# CS422
# Database systems

Data warehouses and Decision Support Systems

Data-Intensive Applications and Systems (DIAS) Laboratory
École Polytechnique Fédérale de Lausanne

*“He uses statistics as a drunken man uses lampposts - for support rather than for illumination.”*
*– Andrew Lang*

# Overview

- Introduction
- Data warehouses
- On-line analytical processing (OLAP)
- Views
- Top-N queries

# Introduction

- On-Line Transaction Processing: OLTP
  - Updates and queries involving few tuples
    - Register a purchase, read/update the clients' balance, …
    - Dynamic data, exact results required
    - DBA/developer knows exactly what to ask for
- Decision support systems (DSS)
  - **Long-running** queries over (almost) **all** data
    - Get interesting insights from the data:
      - E.g., smoking causes cancer, more toys sold near xmas
    - Fairly static data
    - Exploratory process – ad-hoc queries

# Introduction (2)

Three complementary trends

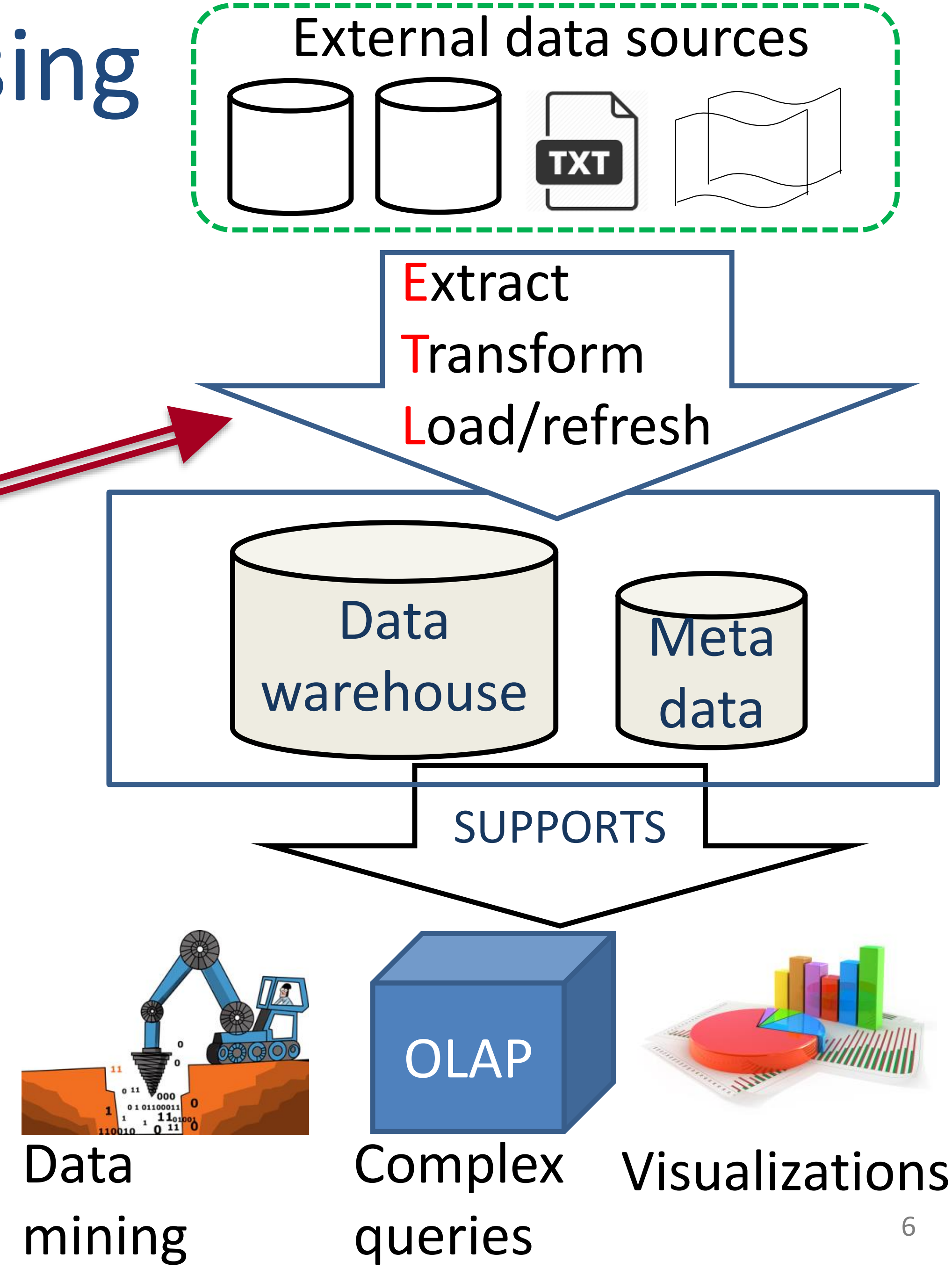
- Data warehousing
  - Consolidate data from many sources in one large repository
- On-Line Analytic Processing (OLAP)
  - Complex SQL queries and views
  - Aggregates and group bys
- Data mining (not in this course)
  - Clustering, classification, decision trees, ...

# Overview

- Introduction
- **Data warehouses**
- On-line analytical processing (OLAP)
- Views
- Top-N queries

# Data Warehousing

- Integrated data spanning long time periods
- Mostly static – ad-hoc updates uncommon
- Complex ad-hoc queries, interactive response times!



# Warehousing issues

- Semantic integration: Normalize data from multiple sources, eliminate mismatches
- Heterogeneous sources: Access data from different source formats and repositories
- Load, Refresh, Purge cycle: Load data, periodically refresh it, purge old data
- Metadata management: Keep track of sources and metadata (e.g., loading time, format, transformations)

# Warehousing issues (2)

- Querying
  - Huge data
    - Scan-intensive queries, touch (almost) all data
  - Exploratory, ad-hoc queries
  - Difficult to describe
  - Difficult to optimize

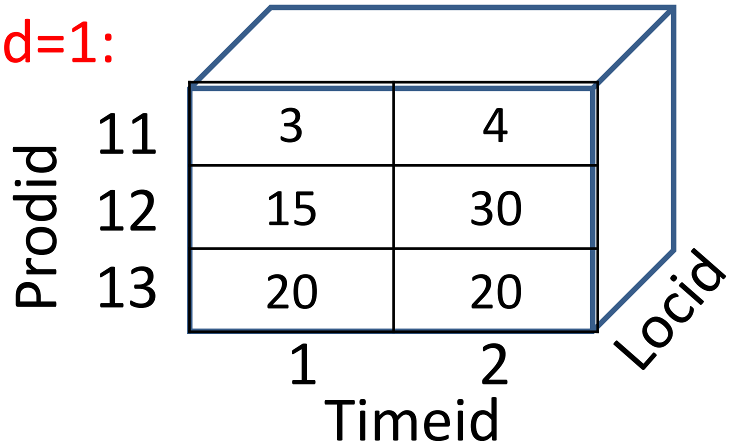
# Overview

- Introduction
- Data warehouses
- **On-line analytical processing (OLAP)**
- Views
- Top-N queries

