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

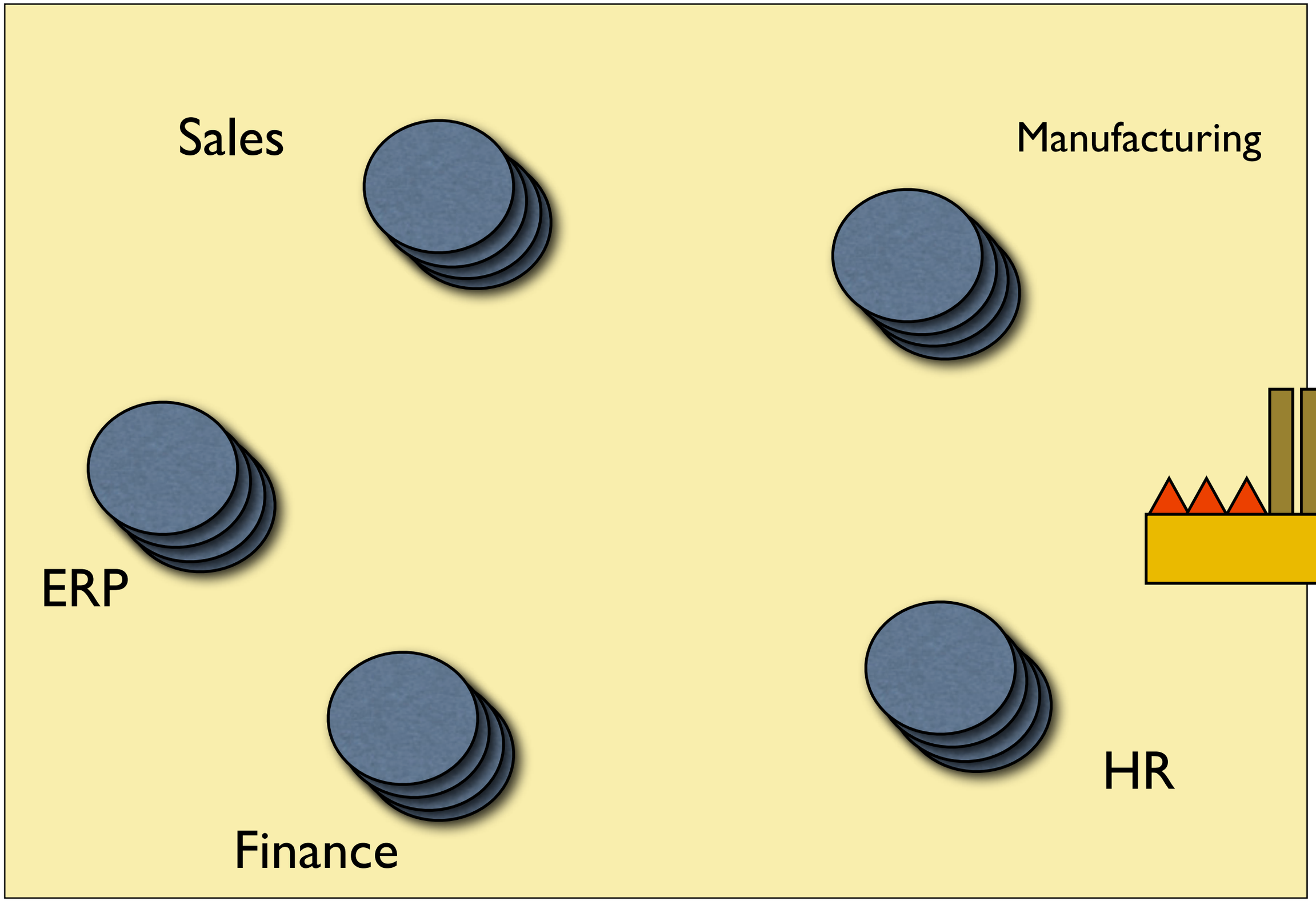
Data Warehousing and OLAP

Introduction.

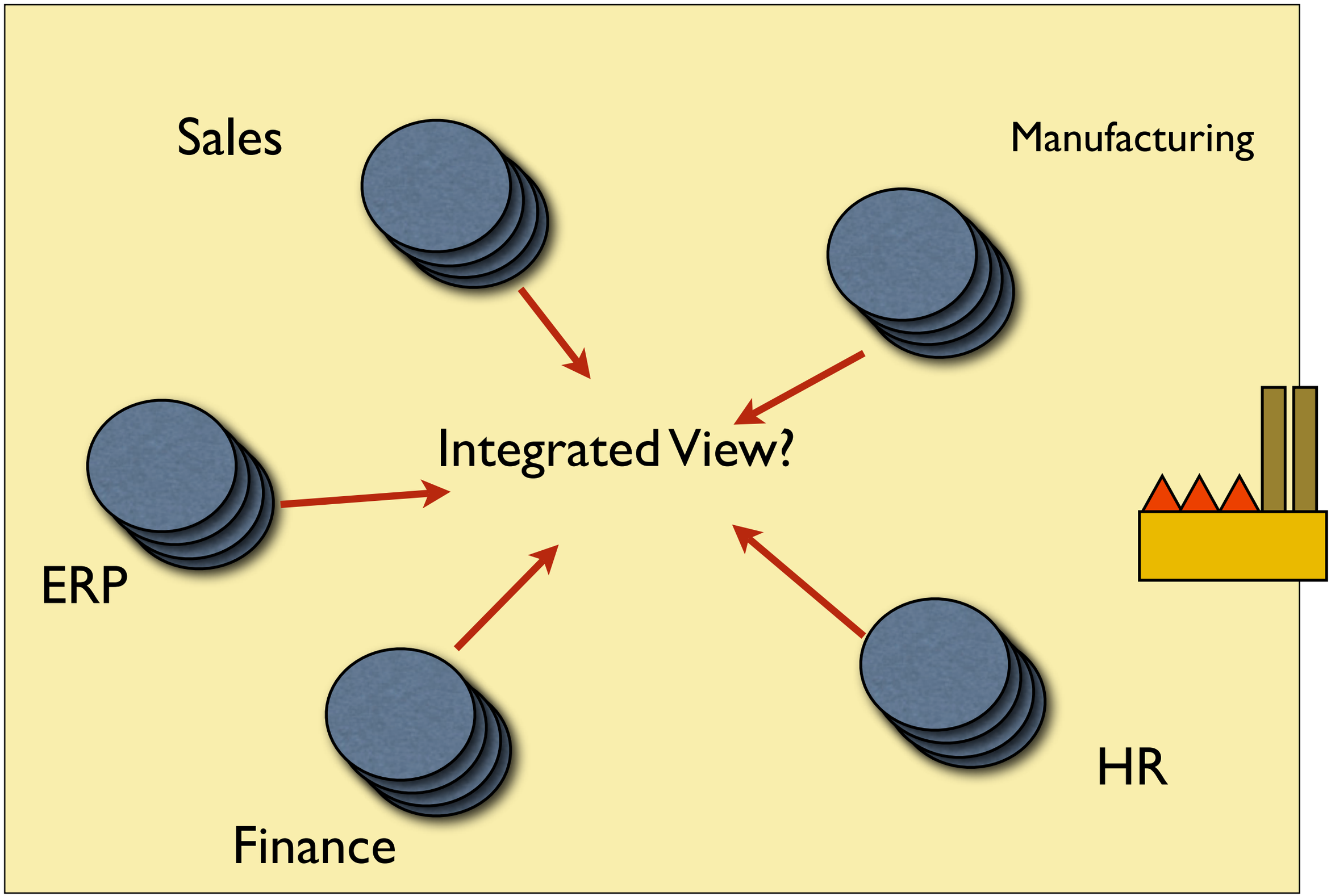
Query Mediation versus Data Warehousing

- Common Problem: Data integration
- Solution 1: Mediation (Informaton integration systems)
- Solution 2: Materialization (Data Warehousing)
- Important Trade-offs

