Business Intelligence: OLAP, Data Warehouse, and Column Store

Thomas Heinis

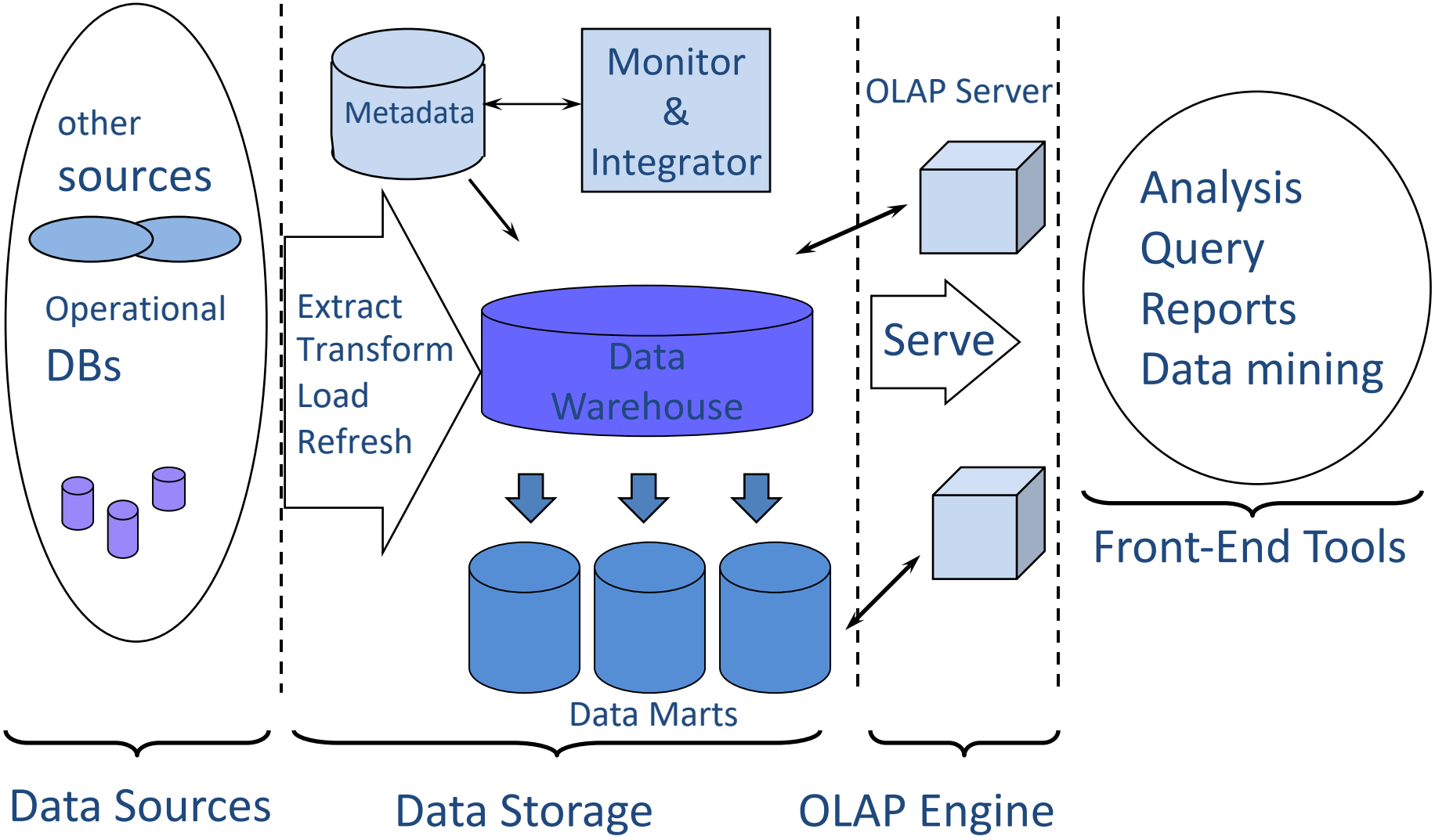
Imperial College London

Data Warehouse Implementation

Implementing a Warehouse

- *Monitoring:* Sending data from sources
- *Integrating:* Loading, cleansing,...
- *Processing:* Query processing, indexing, ...
- *Managing:* Metadata, Design, ...