# Multidimensional data model

- Numeric measures which depend on a set of dimensions
  - Measure Sales, dimensions Product, Location, Time

Slice for locid=1:




| Prod id | Time id | Locid | Sales |
|---|---|---|---|
| 11 | 1 | 1 | 3 |
| 11 | 2 | 1 | 4 |
| 12 | 1 | 1 | 15 |
| 12 | 2 | 1 | 30 |
| 13 | 1 | 1 | 20 |
| 13 | 2 | 1 | 20 |
| 11 | 1 | 2 | 33 |
| 11 | 2 | 2 | 31 |
| … | | | |

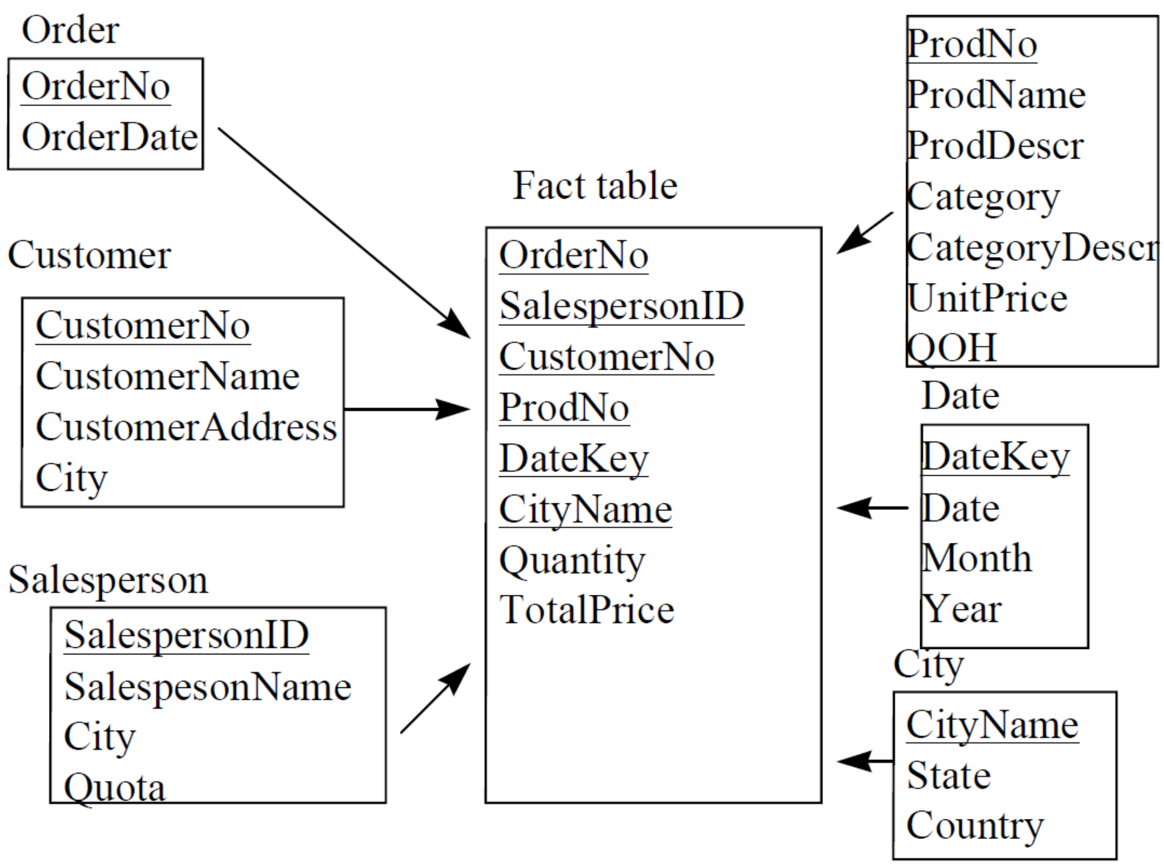
**Store as array (MOLAP) or relations (ROLAP)**

# ROLAP: Relational OLAP

Store multi-dimensional data as relations in a **star schema:**

- Fact table: Huge, stores the measurements/facts
  - **Normalized**
- Dimension tables
  - **Denormalized**

In this example, Quantity and TotalPrice are the only ***measures!***



# Star schema example

- Multi-dimensional data stored as relations
- Fact table
- Dimension tables

| Prod id | Time id | Locid | Sales |
|---|---|---|---|
| 11 | 1 | 1 | 25 |
| 11 | 2 | 1 | 8 |
| 12 | 1 | 1 | 15 |
| 12 | 2 | 1 | 30 |
| 13 | 1 | 1 | 20 |
| 13 | 2 | 1 | 20 |

| Prod id | name | Category | Price |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |

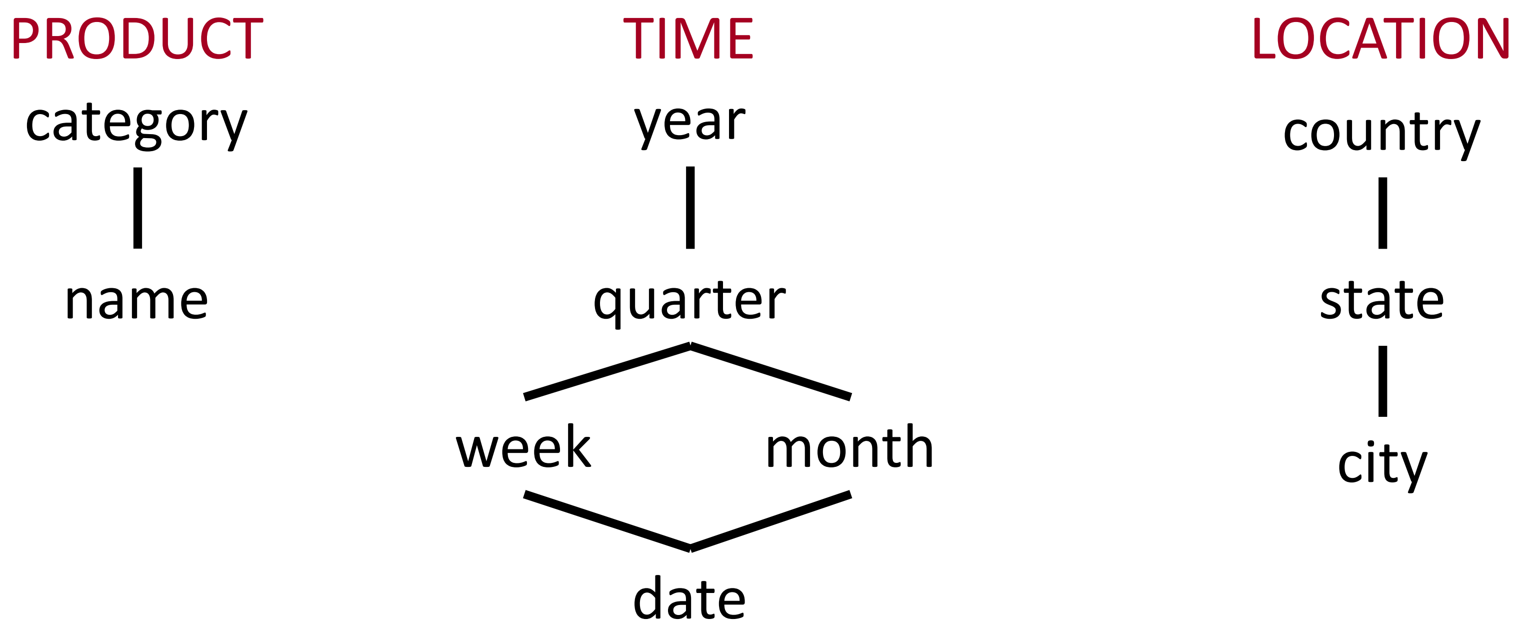
| Locid | City | State | Country |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |

| Time id | date | week | month | quarter | year | holiday flag |
|---|---|---|---|---|---|---|
| 1 | 01.01.17 | 1 | 1 | 1 | 2017 | T |
| 2 | 02.01.17 | 1 | 1 | 1 | 2017 | F |
| 3 | 03.01.17 | 1 | 1 | 1 | 2017 | F |
| | … | | | | | |

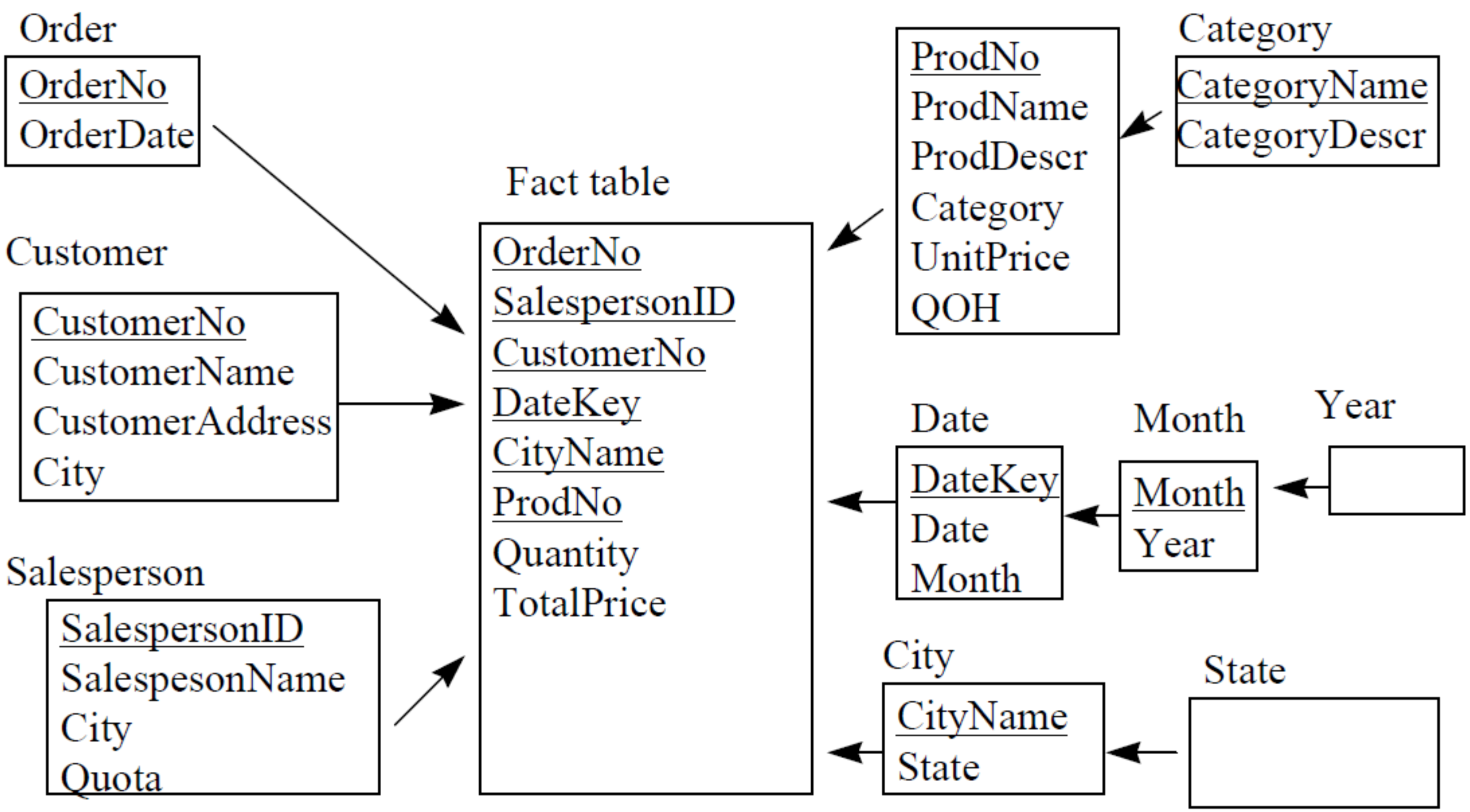
# Dimension hierarchies

- For each dimension, the set of values can be organized in a hierarchy

Product: <prodid, name, category, price>
Time: <timeid,date,week,month,quarter,year>
Location: <locid, city, state, country>

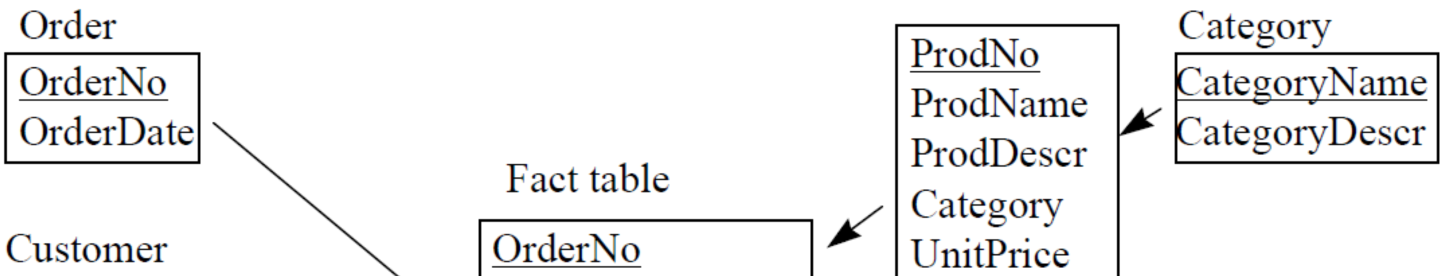


# ROLAP Alternative: Snowflake Schema

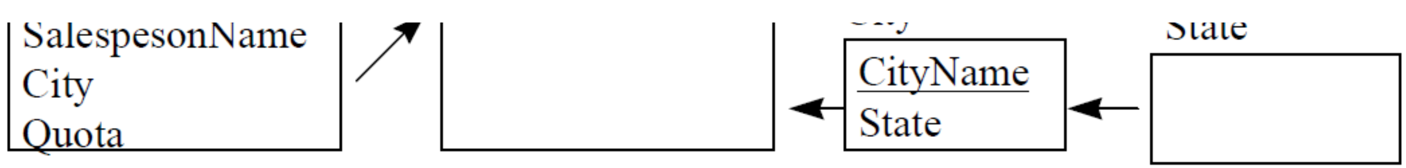



- Normalized dimension tables
  - Space efficient
  - Captures hierarchies
  - Fact-dimension joins more expensive