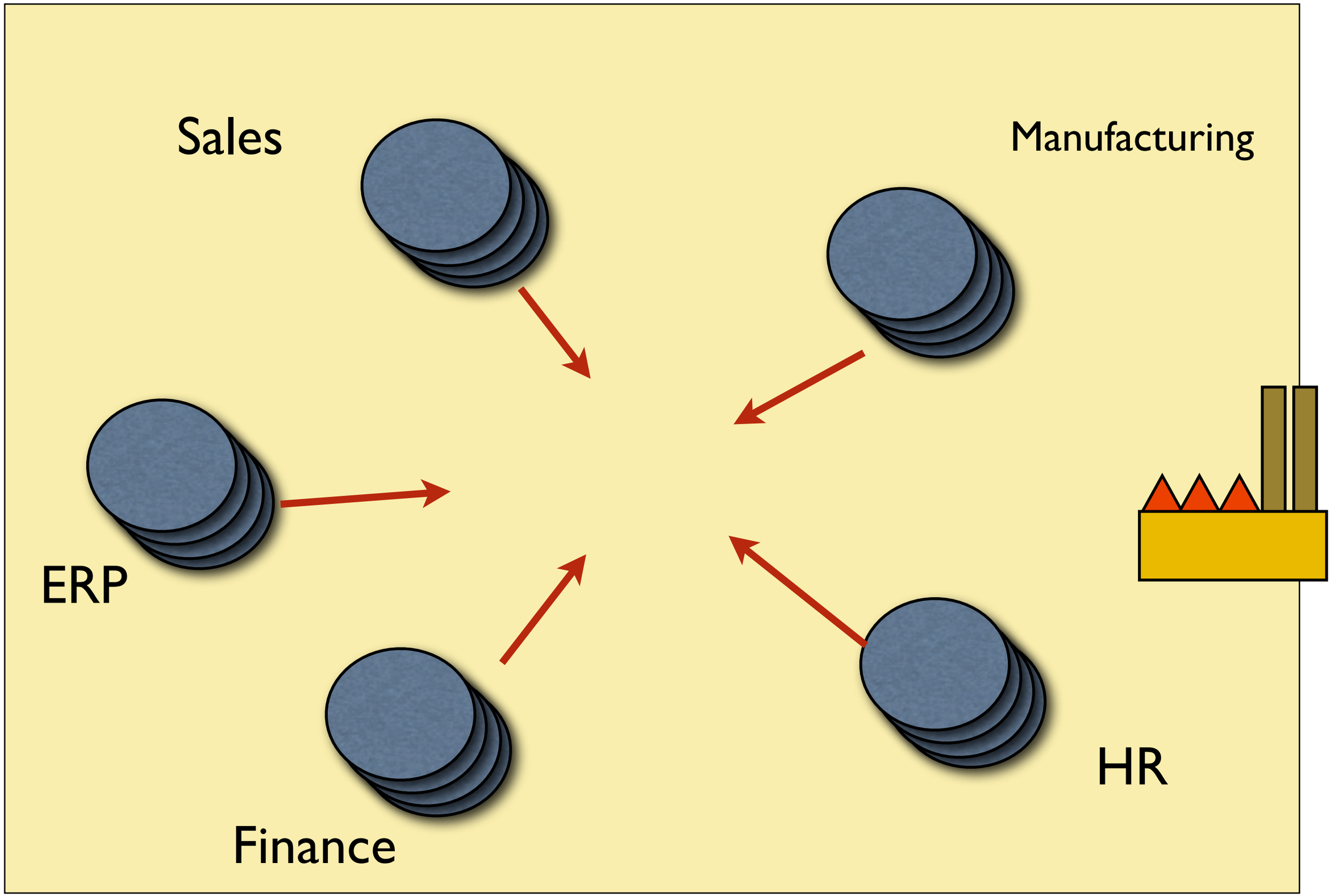
Problem: Data Integration



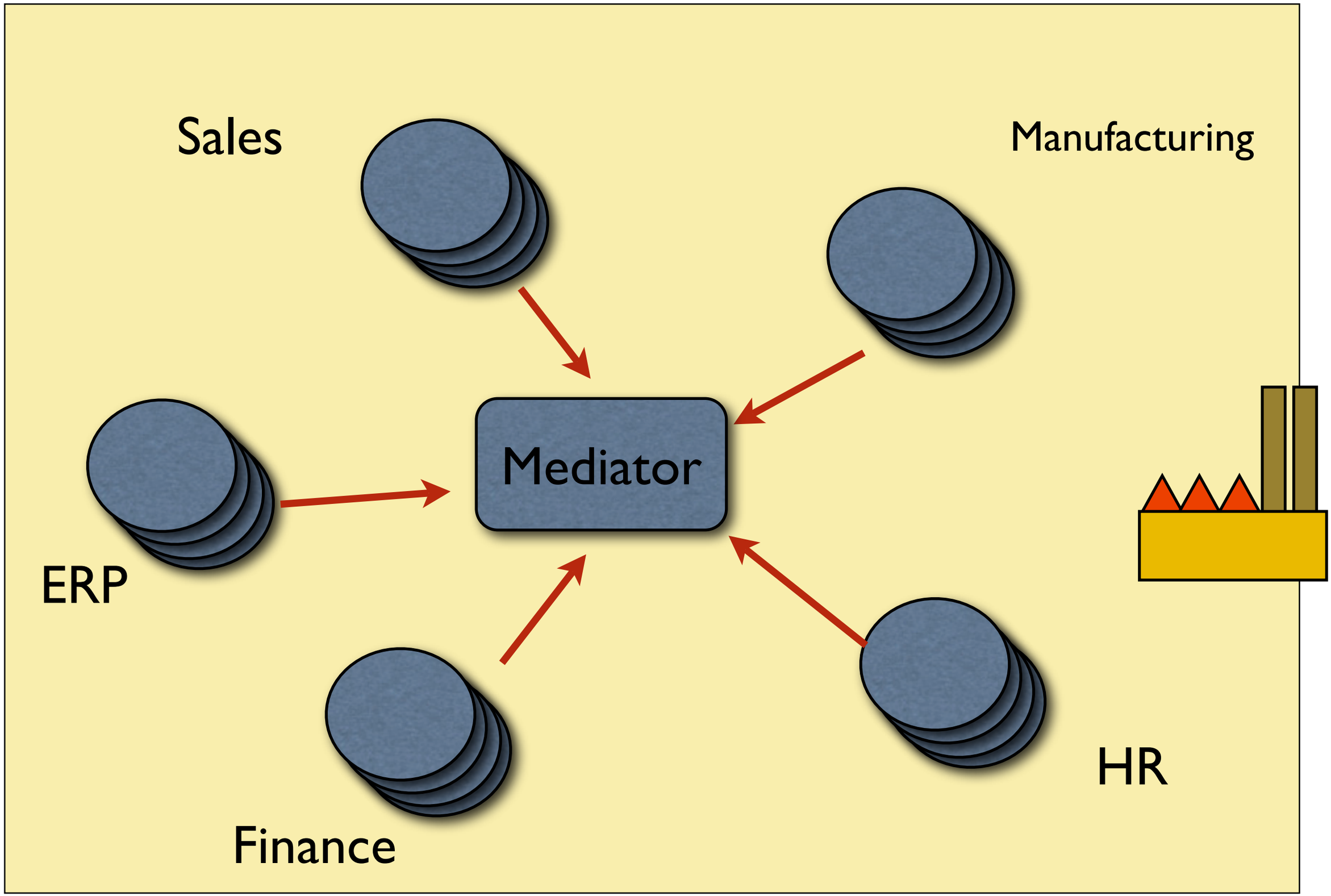
Problem: Data Integration



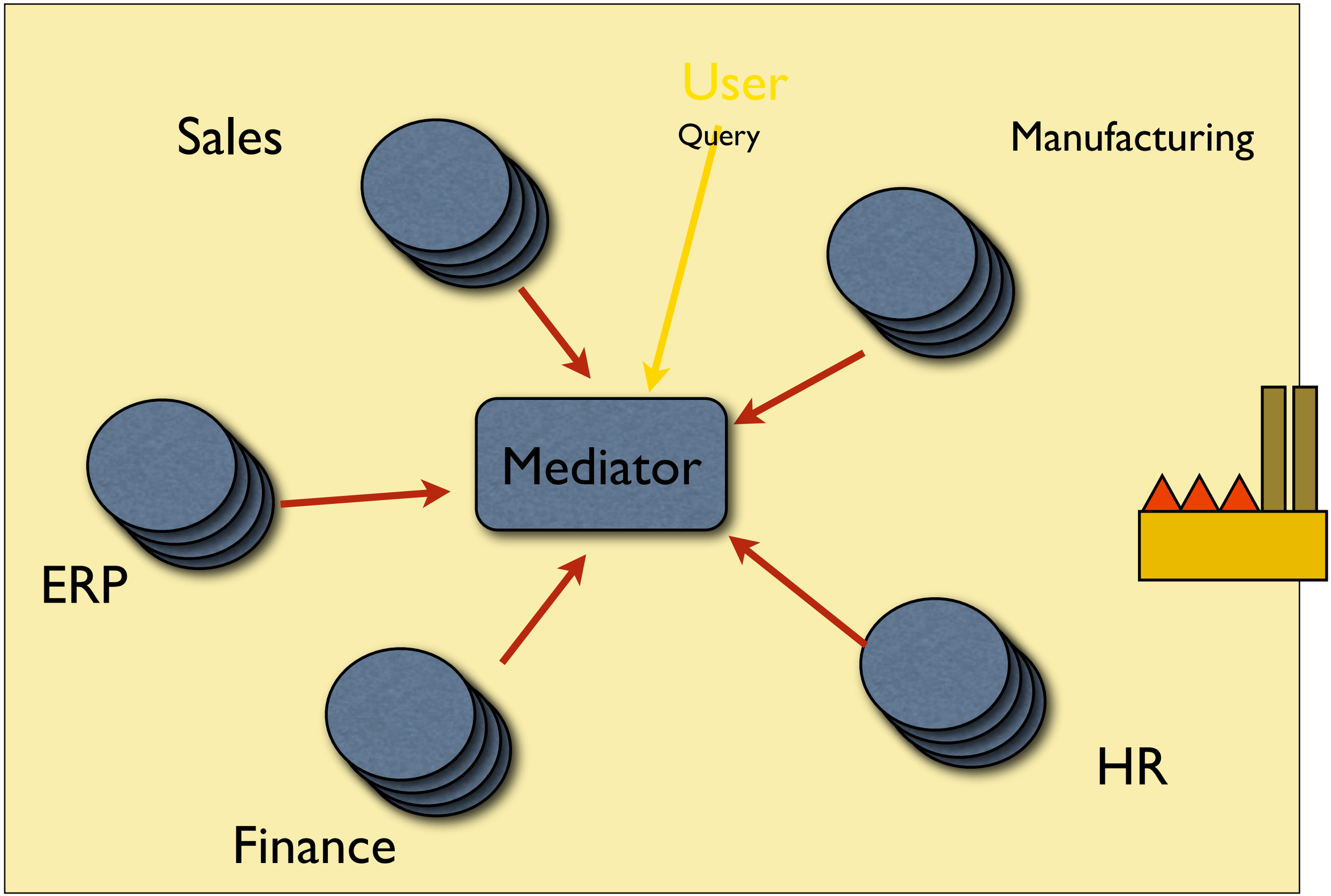
Solution 1: Mediation



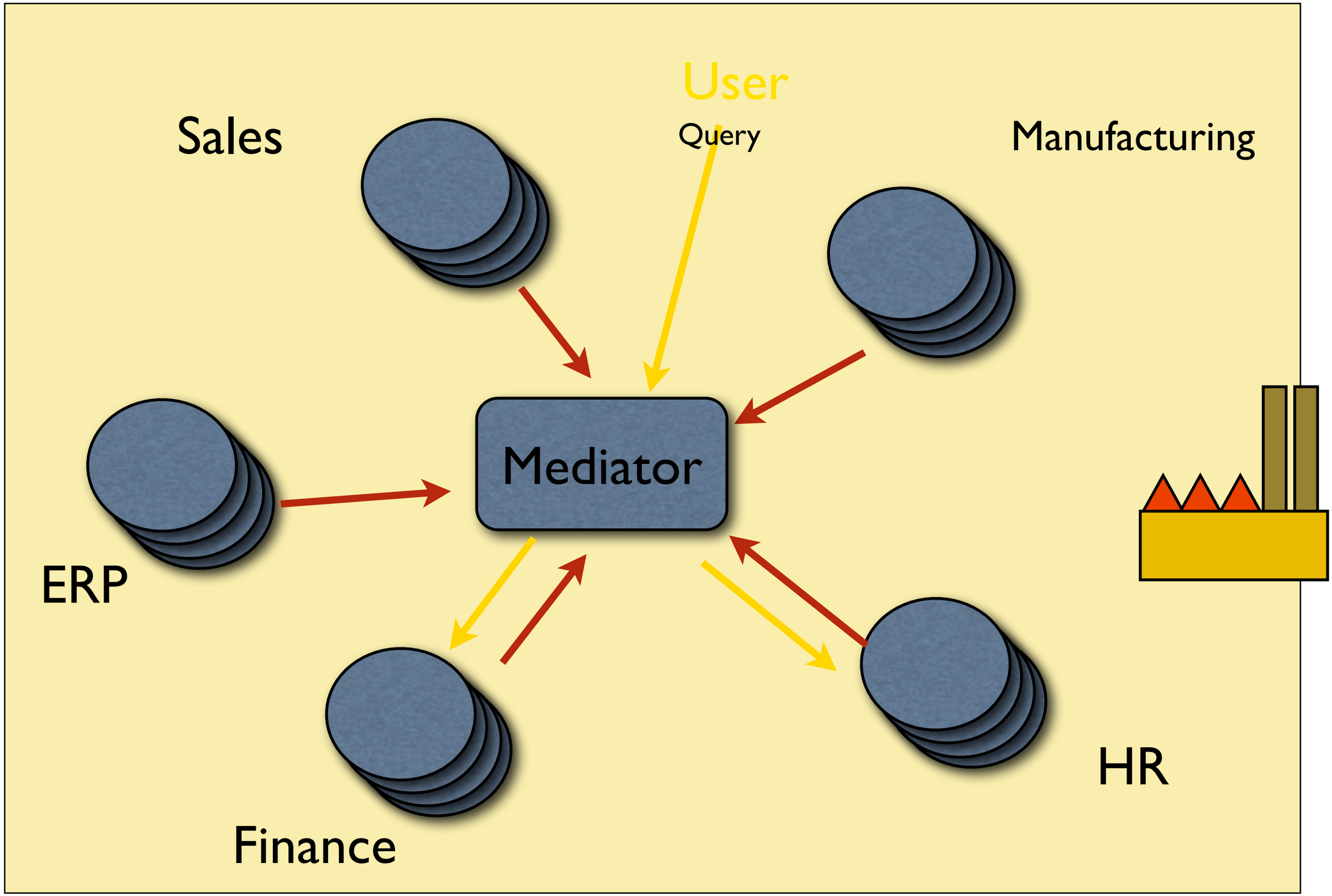
Solution 1: Mediation



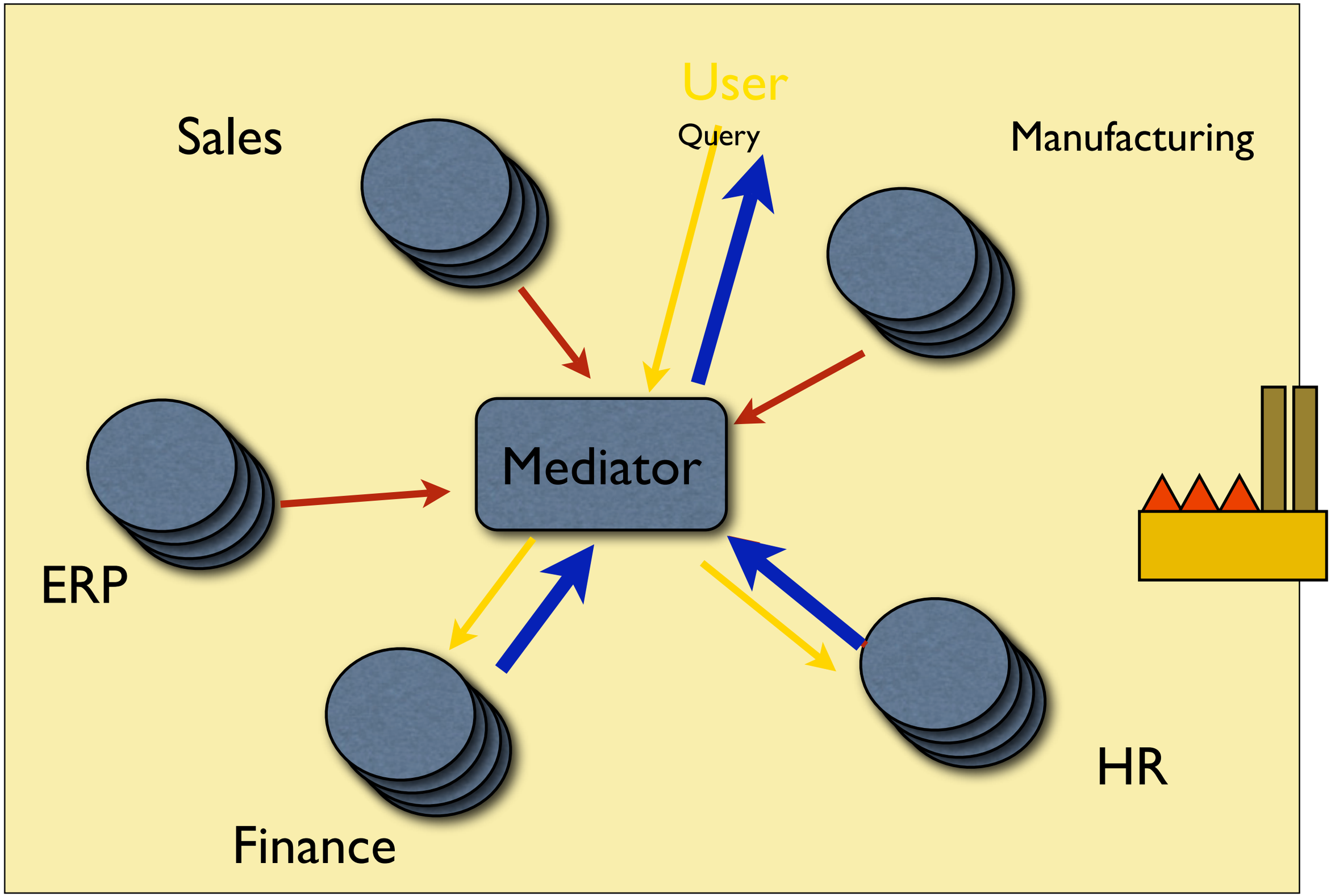
Solution 1: Mediation



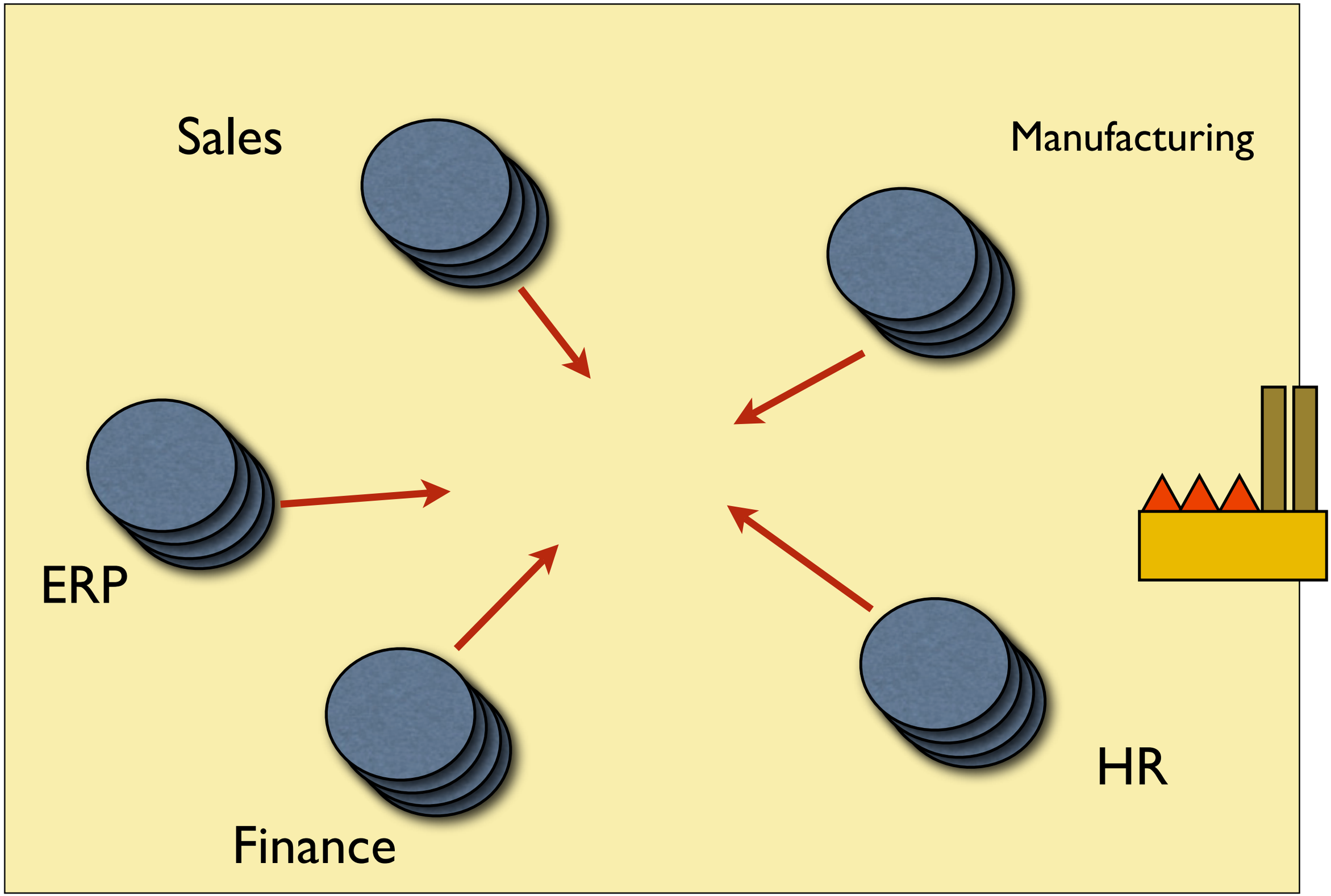
Solution 1: Mediation



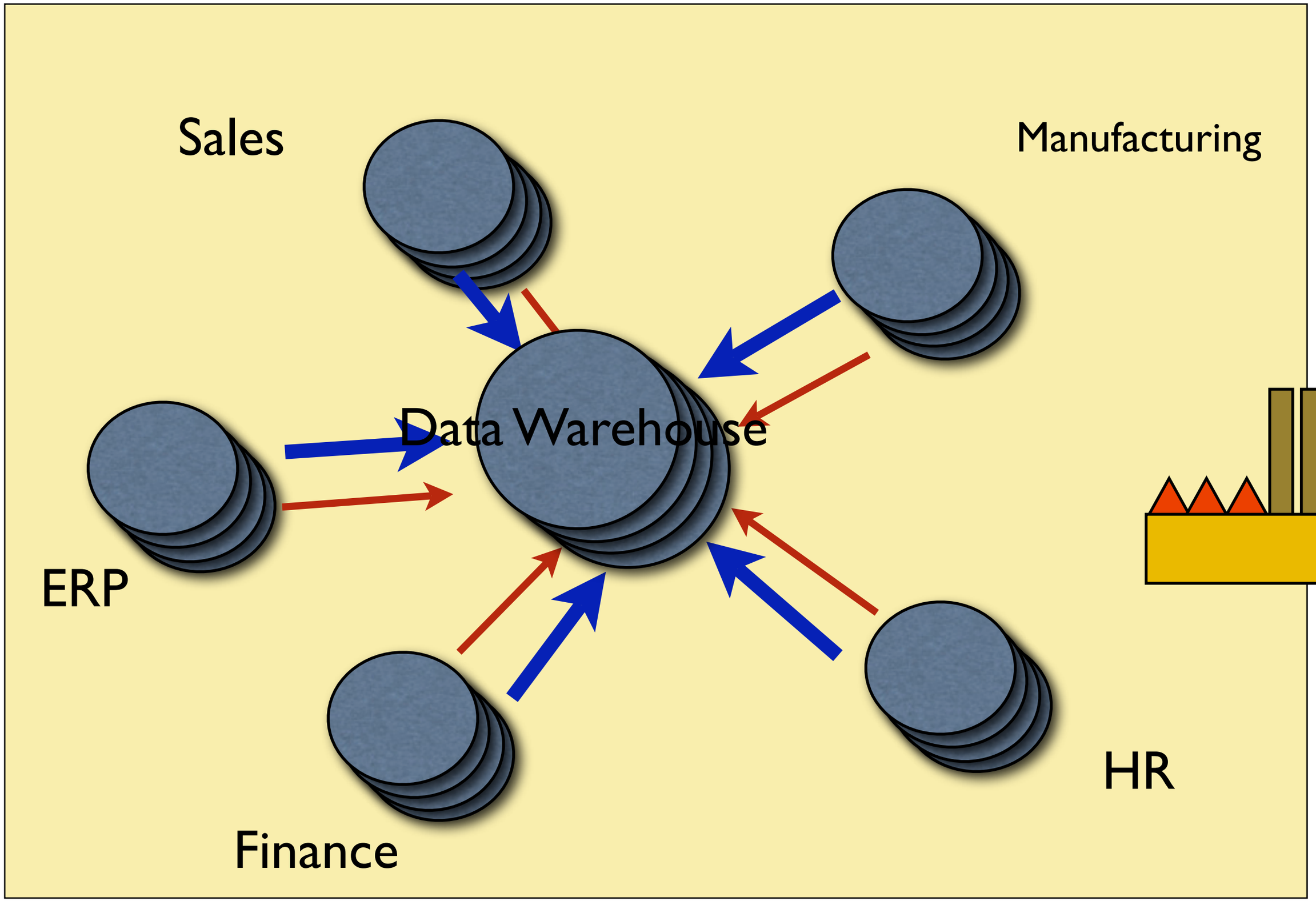
Solution 1: Mediation



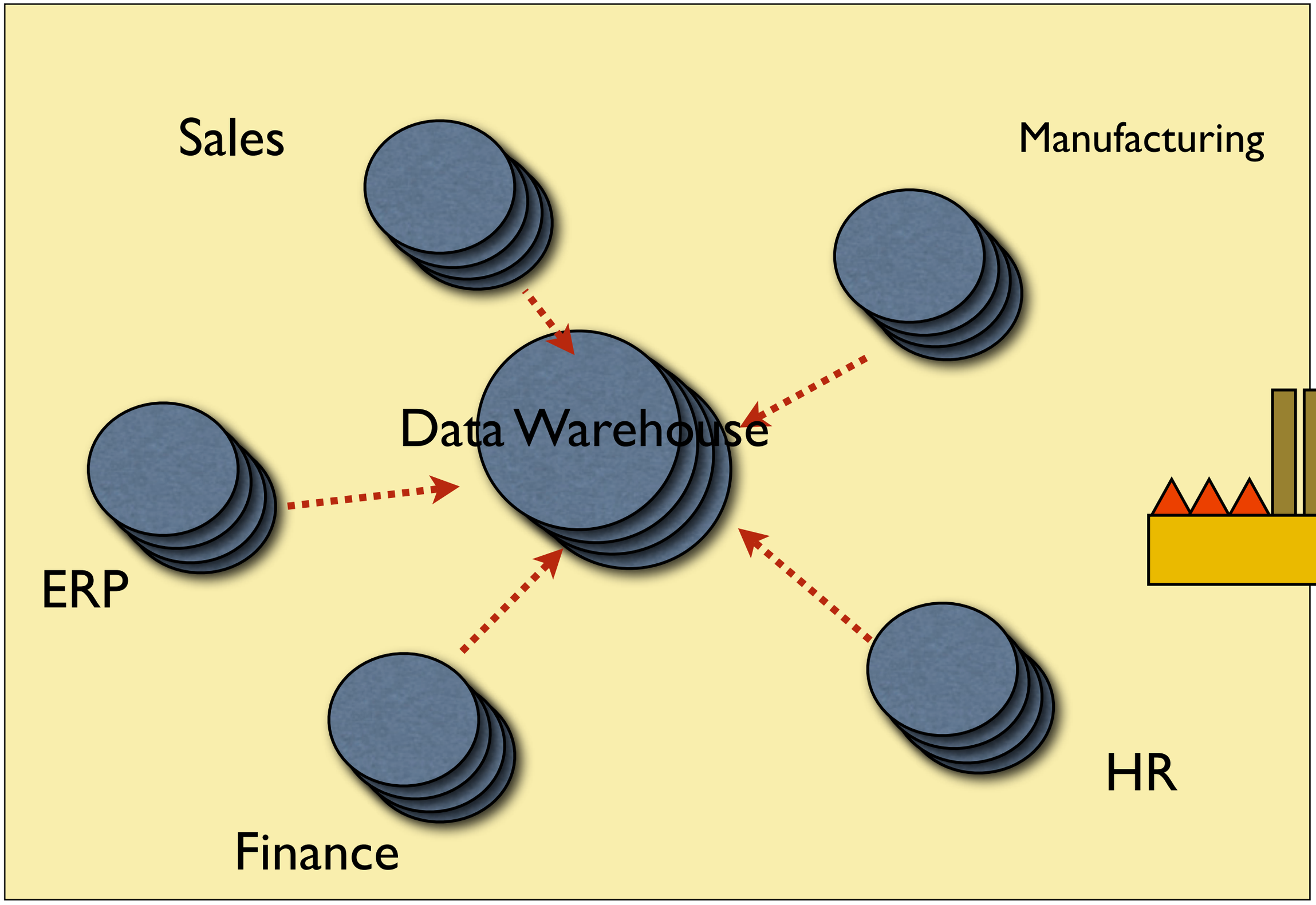
Solution 2: Materialization



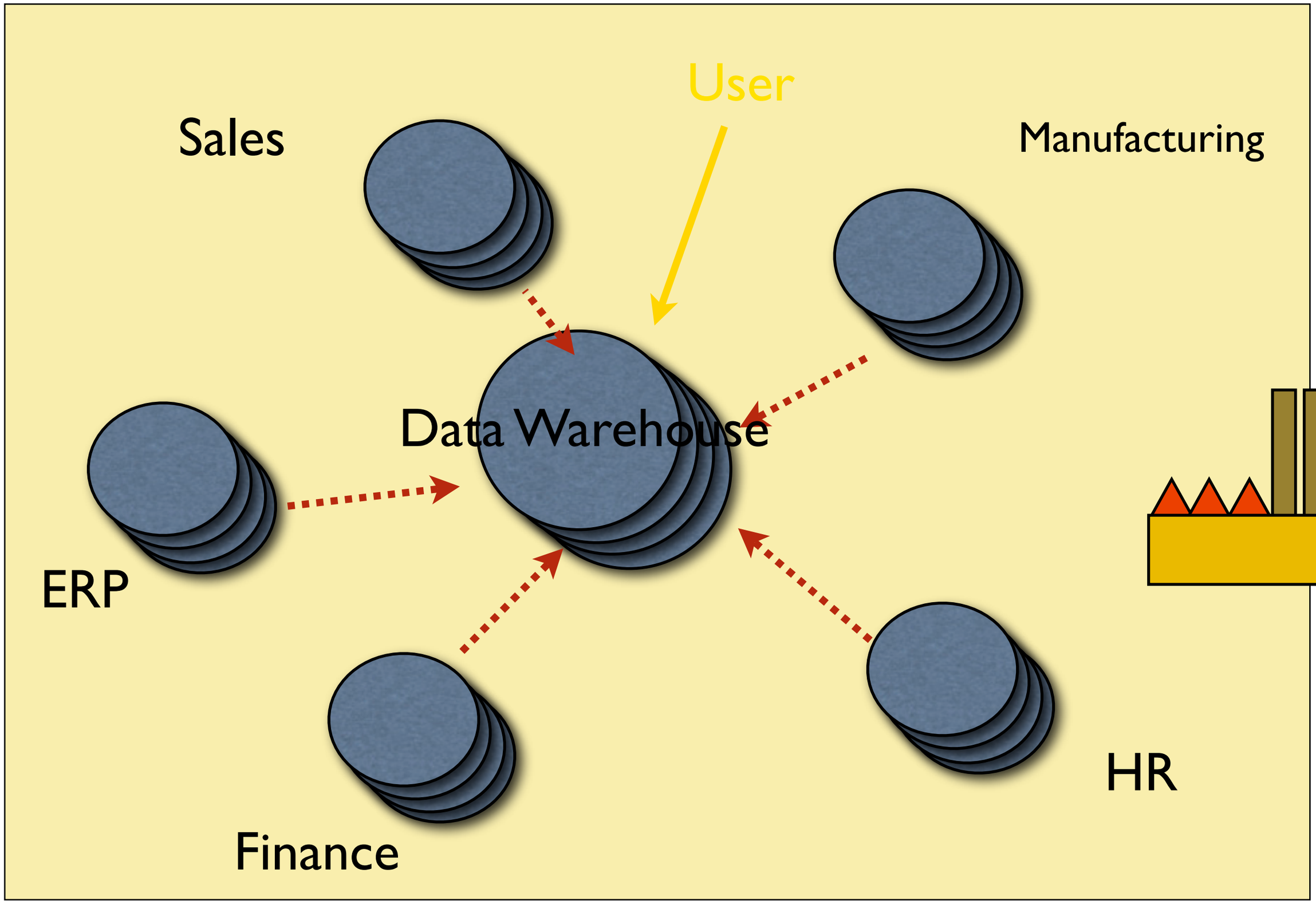
Solution 2: Materialization



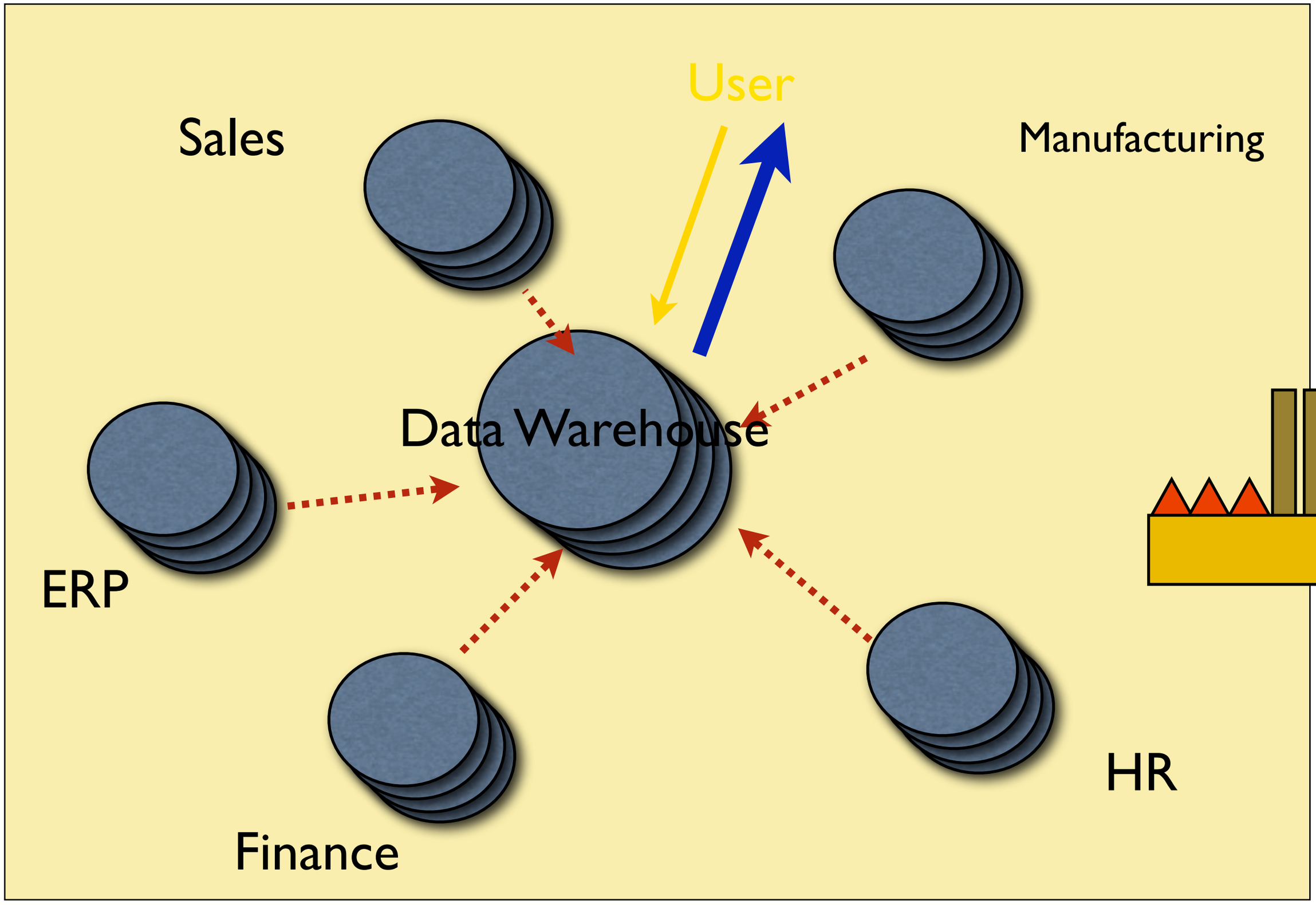
Solution 2: Materialization



Solution 2: Materialization



Solution 2: Materialization



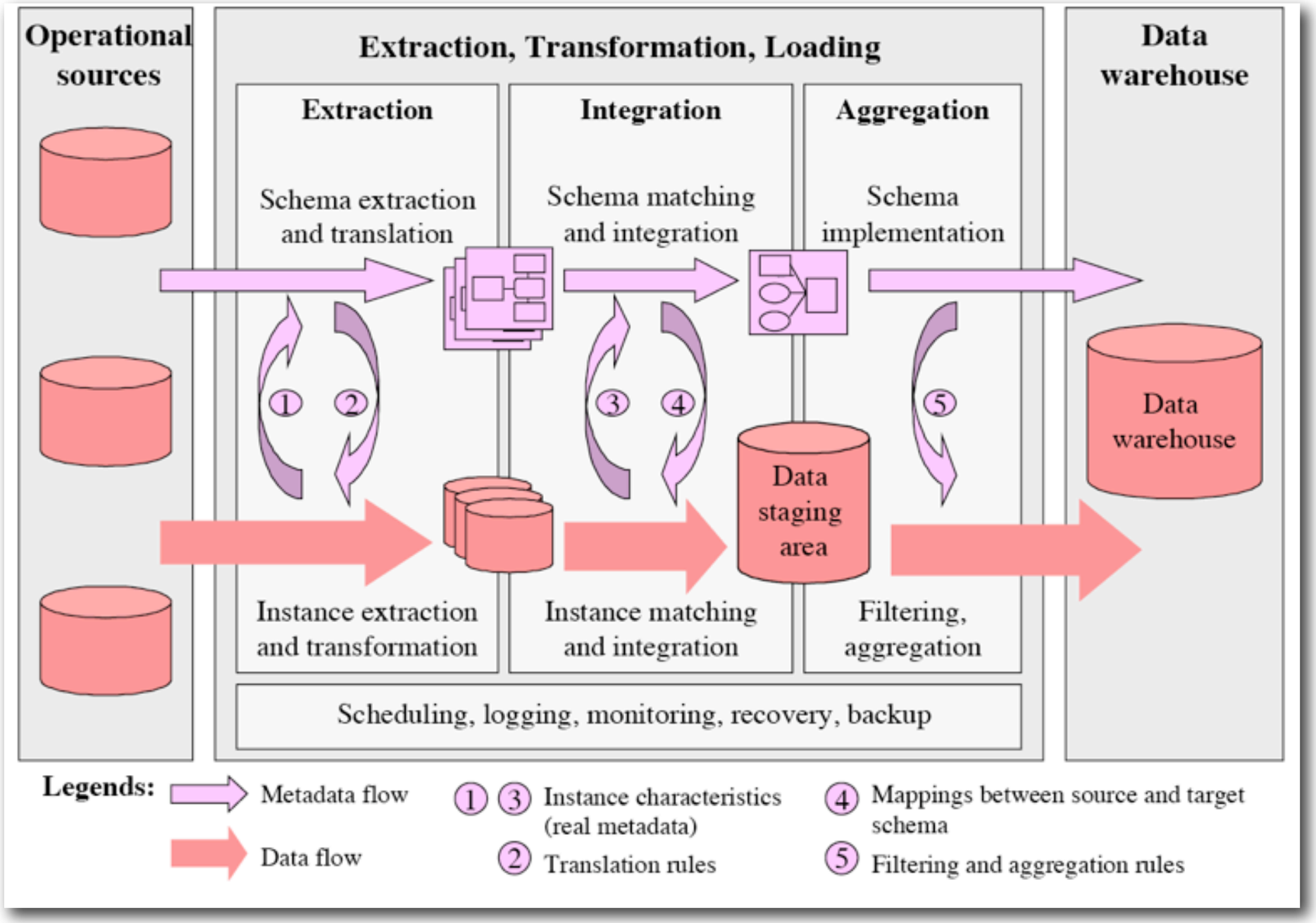
Trade-offs

Mediation	Materialization
up-to-date data	old (stale) data
poor performance	high performance
easy to extend	hard to extend

Summary

