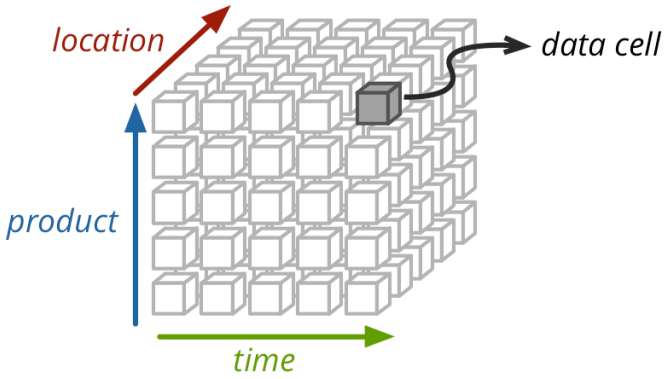
Warehouse Models & Operators

- Data Models
  - relations
  - stars & snowflakes
  - cubes
- Operators
  - slice & dice
  - roll-up, drill down
  - pivoting
  - other



Star

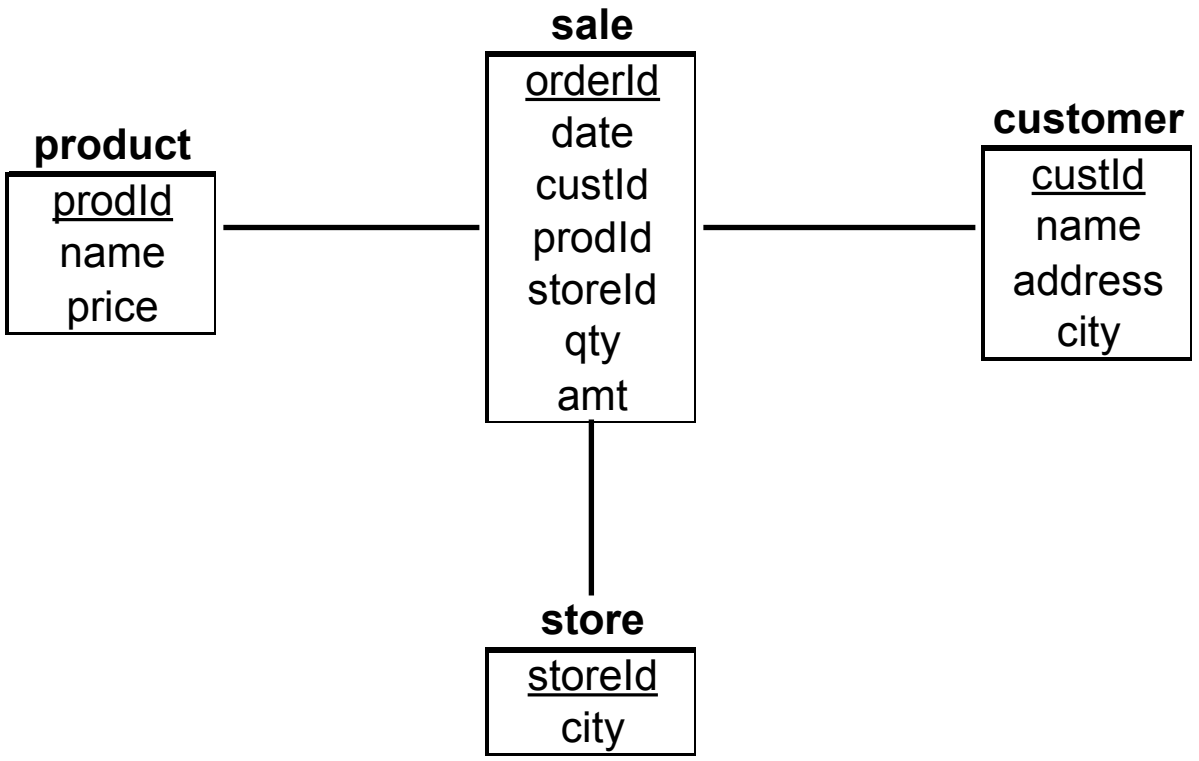
product	prodId	name	price
	p1	bolt	10
	p2	nut	5

store	storeId	city
	c1	nyc
	c2	sfo
	c3	la

sale	oderId	date	custId	prodId	storeId	qty	amt
	o100	1/7/97	53	p1	c1	1	12
	o102	2/7/97	53	p2	c1	2	11
	105	3/8/97	111	p1	c3	5	50

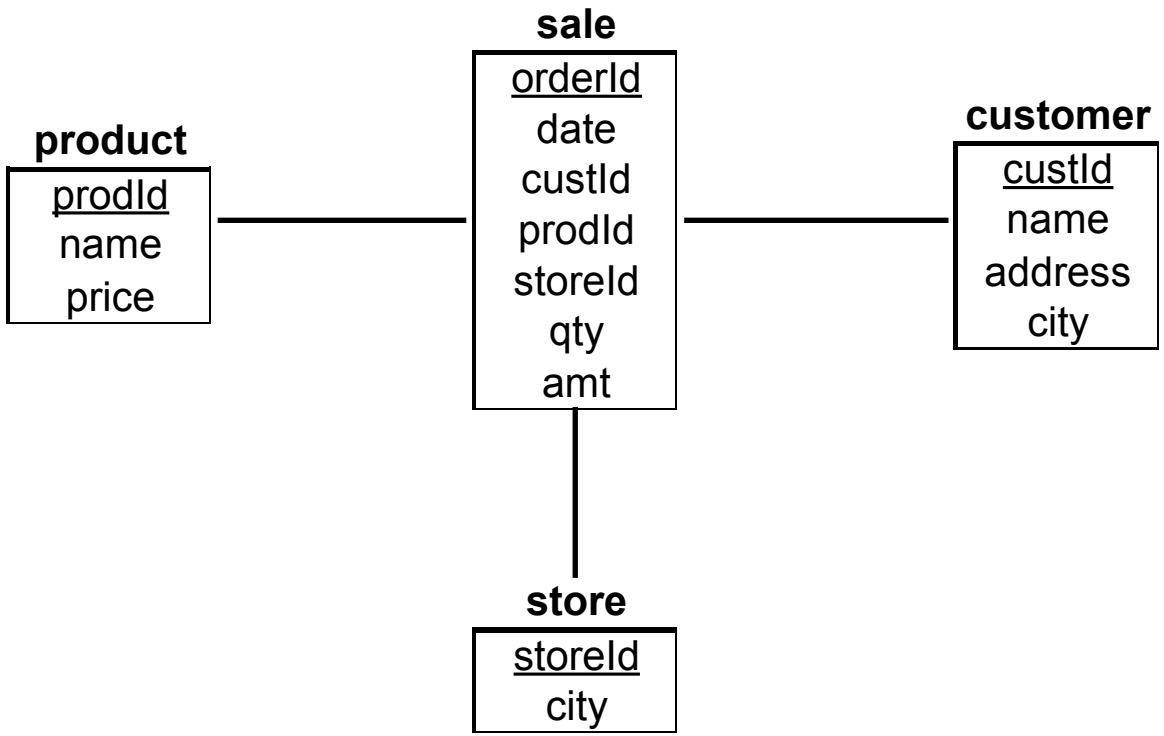
customer	custId	name	address	city
	53	joe	10 main	sfo
	81	fred	12 main	sfo
	111	sally	80 willow	la

Star Schema

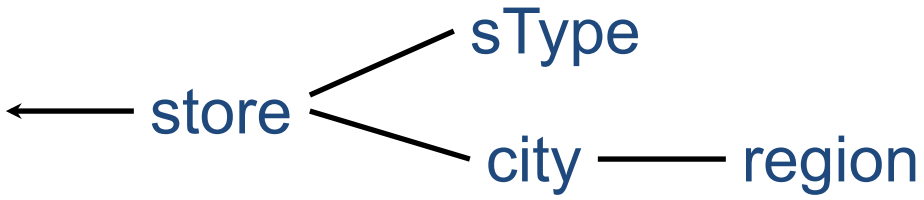

sale
orderId
date
custId
prodId
storeId
qty
amt
product
prodId
name
price
customer
custId
name
address
city
store
storeId
city

Terms

- Fact table
- Dimension tables
- Measures



Dimension Hierarchies



store	storeId	cityId	tId	mgr
	s5	sfo	t1	joe
	s7	sfo	t2	fred
	s9	la	t1	nancy

sType	tId	size	location
	t1	small	downtown
	t2	large	suburbs

city	cityId	pop	regId
	sfo	1M	north
	la	5M	south

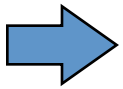
region	regId	name
	north	cold region
	south	warm region

- ➔ snowflake schema
- ➔ constellations

Aggregates

- Add up amounts for day 1
- In SQL: SELECT sum(amt) FROM SALE
  WHERE date = 1

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

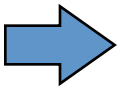


81

Aggregates

- Add up amounts by day
- In SQL: SELECT date, sum(amt) FROM SALE
  GROUP BY date

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

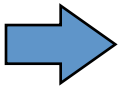


ans	date	sum
	1	81
	2	48

Another Example

- Add up amounts by day, product
- In SQL: SELECT date, sum(amt) FROM SALE
  GROUP BY date, prodId

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4



sale	prodId	date	amt
	p1	1	62
	p2	1	19
	p1	2	48

rollup →

← drill-down

ROLAP vs. MOLAP

- ROLAP:
  Relational On-Line Analytical Processing
- MOLAP:
  Multi-Dimensional On-Line Analytical Processing

Cube

Fact table view:

sale	prodId	storeId	amt
	p1	c1	12
	p2	c1	11
	p1	c3	50
	p2	c2	8

Multi-dimensional cube:

	c1	c2	c3
p1	12		50
p2	11	8	

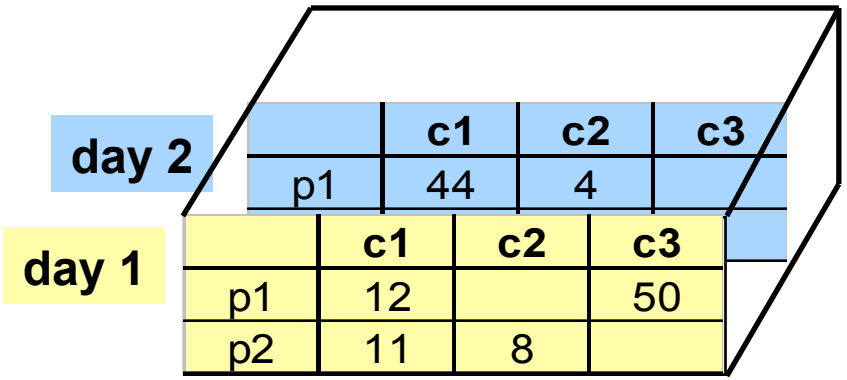
dimensions = 2

3D Cube

Fact table view:

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

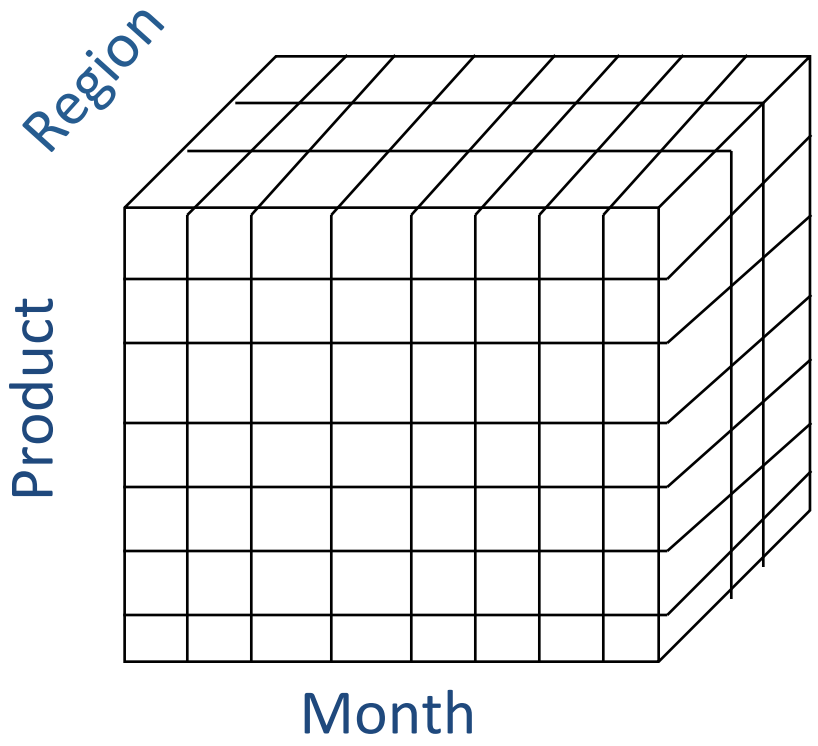
Multi-dimensional cube:



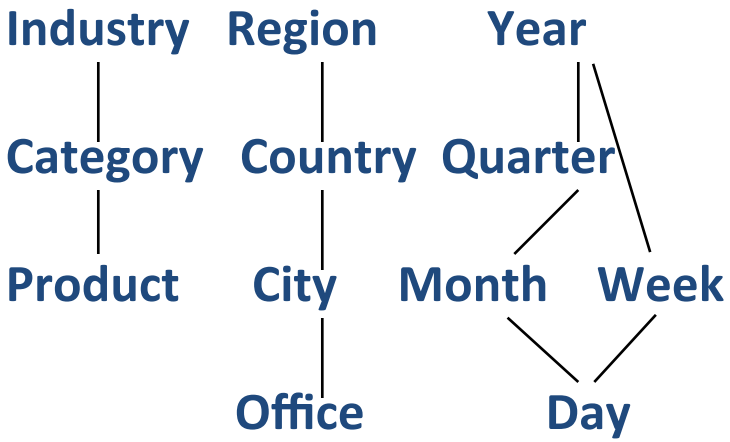
dimensions = 3

Multidimensional Data

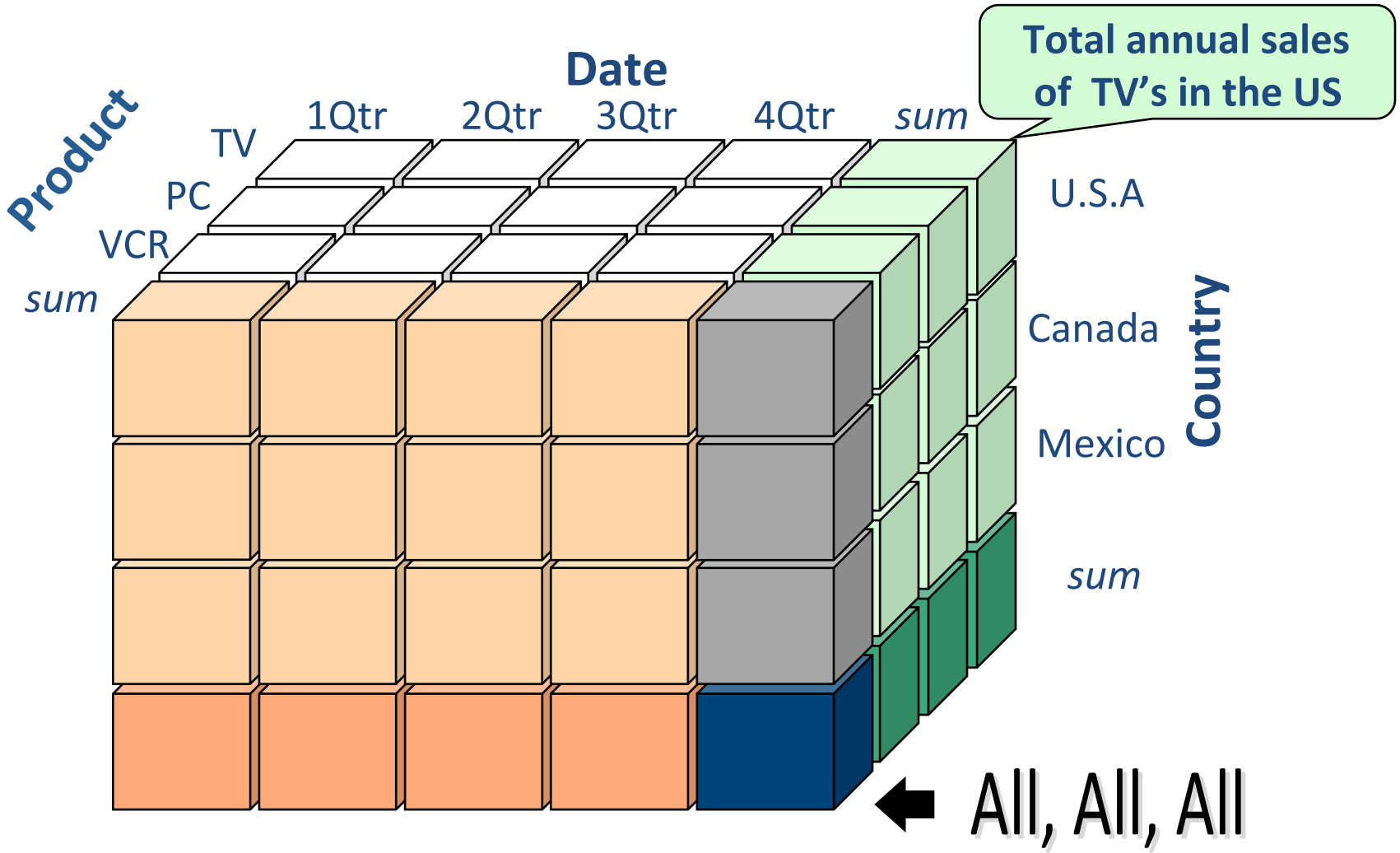
Sales volume as a function of product, month, and region



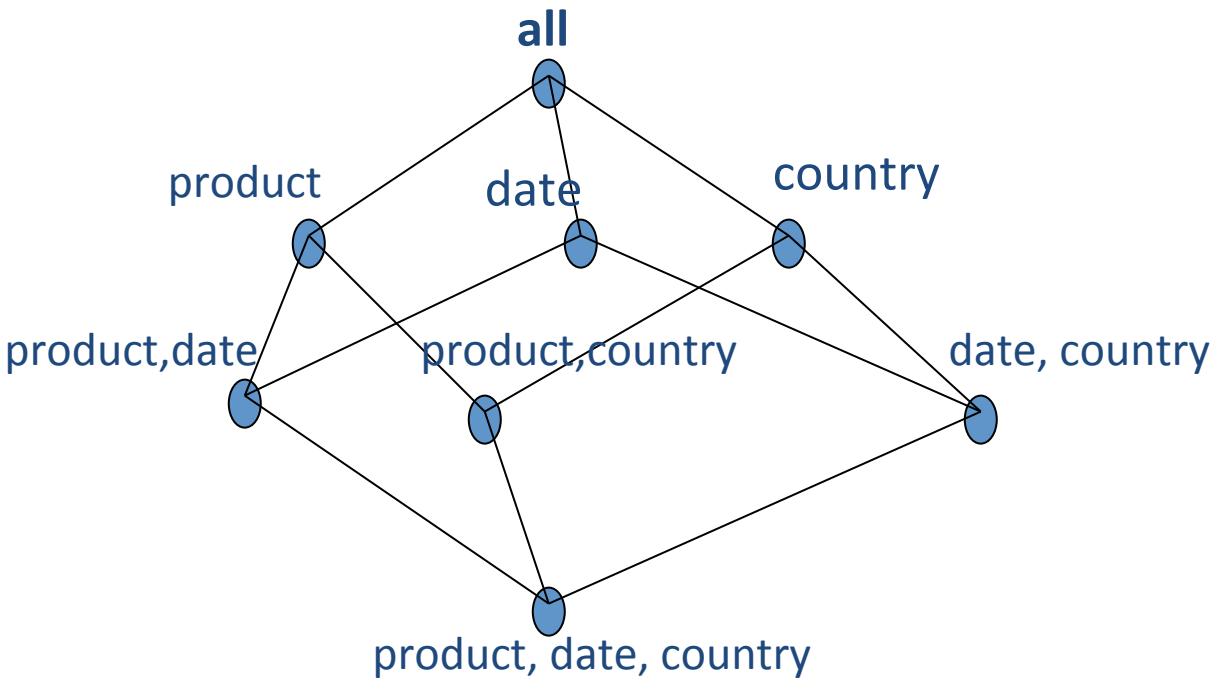
Dimensions: Product, Location, Time
Hierarchical summarization paths



