Data Warehousing & OLAP

Text:

Chapter 25

Other References:

Database Systems: The Complete Book, 2nd edition, by Garcia-Molina, Ullman, & Widom

The Data Warehouse Toolkit, 3rd edition, by Kimball & Ross

Databases: the continuing saga

When last we left databases…

- We had decided they were great things
- We knew how to conceptually model them in ER diagrams
- We knew how to logically model them in the relational model
- We knew how to normalize our database relations
- We could write queries in different languages

Next: what data format to people use for analysis?

Learning Goals

- Compare and contrast OLAP and OLTP processing (e.g., focus, clients, amount of data, abstraction levels, concurrency, and accuracy).
- Explain the ETL tasks (i.e., extract, transform, load) for data warehouses.
- Explain the purpose of the star schema design, including potential tradeoffs.
- Argue for the value of a data cube in terms of:
  - The goals of OLAP (e.g., summarization, abstractions), and
  - The operations that can be performed (drill-down, roll-up, slicing/dicing).
- Estimate the complexity of a data cube, in terms of the number of equivalent aggregation queries.
- Apply the HRU algorithm to find the best *k* views to materialize.

What We Have Focused on So Far

- **OLTP (Online Transaction Processing)**
  - Transaction-oriented applications, typically for data entry and retrieval transaction processing.
  - The system responds immediately to user requests.
  - High throughput and insert- or update-intensive database management. These applications are used concurrently by hundreds of users.
- The key goals of OLTP applications are availability, speed, concurrency and recoverability.

source: Wikipedia

Can We Do More?

- Increasingly, organizations are analyzing current and historical data to identify useful patterns and support long-term strategies.
  - a.k.a. “Decision Support”, “Business Intelligence”
- The emphasis is on complex, interactive, exploratory analysis of very large datasets created by integrating data from across all parts of an enterprise.

Let’s imagine that you’re trying to make some decisions at UBC like…

- Deciding what students to admit to what programs
- Understanding what students are at risk and need more support

These tasks require …

- Data that is large
- Data that comes from multiple sources
- Doesn't need to be up to the minute accurate in order to be useful
- The ability to answer very complex queries very quickly.

OLTP is not the best choice to handle this scenario

What about just putting all the data together?

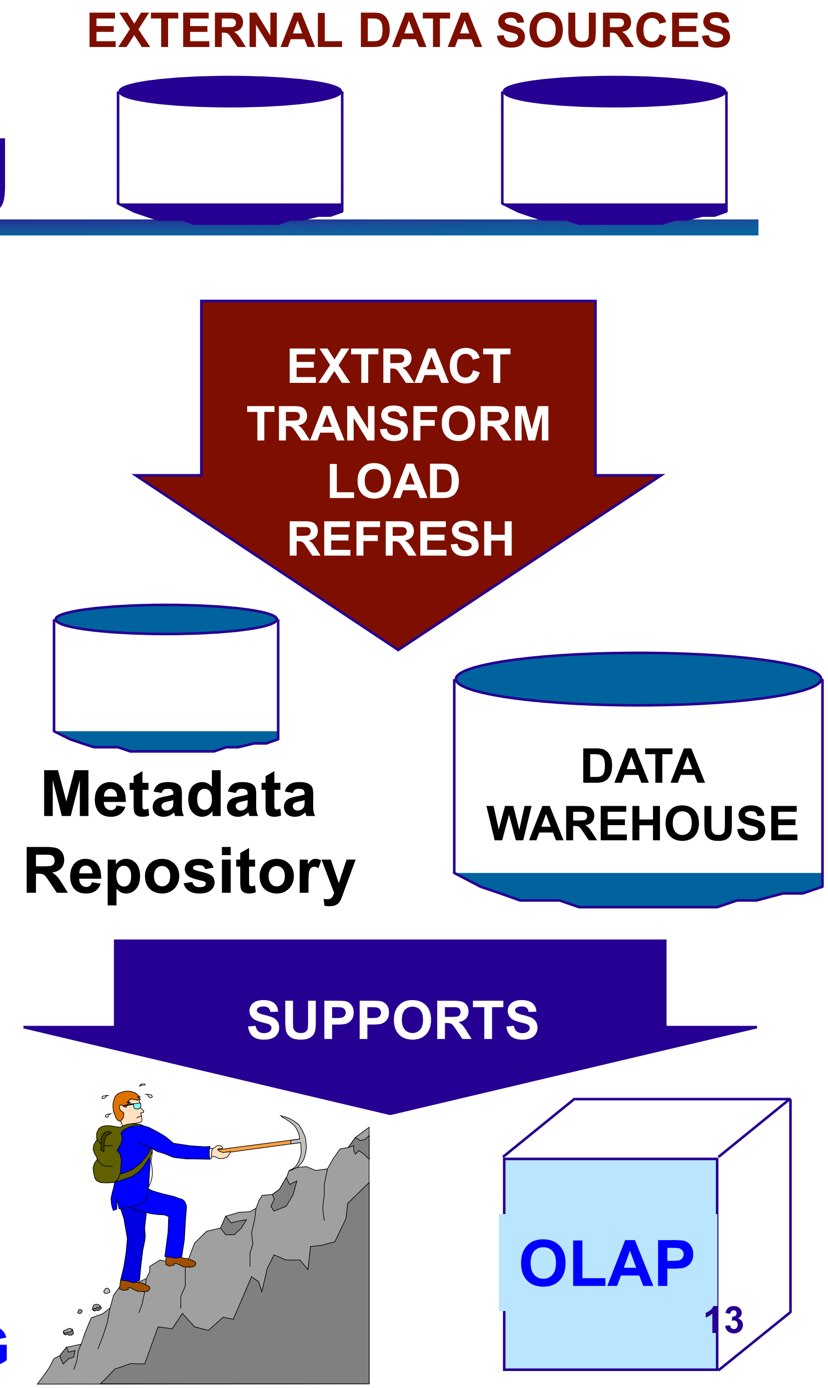
- One current direction (data lakes) says “hey, let’s just put all the data in the same spot, and people who ask queries can figure it out.”
- This is great for getting answers quickly if you don’t care about accuracy.
- But very hard to make super accurate decisions on.
- Among other things, the data might not even be in the same schema.
- Data Lakes are fast to build, but very slow to query.

What about Data Warehouses?

- Data warehouses import all the data into one application
- Data cleaning and integration techniques are applied to ensure consistency in naming conventions, schemas, etc.
- Everything is put into a common format
- Data warehouses are slow to build, but very fast to query.

Data Warehousing

The process of constructing and using data warehouses is called data warehousing.



DATA
MINING

Data Warehouses support

OLAP (Online Analytical Processing)

- Perform complex SQL queries and views, including trend analysis, drilling down for more details, and rolling up to provide more easily understood summaries.
- Queries are normally performed by domain experts rather than database experts.

Data Mining

- Exploratory search for interesting trends (patterns) and anomalies (e.g., outliers, deviations) using more sophisticated algorithms (as opposed to queries).