# ROLAP Alternative: Snowflake Schema




**Stonebraker: "If you are a data warehouse designer and come up with something other than a snowflake schema, you should probably rethink your design."**




- Normalized dimension tables
  - Space efficient
  - Captures hierarchies
  - Fact-dimension joins more expensive

# OLAP queries

- Combination of SQL and spreadsheets

SELECT
product.category, SUM(sales)
FROM sales, product, time
WHERE time.year=2016 AND
sales.timeid=time.timeid AND
sales.prodid=product.prodid
GROUP BY ~~product.category~~
locid

| product.category | locid | time.year | sales | SUM |
|---|---|---|---|---|
| 1 | 1 | 2016 | 721 | |
| 1 | 2 | 2016 | 586 | |

| product.category | locid | time.year | sales | SUM |
|---|---|---|---|---|
| 1 | 1 | 2016 | 256 | |
| 2 | 1 | 2016 | 669 | 1080 |
| 3 | 1 | 2016 | 155 | |
| 1 | 2 | 2016 | 337 | |
| 2 | 2 | 2016 | 208 | 647 |
| 3 | 2 | 2016 | 102 | |
| 1 | 3 | 2016 | 409 | |
| 2 | 3 | 2016 | 881 | 1358 |
| 3 | 3 | 2016 | 68 | |
| 1 | 4 | 2016 | 647 | |
| 2 | 4 | 2016 | 125 | 895 |
| 3 | 4 | 2016 | 123 | |
| 1 | 5 | 2016 | 775 | |
| 2 | 5 | 2016 | 686 | 2220 |
| 3 | 5 | 2016 | 759 | |
| | | | | **6200** |

# OLAP queries (2)

- Combination of SQL and spreadsheets
- Key operation: Aggregations & group by
  - Find total sales
  - Find total sales for each city/state/location/category/…
  - Find top-5 products ranked by average sales
  - …

# Examples of OLAP

- Comparisons (this period v.s. last period)
  - Show me the sales per region for this year and compare it to that of the previous year to identify discrepancies
- Multidimensional ratios (percent to total)
  - Show me the contribution to weekly profit made by all items sold in the northeast stores between may 1 and may 7
- Ranking and statistical profiles (top N/bottom N)
  - Show me sales, profit and average call volume per day for my 10 most profitable salespeople
- Custom consolidation (market segments, ad hoc groups)
  - Show me an abbreviated income statement by quarter for the last four quarters for my northeast region operations

# OLAP queries (3)

- Roll-up: Aggregation at different levels
  - E.g., given total sales by city, roll-up to get sales by state

country

state

city

<locid, city, state, country>

SELECT city, state, SUM(sales)

FROM sales, locations

WHERE sales.locid=locations.locid

GROUP BY ROLLUP(state,city)

| STATE | CITY | SUM(sales) |
|---|---|---|
| NY | New York | 43324 |
| NY | Albany | 6343 |
| NY | Buffalo | 5535 |
| NY | | 55202 |
| CA | Berkeley | 44200 |
| CA | Davis | 553 |
| CA | | 44753 |
| | | 99955 |

# OLAP queries (4)

- **Roll-up**: Aggregation at different levels
  - E.g., given total sales by city, roll-up to get sales by state

country
|
state
|
city

- **Drill-down**: Inverse of roll-up
  - E.g., given total sales by state, drill-down to get total sales by city, by product, by quarter, …
- **Pivoting**: Aggregation on selected dimensions
  - Result: cross-tabulation
  - E.g., pivoting on Location and Time:

| Time \ Location | 1 | 2 |
|---|---|---|
| 1 | 30 | 44 |
| 2 | 52 | 113 |
| 3 | 55 | 442 |

# Comparison with SQL queries

- Cross-tabulation obtained by pivoting can be computed using a collection of SQL queries

```
Q1: SELECT SUM(S.sales)
FROM sales S,
times T, locations L WHERE
S.timeid=T.timeid AND
S.locid=L.locid
```

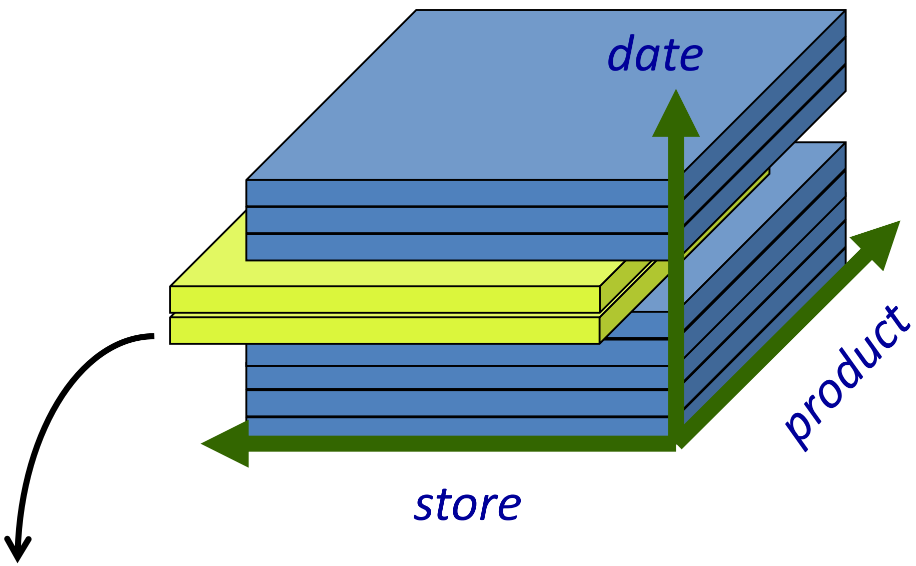
| Year \ State | IL | CA | Total |
|---|---|---|---|
| 07 | 30 | 44 | 74 |
| 08 | 52 | 113 | 165 |
| 09 | 55 | 442 | 497 |
| Total | 137 | 599 | **736** |

```
Q2: SELECT SUM(S.sales) FROM sales S,
times T WHERE S.timeid=T.timeid
GROUP BY T.year
```

