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 (5/6)

- large systems
  - global scale data management
  - map/reduce
  - pig and pig latin
  - search engines
  - data warehousing and OLAP
- write-optimized system concepts
  - OLTP
  - publish/subscribe
  - streaming
  - moving objects
- management of geographical data
  - basic concepts
  - GIS, google maps

SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Data Warehousing and OLAP

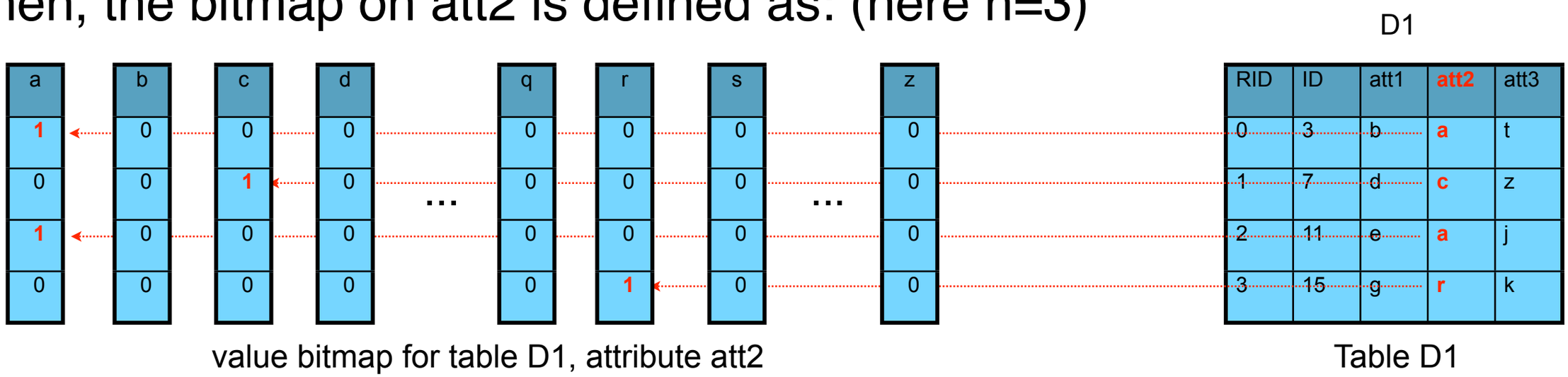
Part II.

Join Index Implementation

- How to represent result RID lists? Two possibilities:
  1. as lists of RIDs
     - uncompressed
       value -> [int,int,int, terminator] or:
       value -> [number of RIDS, int,int,int]
     - using some sort of compression
       value -> [int, diff to previous value, diff to previous value]
       or any other suitable techniques
     - again many similarities to IR technology
  2. as bitmaps
     - works if total number of RIDs is bounded and the number of attribute values is “low“
     - will be discussed in the following
- similar techniques apply for B^+-trees...

Recap: Value Bitmap

- Lets assume that the domain of attribute att2 is {a,...,z}.
- Lets assume that the number of RIDs is bounded by n.
- Then, the bitmap on att2 is defined as: (here n=3)



value bitmap for table D1, attribute att2

Table D1

- This means, for each attibute value v of attribute att2 we define a separate bit list BL_v of length n+1.
- If row RID has value v for attribute att2, then BLv[RID] := 1, otherwise BLv[RID] := 0.
- Size of the bitmap: l{a,...,z}l * (n+1) = 26*4 = 104 Bits = 13 Bytes
- see also Week 4, Slide 54 (and blackboard)

Bitmap Operations

- Why are bitmaps a good idea?
- Assume we have multiple bitmaps for different attributes of a table D1
- AND
  condition **D1.att1 == 42 AND D1.att4 == 47**
  computation: lookup bit lists in both bitmaps and compute boolean **and**
- OR
  condition **D1.att1 == 42 OR D1.att1 == 43**
  computation: lookup both bit lists in bitmap for att1 and compute boolean **or**
- NOT
  condition **D1.att1 != 42**
  computation: lookup bit list in bitmap for att1 and compute boolean **not**
- etc.
- bit operations are very efficient as they combine 32/64 bits in a single CPU operation
- vectorized (**SIMD**) processing may even compute longer bit chunks at a time

Bitmap Compression

- Many entries of a bit map may be zero
- Therefore compression techniques may be applied.
- At query time bitmap operations have to decompress the bitmaps
- Techniques
  - run length encoding RLE: good if only few bits are set
  - approximative bitmaps: similarities to kd-tries
  - others

Switching Bitmap Representations

- A small note:
  depending on the cardinality of an attribute value, RID list representations may be switched
- For example
  - attribute value frequent: bitmaps may be a good choice
  - attribute value rare: RID lists may be better
- A clever index implementation may switch between the two representations at any time.
- This observation holds for B^+-trees as well as for inverted lists.
- Therefore, an ordered RID list can be regarded as a way to compress a bitmap (and vice-versa).

Bitmap Join Index Example: Type 2

- = a join index represented by a bitmap

D1

RID	ID	att1	att2	att3
0	3	b	a	t
1	7	d	c	z
2	11	e	a	j
3	15	g	r	k

1 — n

F

RID	D1_ID	...	...
0	3		
1	11		
2	7		
3	15		
4	3		
5	7		
6	15		
7	11		
8	11		
9	7		
10	3		

2. D1.RID -> {F.RID}

- Logical Index:
  0 -> {0,4,10}
  1 -> {2,5,9}
  2 -> {1,7,8}
  3 -> {3,6}

2. D1.RID -> {F.RID}

- Bitmap Join Index:

0	1	2	3
1	0	0	0
0	0	1	0
0	1	0	0
0	0	0	1
1	0	0	0
0	1	0	0
0	0	0	1
0	0	1	0
0	0	1	0
0	1	0	0
1	0	0	0

Bit list length = |F|

Bitmap Join Index Example: Type 3

D1

RID	ID	att1	att2	att3
0	3	b	a	t
1	7	d	c	z
2	11	e	a	j
3	15	g	r	k

1 — n

F

RID	D1_ID	...	...
0	3		
1	11		
2	7		
3	15		
4	3		
5	7		
6	15		
7	11		
8	11		
9	7		
10	3		

3. D1.att2 -> {F.RID}

- Logical Index:
  a -> {0,1,4,7,8,10}
  c -> {2,5,9}
  r -> {3,6}

2. D1.RID -> {F.RID}

- Bitmap Join Index:

0	1	2	3
1	0	0	0
0	0	1	0
0	1	0	0
0	0	0	1
1	0	0	0
0	1	0	0
0	0	0	1
0	0	1	0
0	0	1	0
0	1	0	0
1	0	0	0

