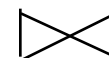
Table A₁

ID			
1			
5			
9			

Node 2

Table B₁

a_ID			
5			
9			

Table A₂

ID			
2			
6			
10			

Node 3

Table B₂

a_ID			
2			
10			

Table A₃

ID			
3			
7			
11			

Node 4

Table B₃

a_ID			
7			
11			



Data Partitioning in the Presence of Joins

- Remember the Grace Hash Join from the Implementation and Architecture of DBMS lecture?
- The core idea is the same.
- Thus, what we are doing is a grace hash join where the partitioning phase was done at indexing time!
- In fact, whenever you can apply partitioning or hashing in some algorithm it is worth looking whether this serves as an opportunity to parallelize that algorithm.

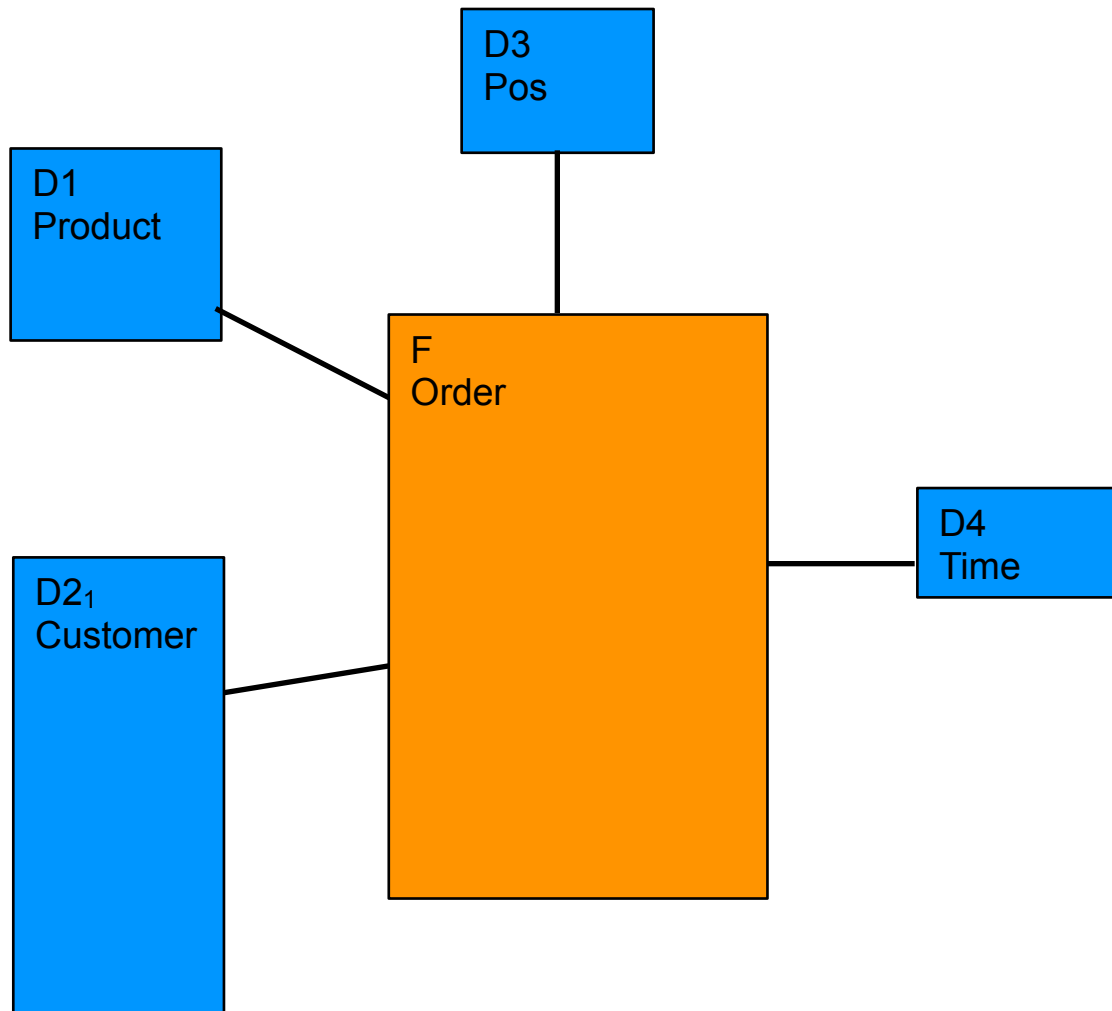
Variants of Data Partitioning

- co-located join (F and D 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

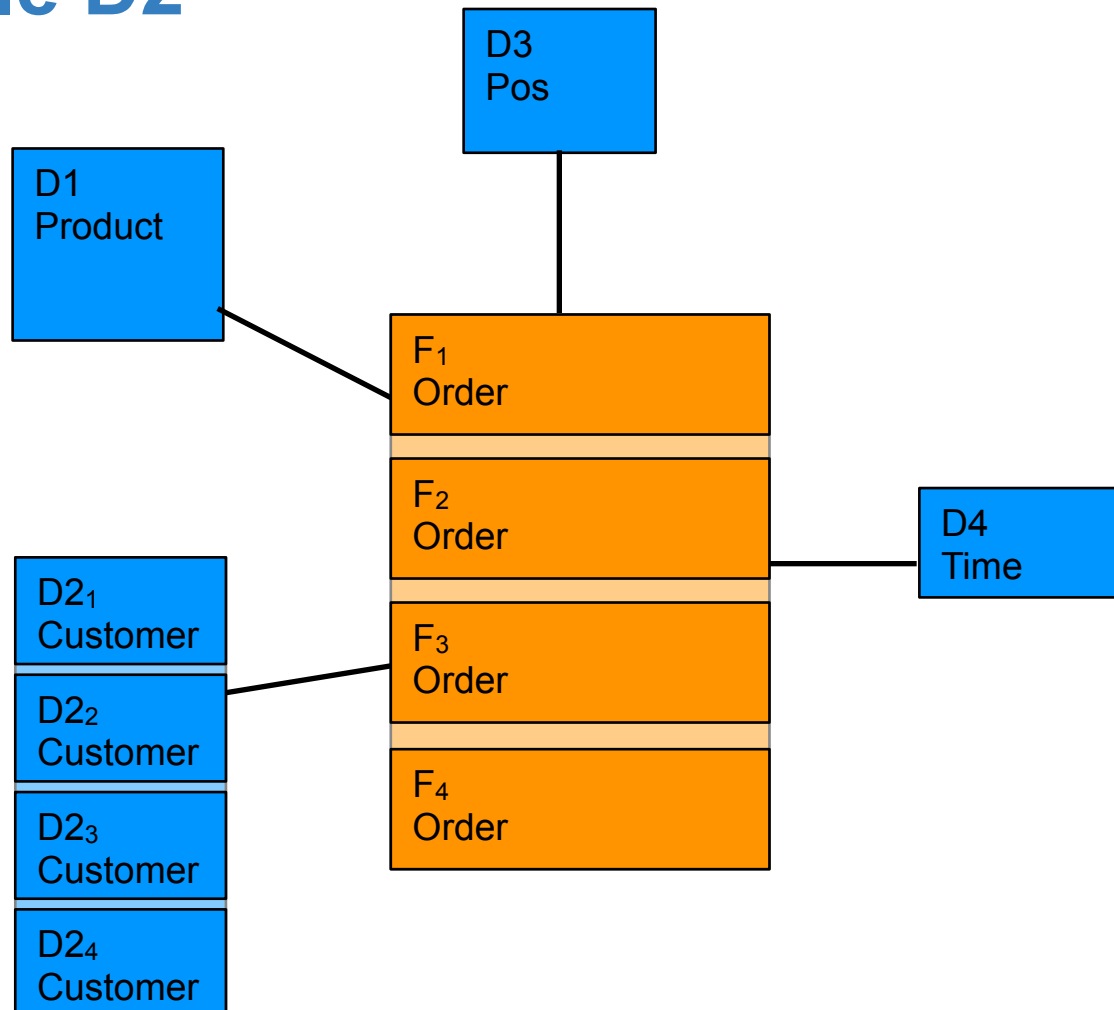
Data Replication in the Presence of Joins