OLTP vs. OLAP

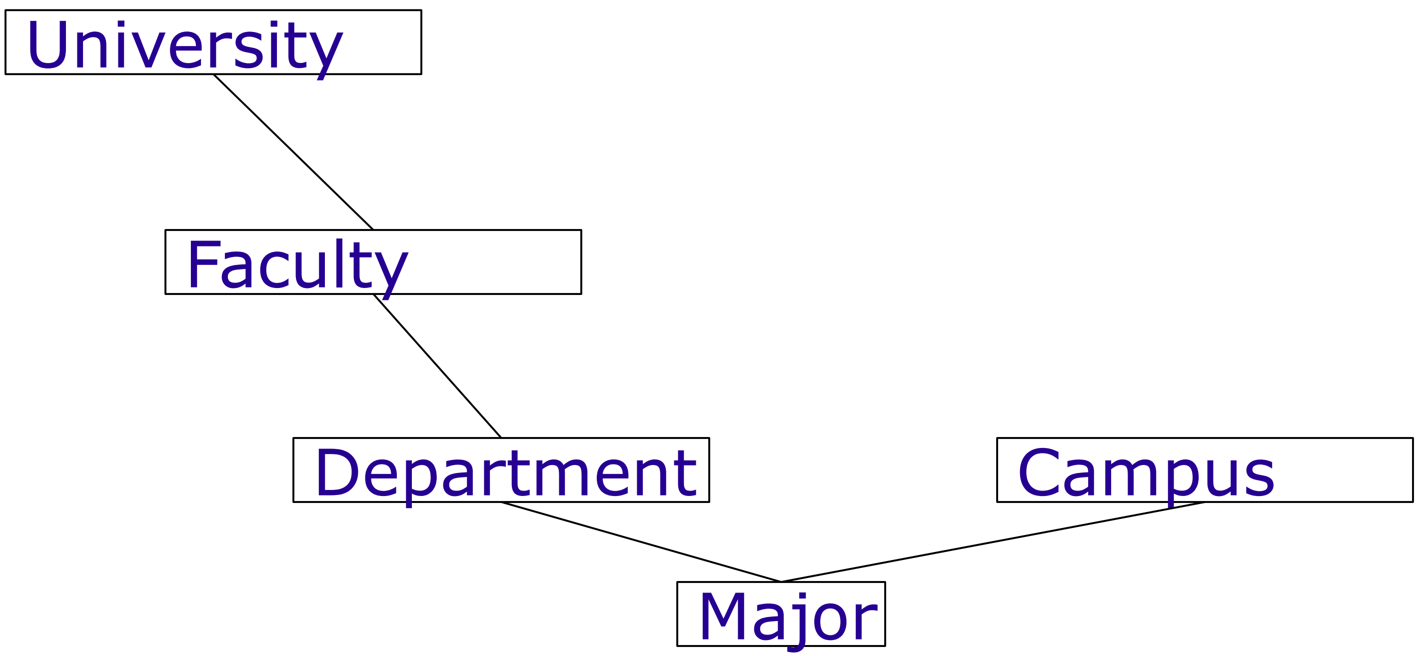
	OLTP	OLAP
Typical User	Basically Everyone (Many Concurrent Users)	Managers, Decision Support Staff (Few)
Type of Data	Current, Operational, Frequent Updates	Historical, Mostly read-only
Type of Query	Short, Often Predictable	Long, Complex
# query	Many concurrent queries	Few queries
Access	Many reads, writes and updates	Mostly reads
DB design	Application oriented	Subject oriented
Schema	E-R model, RDBMS	Star or snowflake schema
Normal Form	Often 3NF	Unnormalized
Typical Size	MB to GB	GB to TB
Protection	Concurrency Control, Crash Recovery	Not really needed
Function	Day to day operation	Decision support

Let's revisit the student table: Student(snum,~~sname~~,major,standing,age)

- There are interesting things that we can investigate if we try various aggregations of major, standing (year, enrolled, etc.), and age
- We're also going to ignore sname because it's not very interesting to do analysis on
- This is over simplistic: there are other things that you want to look at, too, like grades, but let's just use this as an example.

We can do interesting aggregations on major, standing, and age

- You can aggregate them in interesting ways, often in a hierarchy or two
- For example, if we look at majors…



So what we want is a design where…

- We can query about students as our central object
- We can query about majors, standings, and ages, as they relate to students
- OLAP queries are full of groupings and aggregations.
- The natural way to think about such queries is in terms of a ***multidimensional model***, which is an extension of the table model in regular relational databases.

Multidimensional Data Model

- The main relation, which relates dimensions to a measure via foreign keys, is called the ***fact table*** (e.g., students)
- Each dimension can have additional attributes and an associated ***dimension table*** (e.g., majors)
  - Attributes can be numeric, categorical, temporal, counts, sums
- Fact tables are usually *much* larger than dimensional tables.
- There can be multiple fact tables (e.g., students, courses, grades)

Multidimensional Data Model

- This model focuses on:
  - a set of numerical ***measures***: quantities that are important for business analysis, like numbers of majors, etc.
  - a set of ***dimensions***: entities on which the measures depend on, like majors.

Design Issues

- The schema that is very common in OLAP applications, is called a star schema:
  - one table for the fact, and
  - one table per dimension
- The fact table is in BCNF.
- The dimension tables are not normalized. They are small; updates/inserts/deletes are relatively less frequent. So, redundancy is less important than good query performance

Running Example

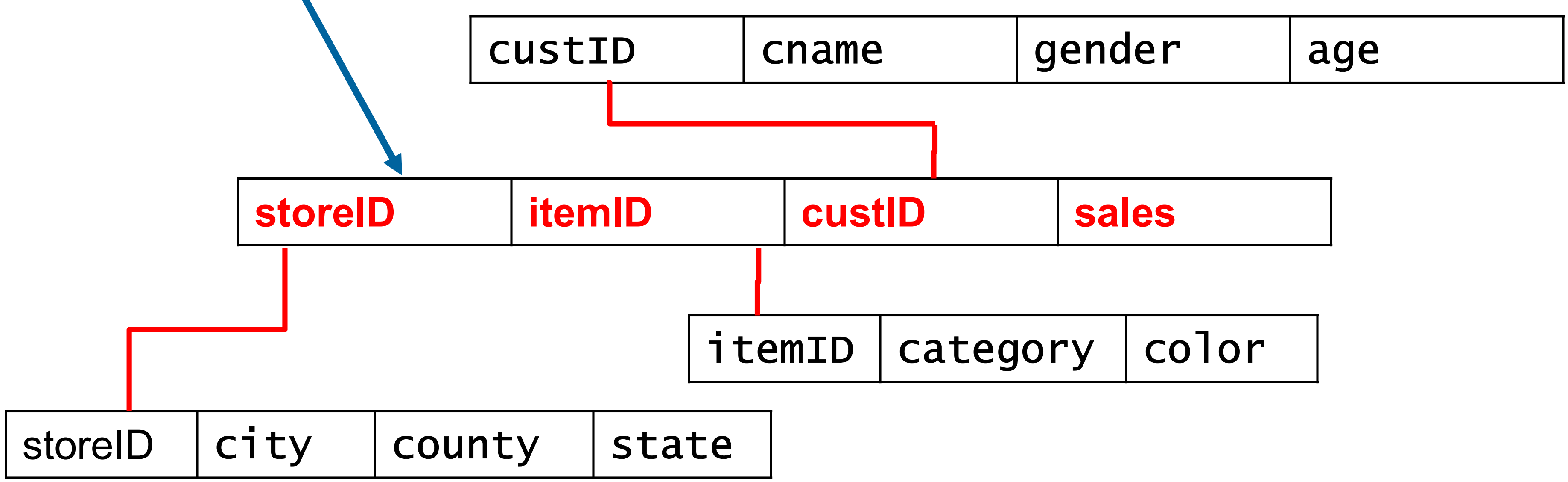
Star Schema – fact table references dimension tables

- Join → Filter → Group → Aggregate

Fact table→

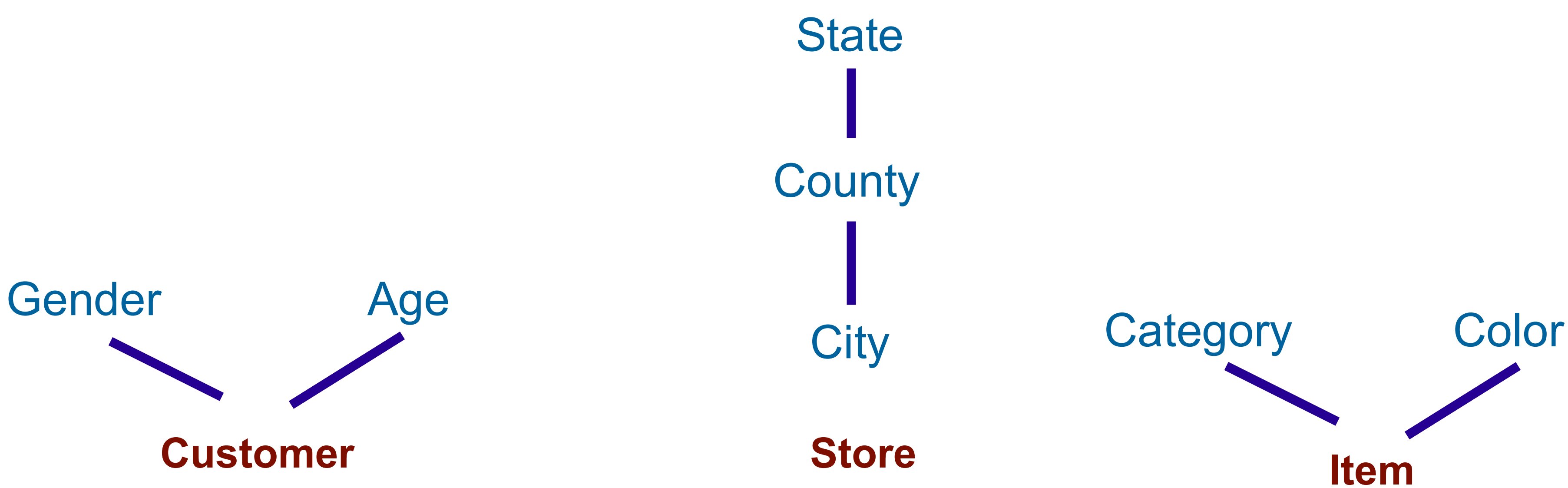
Dimensions

```
AllSales(storeID, itemID, custID, sales)
Store(storeID, city, county, state)
Item(itemID, category, color)
Customer(custID, cname, gender, age)
```