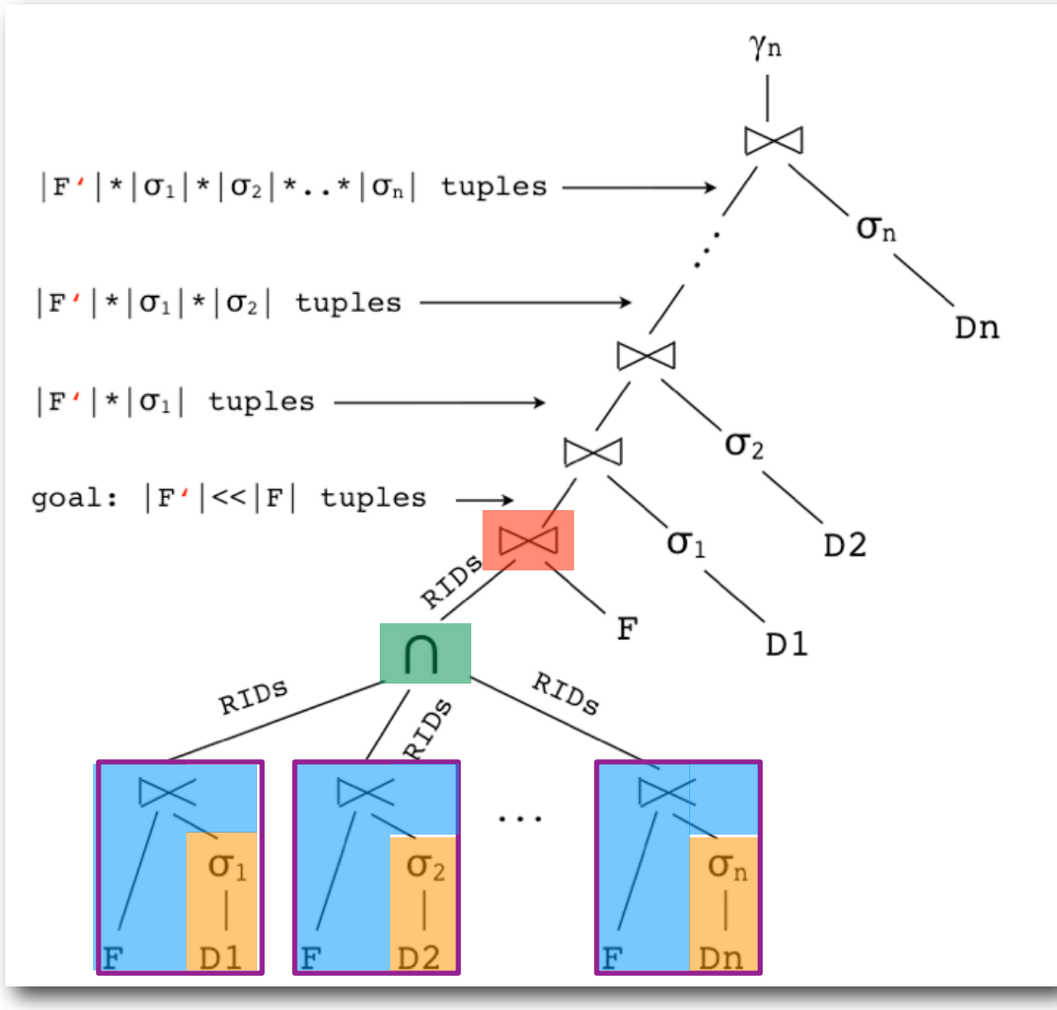
3. D1.att2 -> {F.RID}

- Bitmap Join Index:

a	c	r
1	0	0
1	0	0
0	1	0
0	0	1
1	0	0
0	1	0
0	0	1
1	0	0
1	0	0
0	1	0
1	0	0

Star Join using Bitmap Indices

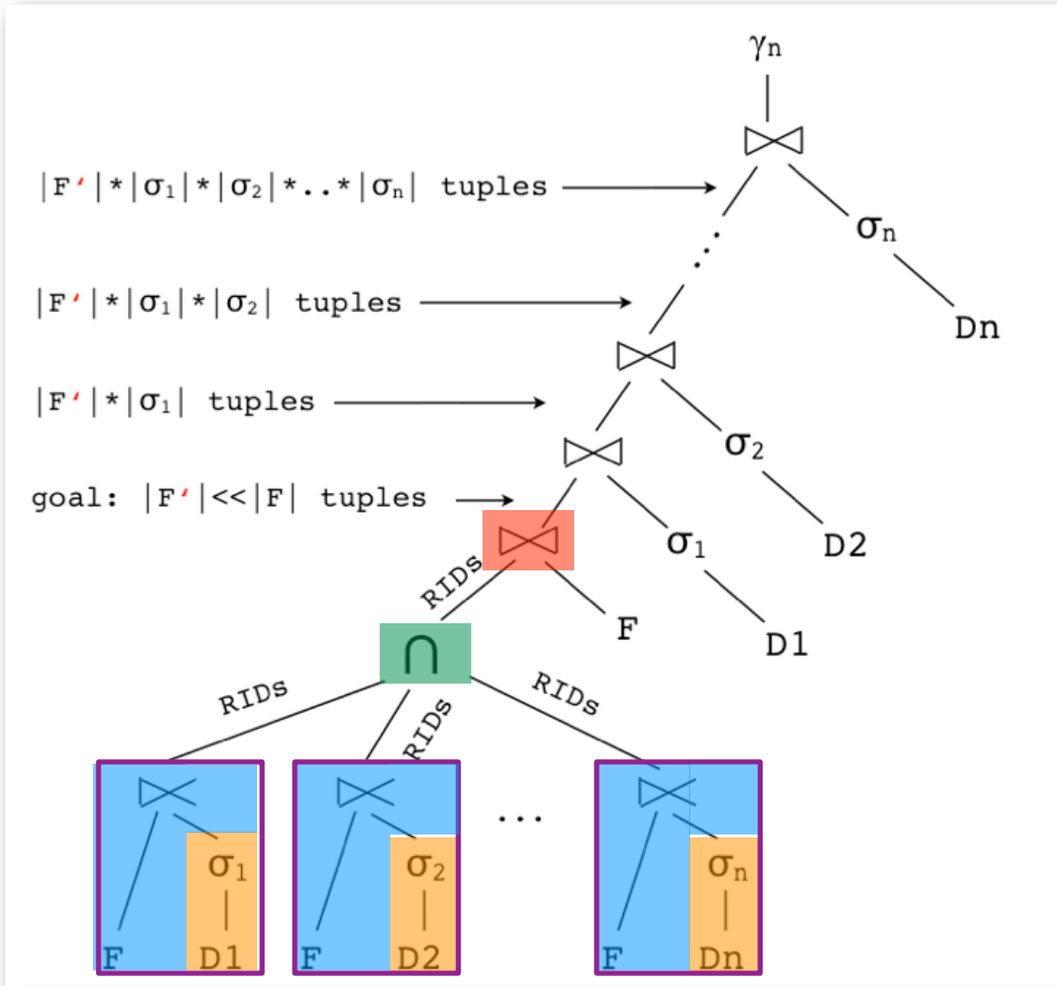
- Let‘s go back to the semi-join plan.
- Now we can use the same plan, but:
  - replace selections by value bitmaps
  - replace semi-joins by Type 2 bitmap join indexes
  - alternatively: replace selections and semi-join by Type 3 bitmap join indexes



- The intersection is implemented by a large merge (a logical AND) of the respective bitmaps.
- The result from that merge operation determines the tuples from F that have to be read to perform the actual join.
- The qualifying tuples can again be fetched from F using an indexed nested-loops join (actually another semi-join) accessing the index on F.