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

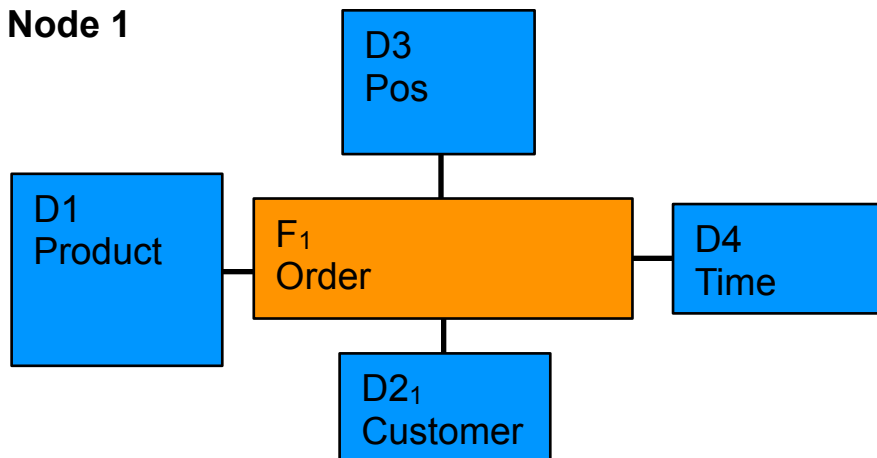
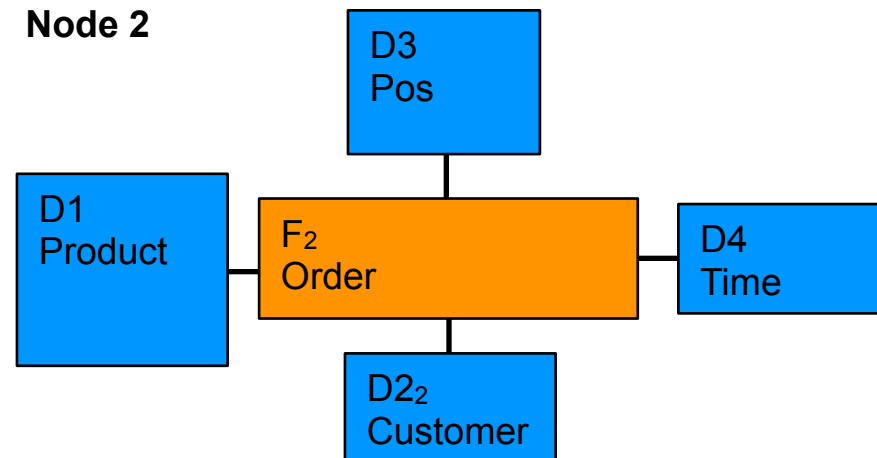
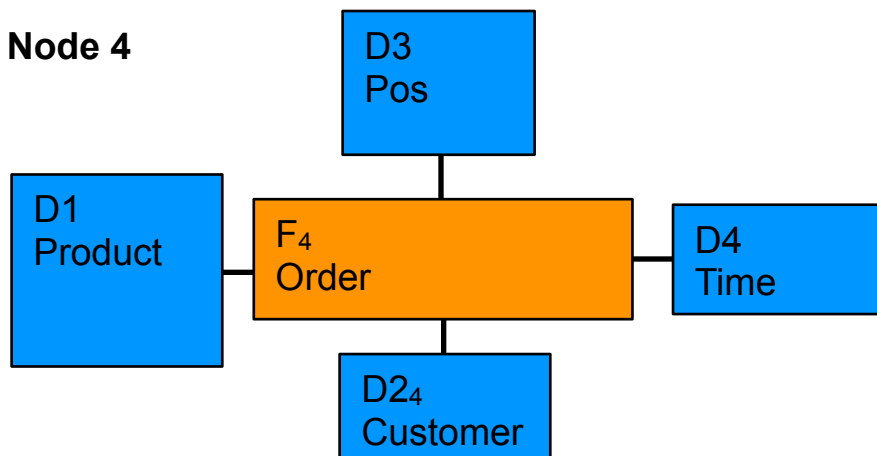
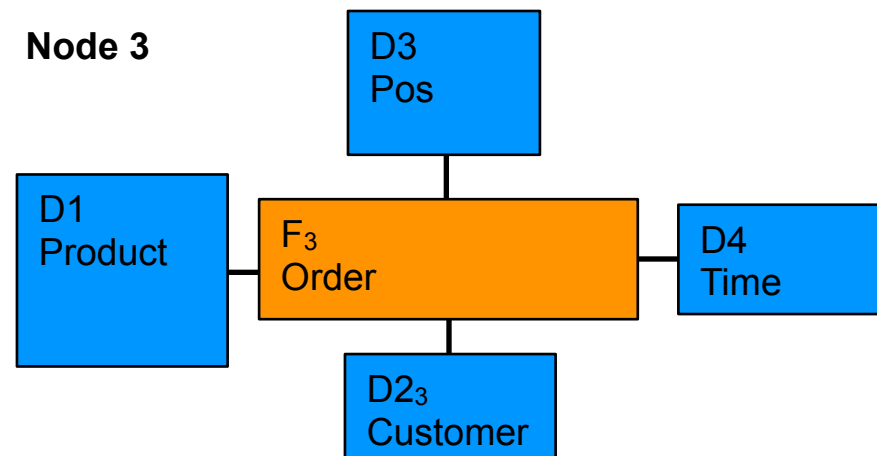
Example for a Simple Star Schema