Multi-Tiered Architecture



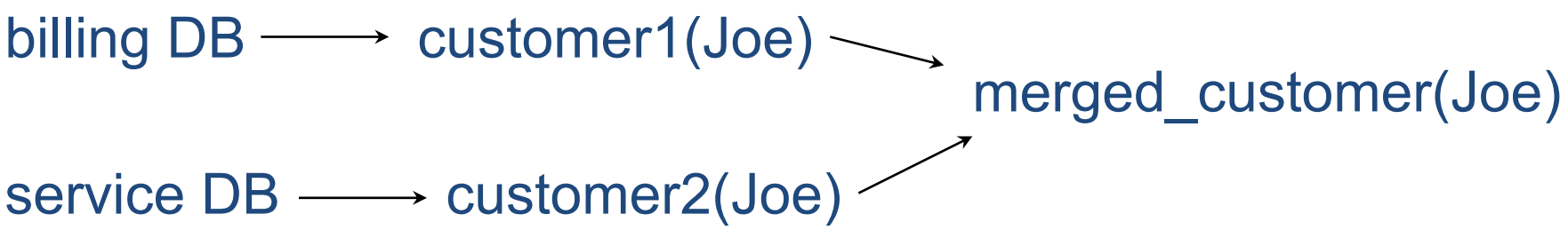
Monitoring

- Source Types: relational, flat file, IMS, WWW, news-wire, …
- Incremental vs. Refresh

customer	id	name	address	city
	53	joe	10 main	sfo
	81	fred	12 main	sfo
	111	**sally**	**80 willow**	**la**

Data Cleaning

- Migration (e.g., yen ⇨ dollars)
- Scrubbing: use domain-specific knowledge (e.g., social security numbers)
- Fusion (e.g., mail list, customer merging)



- Auditing: discover rules & relationships (like data mining)

Loading Data

- Incremental vs. refresh
- Off-line vs. on-line
- Frequency of loading
  - At night, 1x a week/month, continuously
- Parallel/Partitioned load

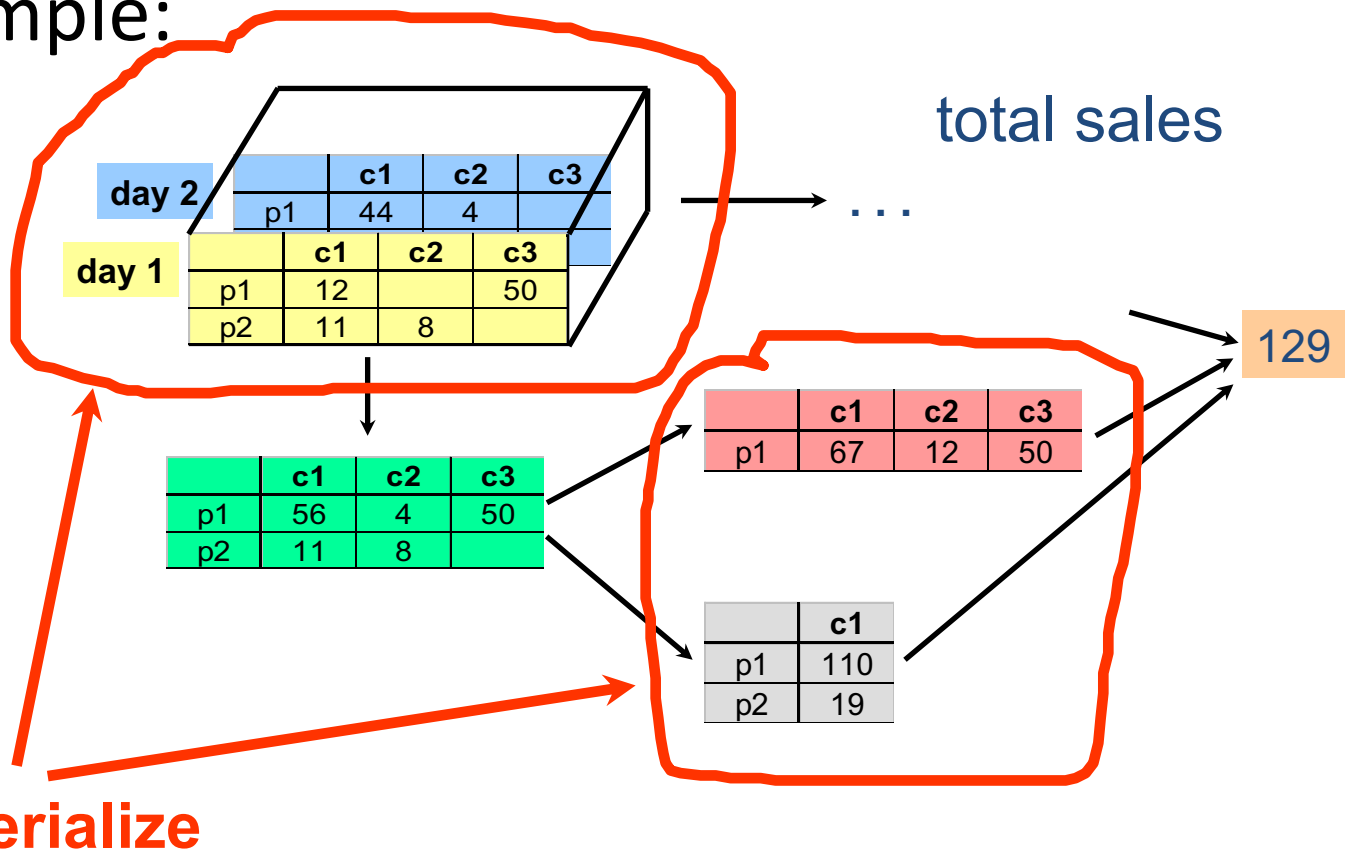
Derived Data

- Derived Warehouse Data
  - indexes
  - aggregates
  - materialized views (next slide)
- When to update derived data?
- Incremental vs. refresh