- Data integration is an important problem
- There are two principal solutions:
  - Mediation
  - Materialization
- Solution based on materialization are called "Data Warehousing"
- Important: Trade-offs among both solutions
- Also note: two solutions can be combined to a third hybrid solution
  - queries partially shipped to data sources and partially executed on materialized data

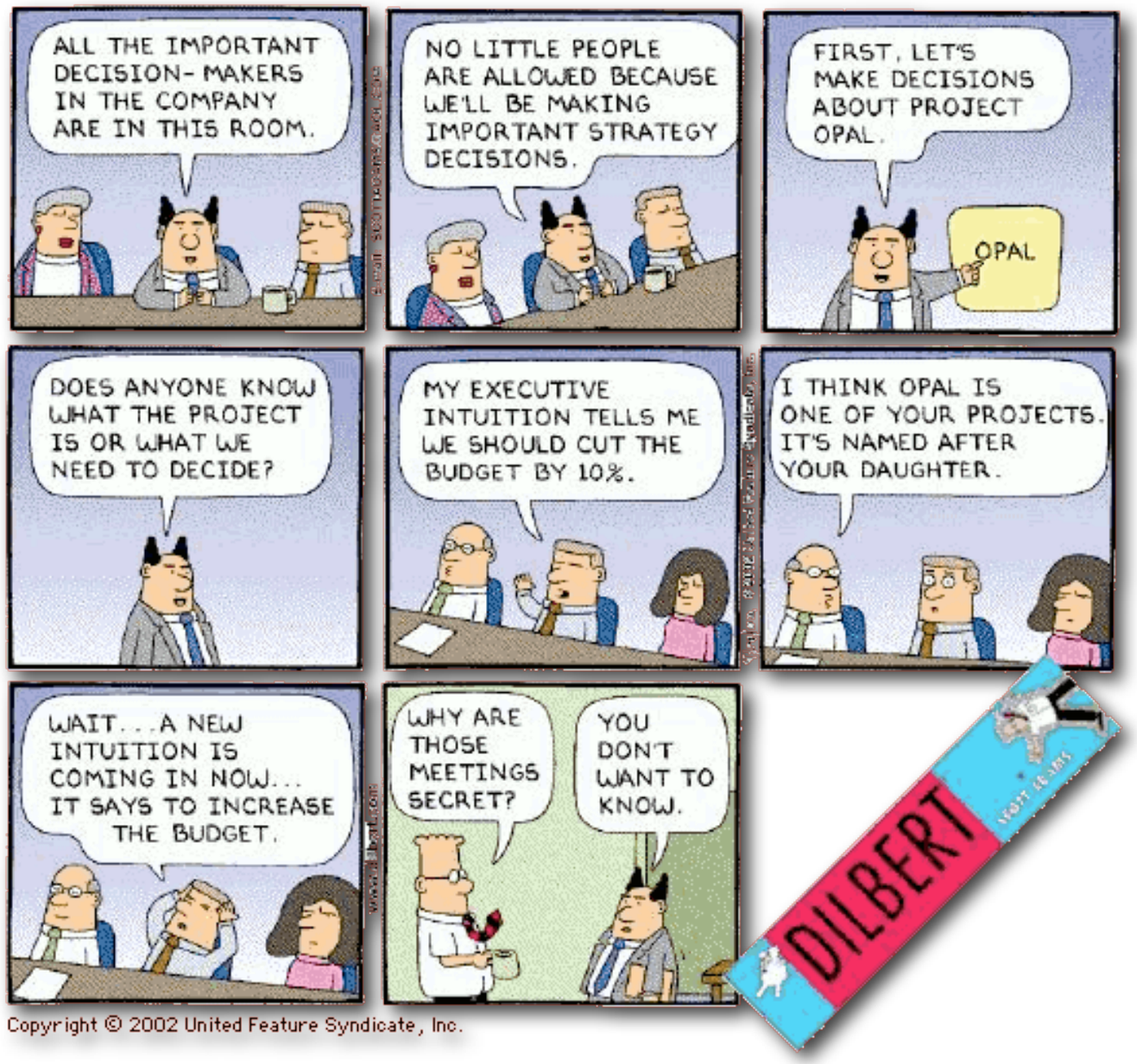
ETL Architecture (Data Warehousing)



Source: Erhard Rahm, Hong Hai Do: Data Cleaning: Problems and Current Approaches. IEEE Data Eng. Bull. 23(4): 3-13 (2000) Source: Erhard Rahm

ETL is a difficult Problem

- DWH used for decision support: what if wrong entries lead to wrong decisions in a company?
- heterogeneous sources: need for wrappers
- duplicate and erroneous data:
  need to cleanse data
- possible large number of different schemas:
  need for schema matching and schema integration