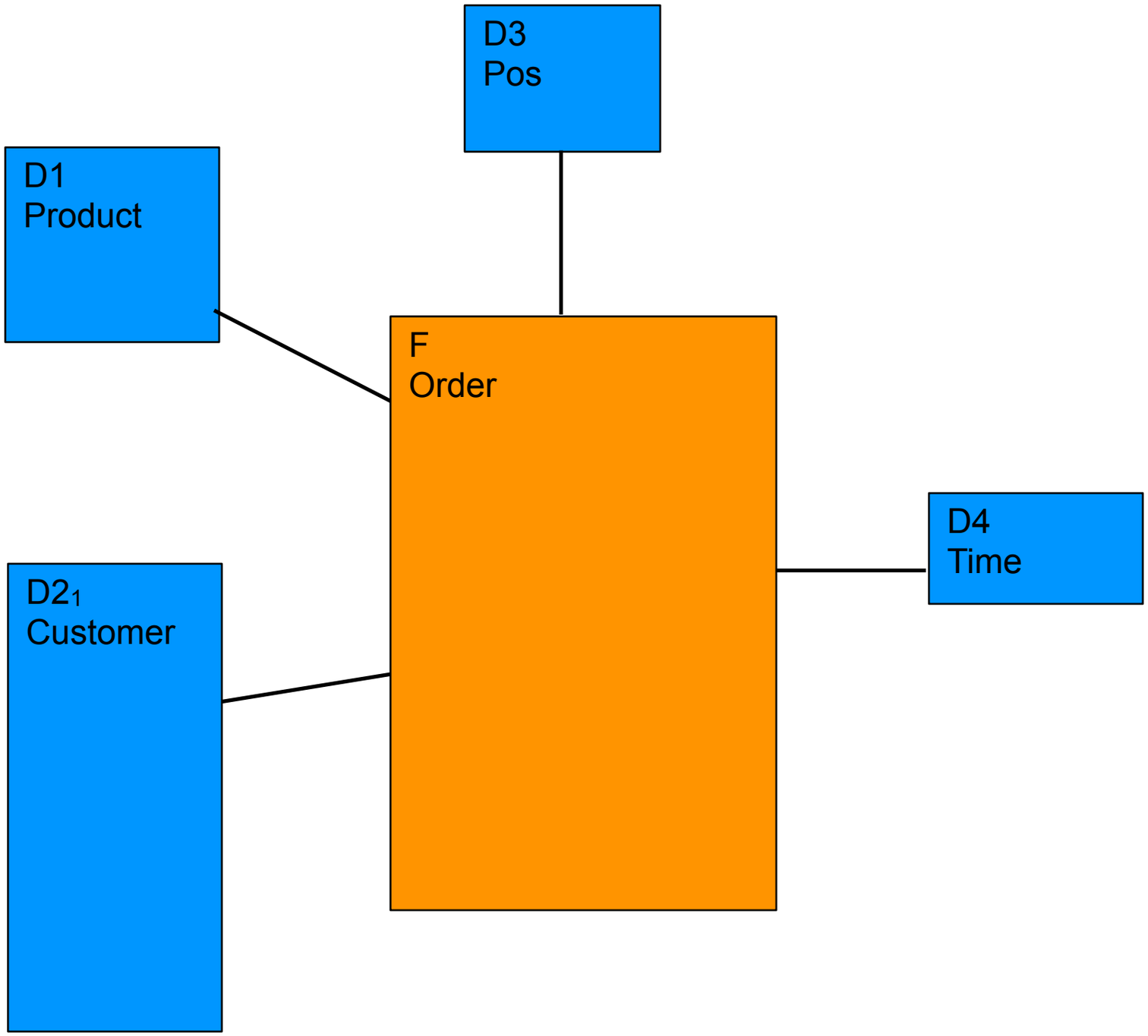
Star Join using Bitmap Indices

- This plan will be very efficient **if** the bitmaps help to reduce the number of tuples that have to be considered from F.
- In addition, the selectivity on the dimension tables should be high.
- Example
  - assume that each predicate has a selectivity of 0.01 (=1%)
  - if all N predicates are statistically independent of each other, we estimate an overall selectivity of 10^{-2N}
  - for N=3 (three predicates) and a fact table containing 10^8 tuples it follows that approximately only 100 RIDs `survive´ the bitmap operation.
  - fetching 100 rows independently using an index nested-loops join accessing the index on F should be faster than reading the entire table F
- However: we need a cost based optimizer to make this decision...

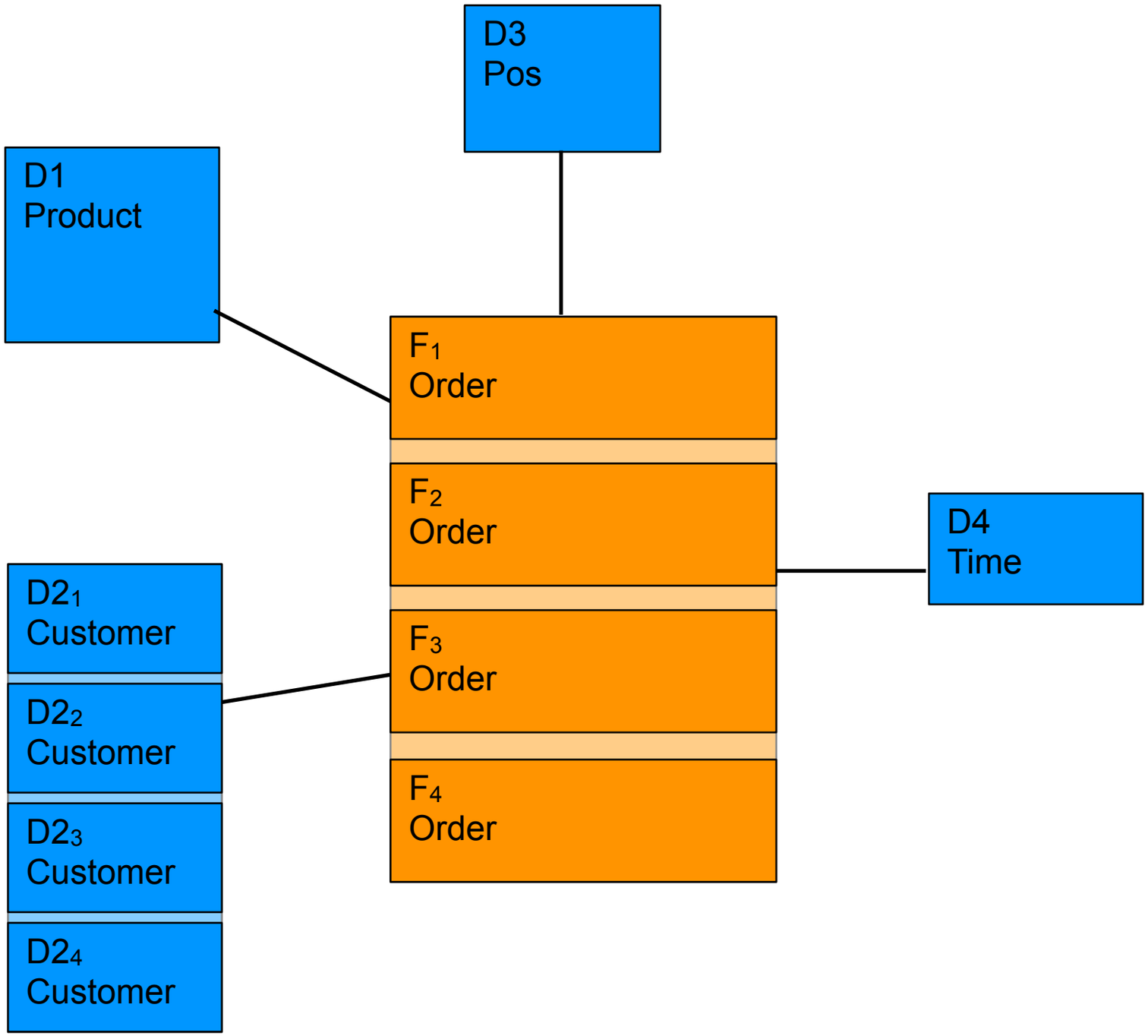


Footnote: Join Indexes in Parallel Databases.