OLAP Implementation

What to Materialize?

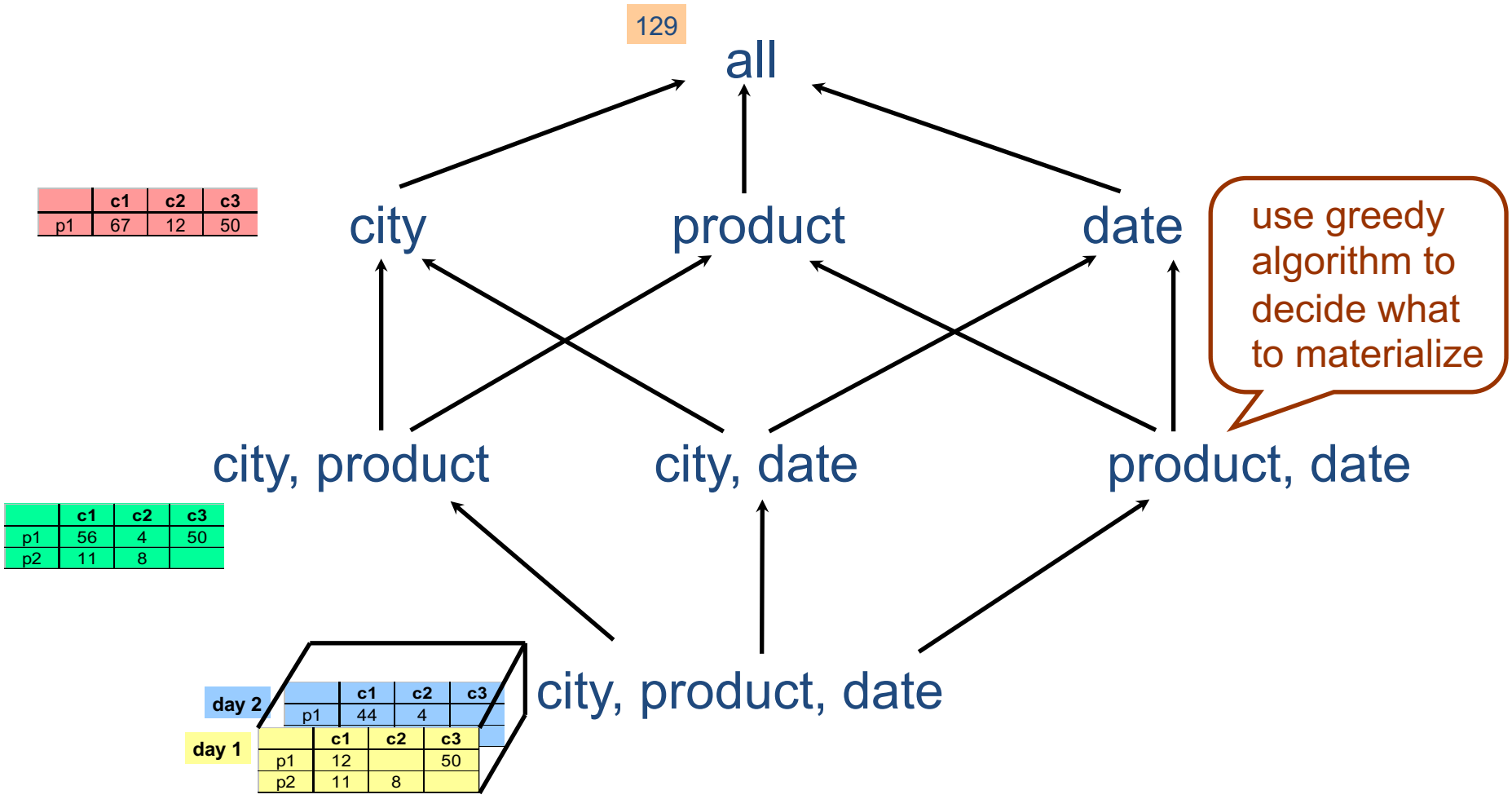
- Store in warehouse results useful for common queries
- Example:



Materialization Factors

- Type/frequency of queries
- Query response time
- Storage cost
- Update cost

Cube Aggregates Lattice



	c1	c2	c3
p1	67	12	50

	c1	c2	c3
p1	56	4	50
p2	11	8	

	c1	c2	c3
p1	44	4	

	c1	c2	c3
p1	12		50
p2	11	8	

Indexing OLAP Data: Bitmap Index

- Index on a particular column
- Each value in the column has a bit vector: bit-op is fast
- The length of the bit vector: # of records in the base table
- The i-th bit is set if the i-th row of the base table has the value for the indexed colum

Base table

Cust	Region	Type
C1	Asia	Retail
C2	Europe	Dealer
C3	Asia	Dealer
C4	America	Retail
C5	Europe	Dealer

Index on Region

RecID	Asia	Europe	America
1	1	0	0
2	0	1	0
3	1	0	0
4	0	0	1
5	0	1	0

Index on Type

RecID	Retail	Dealer
1	1	0
2	0	1
3	0	1
4	1	0
5	0	1

Join Processing

Join

- How does DBMS join two tables?
- Sorting is one way...
- Database must choose best way for each query

Schema for Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long,
  - 100 tuples per page,
  - M = 1000 pages total.
- Sailors:
  - Each tuple is 50 bytes long,
  - 80 tuples per page,
  - N = 500 pages total.

Equality Joins With One Join Column

```
SELECT  *
FROM Reserves R1, Sailors S1
WHERE  R1.sid=S1.sid
```

- In algebra: R⋈S. Common! Must be carefully optimized. R × S is large; so, R × S followed by a selection is inefficient.
- Assume: M tuples in R, p_R tuples per page, N tuples in S, p_S tuples per page.
  - In our examples, R is Reserves and S is Sailors.
- We will consider more complex join conditions later.
- *Cost metric*: # of I/Os. We will ignore output costs.

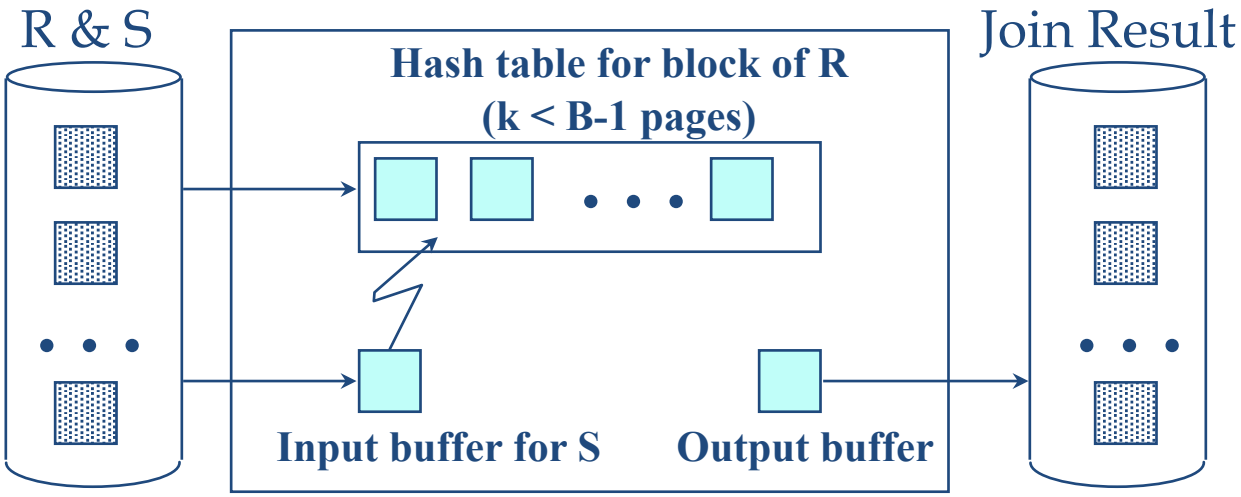
Simple Nested Loops Join

```
foreach tuple r in R do
        foreach tuple s in S do
                if r_i == s_j then add <r, s> to result
```

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: $M + p_R * M * N = 1000 + 100*1000*500$ I/Os.

Block Nested Loops Join

- Use one page as an input buffer for scanning the inner S, one page as the output buffer, and use all remaining pages to hold ``block'' of outer R.
  - For each matching tuple r in R-block, s in S-page, add <r, s> to result. Then read next R-block, scan S, etc.