DWH Query Processing Techniques.

Agenda

- Basics
  - star, snowflake, and galaxy schema
  - native plans
  - Star Joins
    - native
    - cross product
    - semi-joins
  - partitioning (range, hash)
  - mat views
- O‘Neil article (How to index in order to speed-up star-join processing)
  - bitmaps
  - bitmap operations
  - join indices
  - bitmapped join indices
  - star joins using indices

Agenda

- French article (probs with current DBMS architectures)
  - Horizontal partitioning in Oracle
  - vertical partitioning in Oracle
  - projection index
  - adjusting pagesize
- MOLAP (multidimensional OLAP)
  - Dwarf
  - Dwarf recap
- Summary

Recap: Motivation & Overview

declarative query

```
SELECT title
FROM A,B
WHERE A.name = 'Hugo' AND A.id = B.dz;
```

Scanner Parser

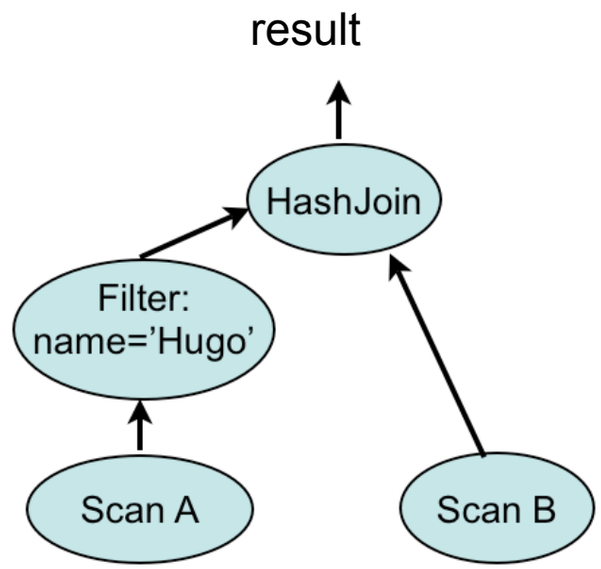
algebraic expression

$\prod_{\texttt{title}}(\sigma_{\texttt{A.name='Hugo' and A.id=B.dz}}(A \times B))$