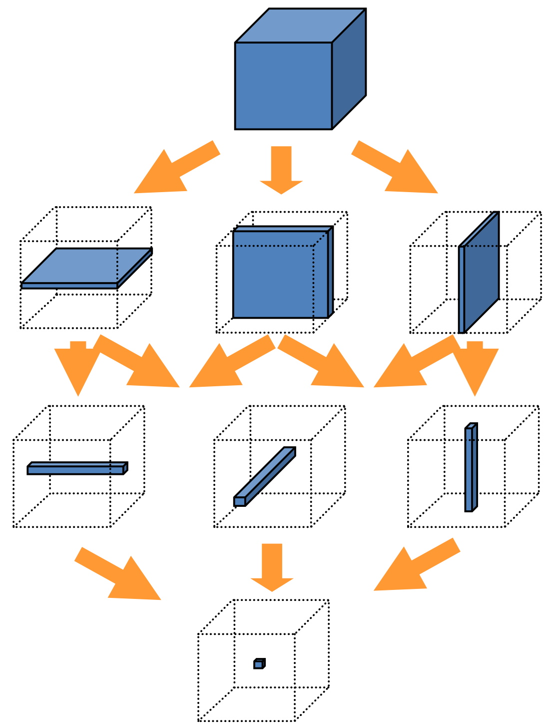
```
Q3: SELECT SUM(S.sales) FROM sales S, locations L
WHERE S.locid=L.locid GROUP BY L.state
```

# OLAP queries (5)

- **Slicing and dicing**: Equality and range selections on one or more dimensions




05.01.17 < date < 16.01.17



# The CUBE operator

- k dimensions → $2^k$ possible SQL GROUP BY queries
  - 10 dimensions → 1024 queries!
- Cube operator: compute **all** combinations

Oracle syntax: modify `GROUP BY` clause:

```
GROUP BY CUBE(prodid,locid,timeid)
```

Alternative syntax: `CUBE BY`

# The CUBE operator (2)

- Cube operator: compute all combinations
  - Equivalent to aggregating sales on all eight subsets of the set (prodid, locid, timeid)
  - Each group corresponds to an SQL query:
    `SELECT SUM(S.sales) FROM sales S`
    **`GROUP BY grouping-list`**

Why CUBE?

Why not individual queries?

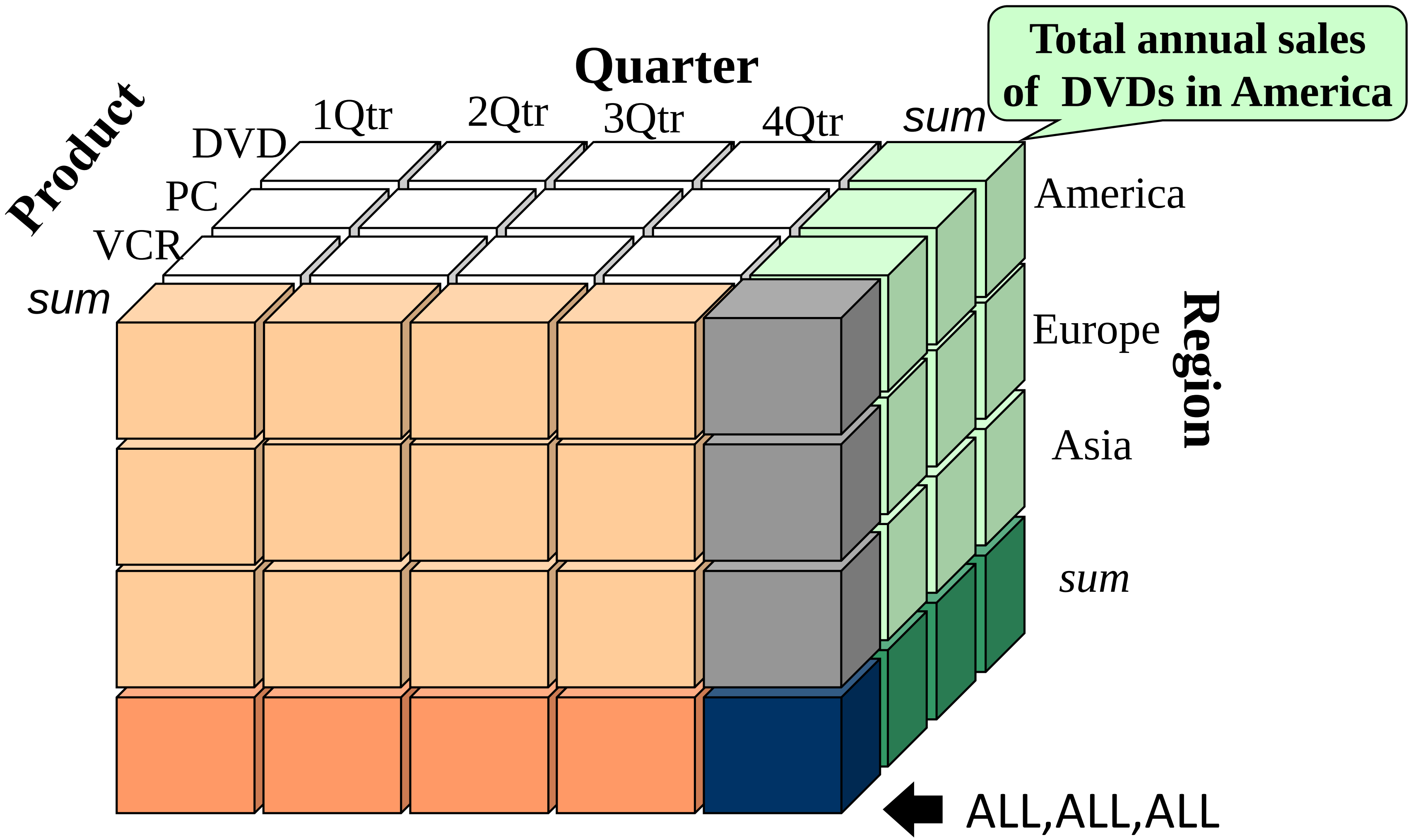
# Relational View of Data Cube

| Sales | | Product | | | | |
|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | ALL |
| Store | 1 | 454 | - | - | 925 | 1379 |
| | 2 | 468 | 800 | - | - | 1268 |
| | 3 | 296 | - | 240 | - | 536 |
| | 4 | 652 | - | 540 | 745 | 1937 |
| | ALL | 1870 | 800 | 780 | 1670 | 5120 |