Index Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S where r_i == s_j do
        add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it the inner and exploit the index.
  - Cost: M + ((M*p_R) * cost of finding matching S tuples)

Sort-Merge Join ($R \bowtie_{i=j} S$)

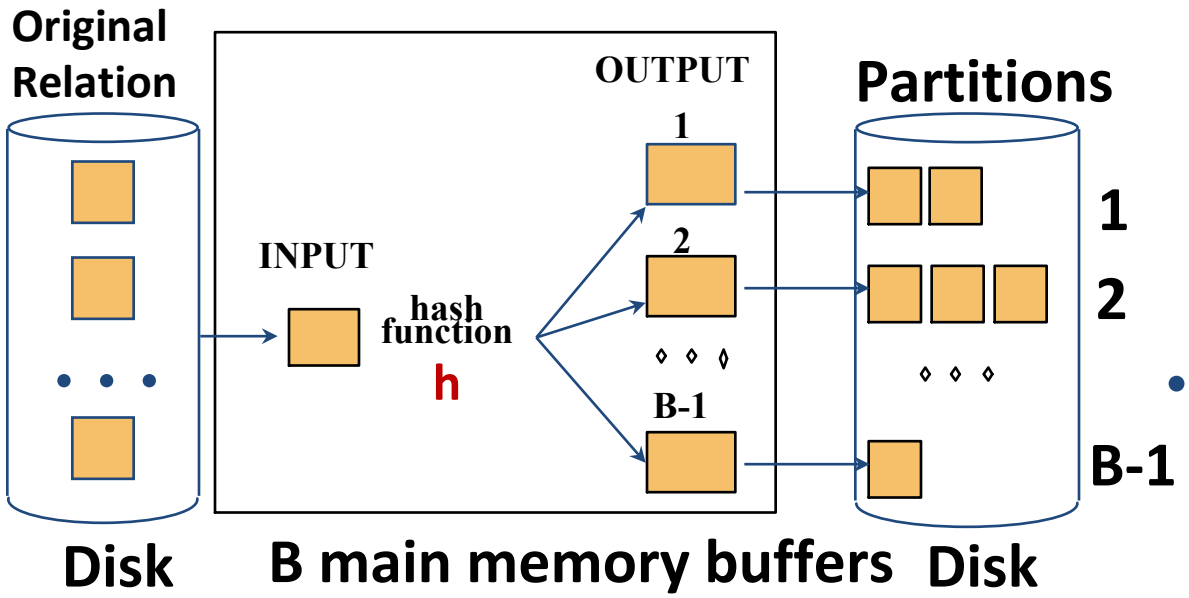
- Sort R and S on the join column, then scan them to do a ``merge'' (on join col.), and output result tuples.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value in Ri (*current R group*) and all S tuples with same value in Sj (*current S group*) *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.
- R is scanned once; each S group is scanned once per matching R tuple. (Multiple scans of an S group are likely to find needed pages in buffer.)