Example Star Schema

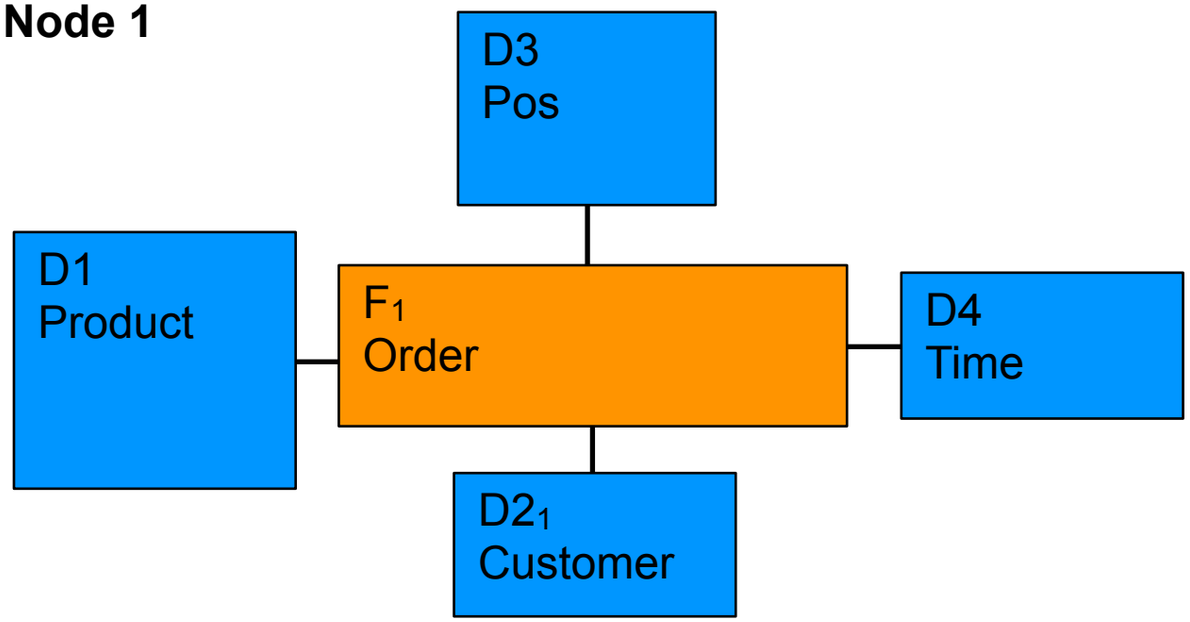


Step1: Partition fact table F and dimension table D2

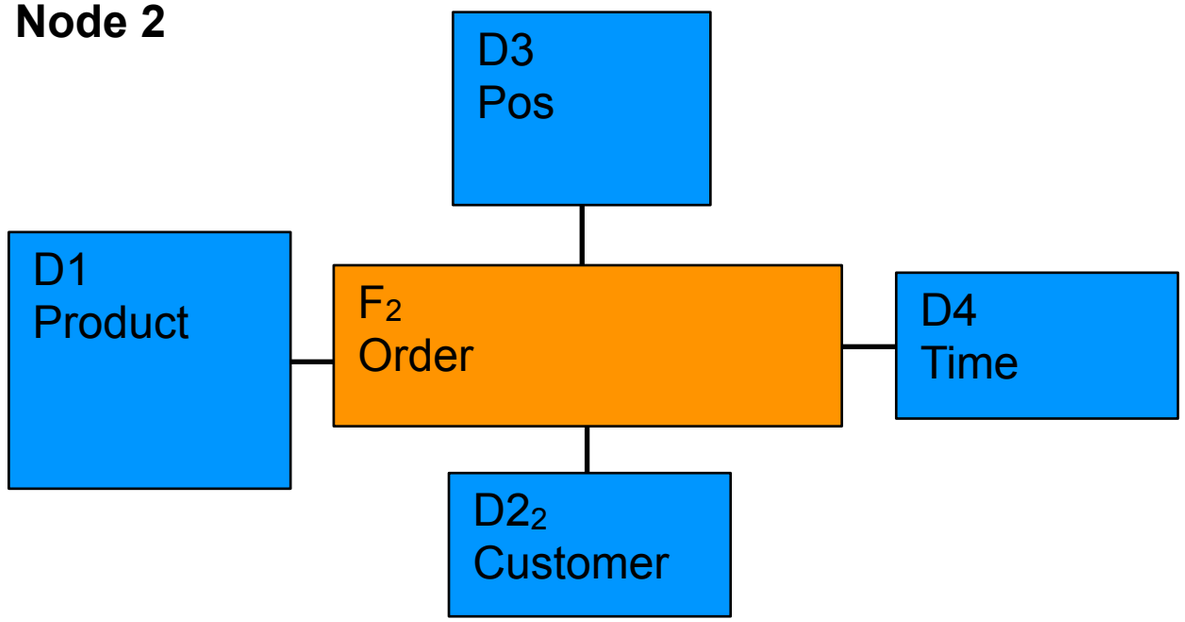


Step2: Replicate D1, D3 and D4 (see Week 11)