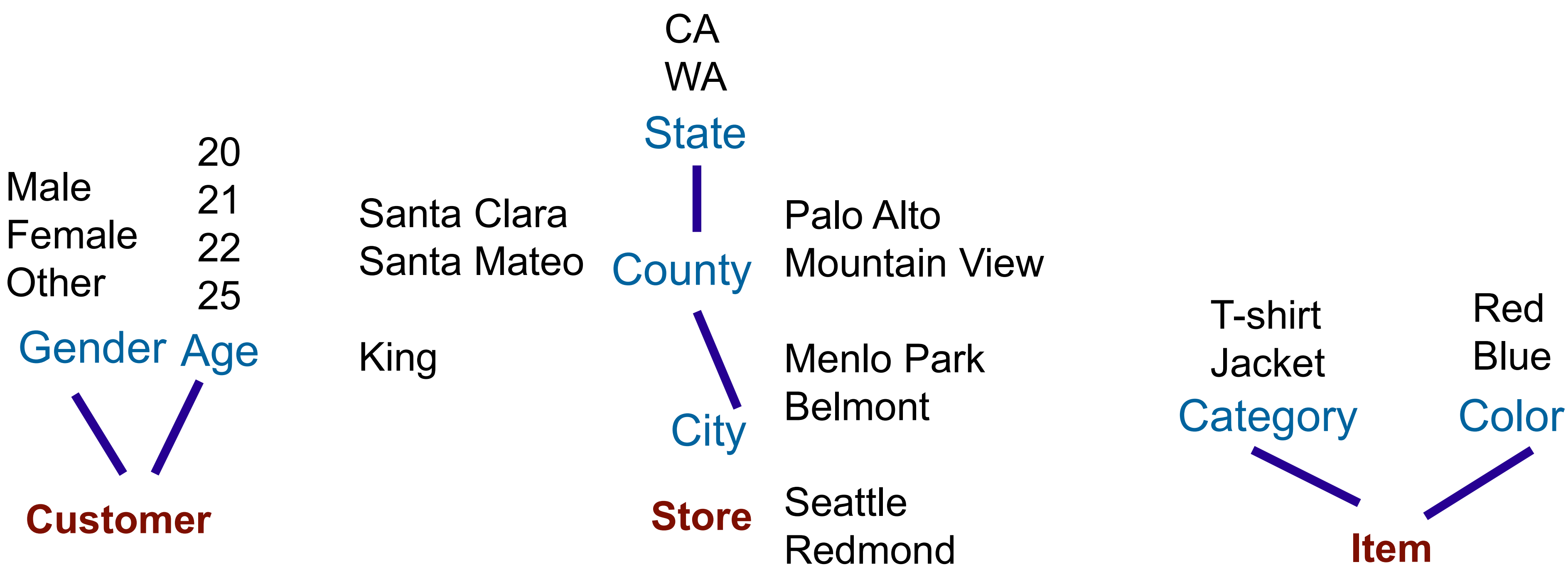
Dimension Hierarchies

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



Running Example (cont.)

```
AllSales(storeID, itemID, custID, sales)
Store(storeID, city, county, state)
Item(itemID, category, color)
Customer(custID, cname, gender, age)
```



Making this concrete

- Spend a couple of minutes thinking about a domain (maybe your project, or, when in doubt, students/grades usually work) that you might want to run OLAP queries on.
- Design a star schema for it
- Design some dimensions & hierarchies
- There are no rights or wrongs here; the goal is just to give you some practice/chance to realize questions about the topic.

Full Star Join

- An example of how to find the *full star join* (or *complete star join*) among 4 tables (i.e., fact table + all 3 of its dimensions) in a Star Schema:
  - Join on the foreign keys

```
SELECT *
FROM    AllSales F, Store S, Item I, Customer C
WHERE  F.storeID = S.storeID and
        F.itemID = I.itemID and
        F.custID = C.custID;
```

- If we join fewer than all dimensions, then we have a *star join*.
- In general, OLAP queries can be answered by computing some or all of the star join, then by filtering, and then by aggregating.

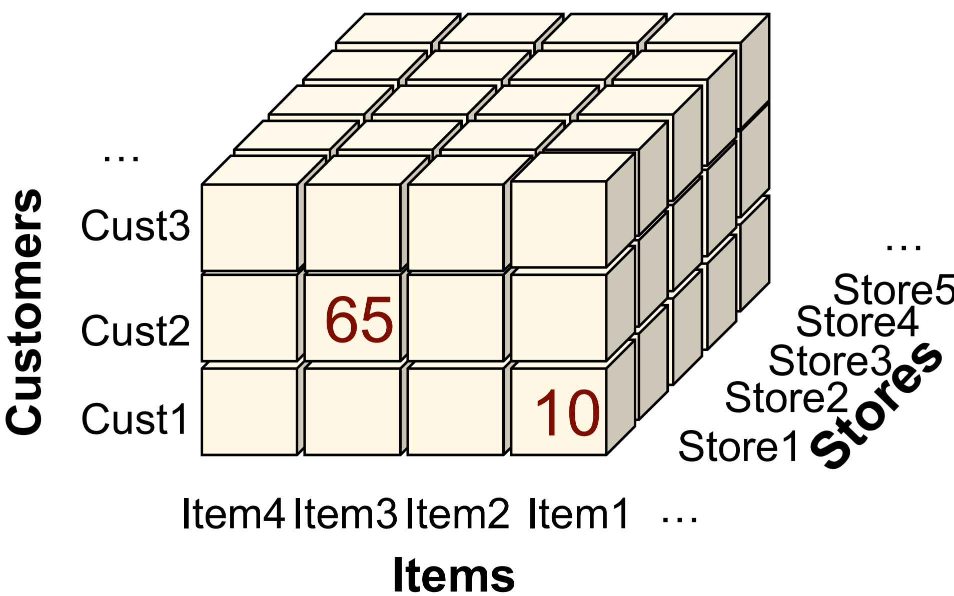
Find total sales by store, item, and customer.

Full Join:

```
SELECT *
FROM     AllSales F, Store S, Item I, Customer C
WHERE   F.storeID = S.storeID and
         F.itemID = I.itemID and
         F.custID = C.custID;
```

Desired outcome

```
SELECT storeID, itemID, custID, SUM(sales)
FROM     AllSales F
GROUP BY storeID, itemID, custID;
```



storeID	itemID	custID	Sum (sales)
store1	item1	cust1	10
store1	item3	cust2	65
…	…	…	…

Great! Now we have a schema!

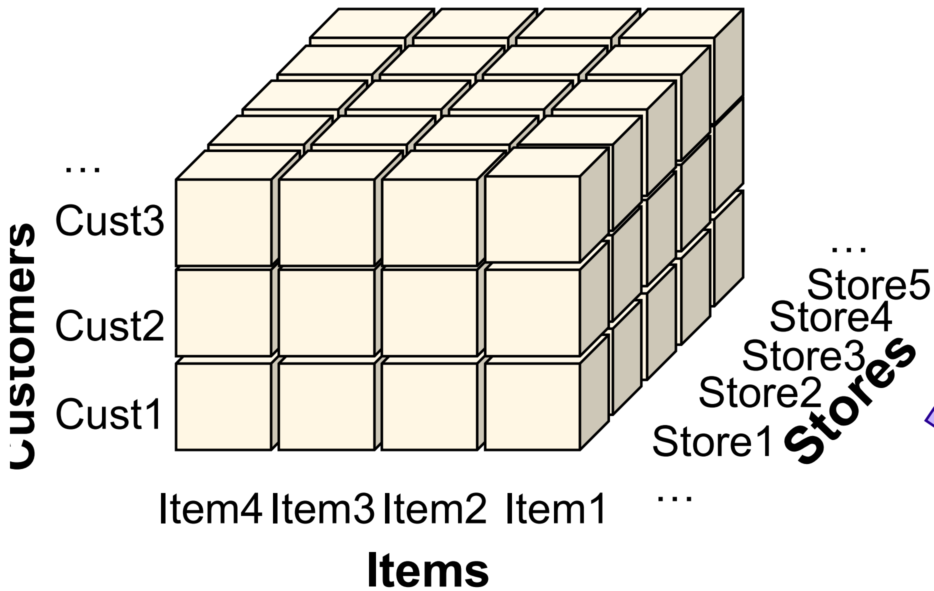
- What can we do with it?