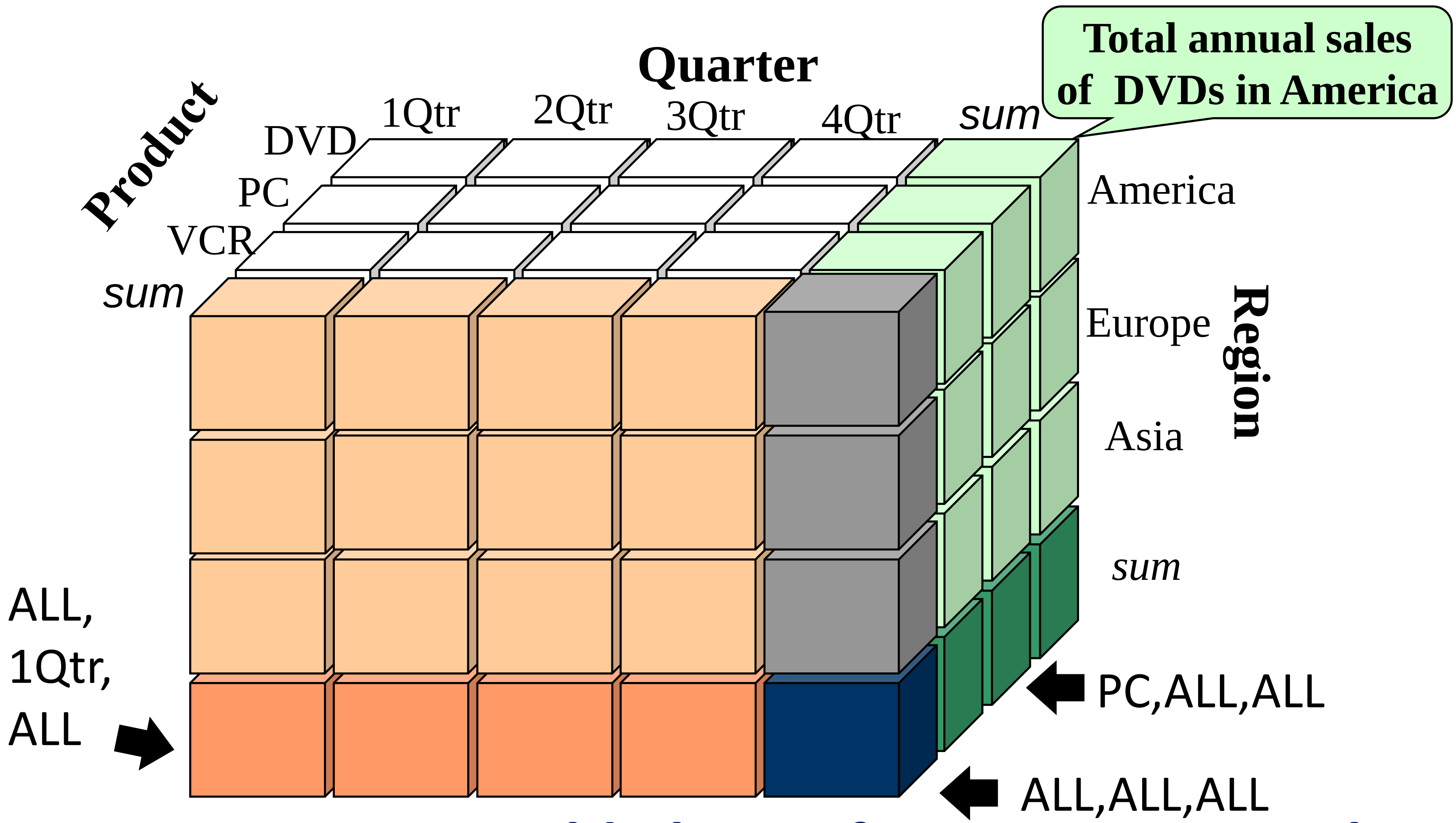
```
SELECT LOCATION.store, SALES.product_key, SUM (amount)
FROM SALES, LOCATION
WHERE SALES.location_key=LOCATION.location_key
CUBE BY SALES.product_key, LOCATION.store
```

| Store | Product_key | sum(amount) |
|---|---|---|
| 1 | 1 | 454 |
| 1 | 4 | 925 |
| 2 | 1 | 468 |
| 2 | 2 | 800 |
| 3 | 1 | 296 |
| 3 | 3 | 240 |
| 4 | 1 | 625 |
| 4 | 3 | 240 |
| 4 | 4 | 745 |
| 1 | ALL | 1379 |
| 2 | ALL | 1268 |
| 3 | ALL | 536 |
| 4 | ALL | 1937 |
| ALL | 1 | 1870 |
| ALL | 2 | 800 |
| ALL | 3 | 780 |
| ALL | 4 | 1670 |
| ALL | ALL | 5120 |

# Data Cube: Multidimensional View



# Cube: Computing "super-aggregates"




**Super-aggregate: Field absent from GROUP BY clause in a query used to build a cube cell => 'ALL'**

# Cube: Computing "super-aggregates"

- What's the type of super-aggregate to compute?
  - COUNT, MIN, MAX, SUM:
    Compute directly from other cells
  - AVG():
    Keep track of SUM, COUNT to compute from other cells
  - Median, Rank:
    Must examine entire dataset

# Optimizations: Bitmap Indexes

- Indexing crucial for performance
- Bitmap indexes

<u>Bit vectors:</u>
1 bit for each possible value.

Many queries can be answered using bit-vector ops!

| *gen* | | |
|---|---|---|
| 1 | 0 | |
| 1 | 0 | |
| 0 | 1 | |
| 1 | 0 | |

| custid | name | gen | rating |
|---|---|---|---|
| 114 | Joe | M | 3 |
| 113 | Sam | M | 5 |
| 115 | Sue | F | 1 |
| 118 | John | M | 4 |