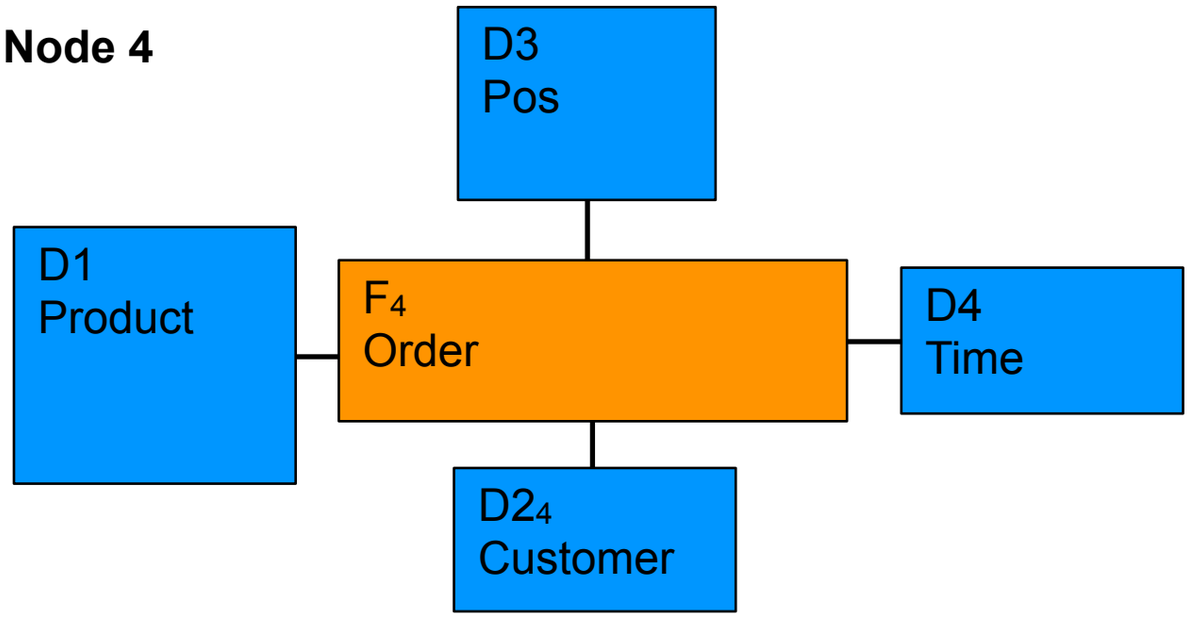
Node 1



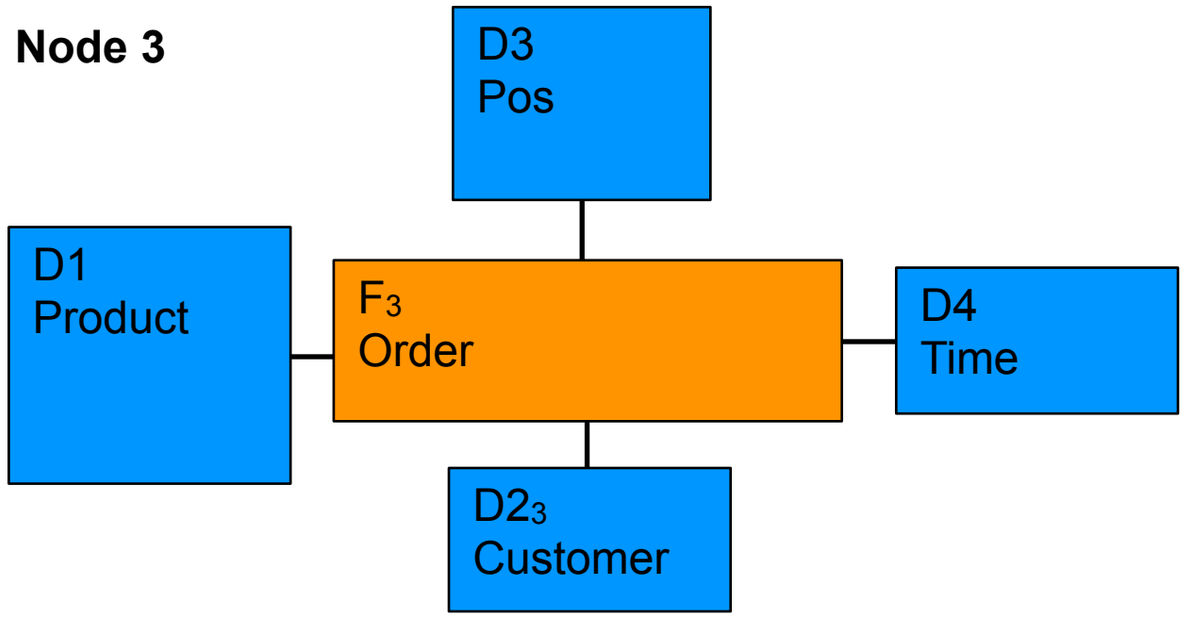
Node 2



Node 4



Node 3

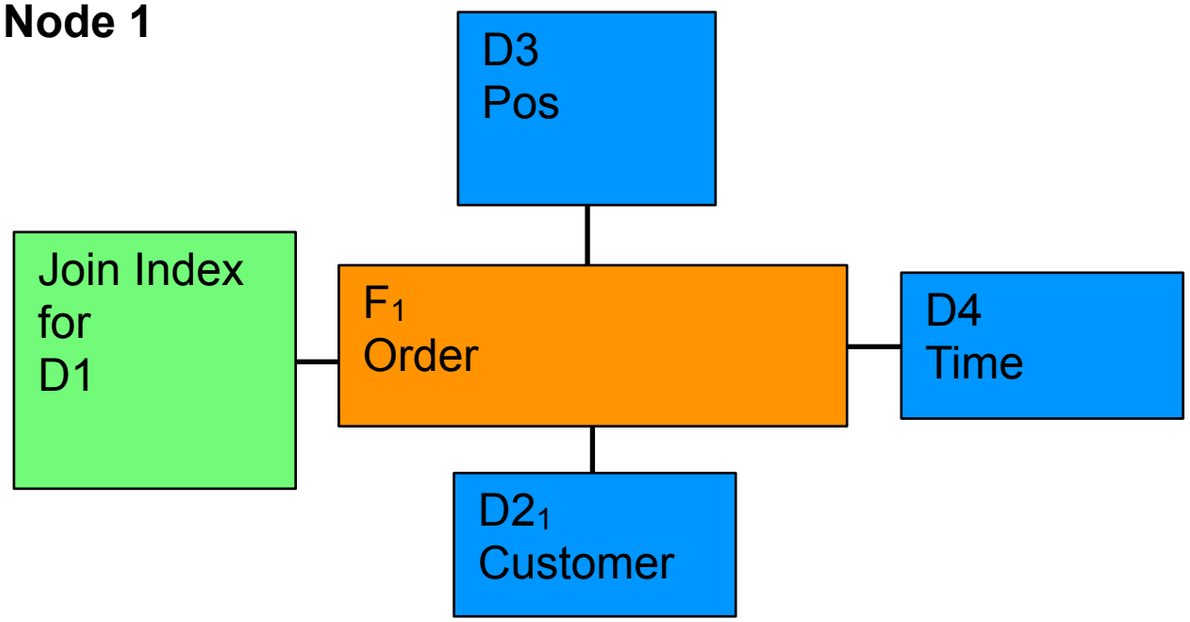


Discussion

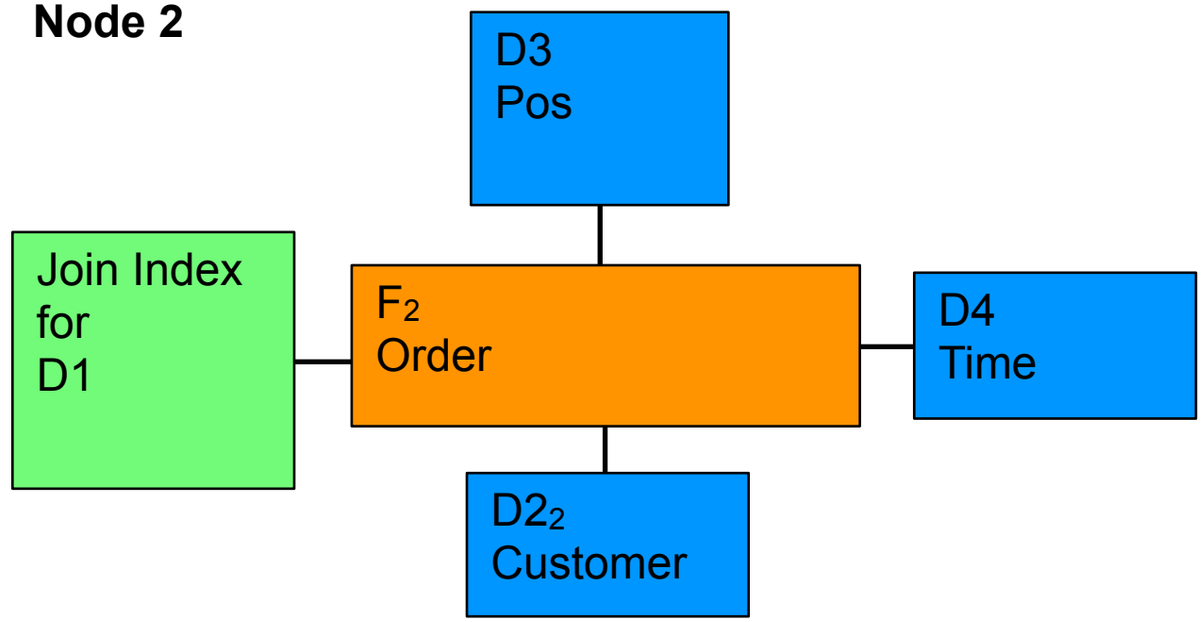
- Partitioning combined with replication is limited:
  - consider having **more than** one large dimension table, e.g., D1
  - consider having dimension tables that are **shared** among multiple star schemas (galaxy schema)
- Also:
  - check for functional dependencies
  - if FDs exist among dimensions, you should revise the schema
  - if that is not possible, FDs may be exploited for partitioning
- 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 would not have to send a bitmap
  - actual semijoin based on join index would be first step