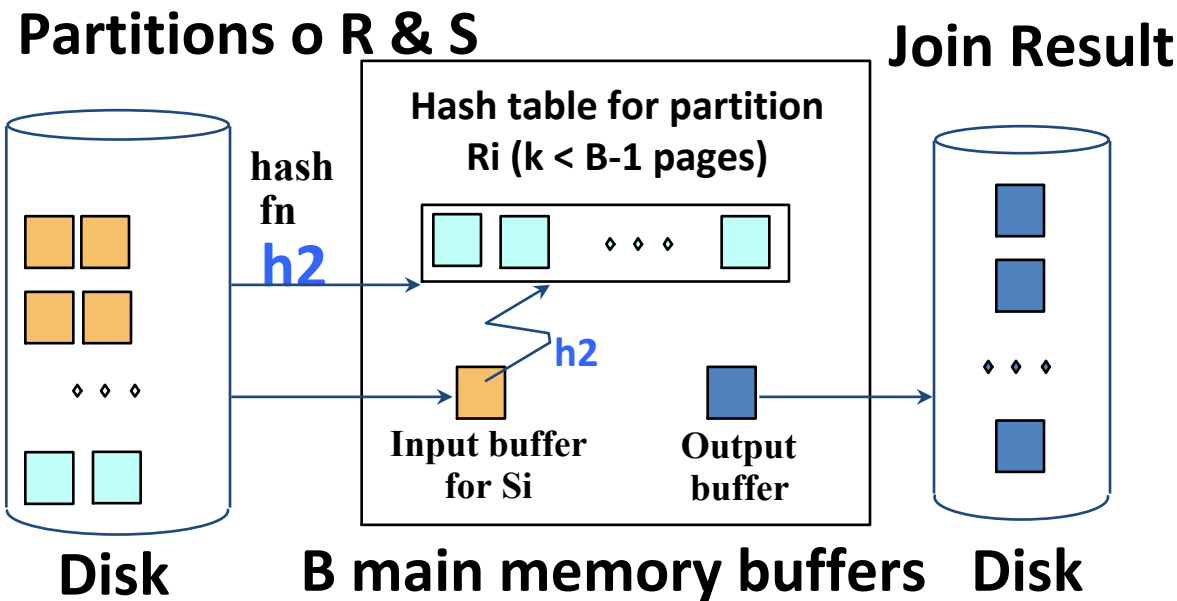
Example of Sort-Merge Join

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Hash-Join

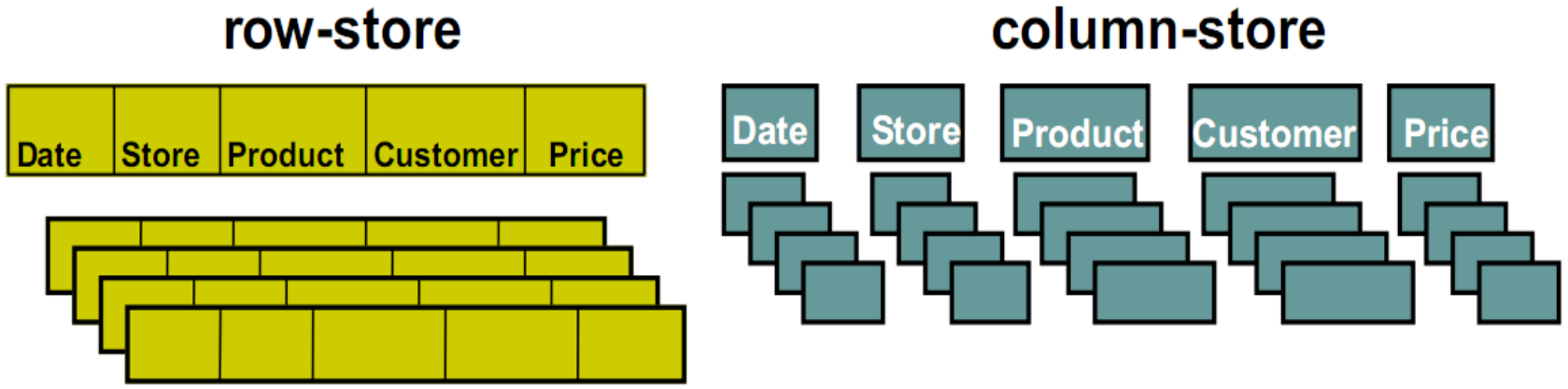




- Partition both relations using hash fn h: R tuples in partition i will only match S tuples in partition i.
- Read in a partition of R, hash it using **h2 (<> h!)**. Scan matching partition of S, search for matches.

An invention in 2000s: Column Stores for OLAP

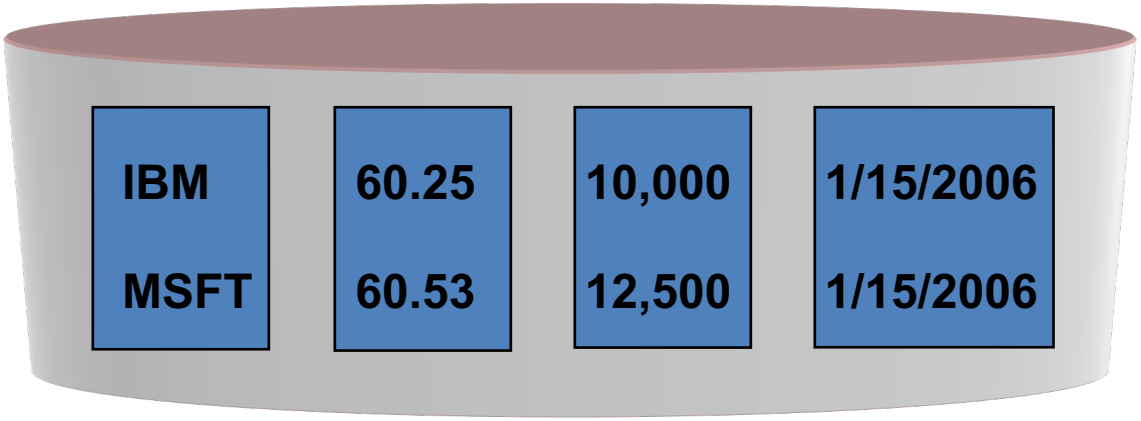
Row Store and Column Store



- In row store data are stored in the disk tuple by tuple.
- Where in column store data are stored in the disk column by column