Step1: Partition fact table F and dimension table D2



Step2: Replicate D1, D3 and D4

Node 1**Node 2****Node 4****Node 3**



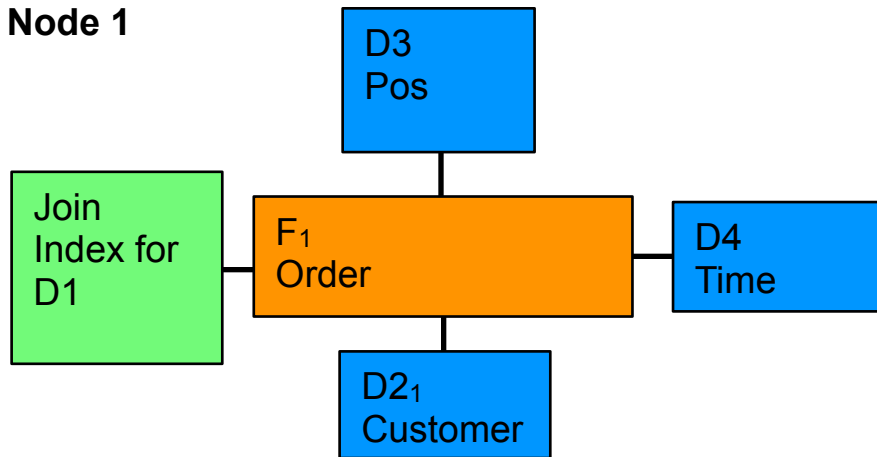