OLAP Queries – Roll-up

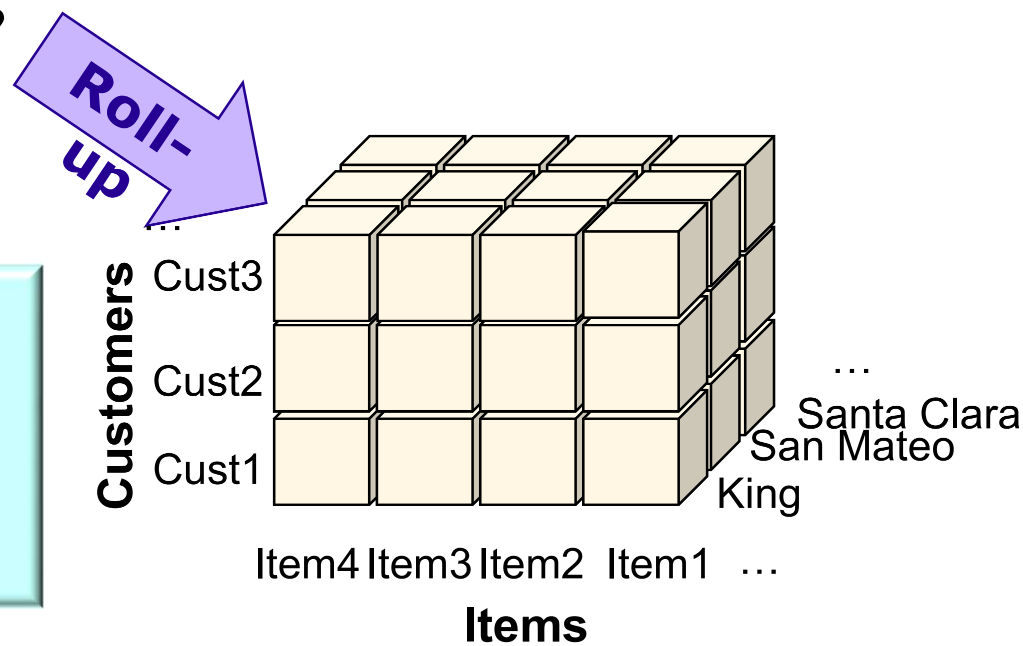
- Roll-up allows you to summarize data by:
  - Changing the level of granularity of a particular dimension
  - Dimension reduction

Roll-up Example 1 (Hierarchy)

Use Roll-up on total sales by store, item, and customer to find total sales by item and customer for each county.



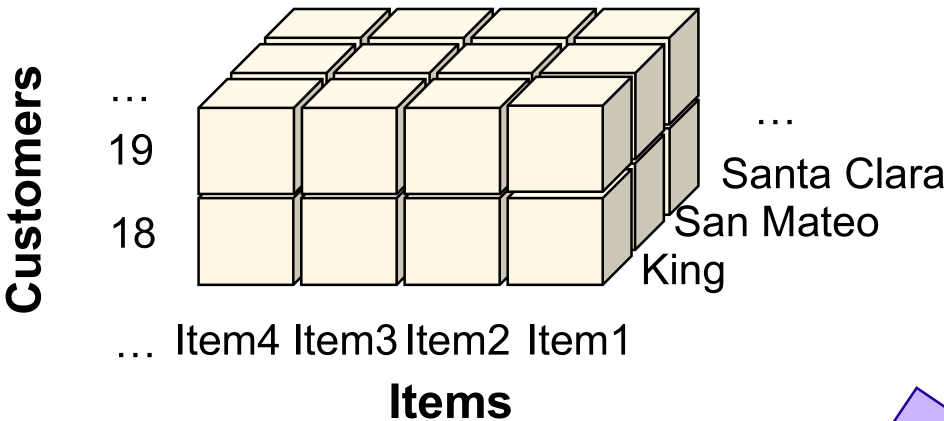

```
SELECT storeID, itemID, custID,
            SUM(sales)
FROM    AllSales F
GROUP BY storeID, itemID, custID;
```




```
SELECT  county, itemID, custID,
             SUM(sales)
FROM     AllSales F, Store S
WHERE   F.storeID = S.storeID
GROUP BY county, itemID, custID;
```

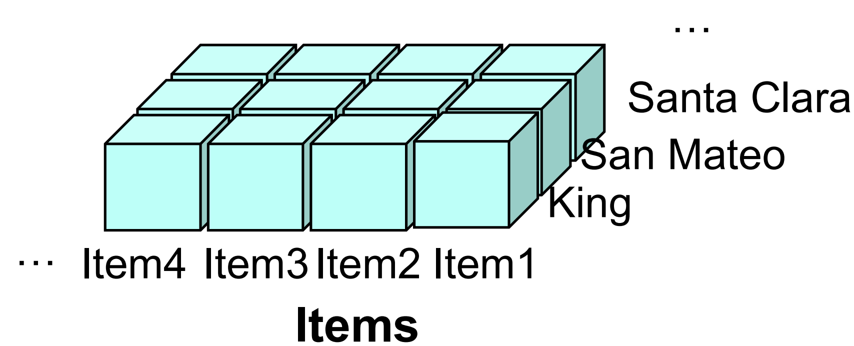
Roll-up Example 3 (Dimension)

Use Roll-up on total sales by item, age and county to find total sales by item for each county.




```
SELECT county, itemID, age,
          SUM(sales)
FROM    AllSales F, Store S,
          Customer C
WHERE F.storeID = S.storeID AND
          F.custID = C.custID
GROUP BY county, itemID, age;
```


```
SELECT county, itemID, SUM(sales)
FROM    AllSales F, Store S
WHERE F.storeID = S.storeID
GROUP BY county, itemID;
```

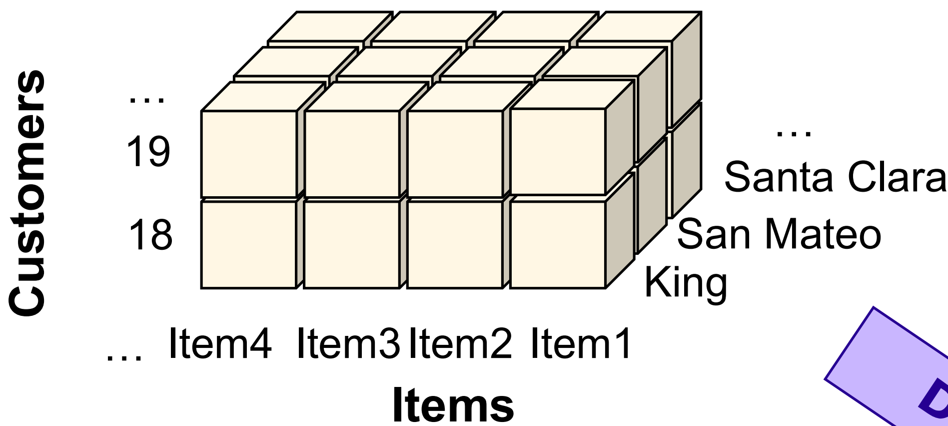


OLAP Queries – Drill-down

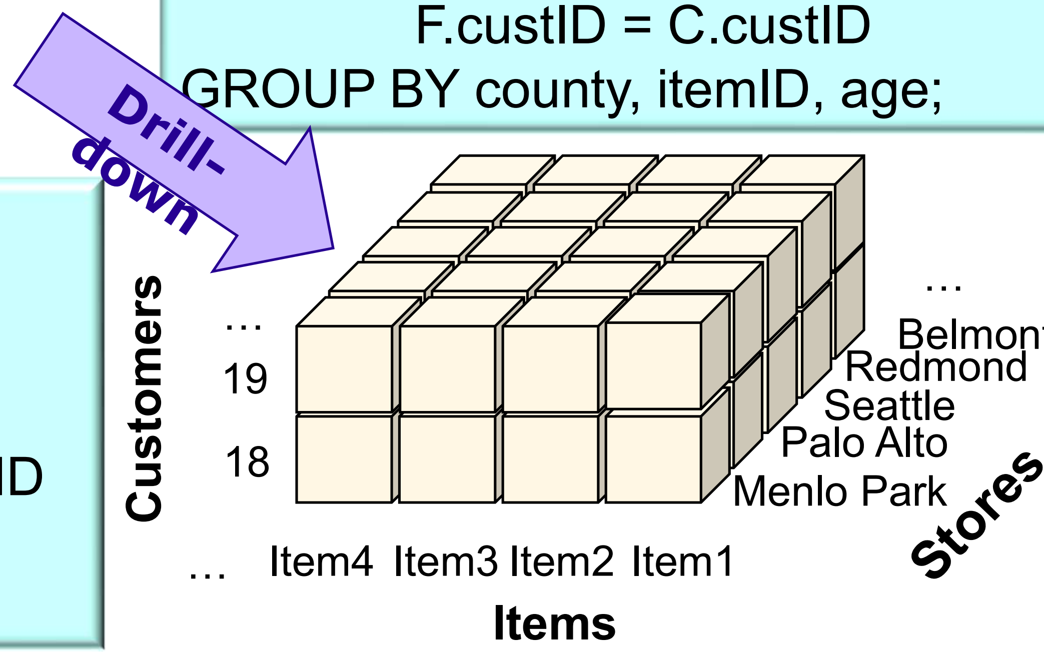
- Drill-down: reverse of roll-up
  - From higher level summary to lower level summary (i.e., we want more detailed data)
  - Introducing new dimensions

Drill-down Example 1 (Hierarchy)

Use Drill-down on total sales by item and age for each county to find total sales by item and age for each city.