A Sample Data Cube



Cuboids Corresponding to the Cube



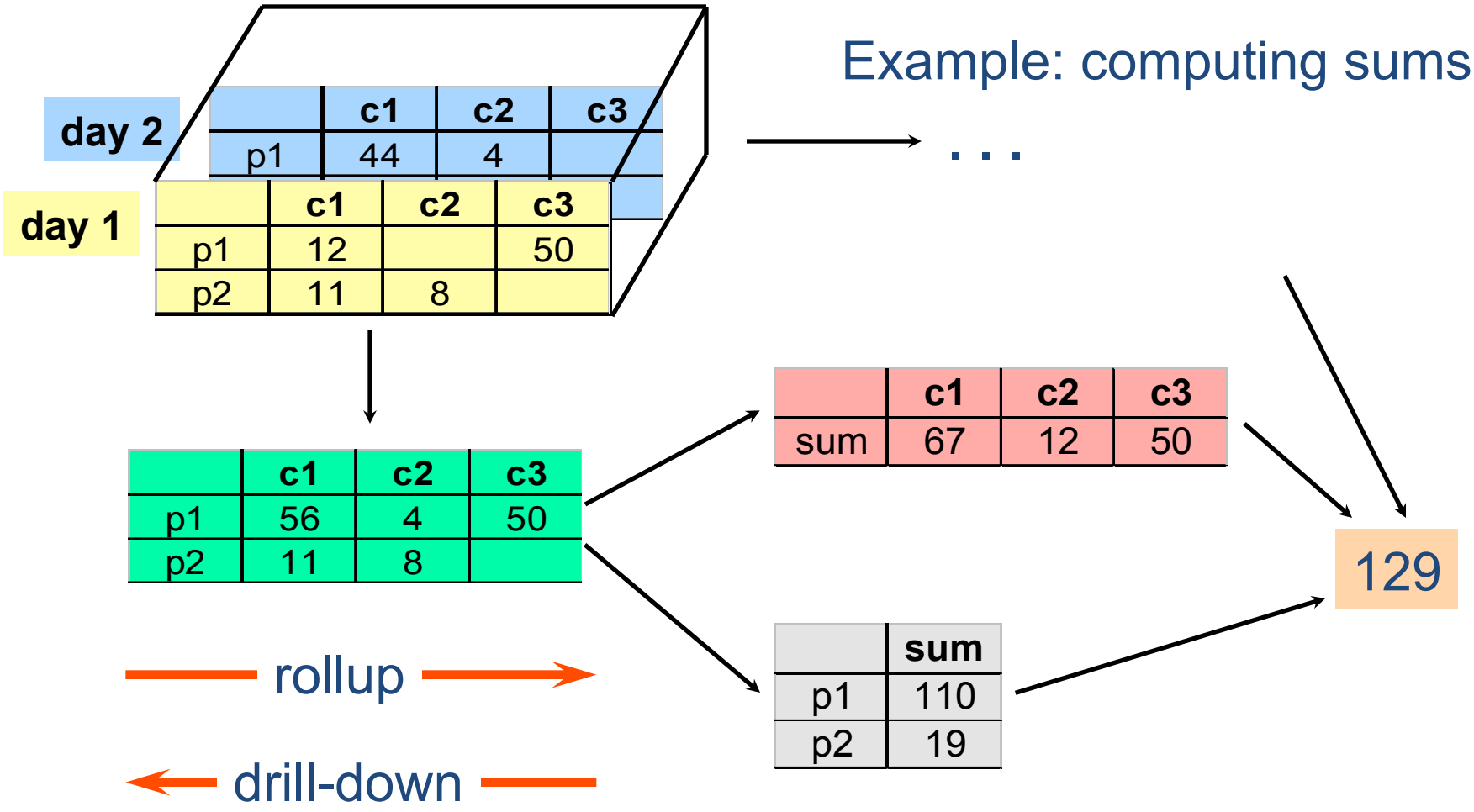
0−D(apex) cuboid

1−D cuboids

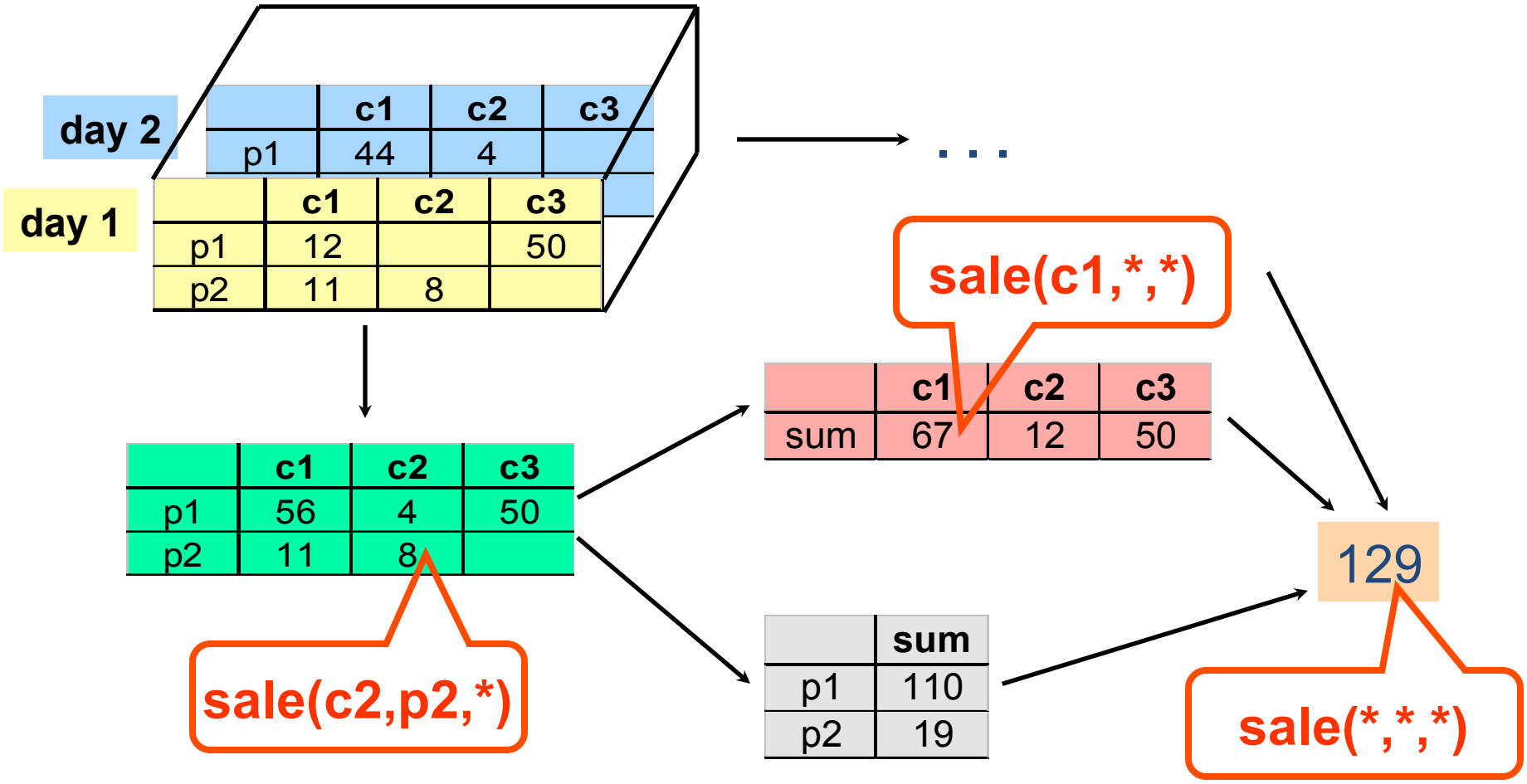
2−D cuboids

3−D(base) cuboid

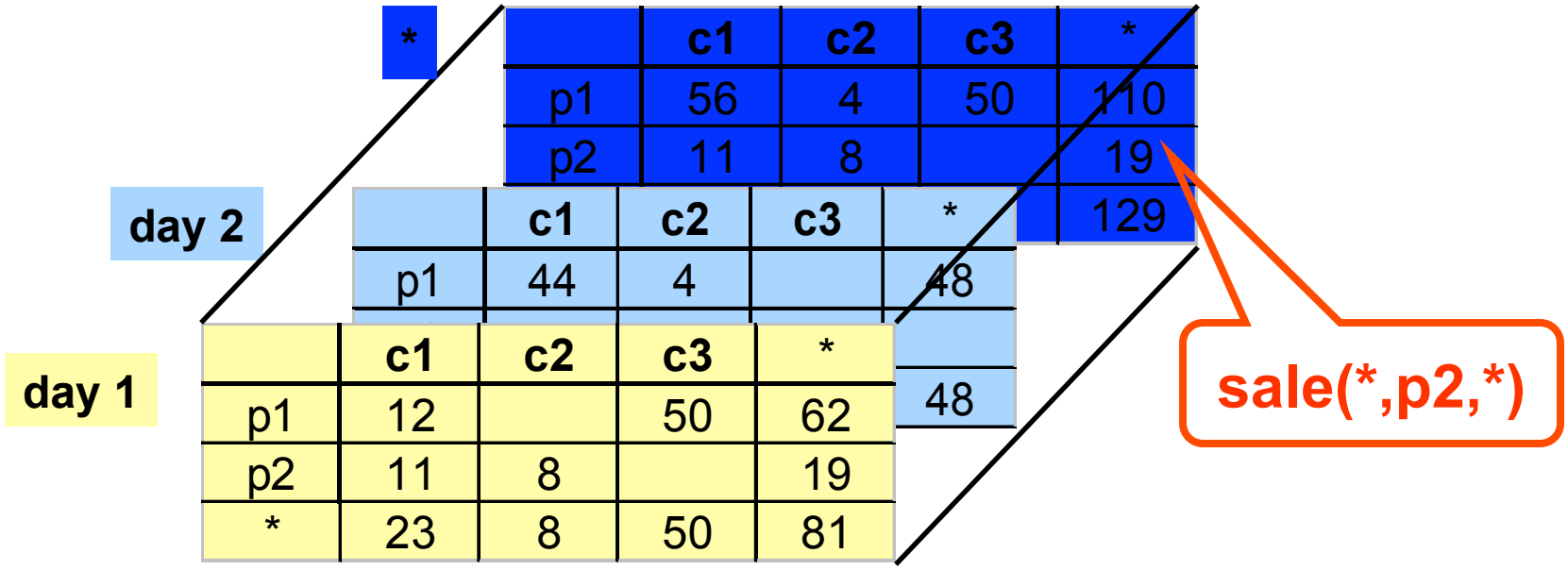
Cube Aggregation



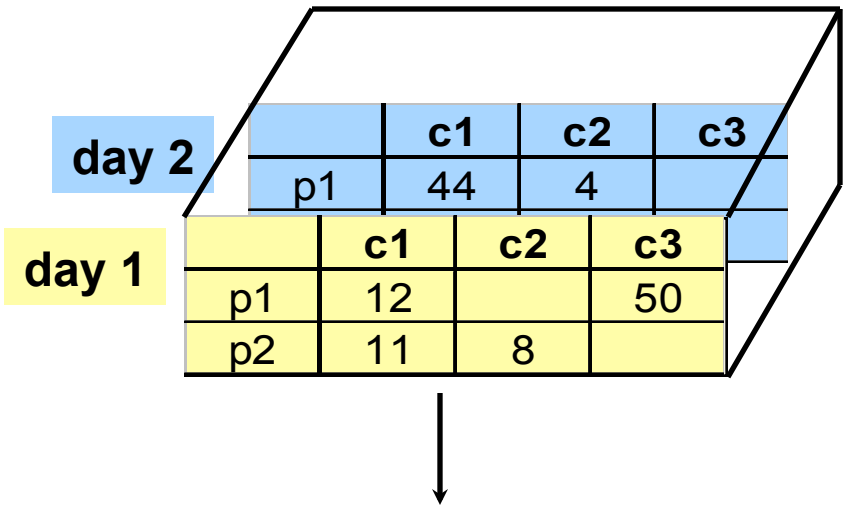
Cube Operators



Extended Cube



Aggregation Using Hierarchies



	region A	region B
p1	56	54
p2	11	8

customer

region

country

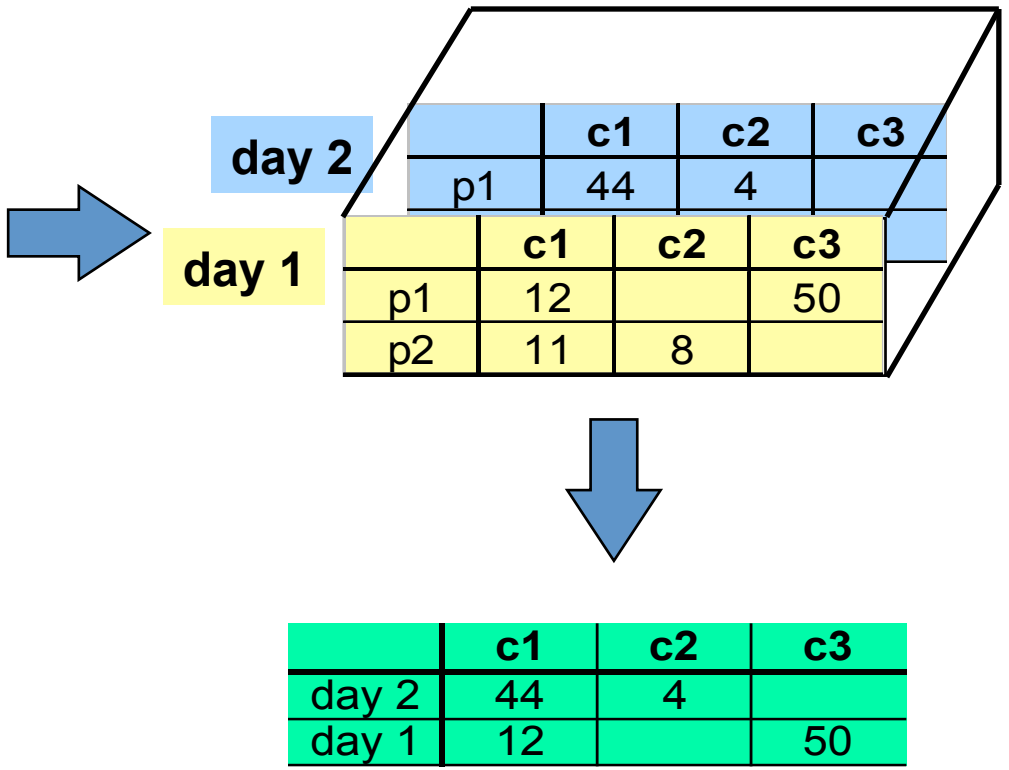
(customer c1 in Region A;
customers c2, c3 in Region B)

Pivoting

Fact table view:

sale	prodId	storeId	date	amt
	p1	c1	1	12
	p2	c1	1	11
	p1	c3	1	50
	p2	c2	1	8
	p1	c1	2	44
	p1	c2	2	4

Multi-dimensional cube:



	c1	c2	c3
day 2	44	4	
day 1	12		50

CUBE Operator (SQL-99)

Chevy Sales Cross Tab				
Chevy	*1990*	*1991*	*1992*	*Total (ALL)*
black	50	85	154	289
white	40	115	199	354
Total (ALL)	90	200	353	1286

```
SELECT      model, year, color, sum(sales) as sales
FROM        sales
WHERE       model in ('Chevy')
AND         year BETWEEN 1990 AND 1992
GROUP BY    CUBE (model, year, color);
```

CUBE Contd.

```
SELECT      model, year, color, sum(sales) as sales
FROM        sales
WHERE       model in ('Chevy')
AND         year BETWEEN 1990 AND 1992
GROUP BY    CUBE (model, year, color);
```

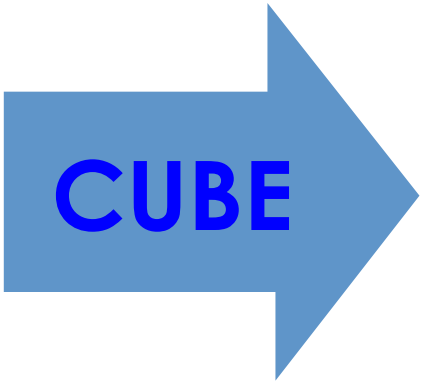
Computes union of 8 different groupings:

{(model, year, color), (model, year), (model, color), (year, color), (model), (year), (color), ()}

Example Contd.

SALES

Model	Year	Color	Sales
Chevy	1990	red	5