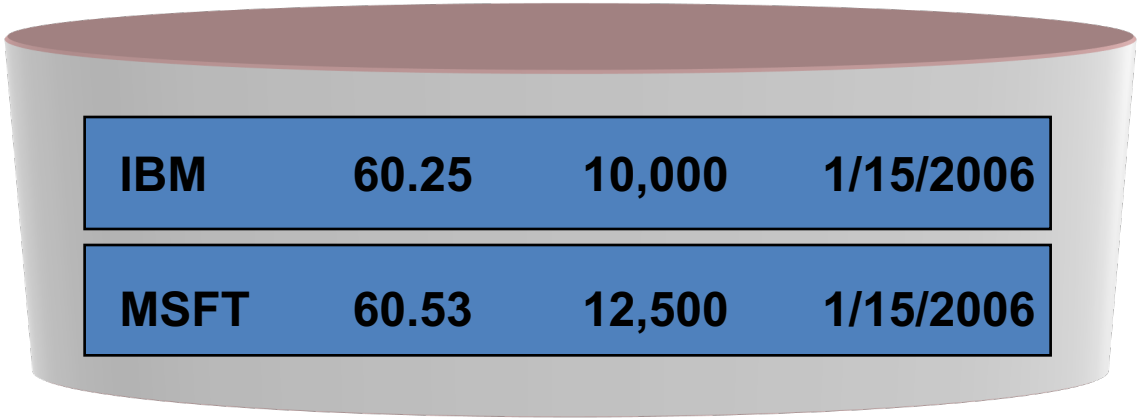
Row Store vs. Column Store

Column Store:



Used in: Sybase IQ, **Vertica**

Row Store:



Used in: Oracle, SQL Server, DB2, Netezza,…

Row Store and Column Store

For example the query

```
SELECT account.account_number,
        sum (usage.toll_airtime),
        sum (usage.toll_price)
FROM usage, toll, source, account
WHERE usage.toll_id = toll.toll_id
AND usage.source_id = source.source_id
AND usage.account_id = account.account_id
AND toll.type_ind in (‘AE’. ‘AA’)
AND usage.toll_price > 0
AND source.type != ‘CIBER’
AND toll.rating_method = ‘IS’
AND usage.invoice_date = 20051013
GROUP BY account.account_number
```

Row-store: one row = 212 columns!

Column-store: 7 attributes

Row Store and Column Store

Row Store	Column Store
(+) Easy to add/modify a record	(+) Only need to read in relevant data
(-) Might read in unnecessary data	(-) Tuple writes require multiple accesses

Column stores are suitable for read-mostly, read-intensive, large data repositories

Column Stores: High Level

- Read only what you need
  - "Fat" fact tables are typical
  - Analytics read only a few columns
- Better compression
- Execute on compressed data
- Materialized views help row stores and column stores about equally

Data Model (Vertica/C-Store)

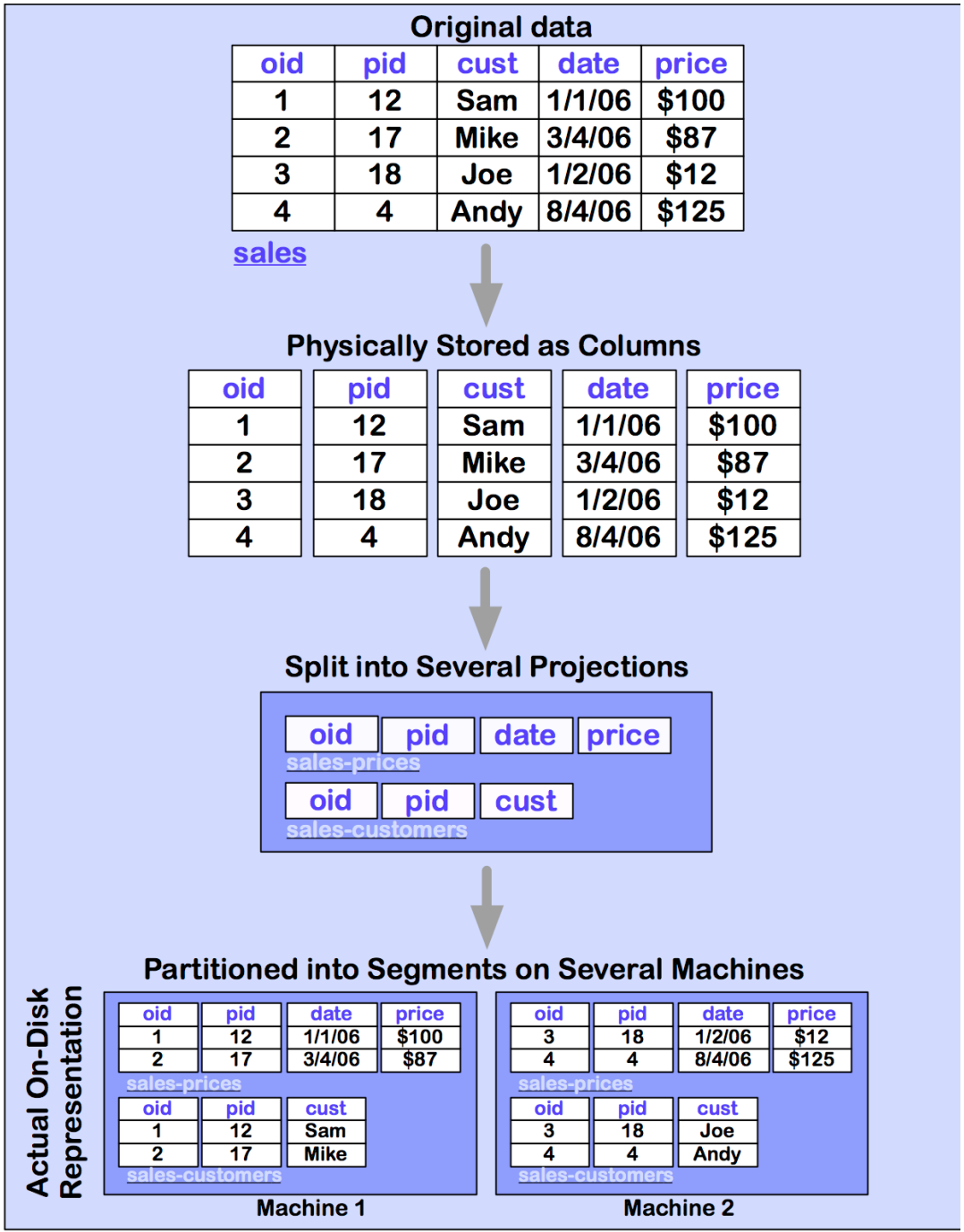
- Same as relational data model
  - Tables, rows, columns
  - Primary keys and foreign keys
  - **Projections**
    - From single table
    - Multiple joined tables
- Example