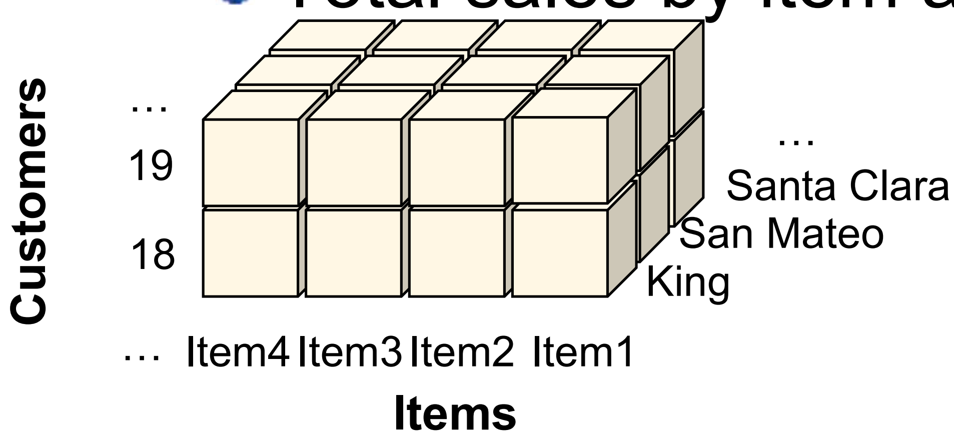

```
SELECT county, itemID, age,
            SUM(sales)
FROM    AllSales F, Store S,
            Customer C
WHERE F.storeID = S.storeID AND
            F.custID = C.custID
GROUP BY county, itemID, age;
```




```
SELECT city, itemID, age,
SUM(sales)
FROM    AllSales F, Store S,
Customer C
WHERE F.storeID = S.storeID AND
            F.custID = C.custID
GROUP BY city, itemID, age;
```

OLAP Queries – Slicing

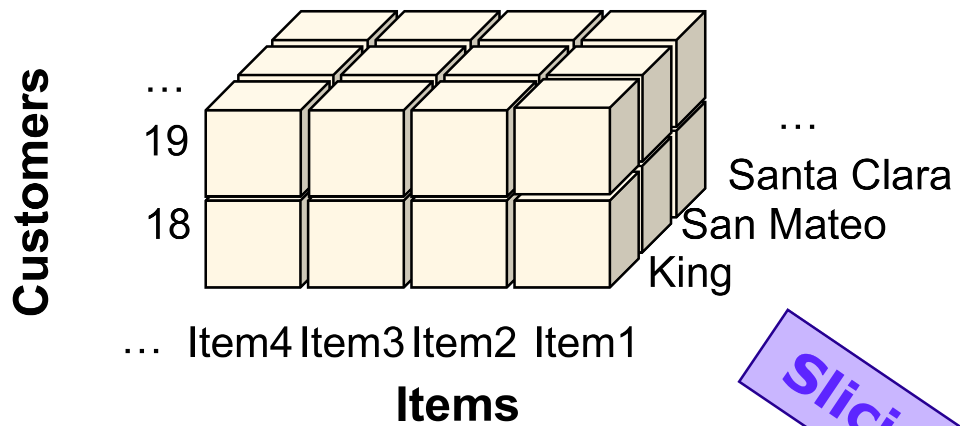
- The slice operation produces a slice of the cube by picking a range or a specific value for one of the dimensions.
- To start our example, let's specify:
  - Total sales by item and age for each county



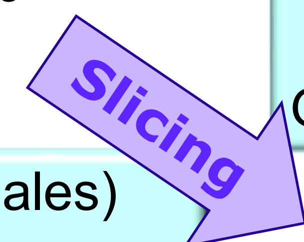

```
SELECT county, itemID, age,
            SUM(sales)
FROM    AllSales F, Store S,
           Customer C
WHERE F.storeID = S.storeID AND
            F.custID = C.custID
GROUP BY county, itemID, age;
```

Slicing Example 1

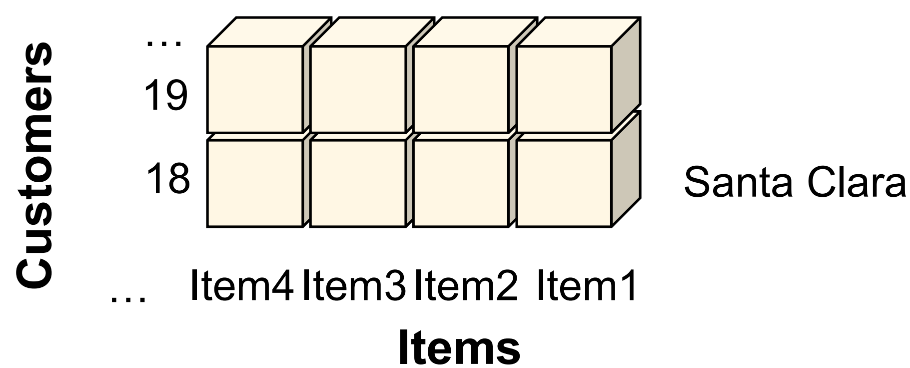
Use Slicing on total sales by item and age for each county to find total sales by item and age for Santa Clara.




```
SELECT county, itemID, age,
          SUM(sales)
FROM    AllSales F, Store S,
Customer C
WHERE F.storeID = S.storeID AND
          F.custID = C.custID
GROUP BY county, itemID, age;
```

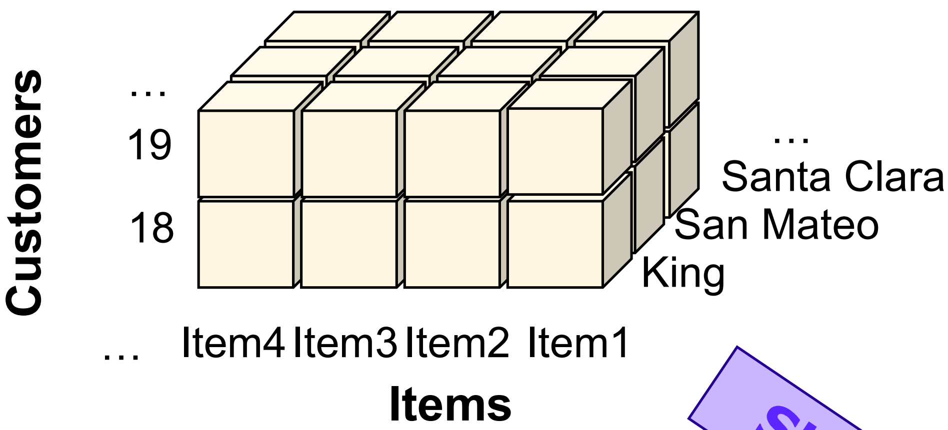



```
SELECT itemID, age, SUM(sales)
FROM    AllSales F, Store S,
Customer C
WHERE F.storeID = S.storeID AND
          F.custID = C.custID AND
          S.county = 'Santa Clara'
GROUP BY itemID, age;
```