Chevy	1990	white	87
Chevy	1990	blue	62
Chevy	1991	red	54
Chevy	1991	white	95
Chevy	1991	blue	49
Chevy	1992	red	31
Chevy	1992	white	54
Chevy	1992	blue	71
Ford	1990	red	64
Ford	1990	white	62
Ford	1990	blue	63
Ford	1991	red	52
Ford	1991	white	9
Ford	1991	blue	55
Ford	1992	red	27
Ford	1992	white	62
Ford	1992	blue	39



DATA CUBE

Model	Year	Color	Sales
ALL	ALL	ALL	942
chevy	ALL	ALL	510
ford	ALL	ALL	432
ALL	1990	ALL	343
ALL	1991	ALL	314
ALL	1992	ALL	285
ALL	ALL	red	165
ALL	ALL	white	273
ALL	ALL	blue	339
chevy	1990	ALL	154
chevy	1991	ALL	199
chevy	1992	ALL	157
ford	1990	ALL	189
ford	1991	ALL	116
ford	1992	ALL	128
chevy	ALL	red	91
chevy	ALL	white	236
chevy	ALL	blue	183
ford	ALL	red	144
ford	ALL	white	133
ford	ALL	blue	156
ALL	1990	red	69
ALL	1990	white	149
ALL	1990	blue	125
ALL	1991	red	107
ALL	1991	white	104
ALL	1991	blue	104
ALL	1992	red	59
ALL	1992	white	116
ALL	1992	blue	110

Aggregates

- Operators: sum, count, max, min, median, ave
- “Having” clause
- Cube (& Rollup) operator
- Using dimension hierarchy
  - average by region (within store)
  - maximum by month (within date)

Query & Analysis Tools

- Query Building
- Report Writers (comparisons, growth, graphs,…)
- Spreadsheet Systems
- Web Interfaces
- Data Mining

Other Operations

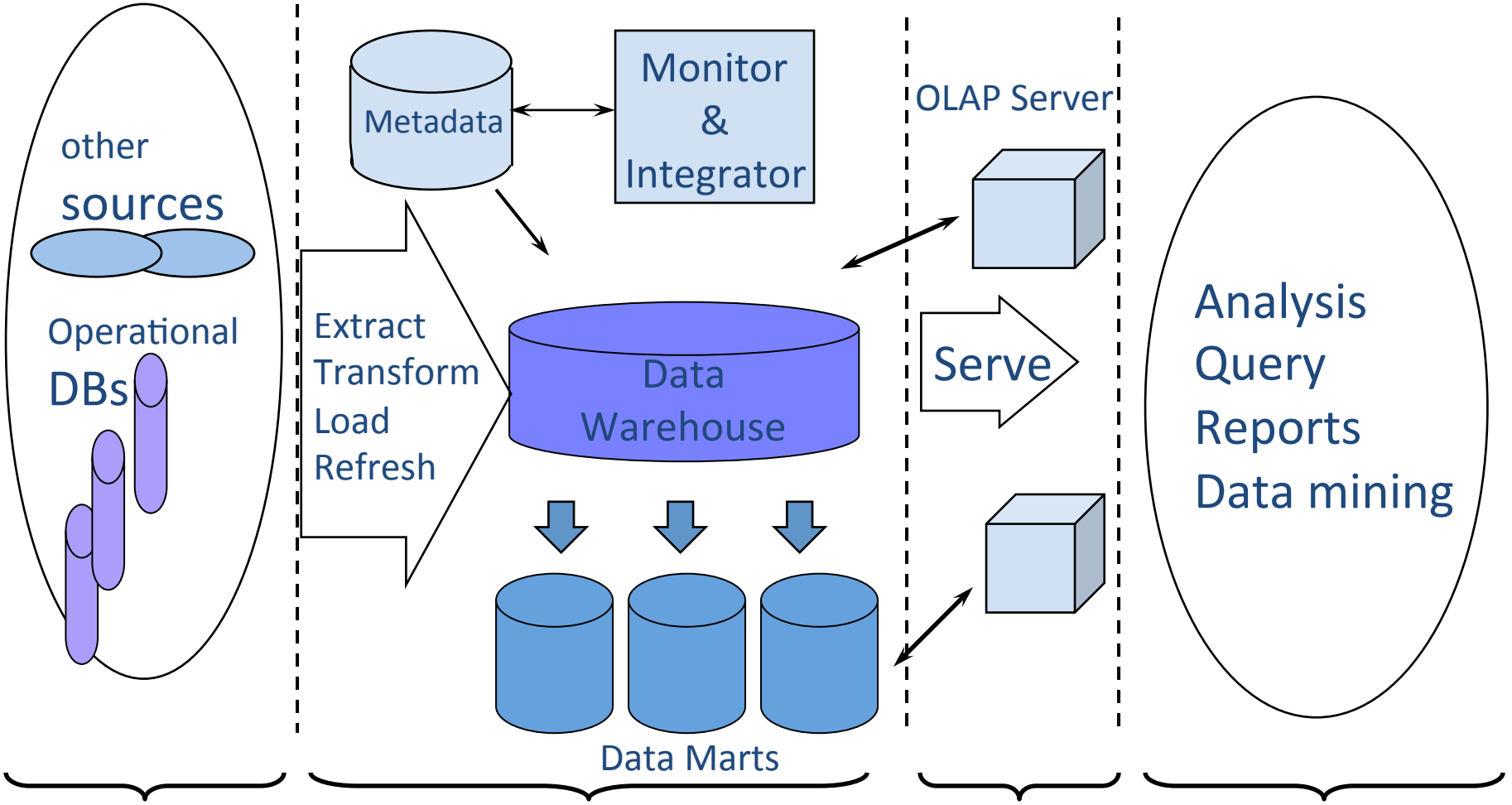
- Time functions
  - e.g., time average
- Computed Attributes
  - e.g., commission = sales * rate
- Text Queries
  - e.g., find documents with words X AND B
  - e.g., rank documents by frequency of words X, Y, Z