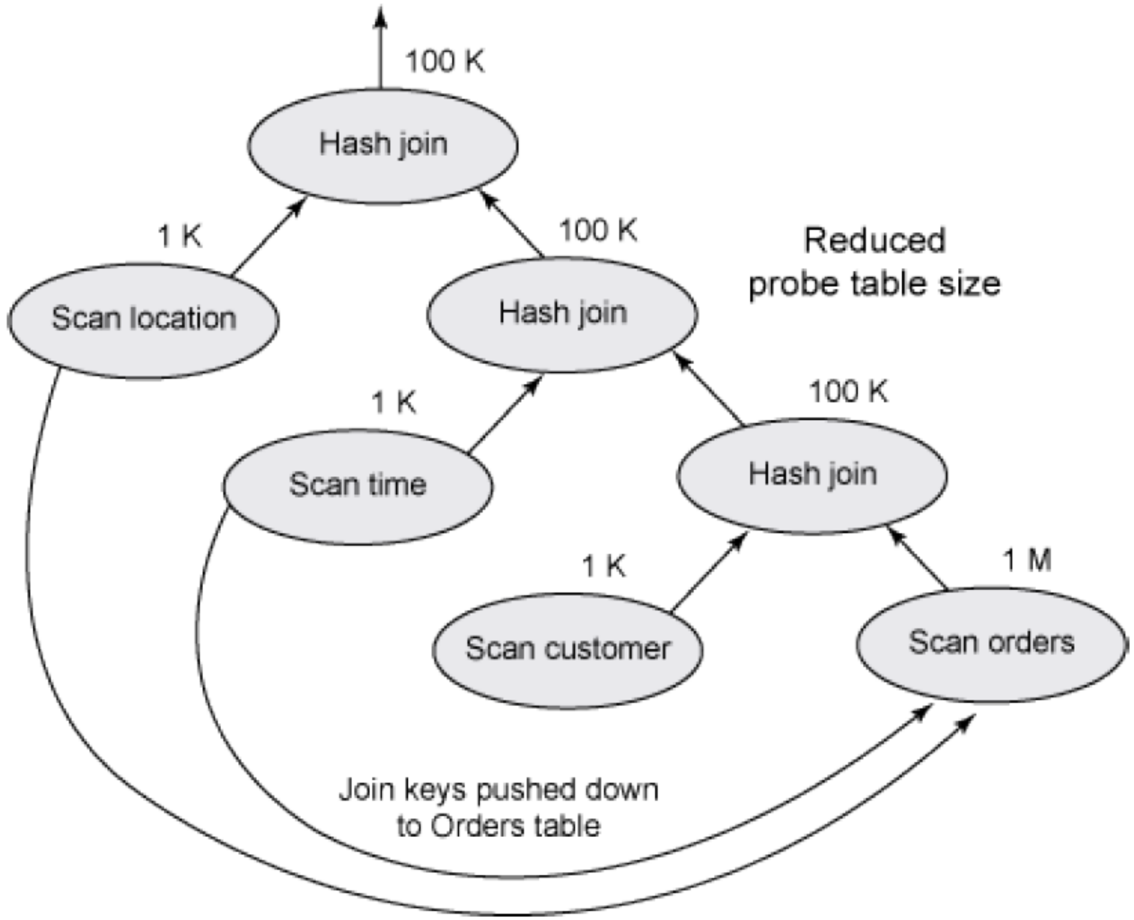
| *rating* |
|---|
| 00100 |
| 00001 |
| 10000 |
| 00010 |

Why is this faster?

# Optimizations: Join Indexes

- Consider
  Sales $\bowtie$ Products $\bowtie$ Times $\bowtie$ Locations
- Join index: compute join and store [s,*p*,*t*,*l*]
  - s, p, t, l: record ids in Sales, Products, Times, Locations
- Problem: Number of join indexes grows rapidly
- Alternative
  - For each dimension table with column c, compute [c,s]
    - Include selection attributes inside the index!
  - Join-merge the indexes at query time

# Optimizations: Star Join




– Apply filters on dimensions during hashtable creation
– Get qualifying keys from each dimension, and use them to prune Fact table => Reduce fact tuples used in joins


OTHERSIDE
RED HOT CHILI PEPPERS

# Overview

- Introduction
- Data warehouses
- On-line analytical processing (OLAP)
- **Views**
- Top-N queries

# Views

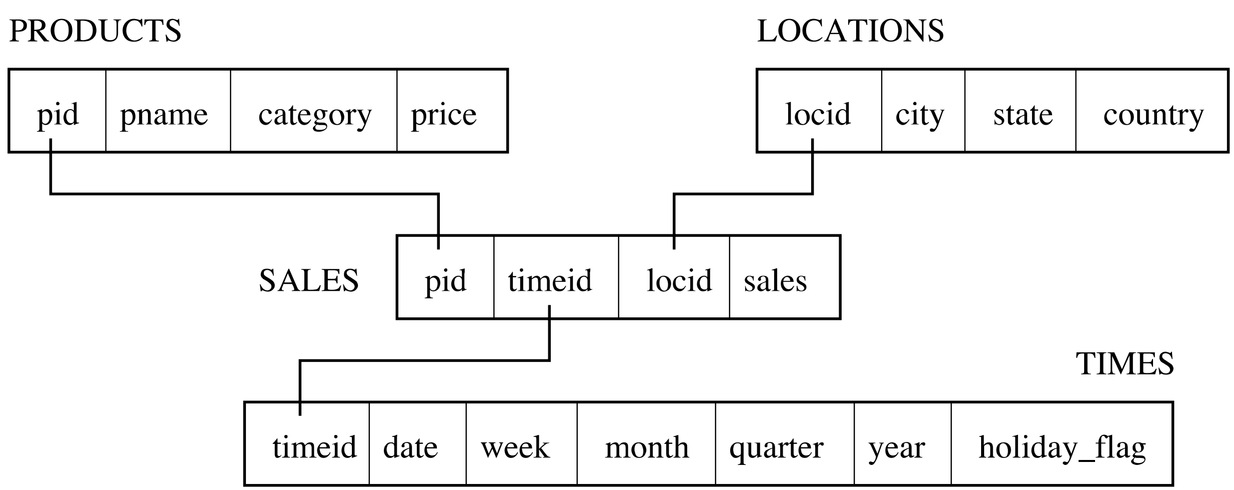
- Most OLAP queries: aggregate queries
  - Cube – large collection of aggregate queries
  - Expensive – precomputation is essential

- Views: A key component of warehouses
  - Evaluate-on-demand
  - Pre-computed – materialized
    - Different refresh policies

# Evaluate-on-demand views

```
CREATE VIEW RegionalSales
(category,sales,state) AS SELECT P.category,
S.sales, L.state FROM Products P, Sales S,
Locations L WHERE S.pid=P.pid AND
S.locid=L.locid
```

```
SELECT category,state,SUM(sales) FROM
RegionalSales GROUP BY category,state
```



# Evaluate-on-demand views (2)

```
CREATE VIEW RegionalSales
(category,sales,state) AS SELECT P.category,
S.sales, L.state FROM Products P, Sales S,
Locations L WHERE S.pid=P.pid AND
S.locid=L.locid
```

```
SELECT category,state,SUM(sales) FROM
RegionalSales GROUP BY category,state
```

Modified query:

```
SELECT category,state,SUM(sales) FROM
(SELECT P.category, S.sales, L.state FROM
Products P, Sales S, Locations L WHERE
S.pid=P.pid AND S.locid=L.locid)
GROUP BY category,state
```

# View materialization

- Materialized view: A view whose tuples are stored in the database – a *virtual table*
  – Efficiency: Fast access, similar to a query cache → enable interactive queries
  – Indexing: Ability to add indexes on aggregate queries
  – Reusability: Reuse results across queries/users
- But introducing complexity
  – Data replication → additional space requirements
  – Maintenance/refreshing

# Issues in view materialization

- Which views to materialize - which indexes to build on top of them
- How/when to exploit views
  - Both are application-dependent: expected workload, cost with/without the view, real-time requirements
- How to maintain the views up-to-date