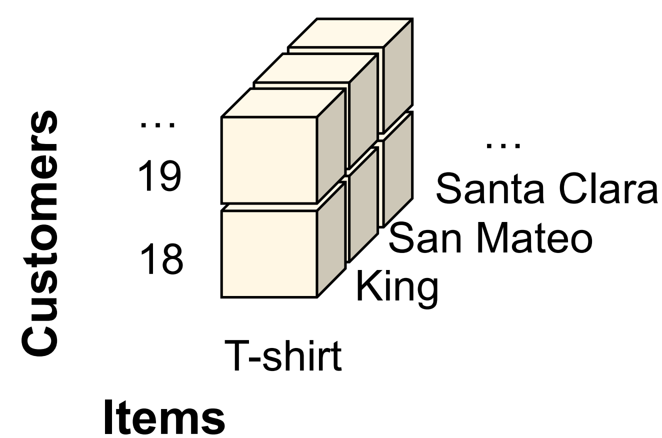
Slicing Example 2

Use Slicing on total sales by item and age for each county to find total sales by age and county for T-shirts.




```
SELECT county, itemID, age,
           SUM(sales)
FROM    AllSales F, Store S,
           Customer C
WHERE F.storeID = S.storeID AND
           F.custID = C.custID
GROUP BY county, itemID, age;
```


```
SELECT county, age, SUM(sales)
FROM    AllSales F, Store S, Customer C, Item I
WHERE F.storeID = S.storeID AND
           F.custID = C.custID AND
           F.itemID = I.itemID AND
           category  = 'Tshirt'
GROUP BY county, age;
```



OLAP Queries – Dicing

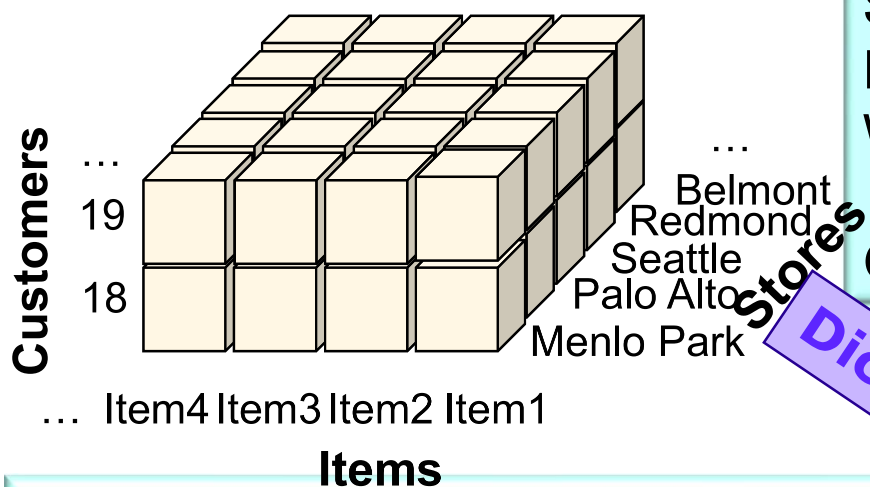
- The dice operation produces a sub-cube by picking ranges or specific values for multiple dimensions.
- To start our example, let's specify:
  - Total sales by age, item, and city


```
SELECT city, itemID, age, SUM(sales)
FROM    AllSales F, Store S, Customer C
WHERE F.storeID = S.storeID AND
             F.custID = C.custID
GROUP BY city, itemID, age;
```

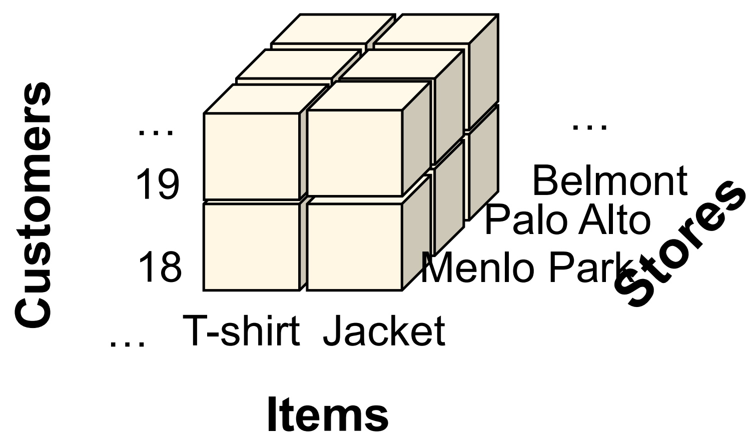
Dicing Example 1

Use Dicing on total sales by age, item, and city to find total sales by age, category, and city for red items in the state of California.

```
SELECT city, itemID, age, SUM(sales)
FROM    AllSales F, Store S, Customer C
WHERE F.storeID = S.storeID AND
            F.custID = C.custID
GROUP BY city, itemID, age;
```




```
SELECT  category, city, age, SUM(sales)
FROM AllSales F, Store S, Customer C, Item I
WHERE  F.storeID = S.storeID AND
             F.custID = C.custID AND
             F.itemID = I.itemID AND
             color = 'red' AND state = 'CA'
GROUP BY category, city, age;
```

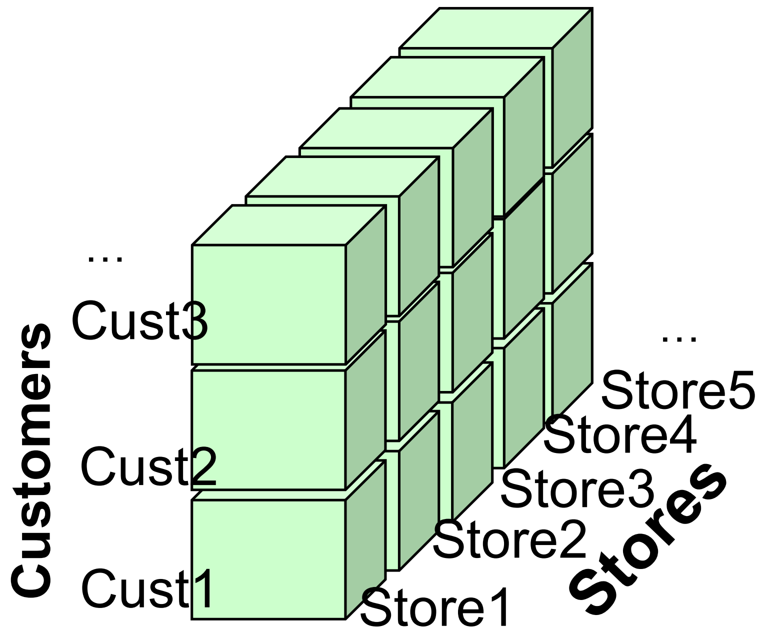


OLAP Queries – Pivoting

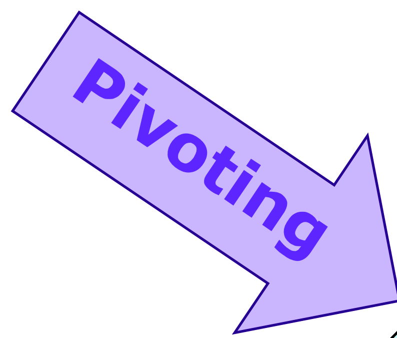
- Pivoting is a visualization operation that allows an analyst to rotate the cube in space in order to provide an alternative presentation of the data.

Pivoting Example 1

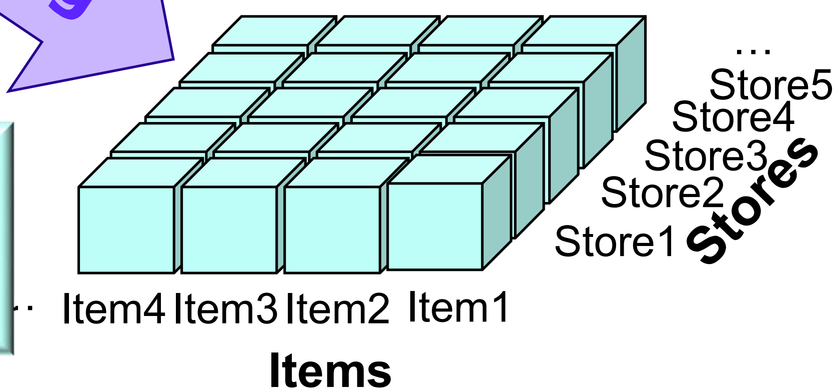
From total sales by store and customer, pivot to find total sales by item and store.




```
SELECT storeID, custID, sum(sales)
FROM  AllSales
GROUP BY storeID, custID;
```