First step: join locally

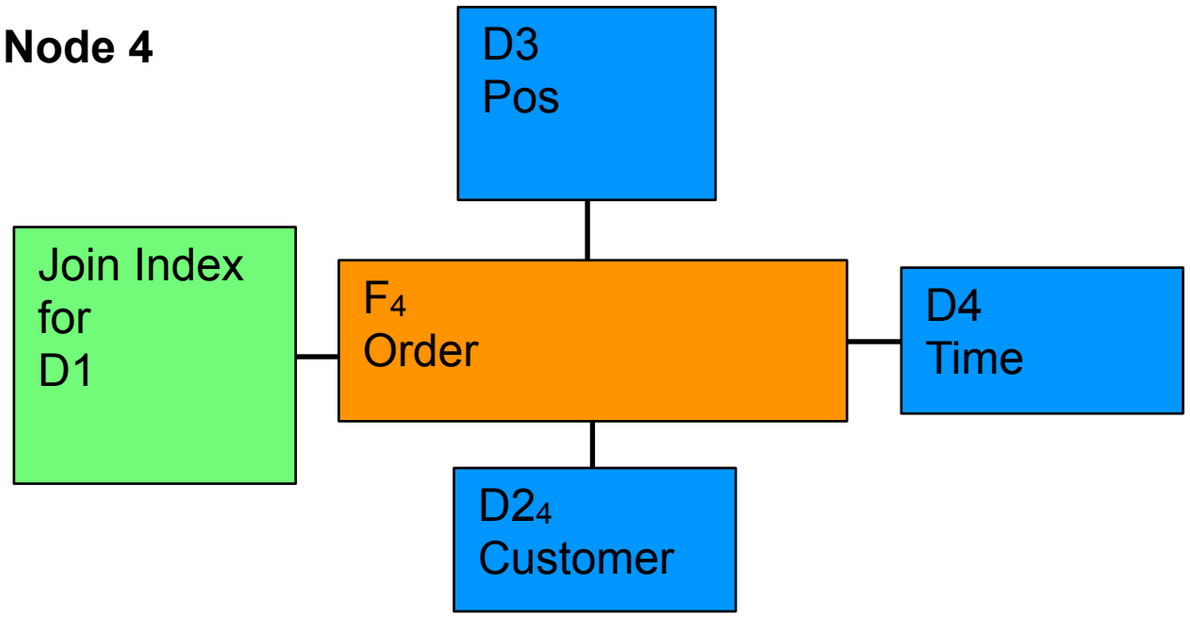
Node 1



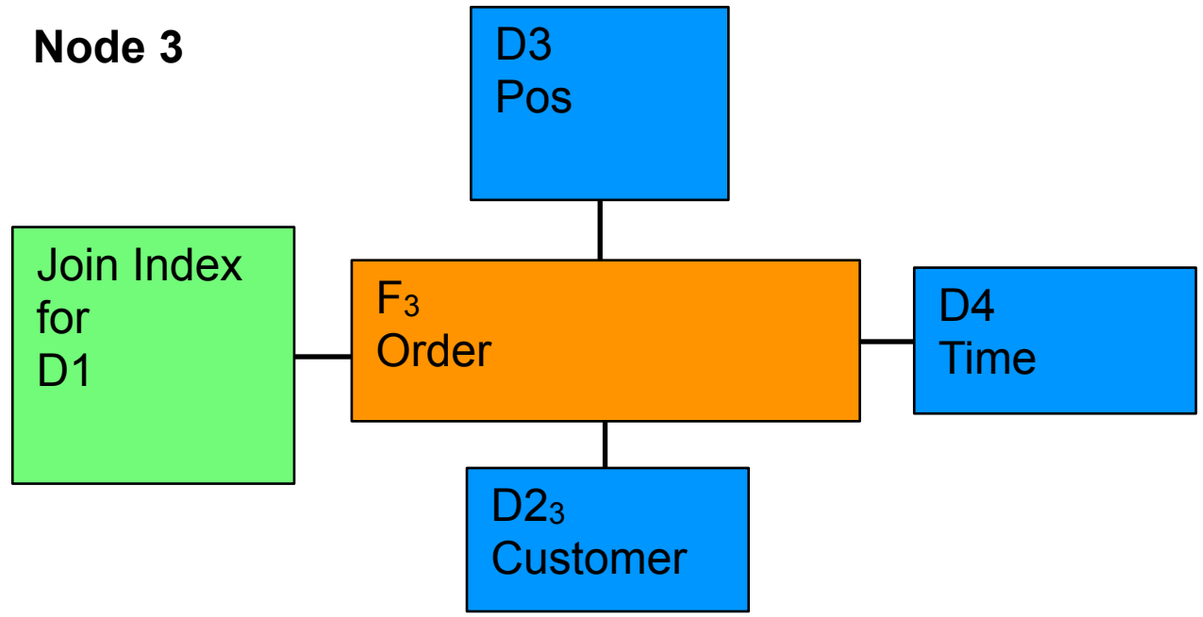
Node 2



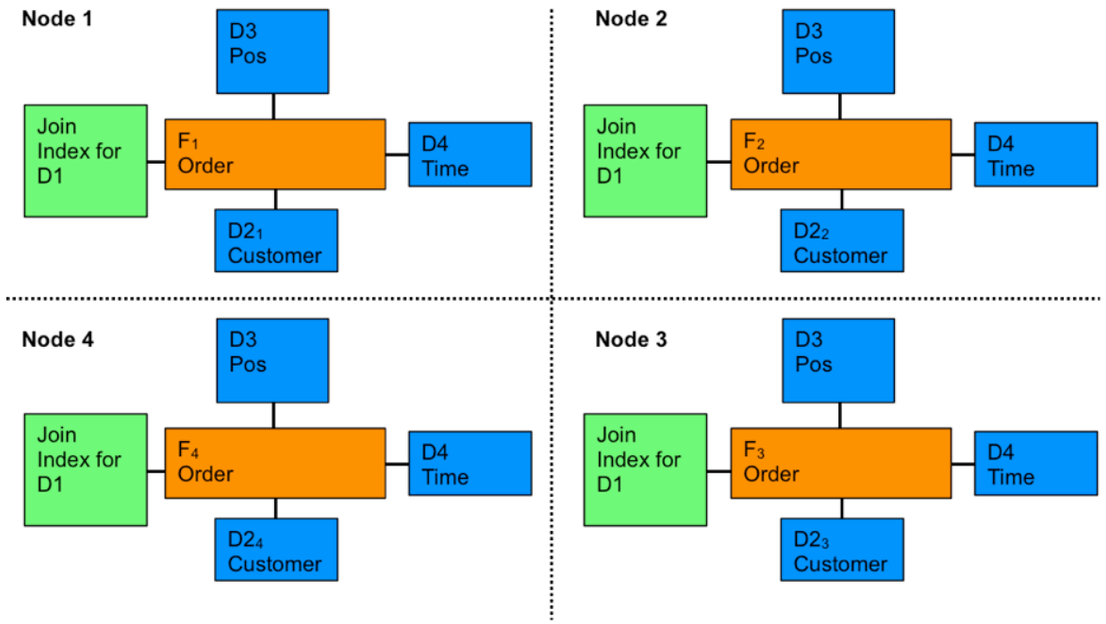
Node 4

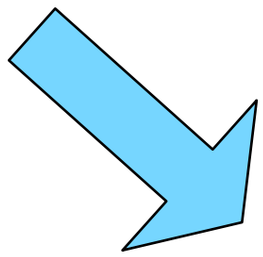


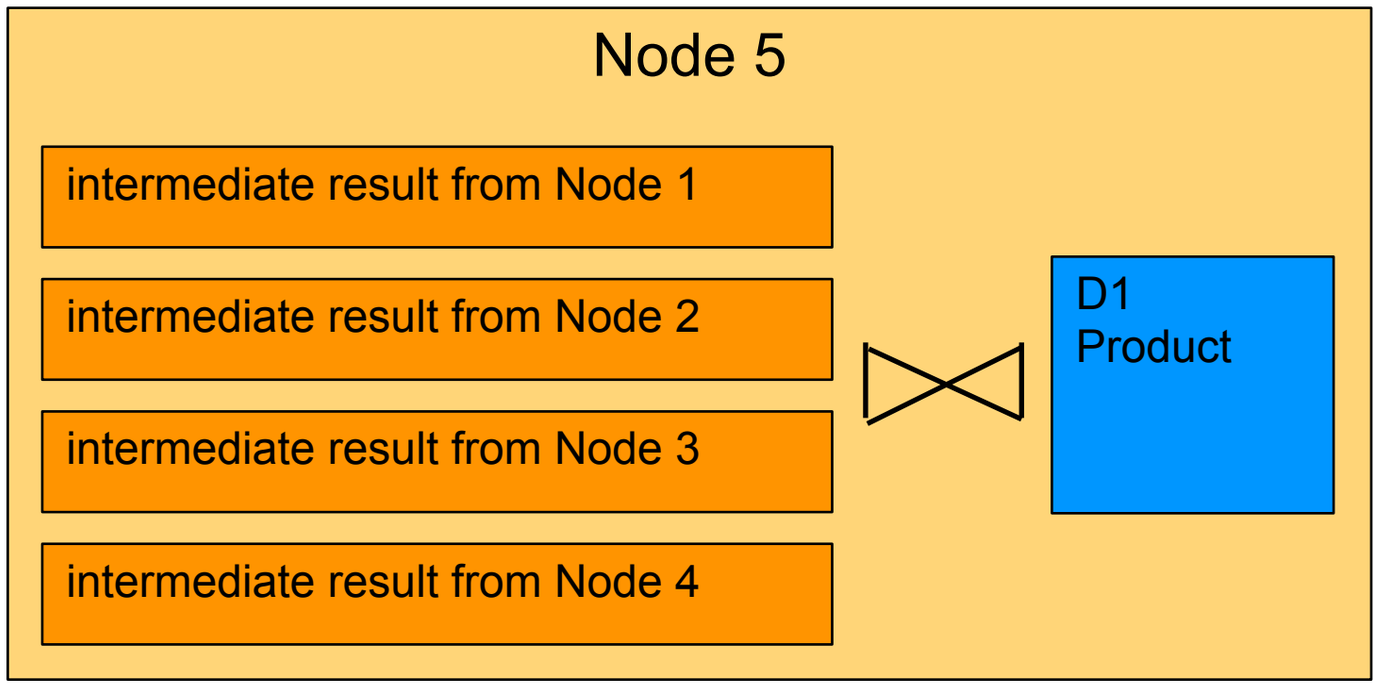
Node 3



Second Step: send intermediate Results to Machine storing D1







Replicated Join Index 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
- similar to replication approach
- difference: replace replicated D1 by join index!

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

How can a Star Join be further improved?

- Horizontal partitioning (range or hash)
  - Grace Hash Join, data distribution, PNUTS, map/reduce, etc.
- vertical partitioning
  - DSM, column stores, data distribution
- projection index
  - = “partial fractured mirrors“
- adjusting pagesize
  - discussion for B+-trees
- considering cache hit rates
  - storage, indexing, algorithms
- compression
  - storage layer compression, cache-optimized trees
- multi-dimensional indexing
  - discussion on VA-file

Horizontal Range Partitioning in Oracle

- Idea: for an attribute F.att$_i$ split F into partitions F$_i$ such that all elements in F$_i$ belong to a certain range
- Example: Partition by time
  - Assume F contains an attribute ‘year‘
  - create partitions F$_{year}$ such that F$_{year}$ contains tuples of that year
  - F$_{2004}$, F$_{2005}$, F$_{2006}$, F$_{2007}$, etc.
  - Any query specifying a time attribute may be restricted to consider only data in some of the partitions.
  - `SELECT ...`
    `FROM ...`
    `WHERE ... F.year=2006 ...`
    This query only has to consider data from partition F$_{2006}$
- Oracle SQL dialect: `PARTITION BY RANGE(year)`

Horizontal Hash Partitioning in Oracle

- Idea: for an attribute F.att$_i$ split F into partitions F_i such that all elements in F_i belong to a certain hash bucket
- the hash bucket is determined by computing
  partition := hash(<some key columns>) % number of partitions

- Example: Partition by hash
  - Assume F contains an attribute ‘year‘
  - create four partitions F_i such that F_i contains tuples of hash(year)
  - F_0, F_1, F_2, F_3
- Oracle SQL dialect: `PARTITION BY HASH(year) PARTITIONS 4;`
- Horizontal partitioning also useful for intra-query parallelism if parallel processing possible

Projection Index

- What should we take: DSM, NSM, or projection index?