Normal relational model

EMP(name, age, dept, salary)
DEPT(dname, floor)

Possible C-store model

EMP1 (name, age)
EMP2 (dept, age, DEPT.floor)
EMP3 (name, salary)
DEPT1(dname, floor)


Original data
oid pid cust date price
1 12 Sam 1/1/06 $100
2 17 Mike 3/4/06 $87
3 18 Joe 1/2/06 $12
4 4 Andy 8/4/06 $125
sales
Physically Stored as Columns
oid pid cust date price
1 12 Sam 1/1/06 $100
2 17 Mike 3/4/06 $87
3 18 Joe 1/2/06 $12
4 4 Andy 8/4/06 $125
Split into Several Projections
oid pid date price
sales-prices
oid pid cust
sales-customers
Partitioned into Segments on Several Machines
Actual On-Disk Representation
oid pid date price
1 12 1/1/06 $100
2 17 3/4/06 $87
sales-prices
oid pid cust
1 12 Sam
2 17 Mike
sales-customers
Machine 1
oid pid date price
3 18 1/2/06 $12
4 4 8/4/06 $125
sales-prices
oid pid cust
3 18 Joe
4 4 Andy
sales-customers
Machine 2

Read Store: Column Encoding/Compression

- Use compression schemes and indices
  - Null Suppression
  - Dictionary encoding
  - Run Length encoding
  - Bit-Vector encoding
  - Self-order (key), few distinct values
    - (value, position, # items)
    - Indexed by clustered B-tree
  - Foreign-order (non-key), few distinct values
    - (value, bitmap index)
    - B-tree index: position → values
  - Self-order, many distinct values
    - Delta from the previous value
    - B-tree index
  - Foreign-order, many distinct values
    - Unencoded

Compression

- Trades I/O for CPU
  - Increased column-store opportunities:
  - Higher data value locality in column stores
  - Techniques such as run length encoding far more useful

Evaluation

- Use TPC-H – decision support queries
- Storage

C-Store	Row Store	Column Store
1.987 GB	4.480 GB	2.650 GB

Query Performance

Query	C-Store	Row Store	Column Store
Q1	0.03	6.80	2.24
Q2	0.36	1.09	0.83
Q3	4.90	93.26	29.54
Q4	2.09	722.90	22.23
Q5	0.31	116.56	0.93
Q6	8.50	652.90	32.83
Q7	2.54	265.80	33.24

Query Performance

- Row store uses materialized views

Query	C-Store	Row Store	Column Store
Q1	0.03	0.22	2.34
Q2	0.36	0.81	0.83
Q3	4.90	49.38	29.10
Q4	2.09	21.76	22.23
Q5	0.31	0.70	0.63
Q6	8.50	47.38	25.46
Q7	2.54	18.47	6.28

Summary: The Performance Gain

- Column representation – avoids reads of unused attributes
- Storing overlapping projections – multiple orderings of a column, more choices for query optimization
- Compression of data – more orderings of a column in the same amount of space
- Query operators operate on

  compressed representation

Take-home Messages

- OLAP
  - Multi-relational Data model
  - Operators
  - SQL
- Data warehouse (architecture, issues, optimizations)
- Join Processing
- Column Stores (Optimized for OLAP workload)