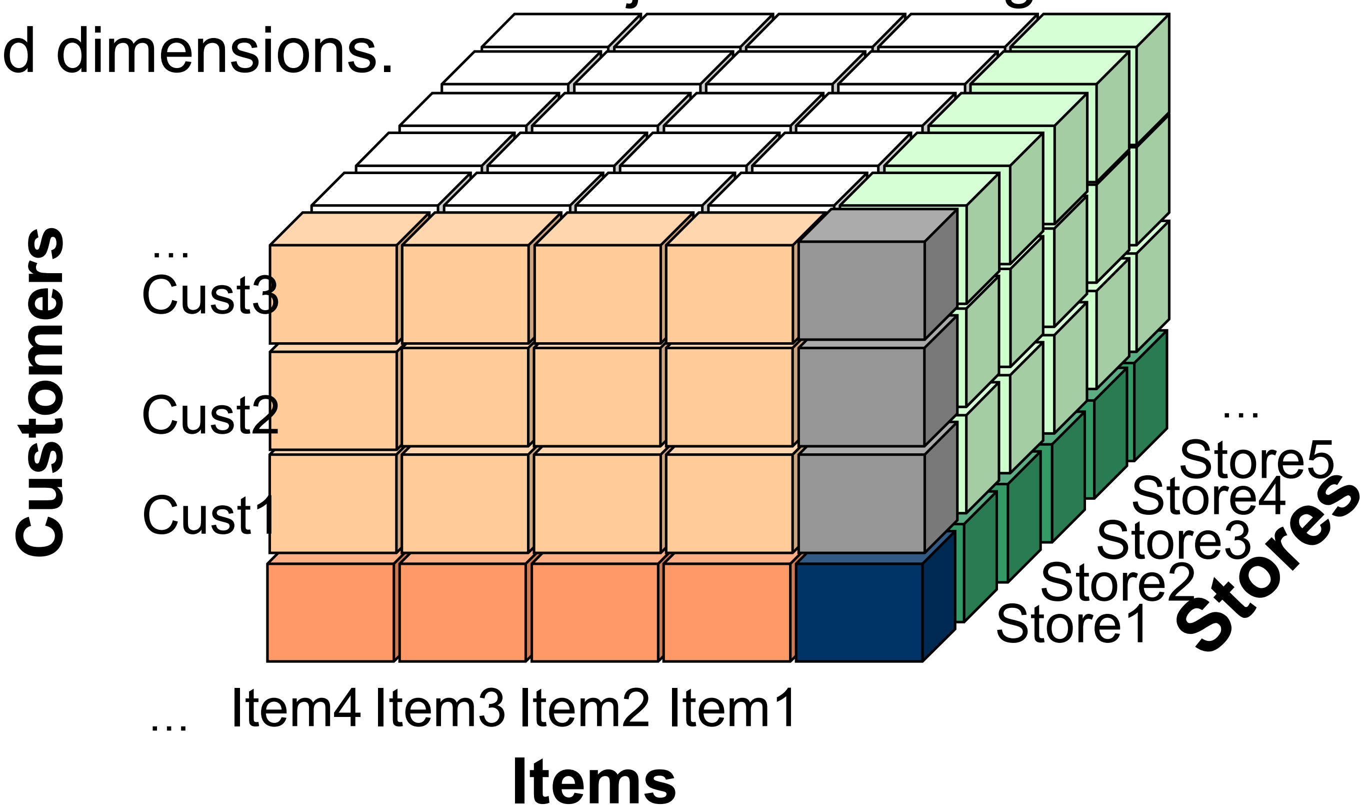



```
SELECT storeID, itemID, sum(sales)
FROM  AllSales
GROUP BY storeID, itemID;
```



Data Cube

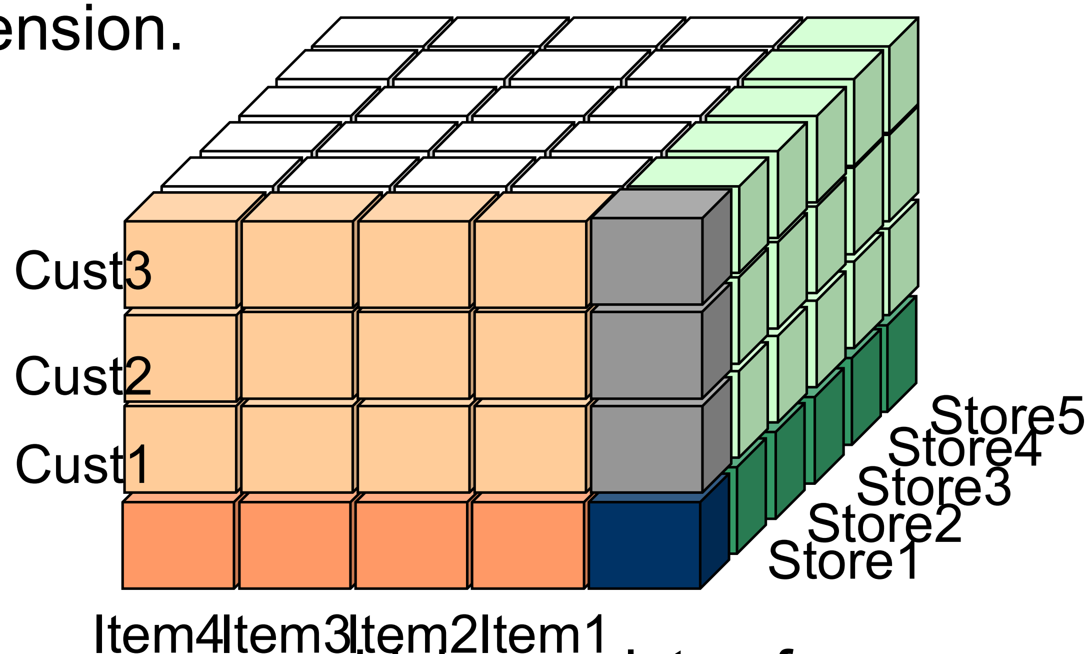
- A *data cube* is a *k*-dimensional object containing both fact data and dimensions.



- A cube contains pre-calculated, aggregated, summary information to yield fast queries.

Data Cube (cont.)

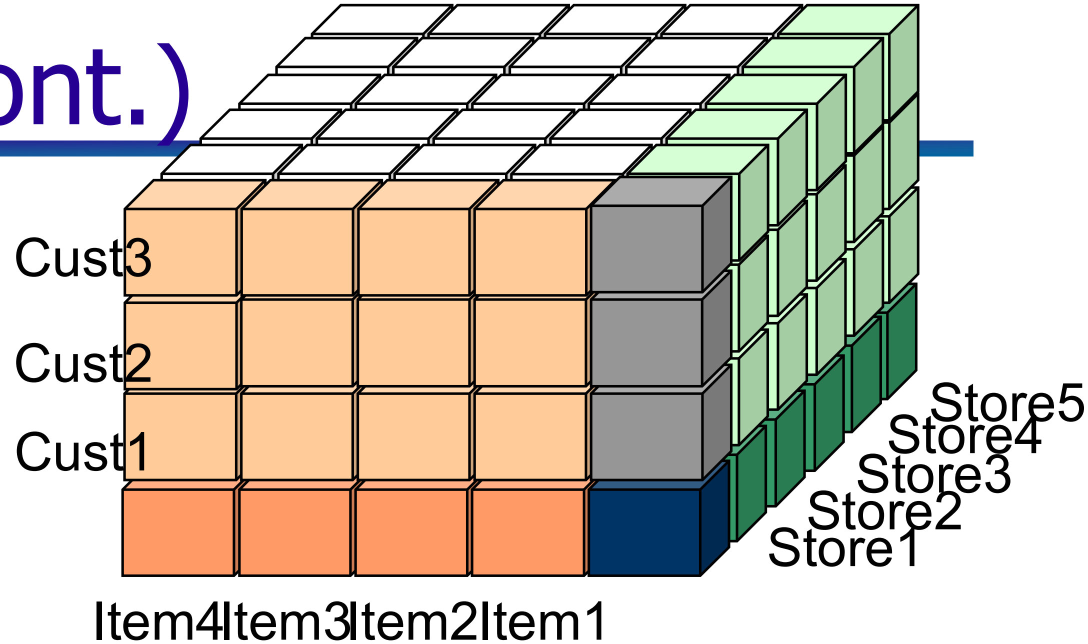
- The small, individual blocks in the multidimensional cube are called *cells*, and each cell is uniquely identified by the *members* from each dimension.



- The cells contain a *measure group*, which consists of one or more numeric *measures*. These are facts (or aggregated facts). An example of a measure is the dollar value in sales for a particular product

Data Cube (cont.)

.