- Idea
  - stick with NSM
  - **but**: create additional **redundant** projections on single attributes
  - **not** redundant columns for **all** attributes (as in fractured mirrors)
    - projections represent vertical partitions of DSM
- Projection Index may be used if few values have to be read, otherwise the NSM representation will be used
- Literature: O'Neil and Quass 97

Projection Index Details

- assume `foo´ is a column of F
- the projection index then consists of the sequence of column values from `foo´ ordered by RID (holes may exist for unused row numbers)
- if column `foo´ is 4 bytes in length, then we can fit 1000 values from `foo´ on a single 4 KByte disk page
- for any given row numer n = m(r) in F we can access the proper disk page and slot to retrieve the appropriate `foo´ value with a simple calculation:

  page = n / 1000
  slot = n % 1000

- the row number of a given page and slot is calculated as:

  n = 1000 * page + slot

Bit-sliced Index

Fact table

F

foo
3
1
4
10
0
1
5
11
5
12
6

binary representation (projection index)

foo
0...000011
0...000001
0...000100
0...001010
0...000000
0...000001
0...000101
0...001011
0...000011
0...001010
0...000110

bit-sliced index

foo3	foo2	foo1	foo0
0	0	1	1
0	0	0	1
0	1	0	0
1	0	1	0
0	0	0	0
0	0	0	1
0	1	0	1
1	0	1	1
0	0	1	1
1	0	1	0
0	1	1	0

- idea: only use as many bits as needed for the doain of the attribute
- for this example only 4 bits = 1/8 of the original space for a PI is required
- for FTS on vertical partition this translates roughly into **times 8** performance improvements!
- Assuming 100 million rows in a projection index this is 50 MB versus 400 MB to read
- versus possibly dozens of gigabytes for reading F

Bit-sliced Index: Aggregation Queries

Fact table

binary representation (projection index)

bit-sliced index

F

foo
3
1
4
10
0
1
5
11
5
12
6

foo
0...000011
0...000001
0...000100
0...001010
0...000000
0...000001
0...000101
0...001011
0...000011
0...001010
0...000110

foo3	foo2	foo1	foo0
0	0	1	1
0	0	0	1
0	1	0	0
1	0	1	0
0	0	0	0
0	0	0	1
0	1	0	1
1	0	1	1
0	0	1	1
1	0	1	0
0	1	1	0

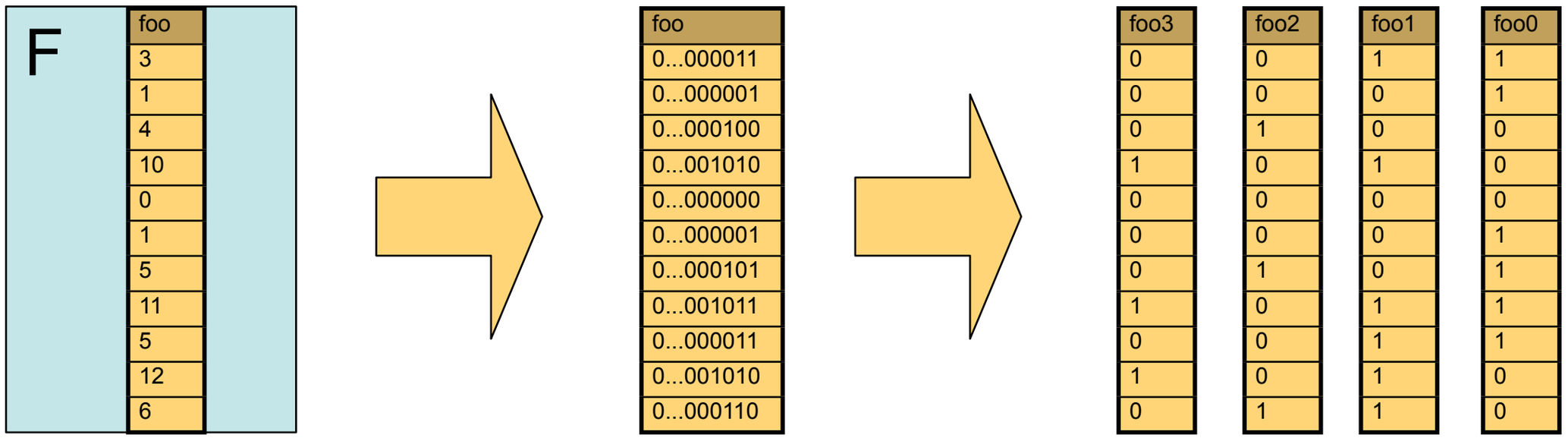
- assume foo is a measure
- let‘s compute the sum
- this could be done as follows:
- $\sum_{i=0}^{3} 2^i * (\#\text{bits in foo<i> set to true})$ (true = “1“)
- => no need to reconstruct individual integers
- => no need to merge bit slices