Query optimizer

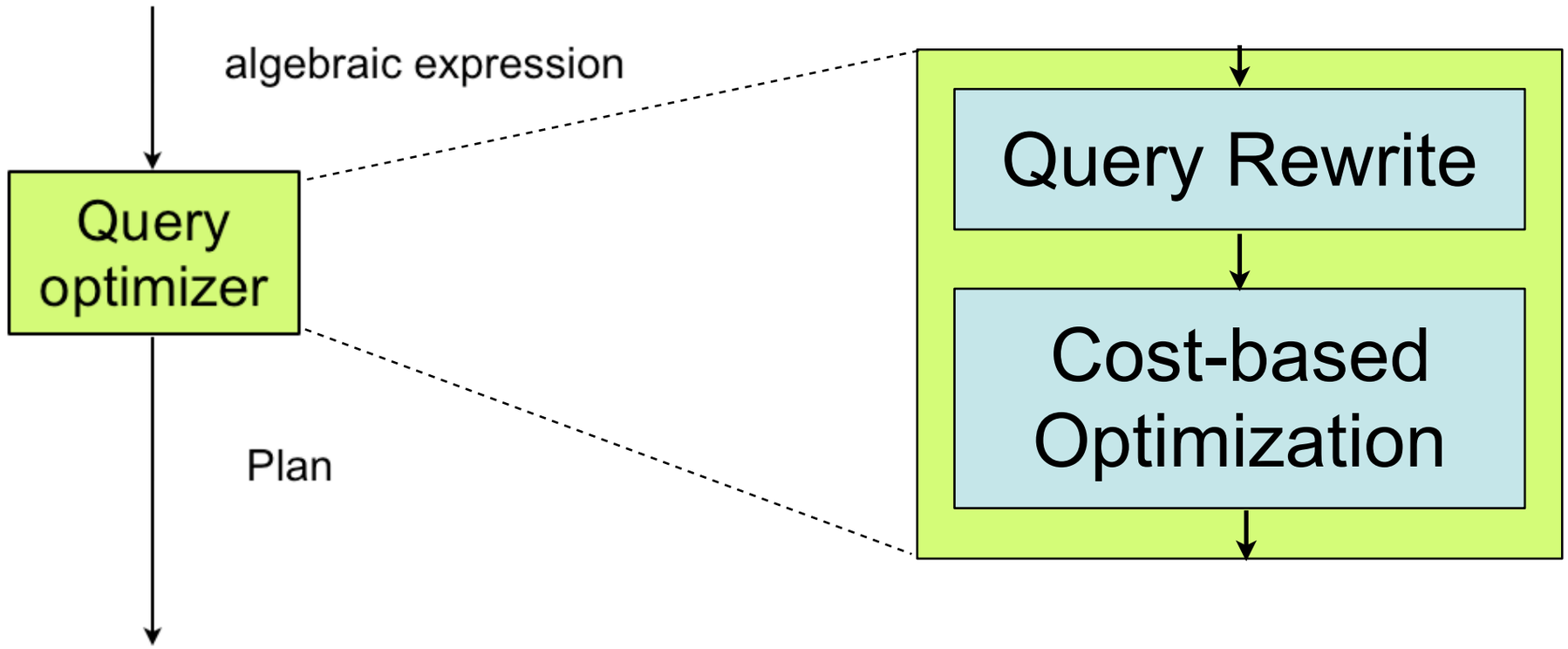
Plan

Code generation and execution



Recap: Query Optimizer

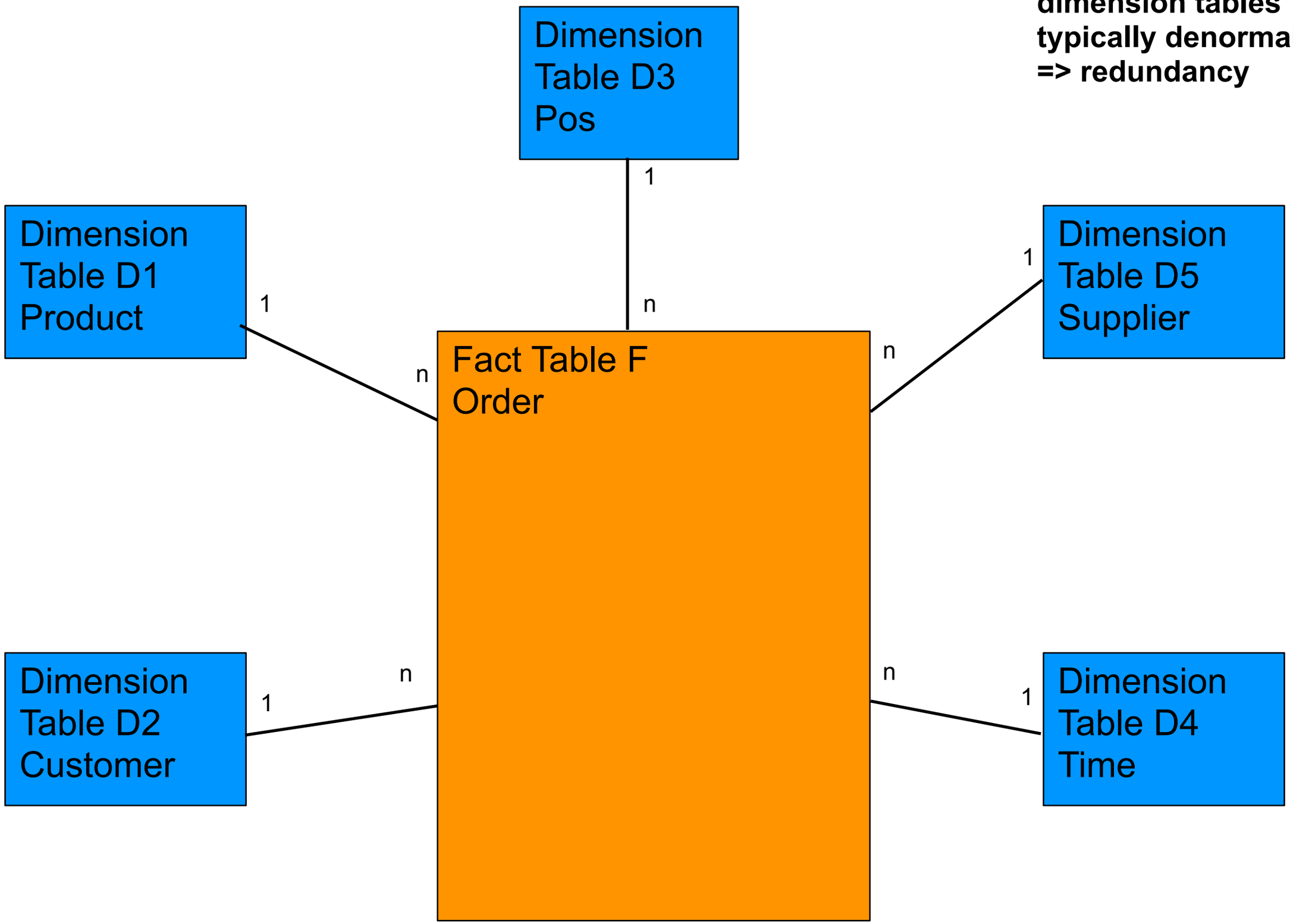
- Query Optimization is divided into two phases



- Query Rewrite
  - rewrite query using rules and heuristics
- Cost-based Optimization
  - Estimate costs of different plans using appropriate cost models
  - Choose plan that has the lowest cost estimate

Star Schema

dimension tables typically denormalized => redundancy



Star Schema

dimension tables typically denormalized => redundancy

D3: Pos

ID	a3	b3	c3	d3	e3
3					
7					
8					
9					
10					
12					
13					

1 — n

D1: Product

ID	a1	b1	c1	d1	e1	f1	g1
1							
3							
5							
8							
11							
42							

1 — n

F

D1_ID	D2_ID	D3_ID	D4_ID	D5_ID	M1	M2
1	11	7	22	5	42,0	5,6
1	11	7	22	333	4,0	3,6
1	11	7	33	8	12,3	4,4
1	11	7	33	333	178	3,3
1	11	7	33	444	11,7	1,0
1	11	7	33	777	12,3	1,0
1	11	7	44	5	11,8	4,6
1	11	7	77	5	9,0	1,0
1	11	7	77	8	2,1	1,1
1	11	7	77	333	3,4	0,5
1	11	8	44	8	3,8	1,9
1	11	8	44	444	1,2	3,0
...	...	...	...	...		

n — 1

D5: Supplier

ID	a5	b5	c5
5			
8			
333			
444			
777			

n — 1

D2: Customer

ID	a2	b2	c2	d2	e2	f2	g2
11							
22							
33							
77							
88							

n — 1

D4: Time

ID	a4	b4	c4	d4
11				
22				
33				
44				
55				
66				
77				

Star Schema

dimension tables typically denormalized => redundancy

D3: Pos

ID	a3	b3	c3	d3	e3
3					
7					
8					
9					
10					
12					
13					

D1: Product

ID	a1	b1	c1	d1	e1	f1	g1
1							
3							
5							
8							
11							
42							

D5: Supplier

ID	a5	b5	c5
5			
8			
333			
444			
777			

F

D1_ID	D2_ID	D3_ID	D4_ID	D5_ID	M1	M2
1	11	7	22	5	42,0	5,6
1	11	7	22	333	4,0	3,6
1	11	7	33	8	12,3	4,4
1	11	7	33	333	178	3,3
1						
1						
1						
1						
1						
1						
1						
1						
...						

1 (D3: Pos) — n (F)
1 (D1: Product) — n (F)
1 (D5: Supplier) — n (F)
1 (D2: Customer) — n (F)

D2: Customer

ID	a2	b2	c2	d2	e2	f2	g2
11							
22							
33							
77							
88							

D4: Time

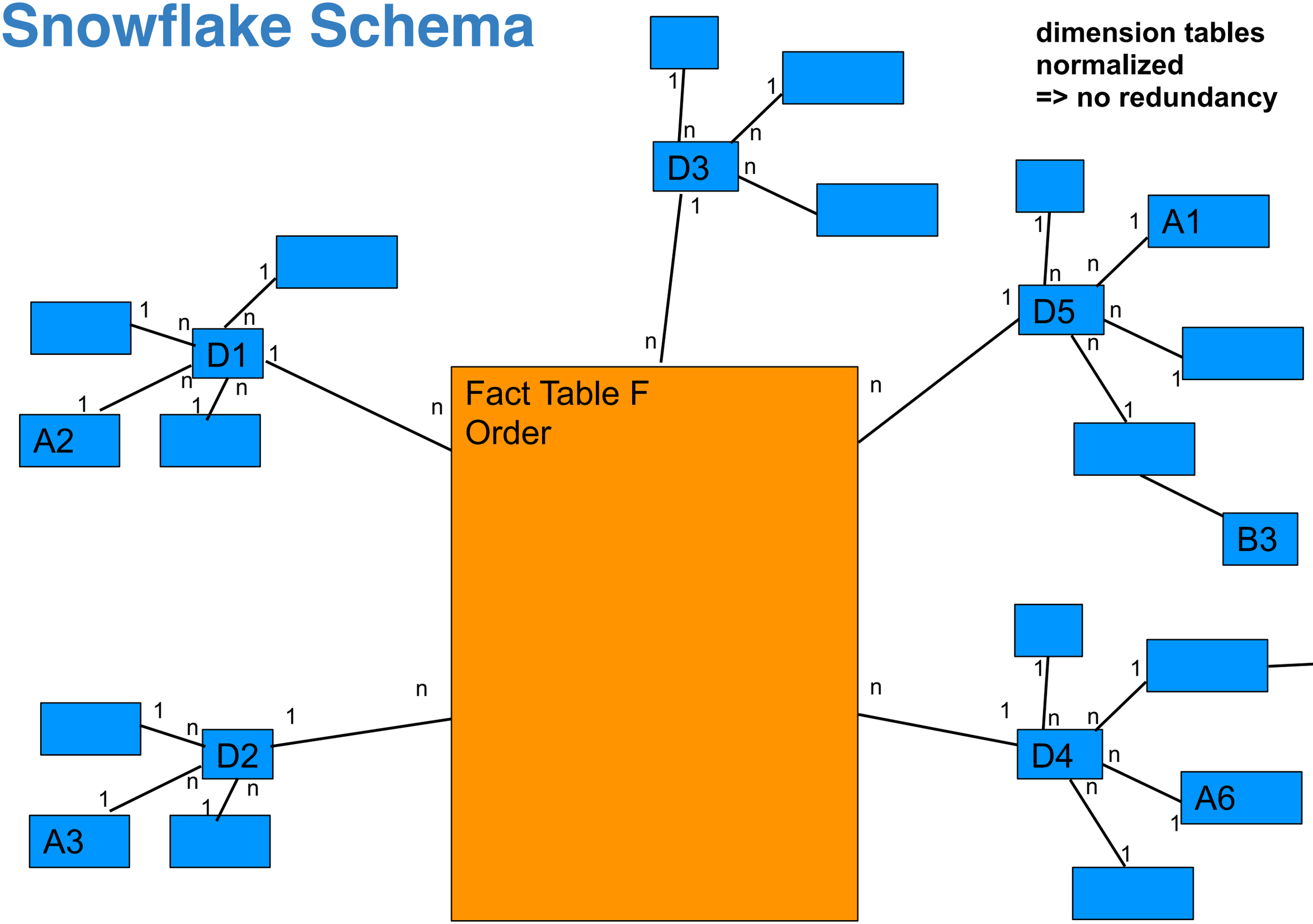
ID	a4	b4	c4	d4
11				
22				
33				
44				
55				
66				
77				

Example Query

Filter:
D1.b1=“laptop x42“
and D2.b2=“Saarland“
and D3.c3=“a356“
and D4.a4=“Q2007“
Display:
D2.b3 (Customer city)
D5.a5 (supplier name)
D4.d4 (week)
Measures:
M1 (sales)
M2 (revenue)