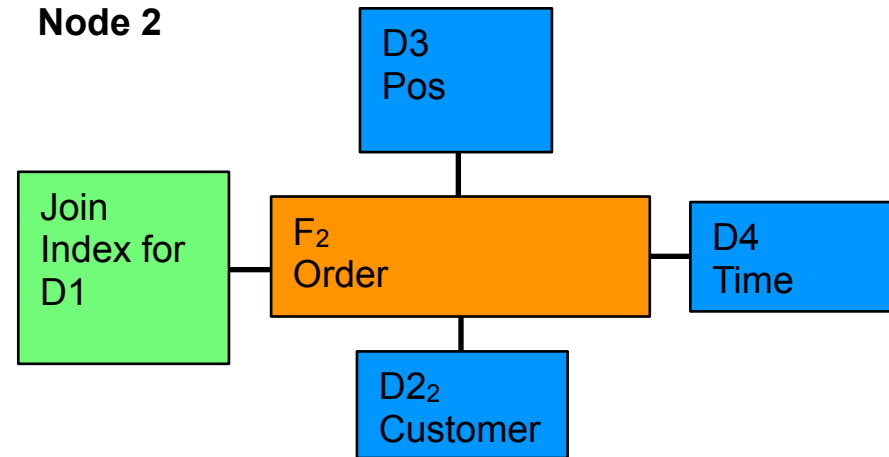
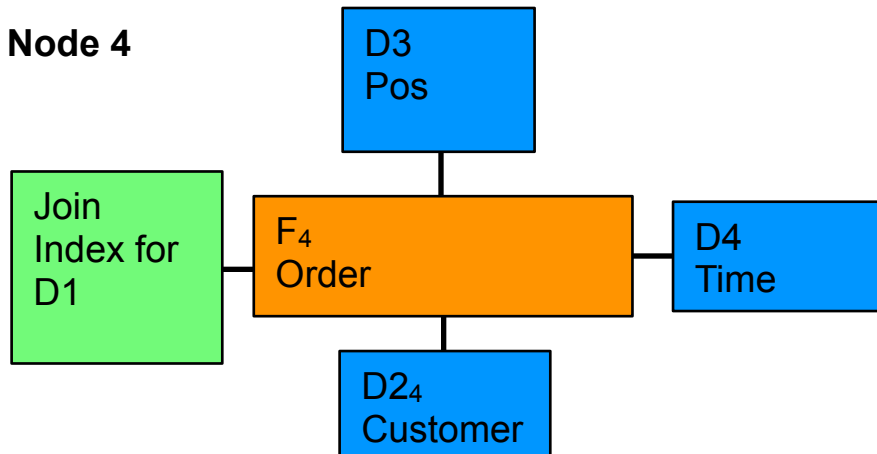
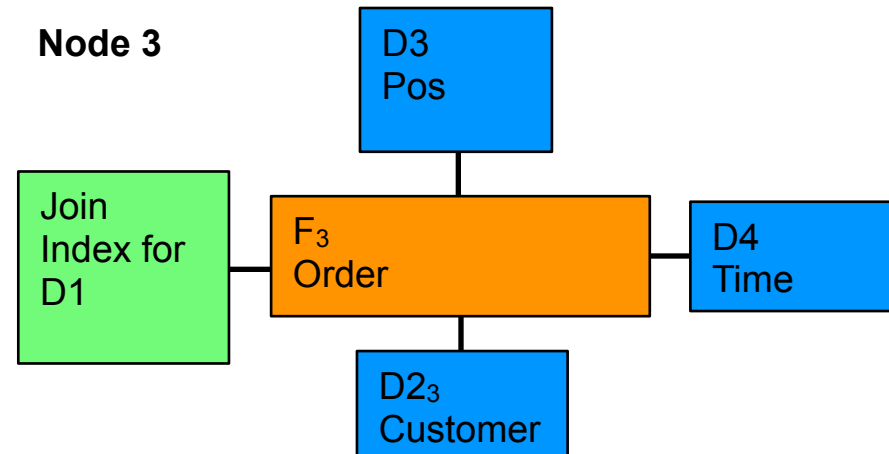
Discussion