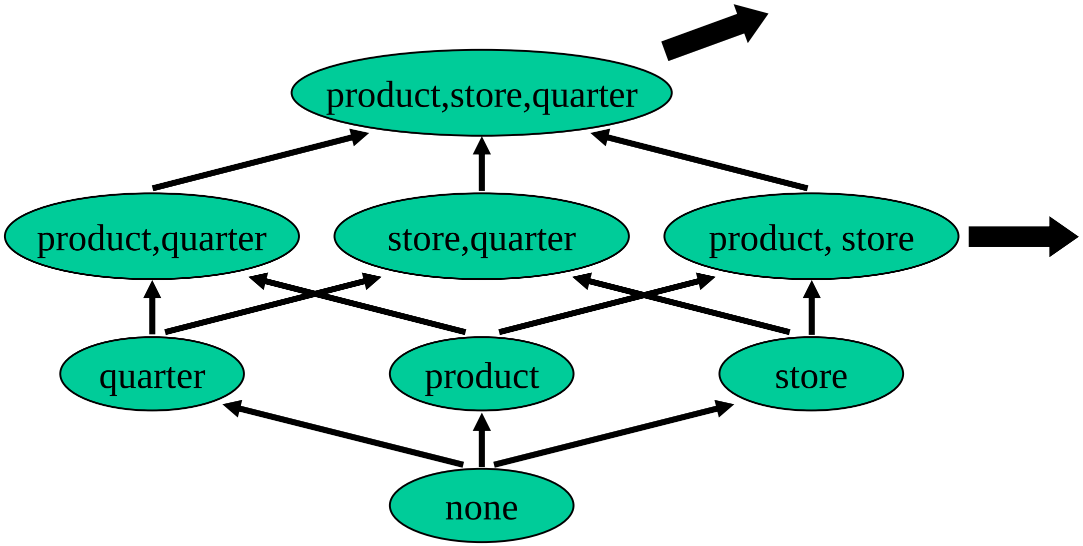
# View maintenance policies

- Immediate view maintenance
- Deferred view maintenance
  - Lazy: update before query
    - Slower queries, faster updates
    - Queries still faster than no-view in most cases
  - Periodic: update at regular intervals
  - Forced: update after a certain number of updates to the base tables
    - Fast queries, batch updating
    - Possibly stale results

# Computing Data Cube From Existing Views

Model dependencies among the aggregates:

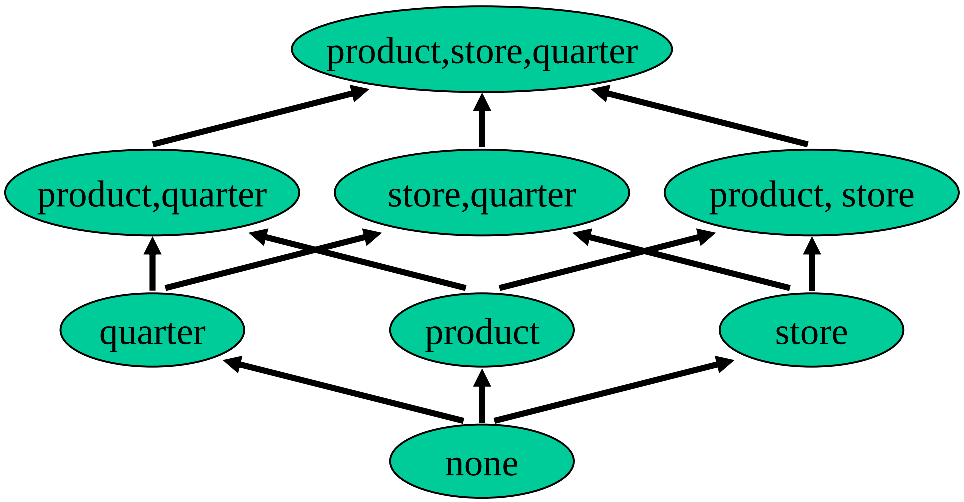
most detailed "view"




can be computed from *view* (product,store,quarter) by summing-up all quarterly sales

# Cube building optimizations

**Agrawal et al [VLDB96]**

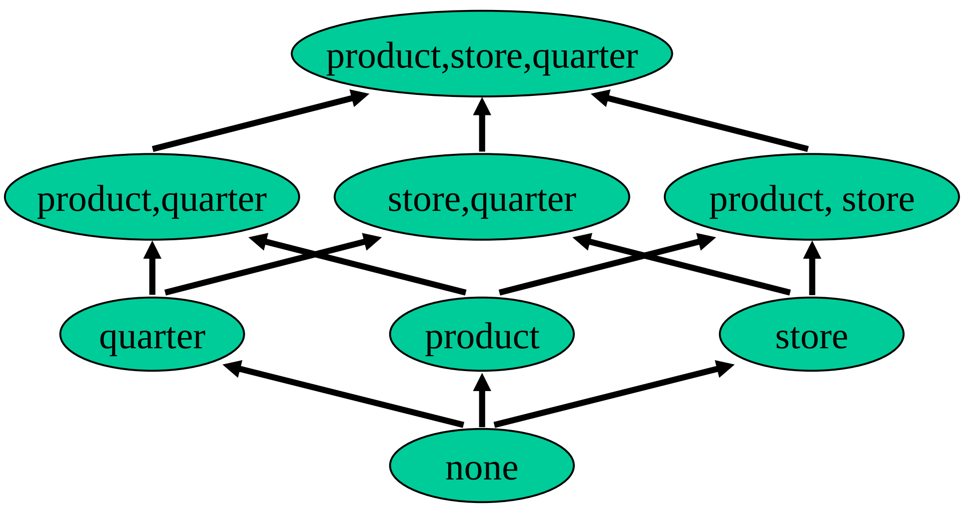



**Optimize hash/sort-based cube computation:**

- **Smallest-parent**
- **Cache-results**
- **Amortize-scans**
- **Share-sorts**
- **Share-partitions**

# Which views to build?

- Use some notion of *benefit* per view
- Limit: *disk space* or *maintenance-time*




**Hanirayan et al SIGMOD'96:**

**Pick views greedily until space is filled**

**Catch: quadratic in the number of views, which is exponential!!!**

# Overview

- Introduction
- Data warehouses
- On-line analytical processing (OLAP)
- Views
- **Top-N queries**

# Top-N queries

Examples

- Find the 20 most expensive products
- Find the 10 products sold most in the USA
- Find the 5 cities where products of category "Apparel" are sold most

Key optimization insight

- Focus on very few results

# Top-N queries (2)

Examples

- Find the 10 products with the highest sales in locid=1 and timeid=3

```
SELECT P.pid, P.pname, S.sales
FROM Sales S, Products P
WHERE S.pid=P.pid AND S.locid=1 AND S.timeid=3
ORDER BY S.sales DESC OPTIMIZE FOR 10 ROWS
```

↕

```
SELECT P.pid, P.pname, S.sales
FROM Sales S, Products P
WHERE S.pid=P.pid AND S.locid=1 AND S.timeid=3
AND S.sales>c ORDER BY S.sales DESC
```

# Summary

- OLAP: Providing decision support for your future managers

- Multidimensional data model, typically represented as special db schema
  - Star, snowflake

- Materialize views for fast retrieval of summaries
  - Keep in mind though: Column stores are changing the picture!!!