Data Warehouse Implementation

Implementing a Warehouse

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

Multi-Tiered Architecture



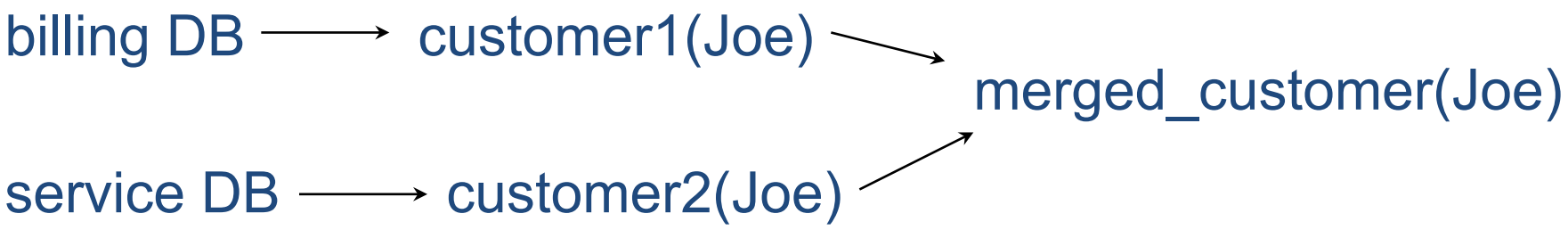
Monitoring

- Source Types: relational, flat file, IMS, VSAM, IDMS, 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

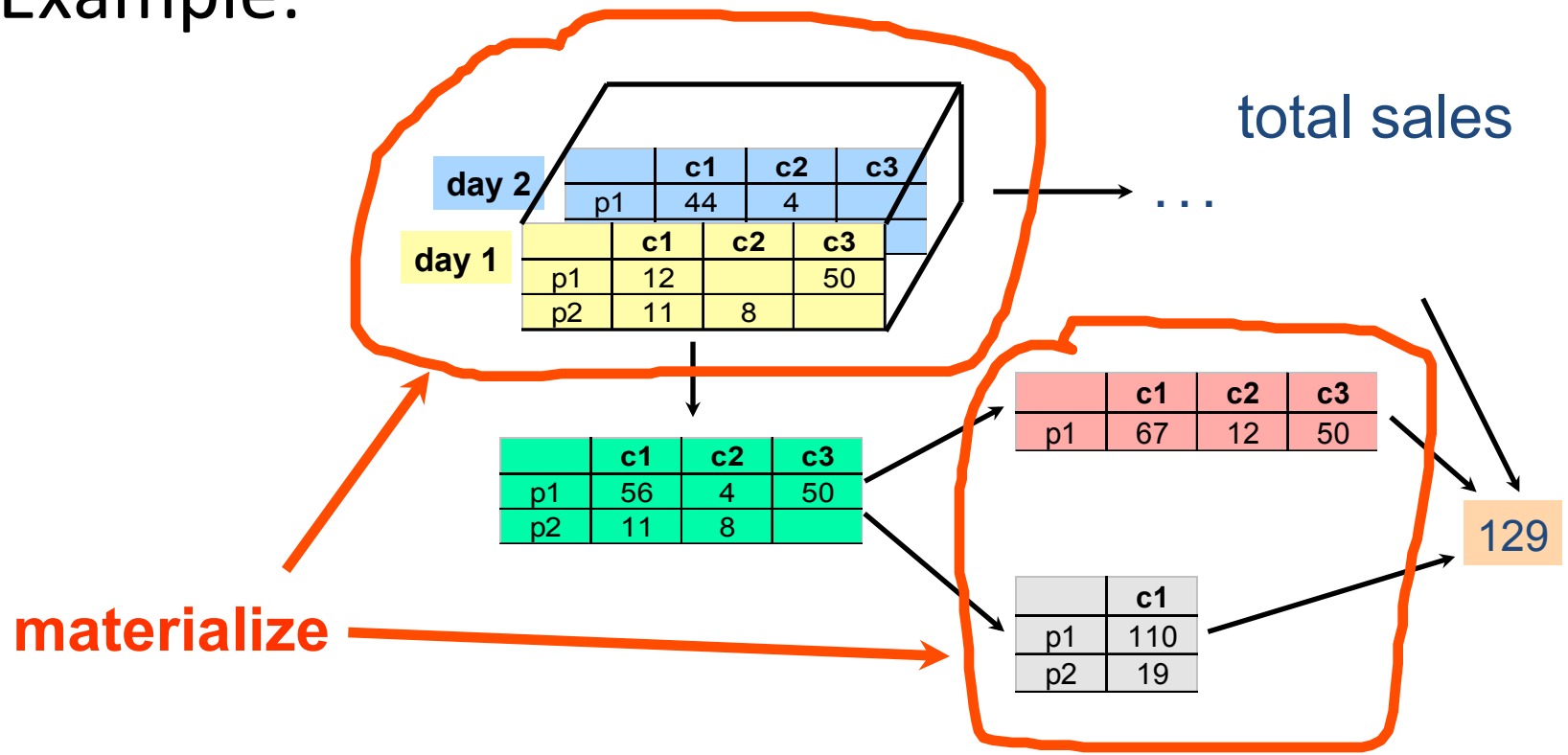
OLAP Implementation

Derived Data

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

What to Materialize?

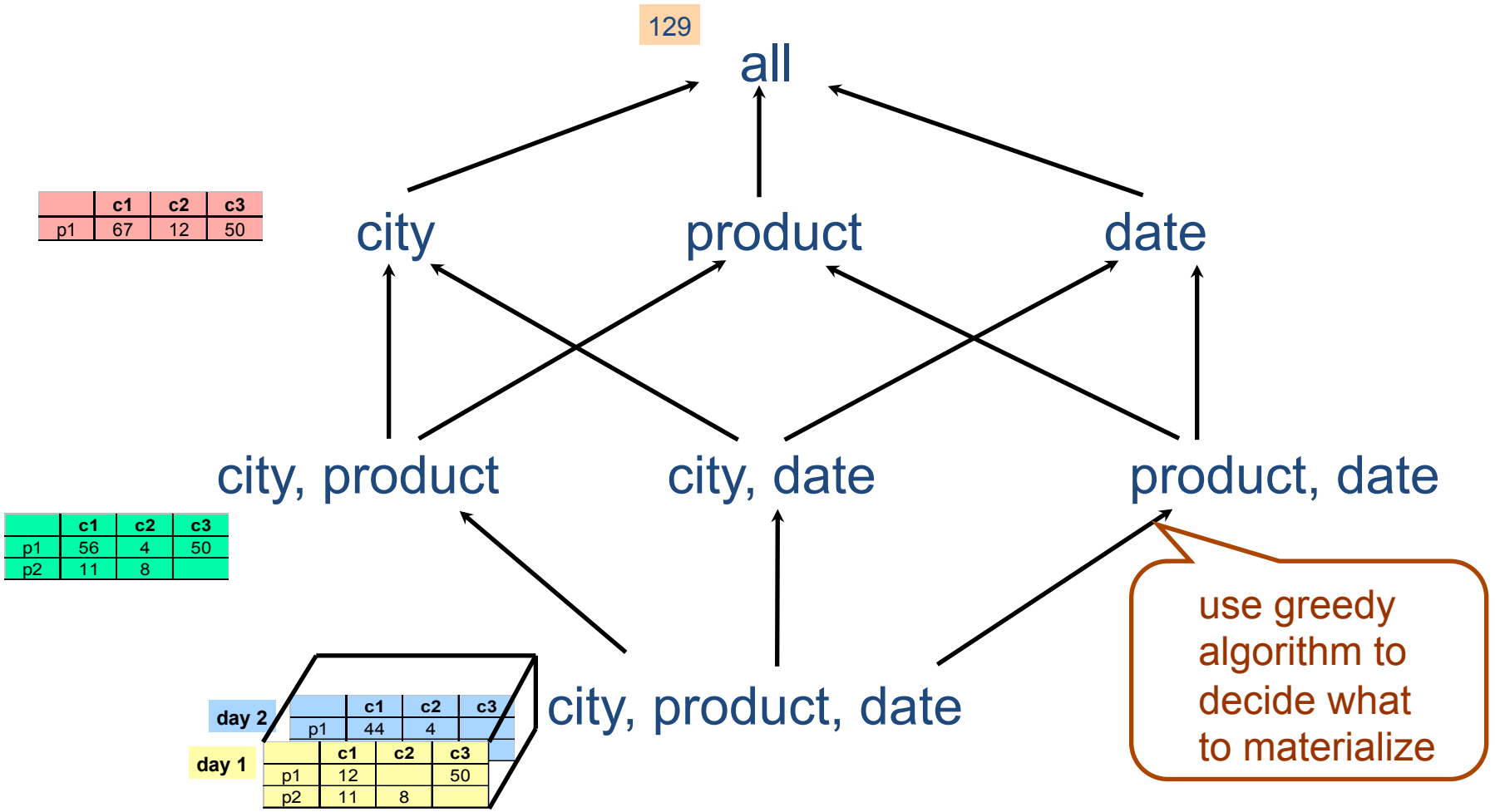
- Store in warehouse results useful for common queries
- Example:



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	

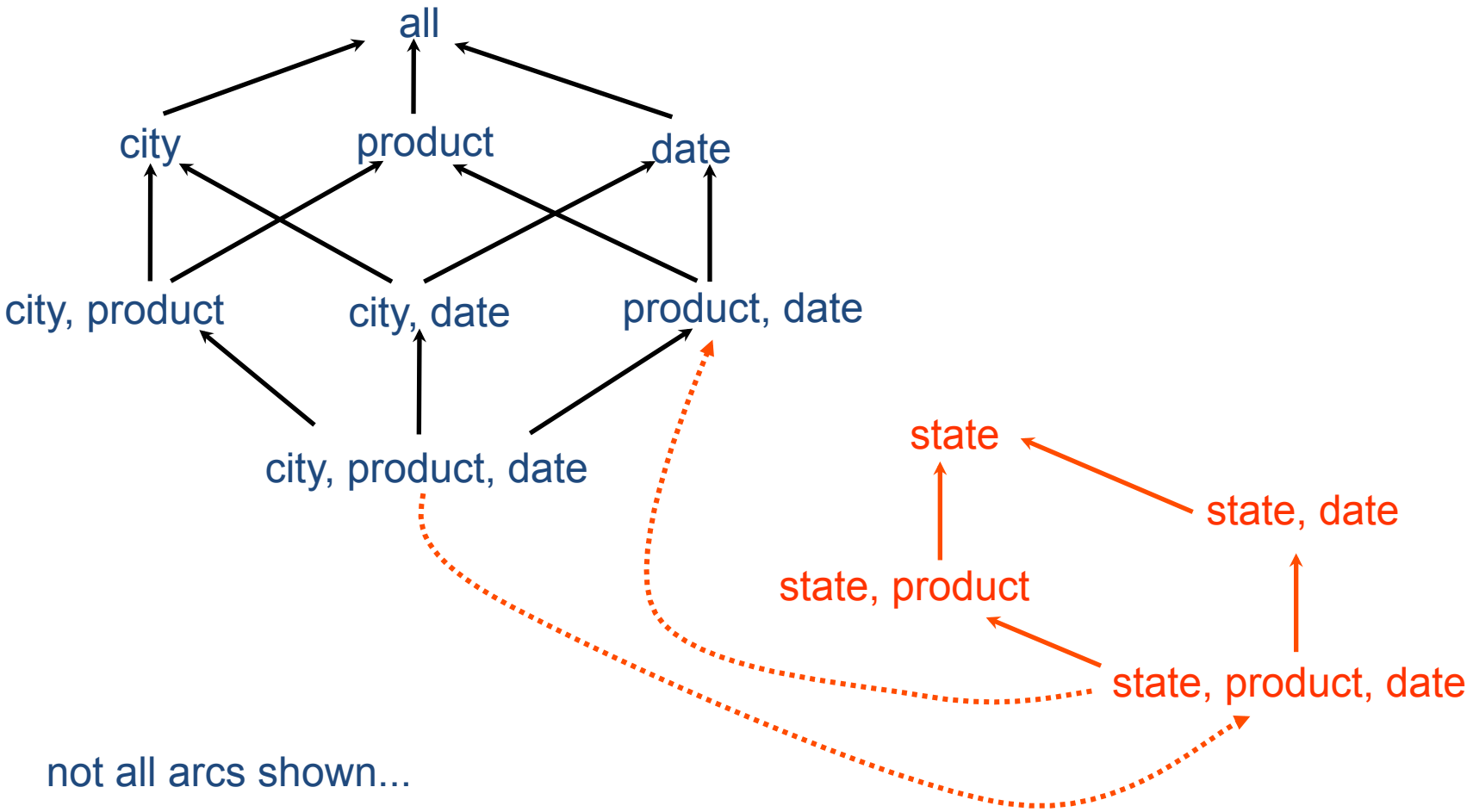
	c1	c2	c3
p1	44	4	

	c1	c2	c3
p1	12		50
p2	11	8	

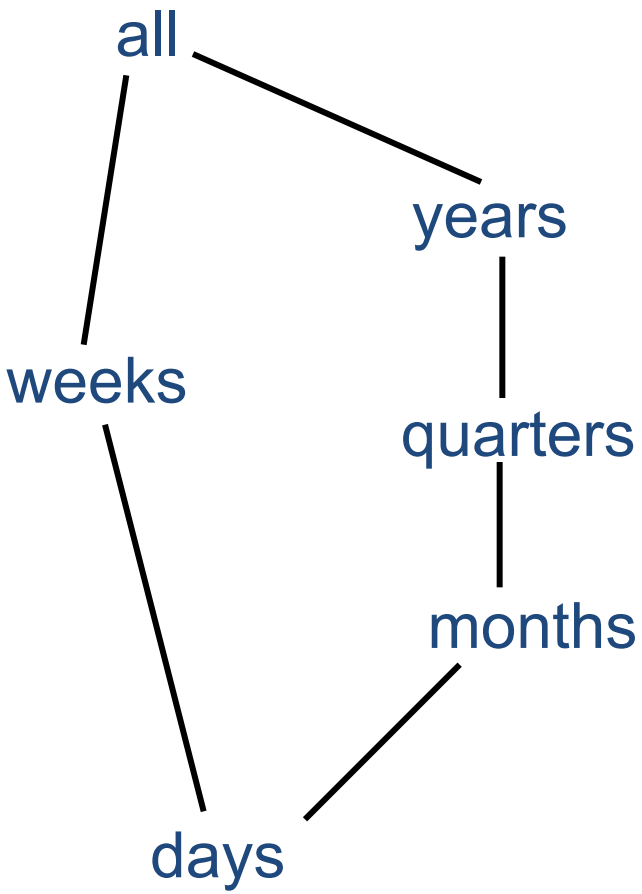
Dimension Hierarchies

cities	city	state
	c1	CA
	c2	NY

Dimension Hierarchies



Interesting Hierarchy



time	day	week	month	quarter	year
	1	1	1	1	2000
	2	1	1	1	2000
	3	1	1	1	2000
	4	1	1	1	2000
	5	1	1	1	2000
	6	1	1	1	2000
	7	1	1	1	2000
	8	2	1	1	2000

conceptual dimension table

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 column
- Not suitable for high cardinality domains

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)

- Similar to old schema; *rname* added for variations.
- 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.