- Partitioning is limited:
 - consider having more than one large dimension table, e.g., D1
 - consider having dimension tables that are shared among multiple star schemas (galaxy schema)
- Cost for replicating tables may at some point become too high
- Remedy: allow some distributed joins
- But: optimize them by providing join indexes, e.g. the semijoin algorithm could start directly with (3) or (4)

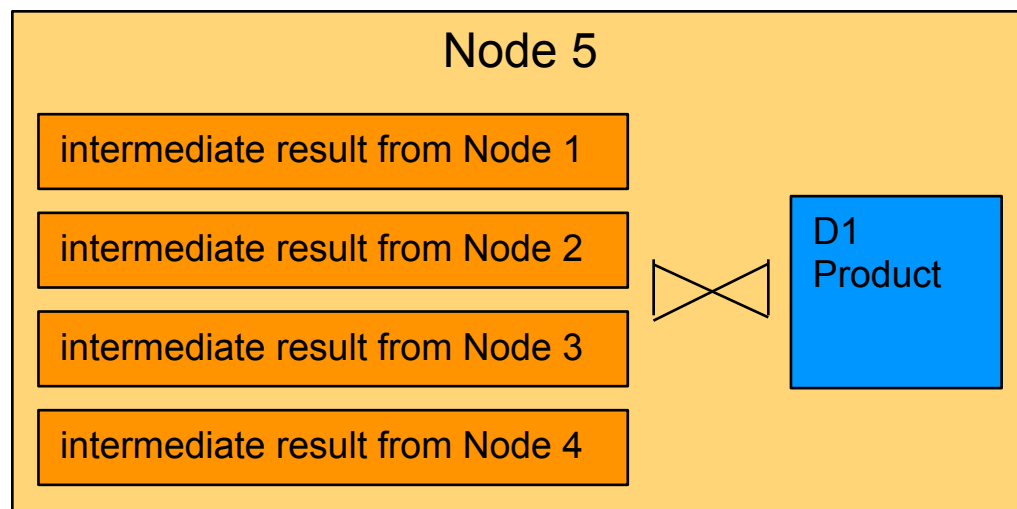
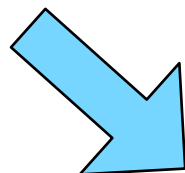
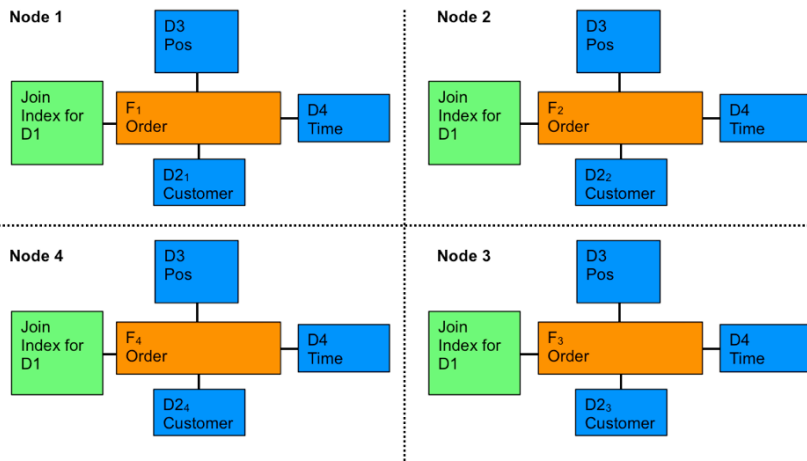
Example

- do not replicate big dimension tables, e.g., D1
- but proceed in two steps
 - (1) process query on partitioned cube ignoring D1
however: join index with D1 may be exploited
 - (2) send intermediate join result to a separate machine performing final join with D2

First step: join locally

Node 1**Node 2****Node 4****Node 3**

Second Step: send intermediate Results to Machine holding D1



Further Optimization Challenges

- Load distribution
- Multiple cubes on same machine
- Multiple queries at the same time
- Redundant machines for same data (inter-query parallelism)
- etc.
- many more techniques