Bit-sliced Index: Predicates

Fact table

binary representation (projection index)

bit-sliced index



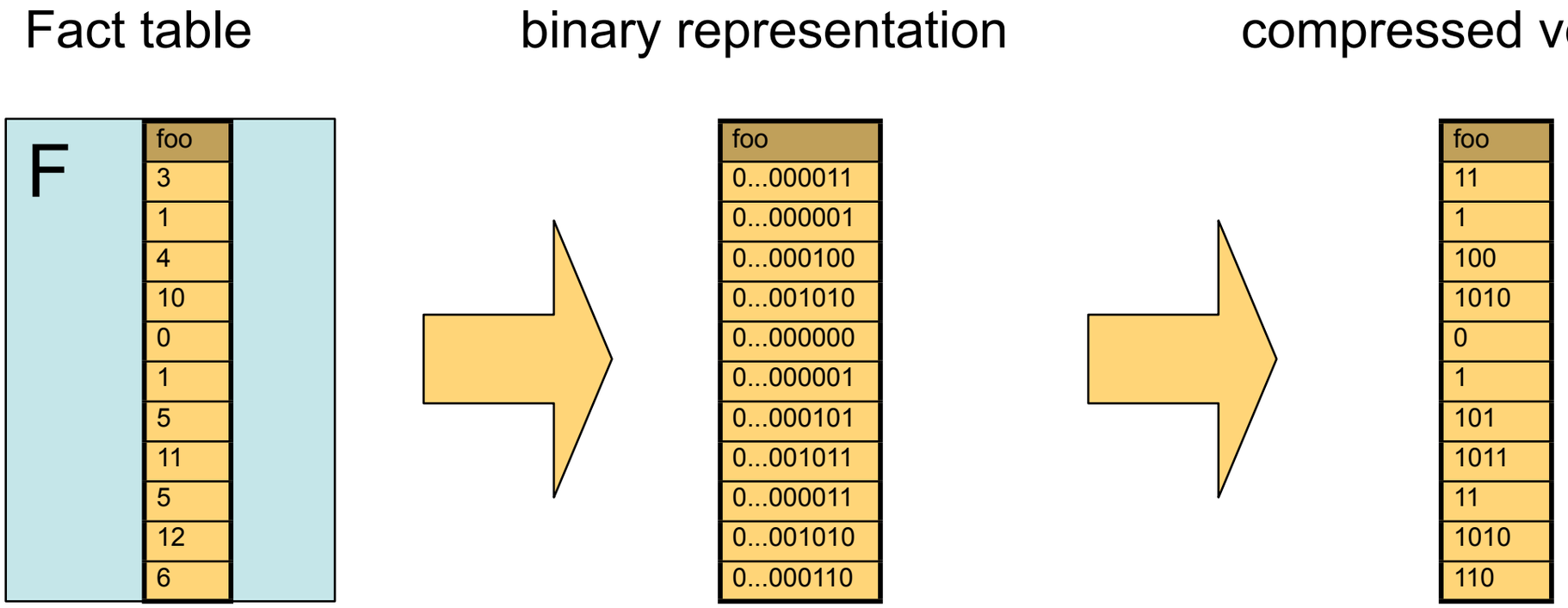
- assume foo is a foreign key of a dimension or a dimension value
- select all odd values = all rows that have a bit set in foo0
- => no need to read bit slices foo1 to foo3
- select all values ≥ 8 <=> all rows that have a bit set in foo3
- => no need to read bit slices foo0 to foo2
- select all values ≥ 12 <=> all rows that have a bit set in foo3 and foo2
- => no need to read bit slices foo0 to foo1

Bit-sliced Index

- Assumptions
  - `foo´ is a column of F of a countable number type
  - domain of numbers is limited, all numbers in 1,...,N
- To represent 1,...,N numbers we need n= $\lceil \log_2 N \rceil$ bits
- Idea
  - for column `foo´ define a bit list D(RID,i) for each bit of each value
  - D(RID,0) = true $\Leftrightarrow$ 2^0 bit of value `foo´ of row RID is set
  - D(RID,1) = true $\Leftrightarrow$ 2^1 bit of value `foo´ of row RID is set
  - ...
  - D(RID,i) = true $\Leftrightarrow$ 2^i bit of value `foo´ of row RID is set
  - ...
  - D(RID,n) = true $\Leftrightarrow$ 2^n bit of value `foo´ of row RID is set
- Literature: O'Neil and Quass, SIGMOD 97

Compressed Vertical Partitions

- Alternative: instead of cutting a vertical partition into bit-slices, compress it directly into a single compressed vertical partition



- similar effect as bit-sliced partitions
- codes for individual row may have different size
- however, no gain for selections
- many different encoding schemes exist for compressing (see Managing Gigabytes book)

Pagesize

- OLTP works best with small page sizes
  - space- and update-efficient system
  - may require lots of seek operations
  - high data fragmentation
- OLAP works best with large page sizes
  - query-efficient system
  - OLAP queries may require large amounts of data to be read to answer a query
  - avoid seek operations whenever possible
  - low data fragmentation

MOLAP: Multidimensional OLAP

- non-relational approach to OLAP
- Core Idea:
  - Do not use relational DBMS for query processing
  - use a multi-dimensional system
  - invent clever index structures that
    - materialize (almost) all aggregates at index time
    - store aggregates in a highly compressed manner
  - at query time just look-up aggregate values or do light-weight post-aggregation

MOLAP: Discussion

- Fight in industry MOLAP versus ROLAP
- Some companies offer MOLAP based tools: Essbase (Hyperion -> Oracle), MS Analysis Services
- companies do not tell people how they actually do it...
- reports on problems with many dimensions, high cardinalities and unexpectedly long indexing times
- therefore some companies favor HOLAP: Hybrid OLAP, i.e., part of the data is in ROLAP another part in MOLAP
- Recommendation
  - be careful
  - given the current hardware charecteristics it is questionable whether MOLAP has a business case if the techniques taught in this lecture are well applied

Summary

- Several methods to speed-up OLAP queries were presented
  - star join plans
  - bitmaps
  - join-indexes
  - mat views
  - partitioning
  - vertical and horizontal partitioning
  - compression
  - cache awareness and obliviousness
  - page size
  - data layout
  - multidimensional databases
- Applying all these techniques well gives excellent OLAP query performance even on TB data warehouses.
- Not all vendors provide all features.

Literature

- Don S. Batory: On Searching Transposed Files. ACM Trans. Database Syst. 1979.
- George P. Copeland, Setrag Khoshafian: A Decomposition Storage Model. SIGMOD Conference 1985: 268-279
- Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, Marios Skounakis: Weaving Relations for Cache Performance. VLDB 2001.
- Patrick E. O'Neil, Goetz Graefe: Multi-Table Joins Through Bitmapped Join Indices. SIGMOD Record 24(3): 8-11 (1995)
- Patrick E. O'Neil, Dallan Quass: Improved Query Performance with Variant Indexes. SIGMOD Conference 1997: 38-4
- Chee Yong Chan, Yannis E. Ioannidis: Bitmap Index Design and Evaluation. SIGMOD Conference 1998: 355-366
- Clark D. French: ``One Size Fits All'' Database Architectures Do Not Work for DDS. SIGMOD Conference 1995: 449-450
- Clark D. French: Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. ICDE 1997: 194-198

Literature

- Ian H. Witten, Alistair Moffat, Timothy C. Bell: Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition. 1999, Morgan Kaufmann