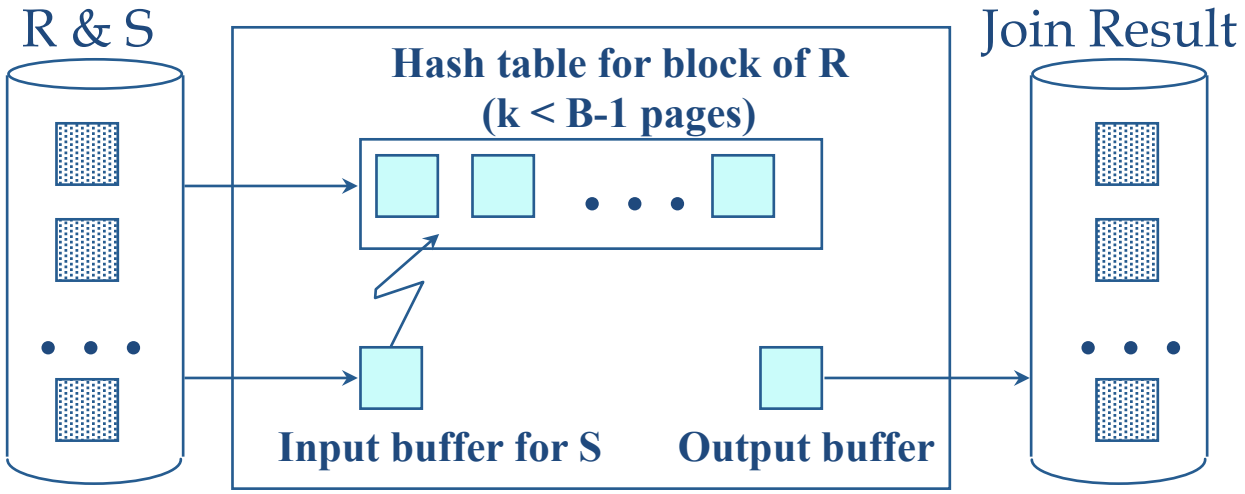
Simple Nested Loops Join

```
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.
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: $M + M*N = 1000 + 1000*500$
  - If smaller relation (S) is outer, cost = 500 + 500*1000

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.



Examples of Block Nested Loops

- Cost: Scan of outer + #outer blocks * scan of inner
  - #outer blocks =
- With Reserves (R) as outer, and 100 pages of R:
  - Cost of scanning R is 1000 I/Os; a total of 10 *blocks*.
  - Per block of R, we scan Sailors (S); 10*500 I/Os.
  - If space for just 90 pages of R, we would scan S 12 times.
- With 100-page block of Sailors as outer:
  - Cost of scanning S is 500 I/Os; a total of 5 blocks.
  - Per block of S, we scan Reserves; 5*1000 I/Os.
- With *sequential reads* considered, analysis changes: may be best to divide buffers evenly between R and S.

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)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples (assuming Alt. (2) or (3) for data entries) depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: upto 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index (Alt. 2) on *sid* of Sailors (as inner):
  - Scan Reserves: 1000 page I/Os, 100*1000 tuples.
  - For each Reserves tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Sailors tuple. Total: 220,000 I/Os.
- Hash-index (Alt. 2) on *sid* of Reserves (as inner):
  - Scan Sailors: 500 page I/Os, 80*500 tuples.
  - For each Sailors tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Reserves tuples. Assuming uniform distribution, 2.5 reservations per sailor (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on whether the index is clustered.

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

- Cost: M log M + N log N + (M+N)
  - The cost of scanning, M+N, could be M*N (very unlikely!)
- With 35, 100 or 300 buffer pages, both Reserves and Sailors can be sorted in 2 passes; total join cost: 7500

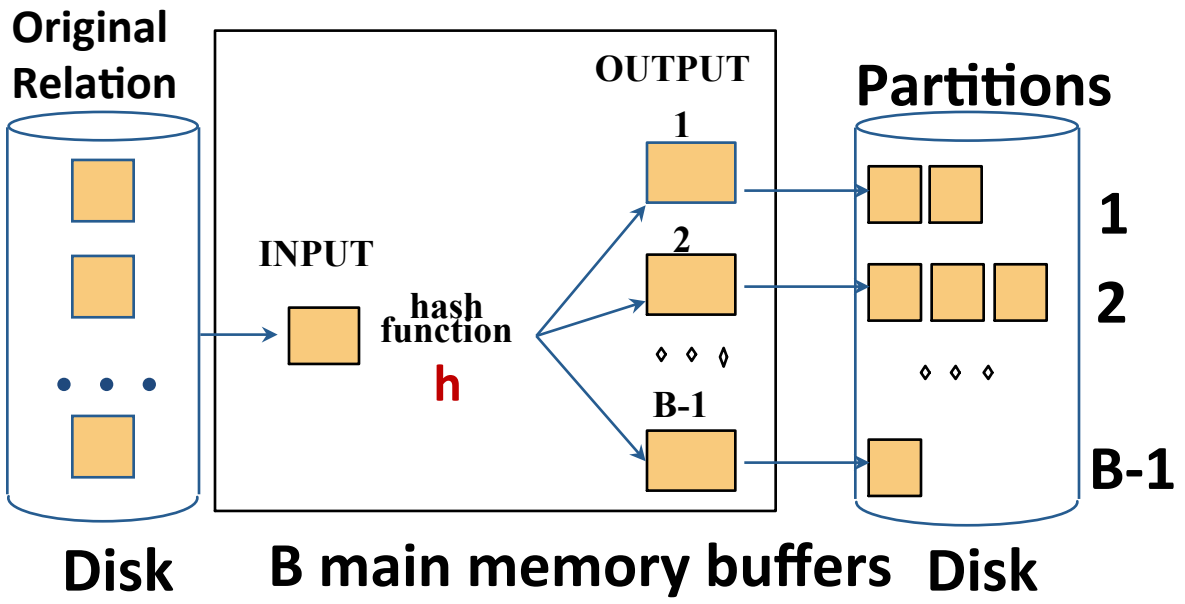
(BNL cost: 2500 to 15000 I/Os)

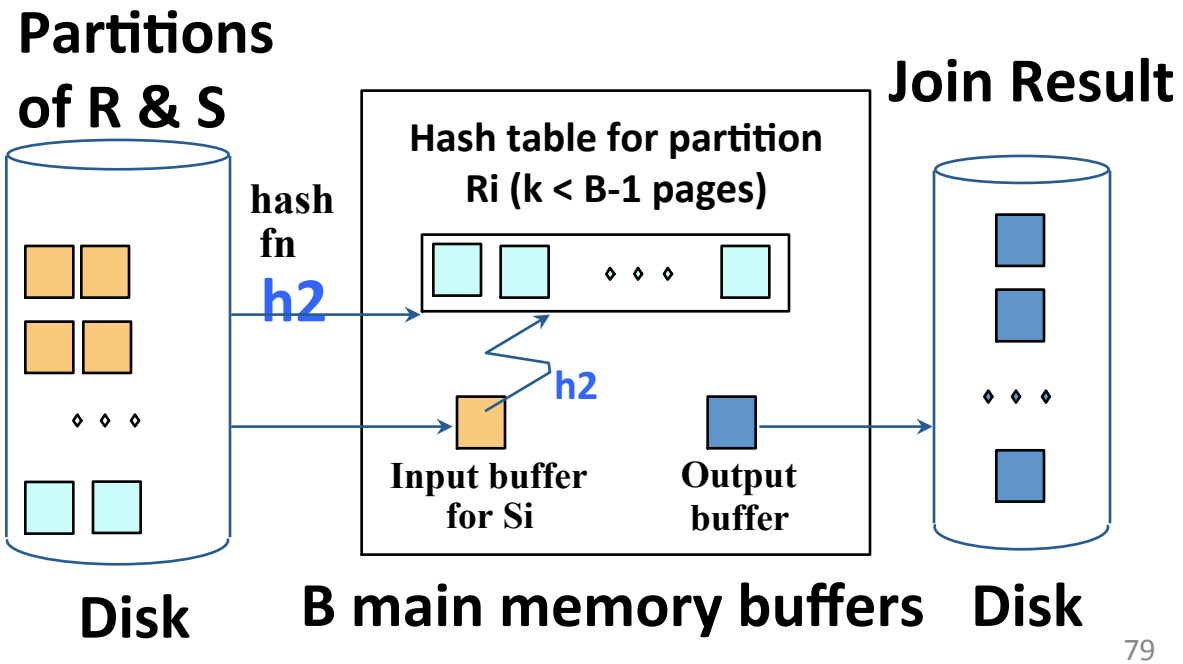
Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - With B > $\sqrt{L}$, where *L* is the size of the larger relation, using the sorting refinement that produces runs of length 2B in Pass 0, #runs of each relation is < B/2.
  - Allocate 1 page per run of each relation, and `merge' while checking the join condition.
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os.
- In practice, cost of sort-merge join, like the cost of external sorting, is *linear*.

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.



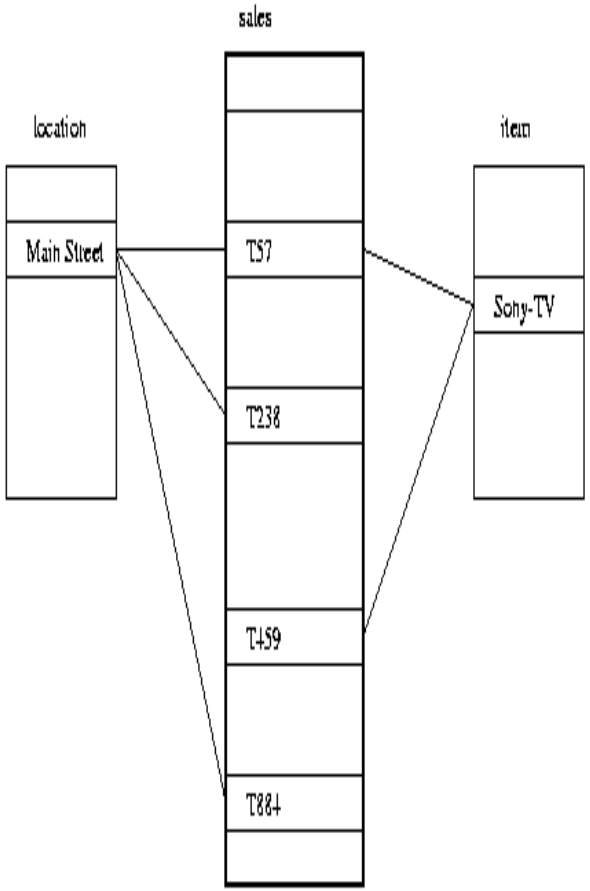


Observations on Hash-Join

- #partitions k < B-1 (why?), and B-2 > size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:
  - k= B-1, and M/(B-1) < B-2, i.e., B must be > $\sqrt{M}$
- If we build an in-memory hash table to speed up the matching of tuples, a little more memory is needed.
- If the hash function does not partition uniformly, one or more R partitions may not fit in memory. Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Join Indices