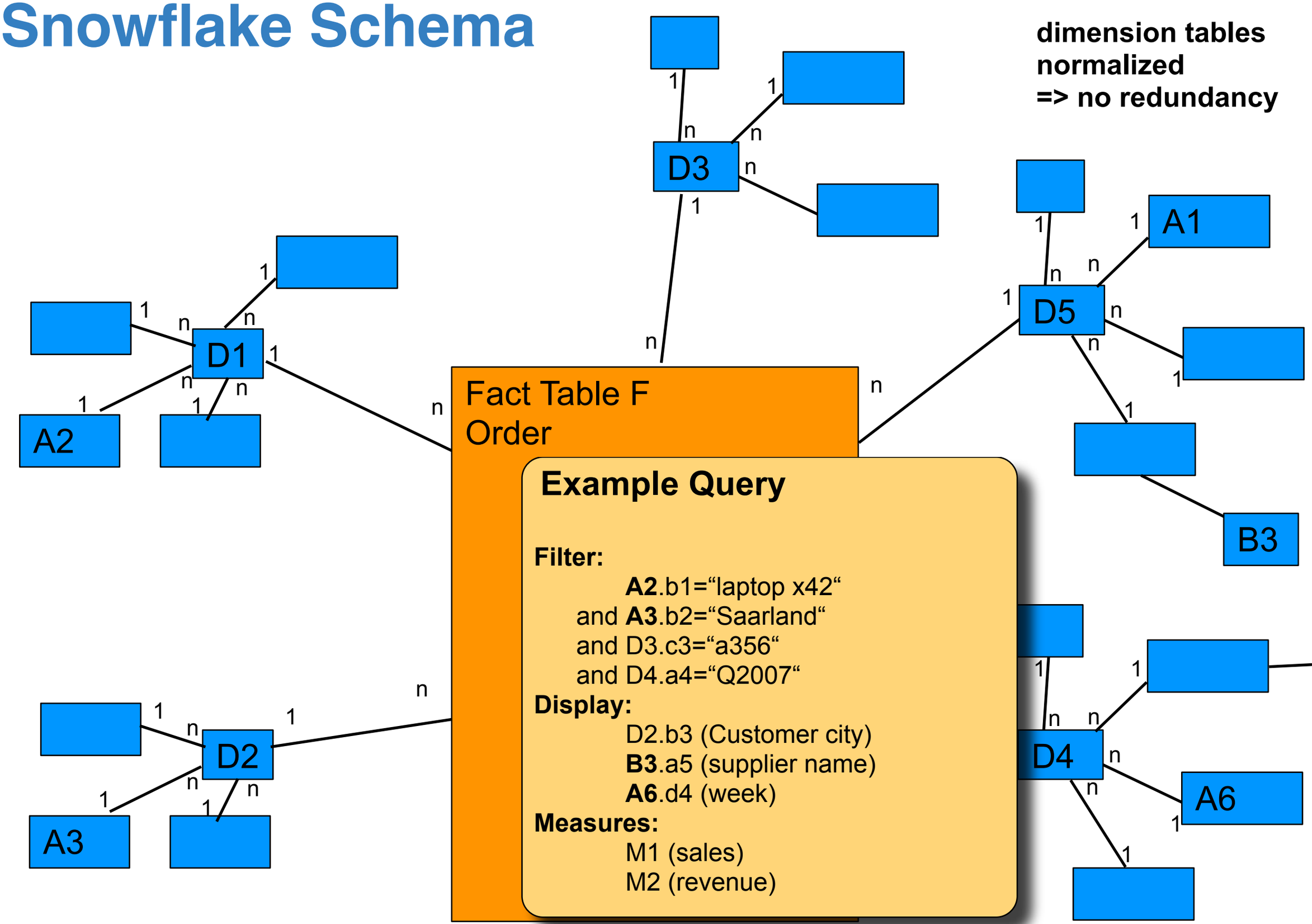
Snowflake Schema

dimension tables
normalized
=> no redundancy



Snowflake Schema

dimension tables
normalized
=> no redundancy

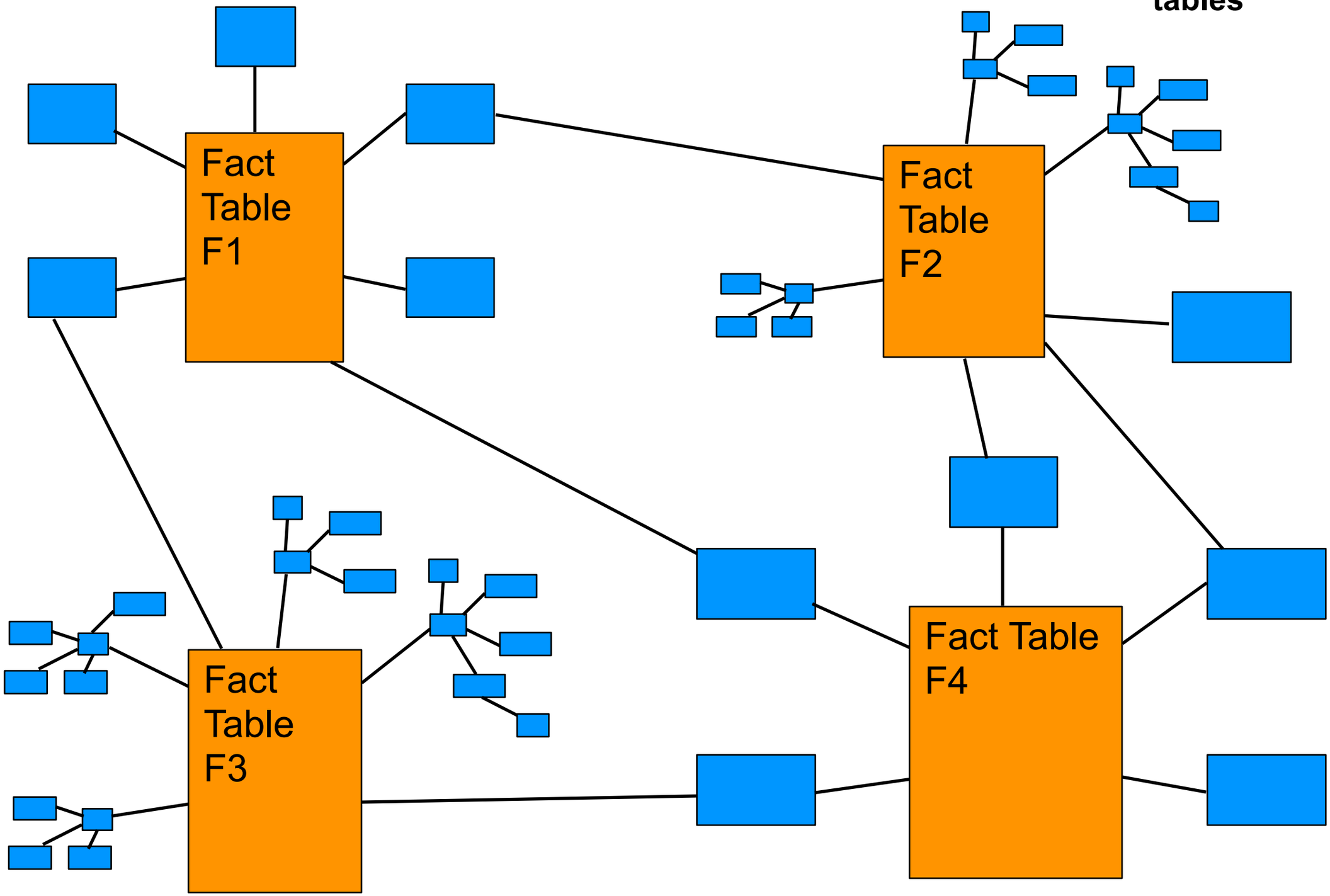


Example Query

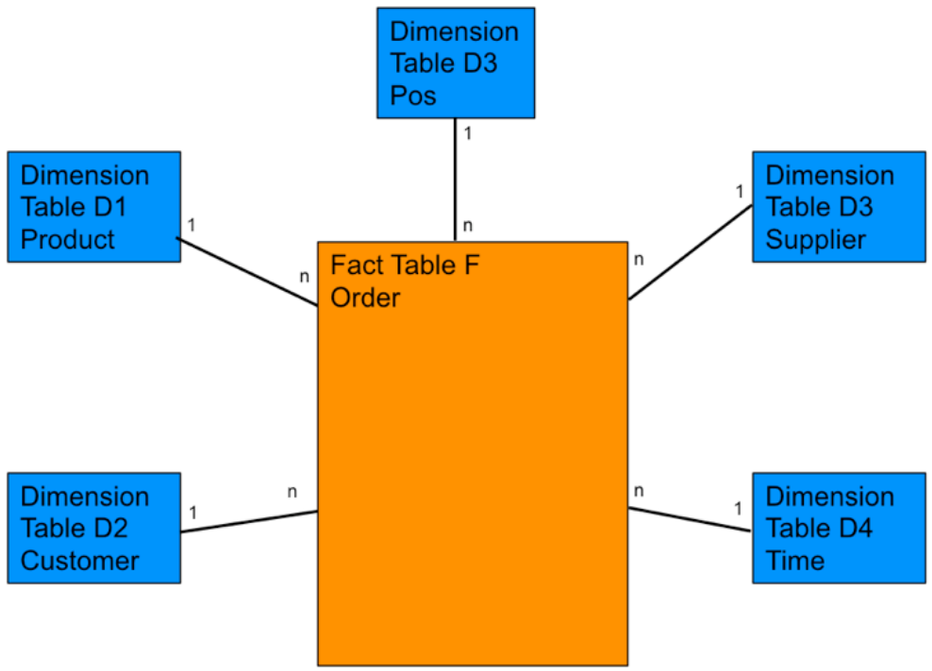
Filter:
A2.b1="laptop x42"
and **A3**.b2="Saarland"
and D3.c3="a356"
and D4.a4="Q2007"
Display:
D2.b3 (Customer city)
B3.a5 (supplier name)
A6.d4 (week)
Measures:
M1 (sales)
M2 (revenue)

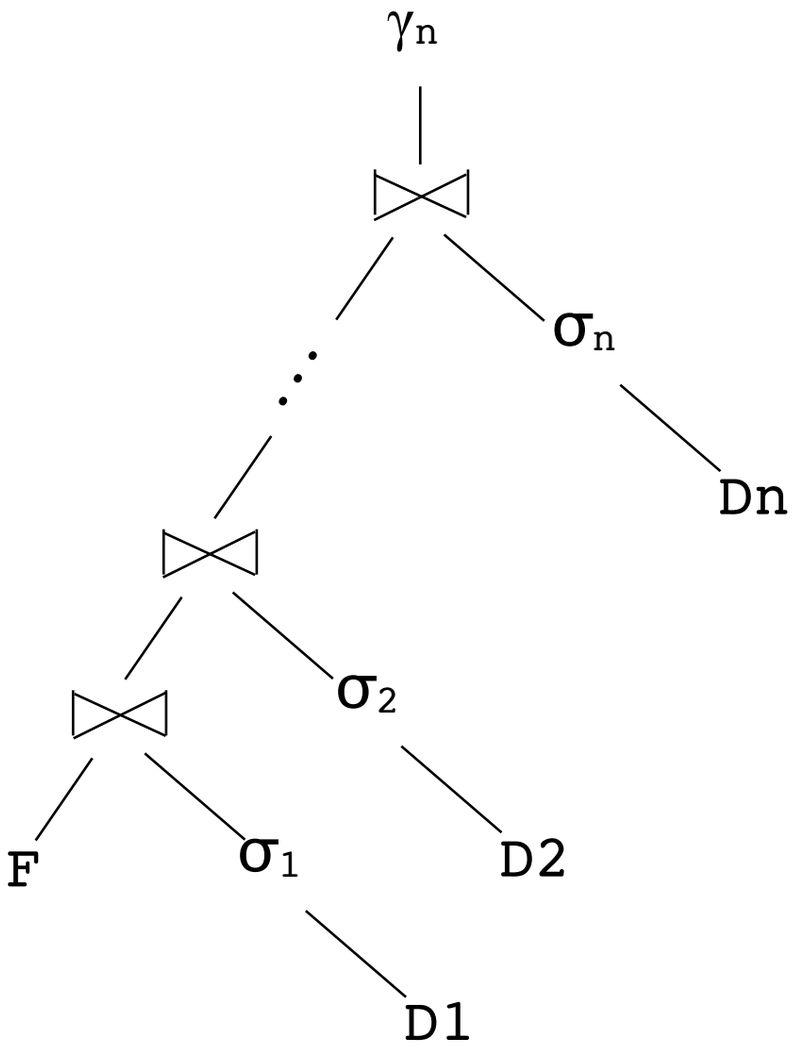
Galaxy Schema

dimension tables shared by different fact tables



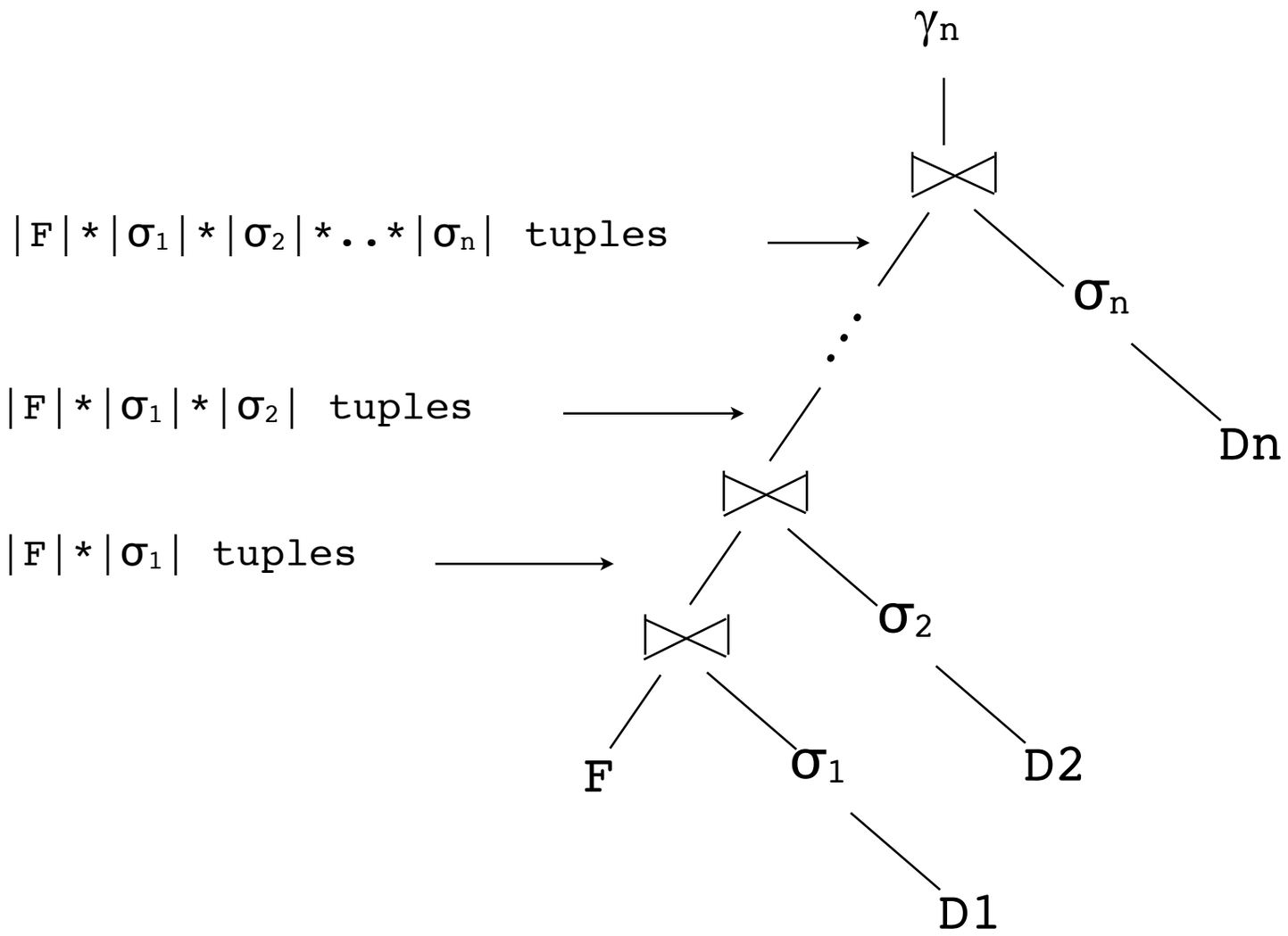
Query Optimization: Naïve Strategy





- Idea: Iteratively join *n* dimension tables to fact table
- first join may exploit primary index on fact table
- this strategy works well if the first intermediate result, i.e., join of D1 and F is small, following joins should decrease size of intermediate result