- Traditional indices map the values to a list of record ids
  - It materializes relational join in JI file and speeds up relational join — a rather costly operation
- In data warehouses, join index relates the values of the dimensions of a start schema to rows in the fact table.
  - E.g. fact table: *Sales* and two dimensions *city* and *product*
    - A join index on city maintains for each distinct city a list of R-IDs of the tuples recording the Sales in the city
  - Join indices can span multiple dimensions

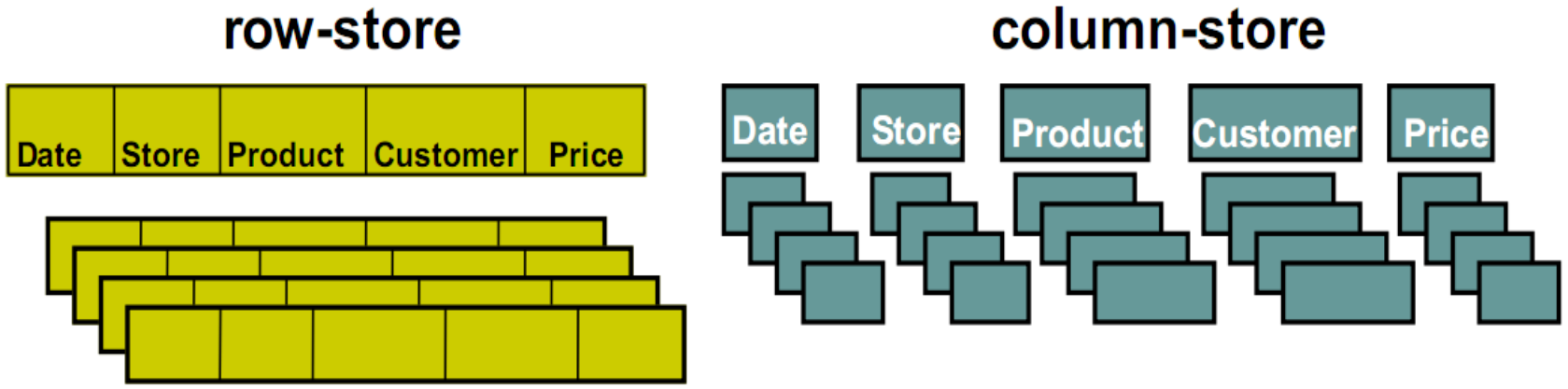


General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index NL, build index on <*sid, sname*> (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.
- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable.
  - Block NL quite likely to be the best join method here.

An invention in 2000s: Column Stores for OLAP

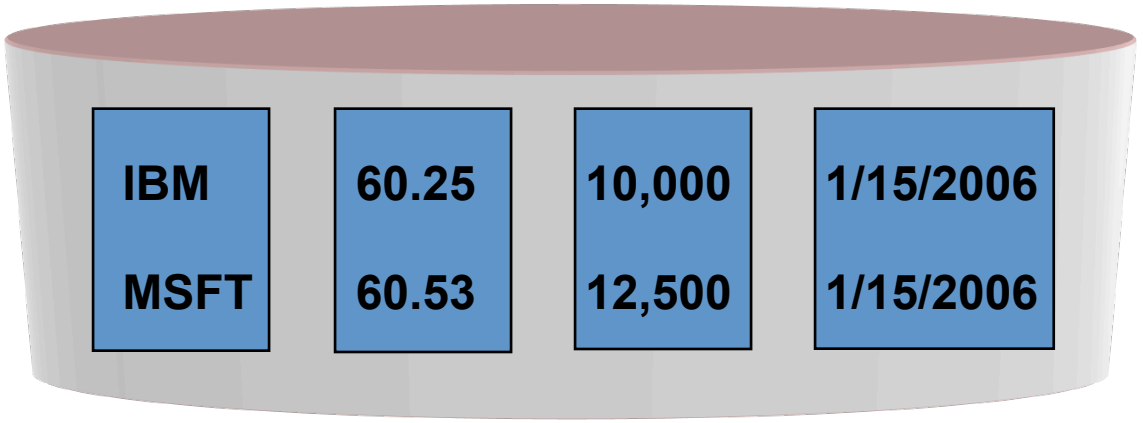
Row Store and Column Store



- 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

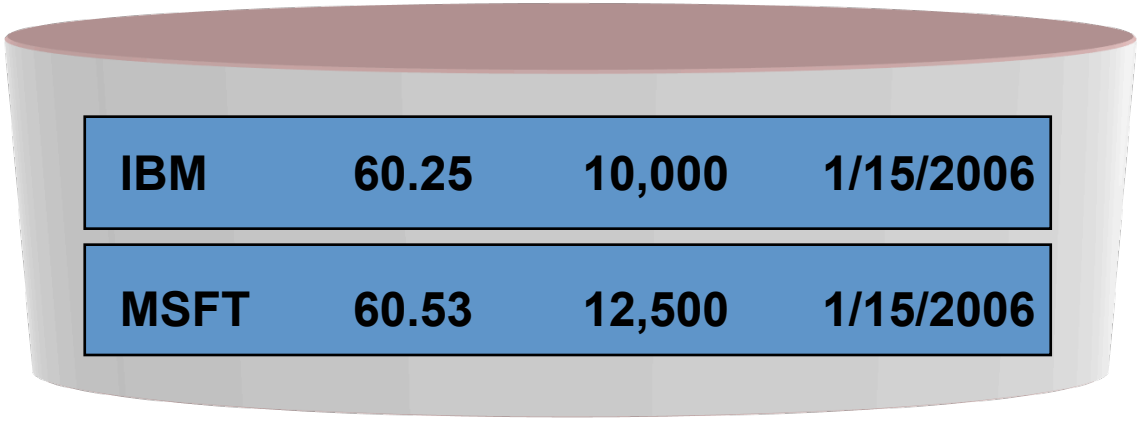
Row Store vs. Column Store

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

- So 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

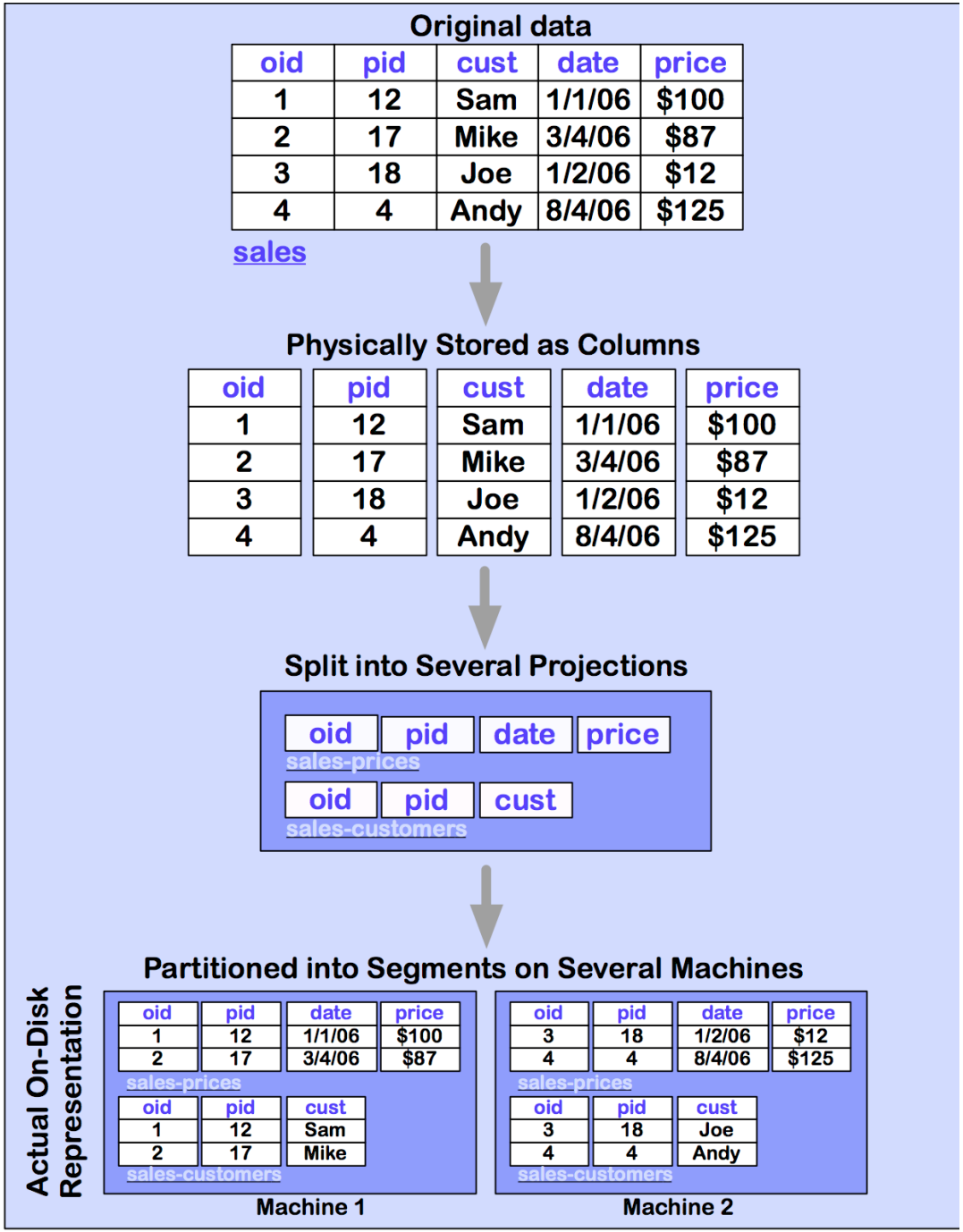
- Example

Normal relational model

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

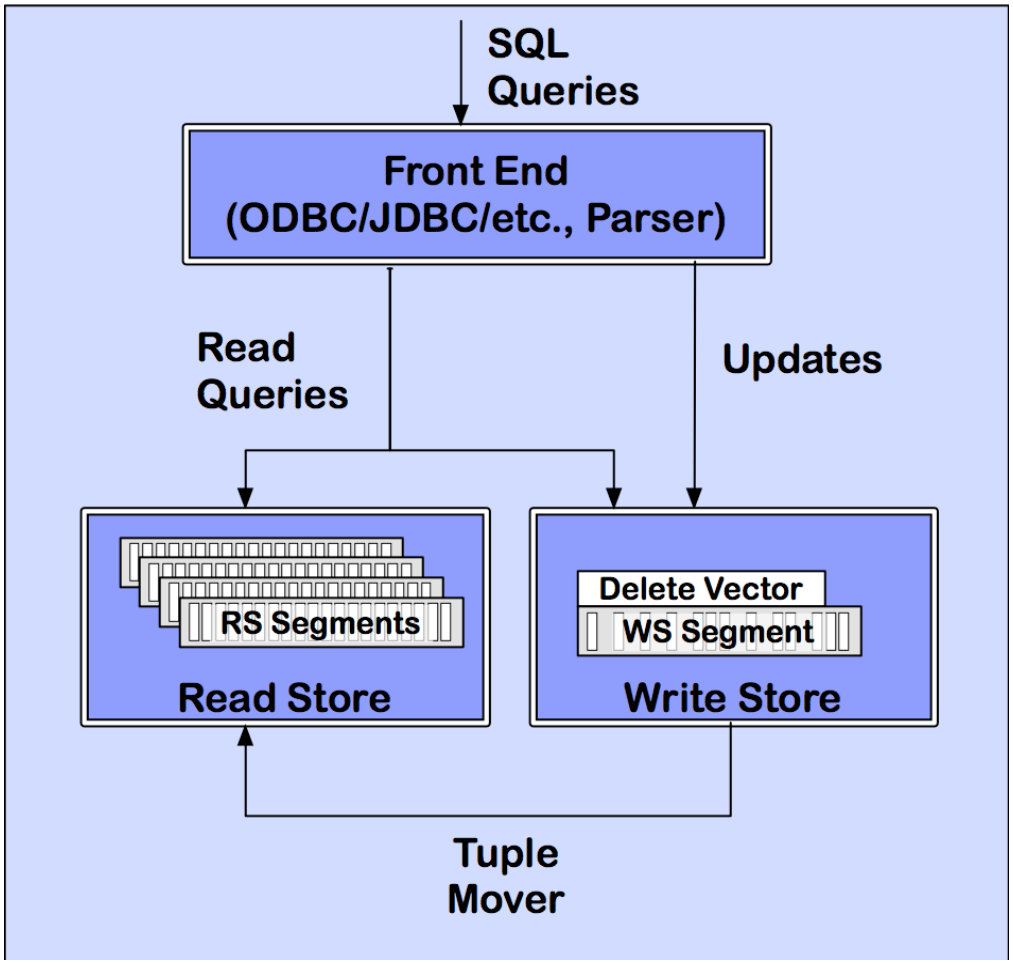
Possible C-store model

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

C-Store/Vertica Architecture

(from vertica Technical Overview White Paper)



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

Write Store

- Same structure, but explicitly use (segment, key) to identify records
  - Easier to maintain the mapping
  - Only concerns the inserted records
- Tuple mover
  - Copies batch of records to RS
- Delete record
  - Mark it on RS
  - Purged by tuple mover

How to Solve Read/Write Conflict

- Situation: one transaction updates the record X, while another transaction reads X.
- Use snapshot isolation

Query Execution – Operators

- **Select:** Same as relational algebra, but produces a bit string
- **Project:** Same as relational algebra
- **Join:** Joins projections according to predicates
- **Aggregation:** SQL like aggregates
- **Sort:** Sort all columns of a projection

Query Execution – Operators

- **Decompress:** Converts compressed column to uncompressed representation
- **Mask** (Bitstring B, Projection Cs) => emit only those values whose corresponding bits are 1
- **Concat:** Combines one or more projections sorted in the same order into a single projection
- **Permute:** Permutes a projection according to the ordering defined by a join index
- **Bitstring operators:** Band – Bitwise AND, Bor – Bitwise OR, Bnot – complement

Benefits in Query Processing

- Selection – has more indices to use
- Projection – some "projections" already defined
- Join – some projections are materialized joins
- Aggregations – works on required columns only

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