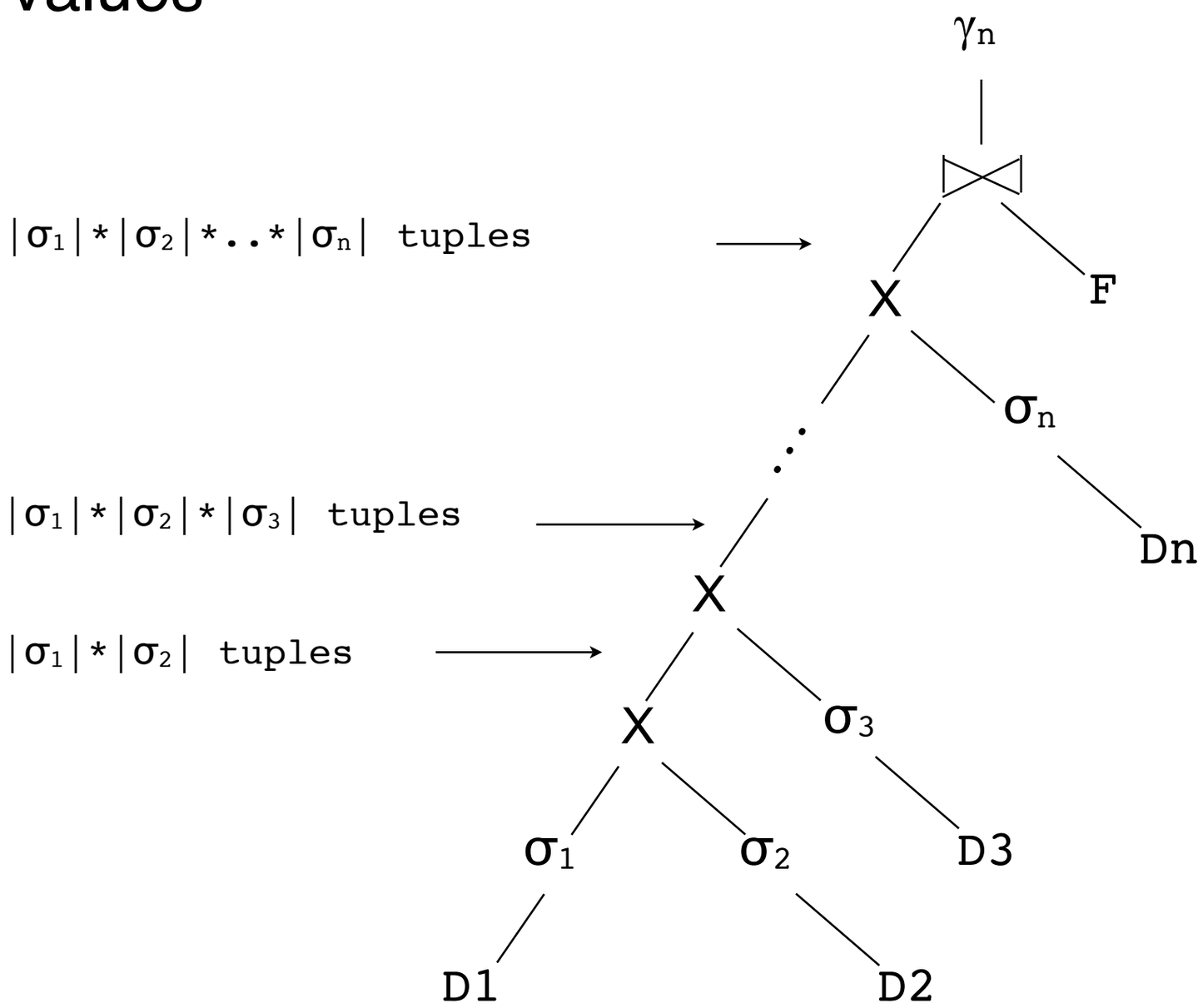
Naïve Strategy: Join Selectivity



- Note: in order to reduce intermediate results, cost based optimizer should reorder dimensions in the plan
- see slides for Week 9

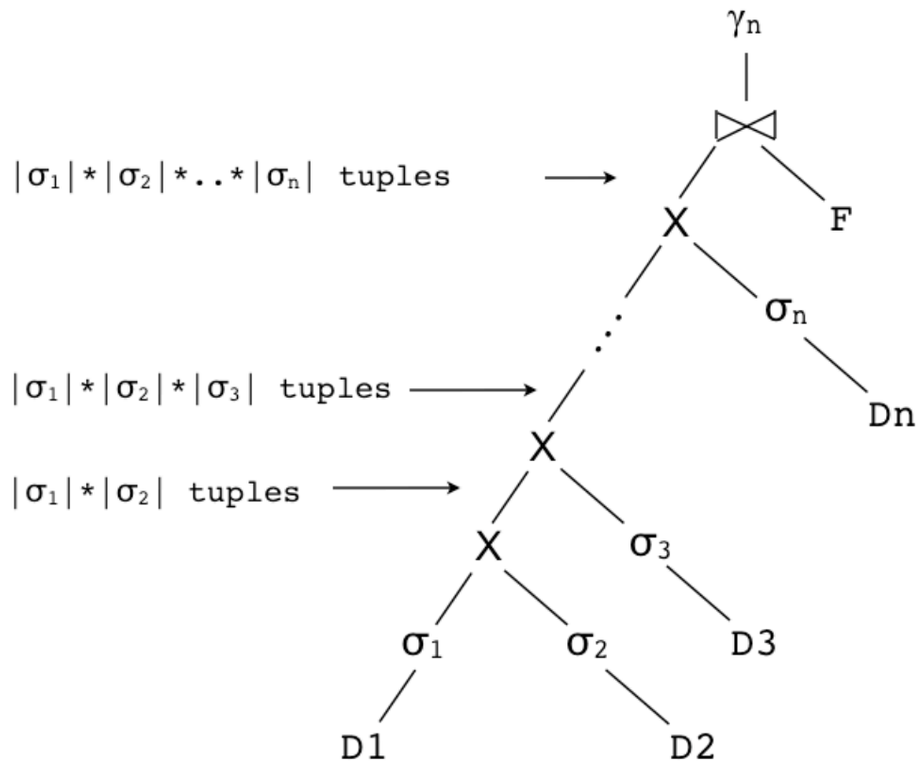
Cross Product Plan

- Idea: create the multidimensional cube and fill that cube with existing values



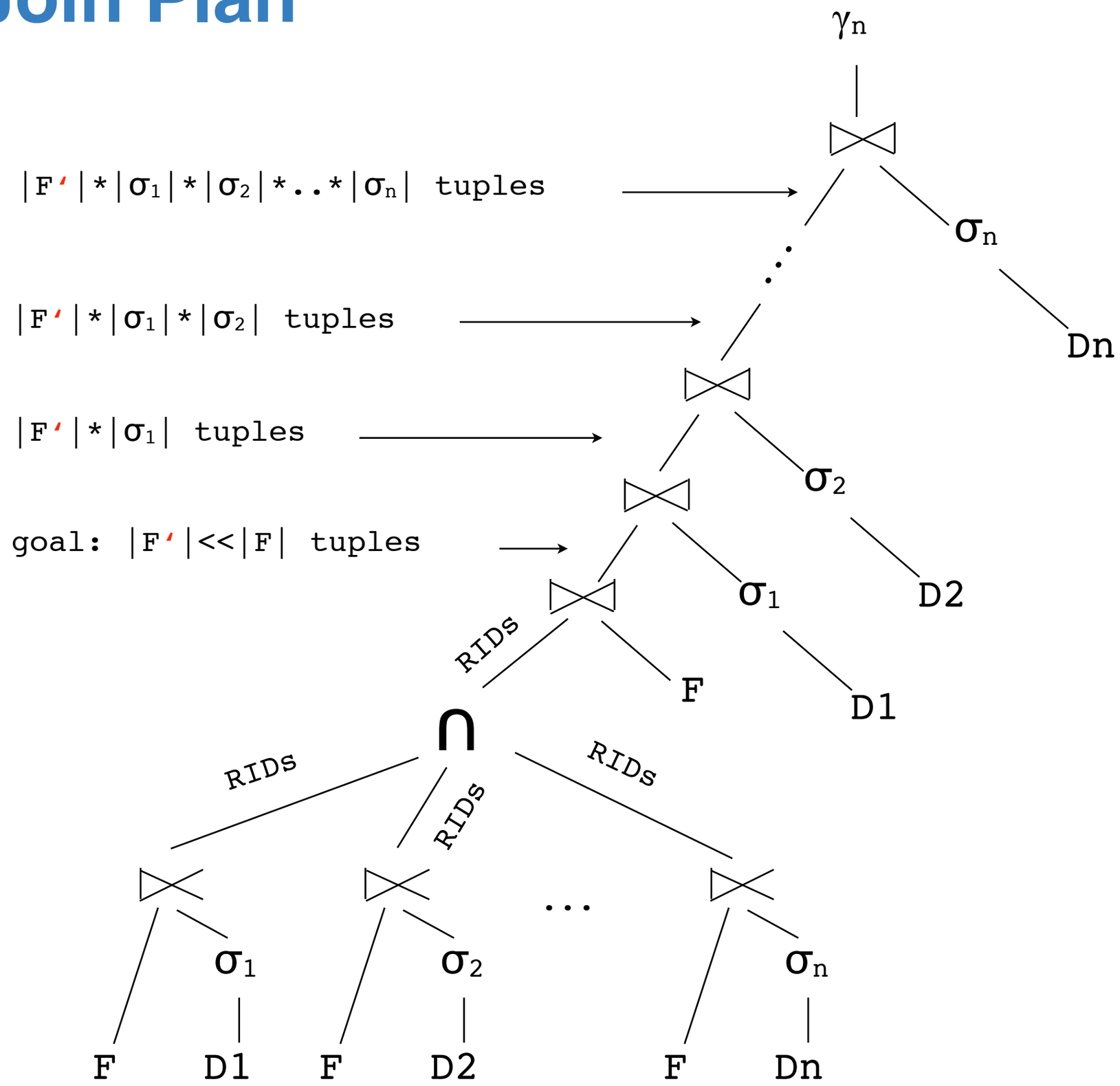
Cross Product Plan

- Idea: create the multidimensional cube and fill that cube with existing values
- this plan may be **very** expensive $O(|D_1|*..*|D_n|)$, i.e., $O(n^2)$ already for two dimensions



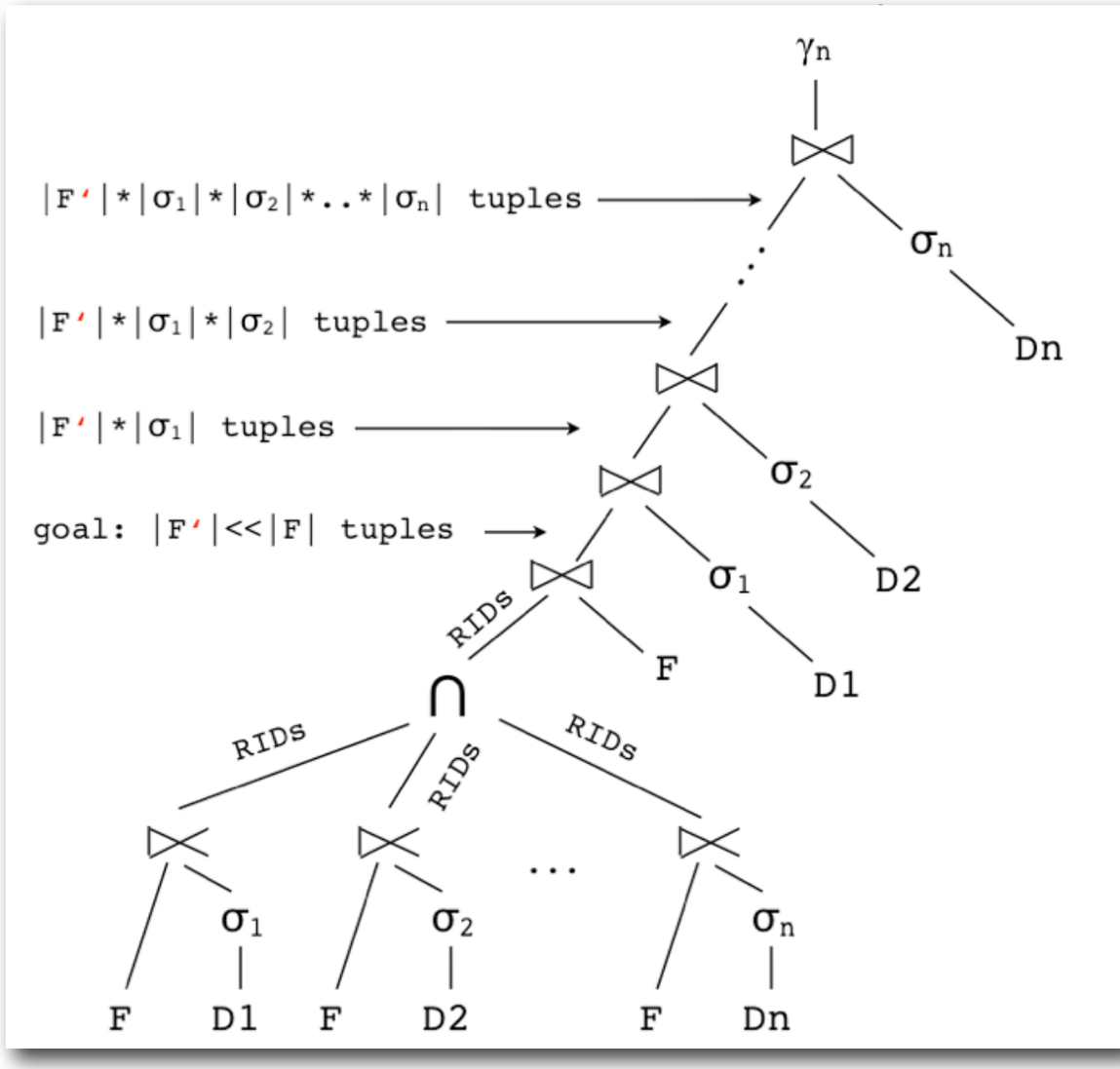
- plan works well if cartesian product on first dimensions is small
- cost based optimizer has to decide whether this plan makes sense
- final join with F may exploit compound index on F
- this plan is also interesting for those cases when client wants to display all cells of the cube anyway no matter whether they contain values or not (CUBE or ROLLUP operators)

Semi-Join Plan



Semi-Join Plan

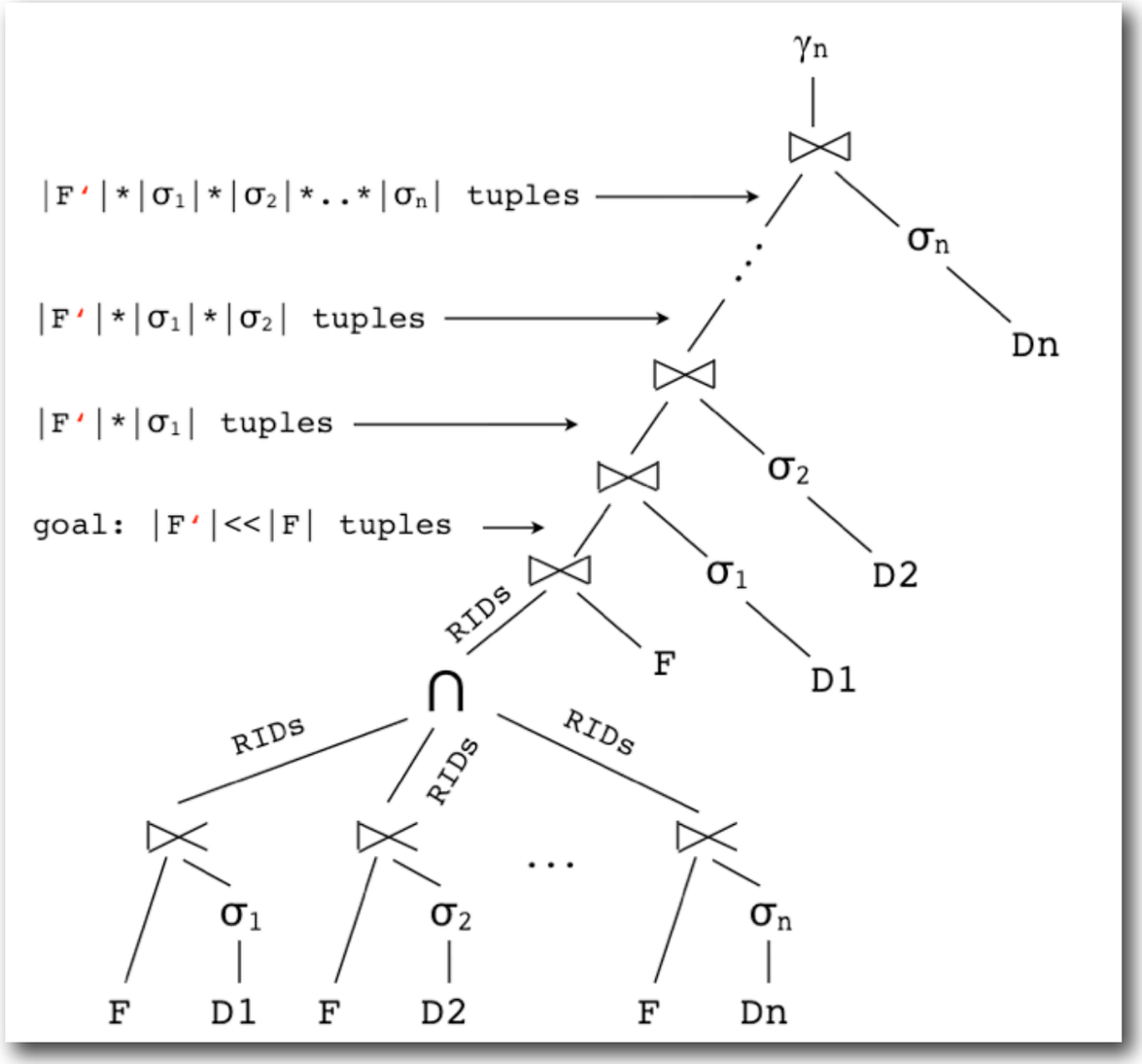
- Idea: reduce size of fact table by preselecting those rows from F that may participate in the join
- preselection is performed using semi-joins of F and D_i
- RID lists of the n semi-joins are then intersected
- final RID list is joined with F



- RID list may be used to enhance the join of the RID list with F by directly accessing tuples in F
- Again: performing n semi-joins may be expensive depending on the selectivity of the selections
- cost based optimizer has to decide whether this plan makes sense

The Story so far

- Different options for star joins
  - naive strategy
  - reordering dimensions
  - cross products
  - semi-joins



- **Open issues...**
  - **How can we improve the performance of the semi-joins?**
  - **How can we improve the performance of the selections?**
  - **How can we improve the performance of the join operations?**

... and their Solutions

- **How can we improve the performance of the semi-joins?**
  - bitmap join index
- **How can we improve the performance of the selections?**
  - secondary index: selection bitmap
  - combination of selection bitmap and bitmap join index
- **How can we improve the performance of the join operations?**
  - materialized views
  - join index (special case of materialized views)
  - partitioning

OLTP vs. OLAP vs. IR

	OLTP	OLAP	IR
Data Access	read/write	read-mostly	read-mostly
Freshness	up-to-date	stale	stale
Query Access	key-oriented	value-oriented	value-oriented
Indexing	update efficient	**query efficient**	**query efficient**
Data	structured	structured	unstructured
Query optimization	heuristic	cost-based	cost-based
Parallelism	inter-query	intra-query	intra-query
Precision&Recall	1	1	<=1
Data volume	small-medium	huge	huge

Indexing Constraints

- OLAP allows for much more expensive indexing techniques
- number of indexes is only limited by
  - available disk space
  - time window available to load and index new data into the DWH
- DBA should select appropriate indexes based on query patterns
  - boss‘s favorite queries should be tuned first...
  - to be on the save side:
    precompute boss‘s favorite queries at indexing time
    (this is **Offline** Analytical Processing, in other words: **Reporting**)
- Claim: using the techniques presented in this lecture a DWH can be configured to have interactive (< 5 sec) response times for all queries
- However: you have to know very well how to use your weapons

Materialized Views

- Idea: materialize (precompute) the result to a join and/or aggregate
- a materialized view is the precomputed result to a query
- the query optimizer may choose to pick one of the materialized views to speed-up query processing
- Trade-off

  additional speedup for query processing

  versus

  additional costs for storing and maintaining materialized views

- Oracle SQL dialect:

```
CREATE MATERIALIZED VIEW matview42
BUILD IMMEDIATE
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
```

- Problem: how to chose which materialized views to create?

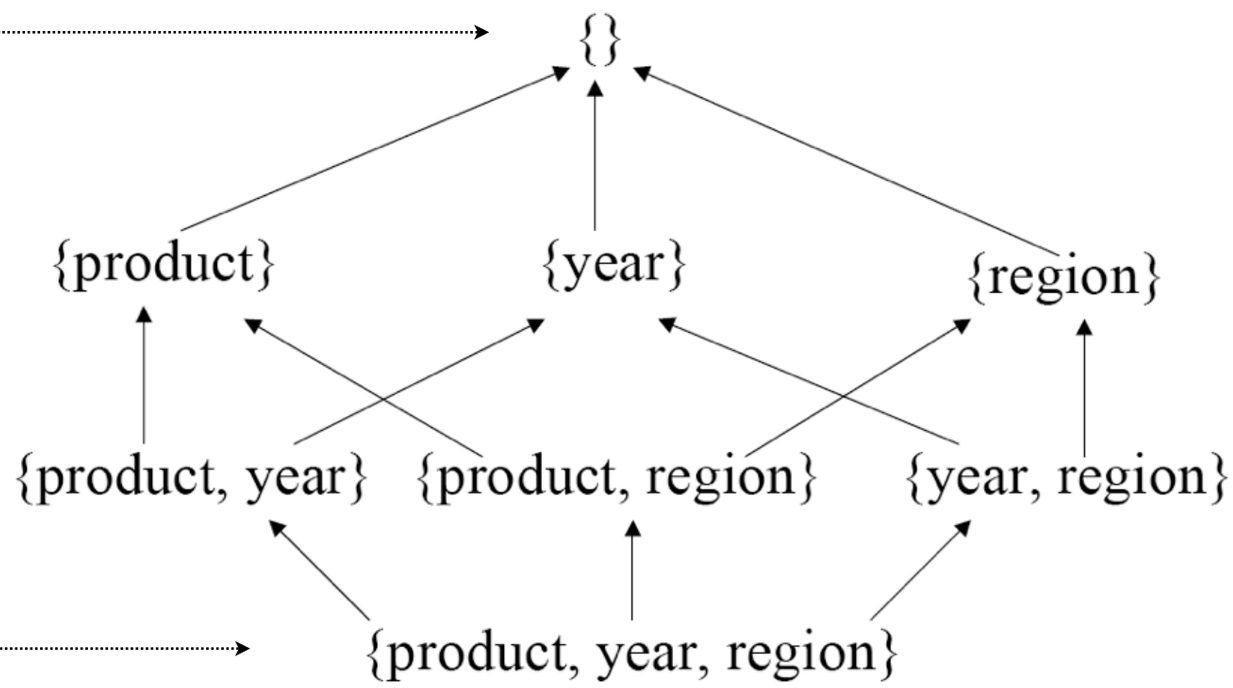
View Lattice

Aggregate of everything

1 attribute in drill-down

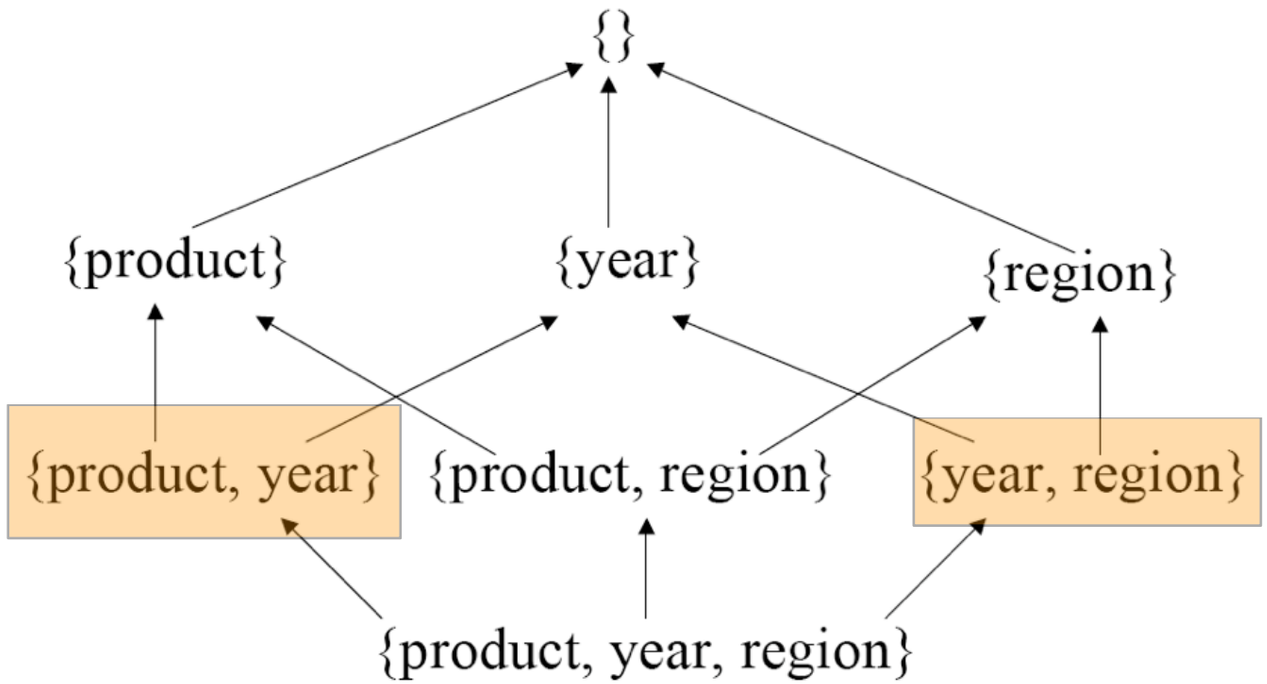
2 attributes in drill-down

3 attributes in drill-down



- Derivability: arrows indicate how a view can be derived from other views
- Note: DWH implementations may have hundreds (!) of attributes
- Number of aggregates for n functional independent attributes = $\mathbf{2^n}$
- For functional dependent attributes polynomial number of aggregates
- => Not all aggregates may be materialized!

Query Answering using Mat. Views Example



- Lets assume we maintain two materialized views (orange boxes)
- All queries referencing aggregates {product}, {year} or {} can be computed using materialized view {product, year}
- All queries referencing aggregates {region}, {year} or {} can be rewritten using materialized view {year, region}

Materialized View Updates

- What happens when new data is loaded into the DWH?
  - depending on the number of materialized views and the view lattice, many materialized views may have to be recomputed or updated
- Recompute
  - existing mat view updates may be combined to provide for more efficient recomputation (multi-query optimization MQO)
  - MQO core idea: for two mat views MV1 and MV2 try to find an aggregate MV3 such that both MV1 and MV2 can be derived from MV3
- Update
  - compute view delta
  - merge delta with existing mat view
  - only allowed for certain types, i.e., additive aggregation functions

Join Index

- special case of a materialized view
- For two tables R and S a join index may have **three** different types
  1. join value -> ({R.RID},{S.RID}) i.e., all rows of R and S that have the specified join column value
  2. R.RID -> {S.RID}, i.e., all rows (RIDs) of S that join with row R.RID
  3. R.attributeValue -> {S.RID}, i.e., all rows of S that join with those rows in R having the specified attribute value (not the join key)
- If more than two tables are involved, these indexes are called *domain indices*.
- Literature: O'Neil and Graefe 95

Join Index Example: Types 1&2

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		

rows in D1 ↓ rows in F ↓

1.join value -> ({D1.RID},{F.RID})

- Index:
  3 -> ({0},{0,4,10})
  7 -> ({1},{2,5,9})
  11 -> ({2},{1,7,8})
  15 -> ({3},{3,6})

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

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

Can be derived from Type 1

similar to result of COGROUP command in Pig Latin

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}

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

Important: we need one bit list per column value. As column values do not have to be unique, the total number of bit lists is typically less than |D1|.

Can be derived by joining Type 2 index with att2->RID