- White: per customer, per item, per store (all separate)
- Dark blue: all customers, all items, all stores (all aggregated)
- Grey: per customer, all items, all stores
- Light orange: per customer, per items, all stores
- Light green: per customer, all items, per store
- Dark orange: all customers, per item, all stores
- Dark green: all customers, all items, per store

Estimating size of a cube

- Consider a car sales cube with dimensions of models, colours, and years
- With 2 models, 2 colours, and 2 years, how many tuples are there in the cube, ***assuming*** there is data for every combo. of (model, year, color)?
- 2 x 2 x 2 tuples in the group-by (model, year, color).
- Each of model, year, colour can independently be “All”.
- (2+1) x (2+1) x (2+1) = 3 x 3 x 3 combos in all.
- Formally, consider a cube with n dimensions with dimension i having C_i values. The size of the cube is $\prod_{i=1}^{n}(Ci + 1)$

Up until now, we've assumed that all cube entries have data

- A cube is *dense* if it has data for all combinations of dimension attributes
  - In practice, a cube is dense if > p% combinations are present for some suitable threshold of p
- Otherwise it is *sparse*

Estimating the size of a sparse cube

- Sparse cube size ≈ dense cube size × sparsity factor
- E.g., suppose car sales cube from previous example has a sparsity factor of 10%. Then the estimated size of sparse car sales cube = 10% of 1386 or 139 tuples
- We care about estimating cube size to help guide how we (1) compute the most "useful" subset of a cube or (2) best compute the full cube – both coming up shortly

The CUBE Operator

Generalizing the previous example, if there are *k* dimensions, we have 2^k possible SQL GROUP BY queries that can be generated through pivoting on a subset of dimensions. A CUBE operator generates that.

- It's equivalent to rolling up AllSales on all eight subsets of the set {storeID, itemID, custID }.
- Each roll-up corresponds to an SQL query of the form:

Lots of research on optimizing the CUBE operator!

```
SELECT SUM (sales)
FROM    AllSales S
GROUP BY grouping-list
```

The CUBE Operator (cont.)

- Roll-up, Drill-down, Slicing, Dicing, and Pivoting operations are expensive.
- SQL:1999 extended GROUP BY to support CUBE (and ROLLUP).
- GROUP BY CUBE provides efficient computation of multiple granularity aggregates by sharing work (e.g., passes over fact table, previously computed aggregates).

WITH CUBE

```
Select dimension-attrs, aggregates
From   tables
Where  conditions
Group By dimension-attrs With Cube
```

Not implemented in MySQL

```
SELECT storeID, itemID, custID,
          sum(sales)
FROM   AllSales
GROUP BY storeID, itemID, custID WITH
CUBE
```

storeID	itemID	custID	Sum
store1	item1	cust1	10
store1	item1	Null	70
store1	Null	cust1	145
store1	Null	Null	325
Null	item1	cust1	10
Null	item1	Null	135
Null	Null	cust1	670
Null	Null	Null	3350

....

WITH ROLLUP

```
Select dimension-attrs, aggregates
From   tables
Where  conditions
Group By dimension-attrs With Rollup
```

Can be used in dimensions that are organized in a hierarchy:

```
SELECT state, county, city, sum(sales)
FROM   AllSales F, Store S
WHERE F.storeID = S.storeID
GROUP BY state, county, city WITH ROLLUP
```

State	County	city	Sum
CA	Santa Clara	Palo Alto	325
CA	Santa Clara	Mountain view	805
CA	Santa Clara	Null	1130
CA	Null	Null	1980
Null	Null	Null	3350

....

State
|
County
|
City

WITH CUBE Example Implemented WITH ROLLUP

Implement the WITH CUBE operator using the WITH ROLLUP operator

```
SELECT storeID, itemID, custID, sum(sales)
FROM AllSales
GROUP BY storeID, itemID, custID with rollup
UNION

SELECT storeID, itemID, custID, sum(sales)
FROM AllSales
GROUP BY itemID, custID, storeID with rollup
UNION

SELECT storeID, itemID, custID, sum(sales)
FROM AllSales
GROUP BY custID, storeID, itemID with rollup;
```

Clicker Question

- Consider a fact table Facts(D1,D2,D3,x), and the following queries:

```
Q1: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3
Q2: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3 with cube
```

- Suppose attributes D1, D2, and D3 have n1, n2, and n3 different values respectively, and assume that each possible combination of values appears at least once in table Facts. Pick the one tuple (a,b,c,d,e) in the list below such that when n1=a, n2=b, and n3=c, then the result sizes of queries Q1 and Q2 are d and e, respectively.
- A: (2, 2, 2, 8, 64)
- B: (5, 4, 3, 60, 64)
- C: (5, 10, 10, 500, 726)
- D: (4, 7, 3, 84, 160)

Hint: It may be helpful to first write formulas describing how d, and e depend on a, b, and c.

Clicker Question

- Consider a fact table Facts(D1,D2,D3,x), and the following queries:

```
Q1: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3
Q2: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3 with cube
```

- Suppose attributes D1, D2, and D3 have n1, n2, and n3 different values respectively, and assume that each possible combination of values appears at least once in table Facts. Pick the one tuple (a,b,c,d,e) in the list below such that when n1=a, n2=b, and n3=c, then the result sizes of queries Q1 and Q2, are d and e, respectively.
- A: (2, 2, 2, 8, 64)
- B: (5, 4, 3, 60, 64)
- C: (5, 10, 10, 500, 726)
- D: (4, 7, 3, 84, 160)

$d = a*b*c$
$e = (a+1)*(b+1)*(c+1)$

Clicker Question

- Consider a fact table Facts(D1,D2,D3,x), and the following queries:

Q1: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3
Q2: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3 with cube
Q3: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3 with rollup

- Suppose attributes D1, D2, and D3 have n1, n2, and n3 different values respectively, and assume that each possible combination of values appears at least once in table Facts. Pick the one tuple (a,b,c,d,e,f) in the list below such that when n1=a, n2=b, and n3=c, then the result sizes of queries Q1, Q2, and Q3 are d, e, and f respectively.
- A: (2, 2, 2, 8, 64, 15)
- B: (5, 4, 3, 60, 64, 80)
- C: (5, 10, 10, 500, 726, 556)
- D: (4, 7, 3, 84, 160, 84)

Hint: It may be helpful to first write formulas describing how d, e, and f depend on a, b, and c.

Clicker Question

- Consider a fact table Facts(D1,D2,D3,x), and the following queries:

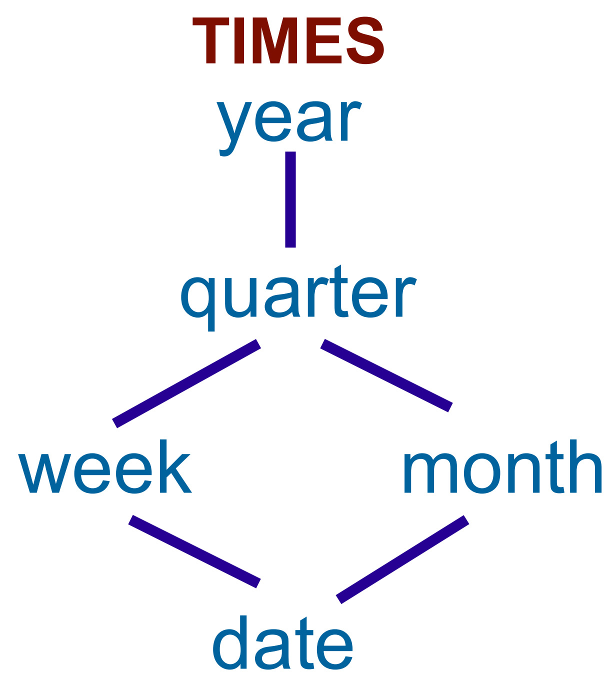
```
Q1: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3
Q2: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3 with cube
Q3: Select D1,D2,D3,Sum(x) From Facts Group By D1,D2,D3 with rollup
```

- Suppose attributes D1, D2, and D3 have n1, n2, and n3 different values respectively, and assume that each possible combination of values appears at least once in table Facts. Pick the one tuple (a,b,c,d,e,f) in the list below such that when n1=a, n2=b, and n3=c, then the result sizes of queries Q1, Q2, and Q3 are d, e, and f respectively.
- A: (2, 2, 2, 8, 64, 15)
- B: (5, 4, 3, 60, 64, 80)
- C: (5, 10, 10, 500, 726, 556)
- D: (4, 7, 3, 84, 160, 84)

d = a*b*c
e = a*b*c+a*b+a*c+b*c+a+b+c+1
e = (a+1)*(b+1)*(c+1)
f = a*b*c + a*b + a + 1

“Date” or “Time” Dimension

- Date or Time is a special kind of dimension.
- It has some special and useful OLAP functions.
  - e.g., durations or time spans, fiscal years, calendar years, and holidays
  - Business intelligence reports often deal with time-related queries such as comparing the profits from this quarter to the previous quarter … or to the same quarter in the previous year.



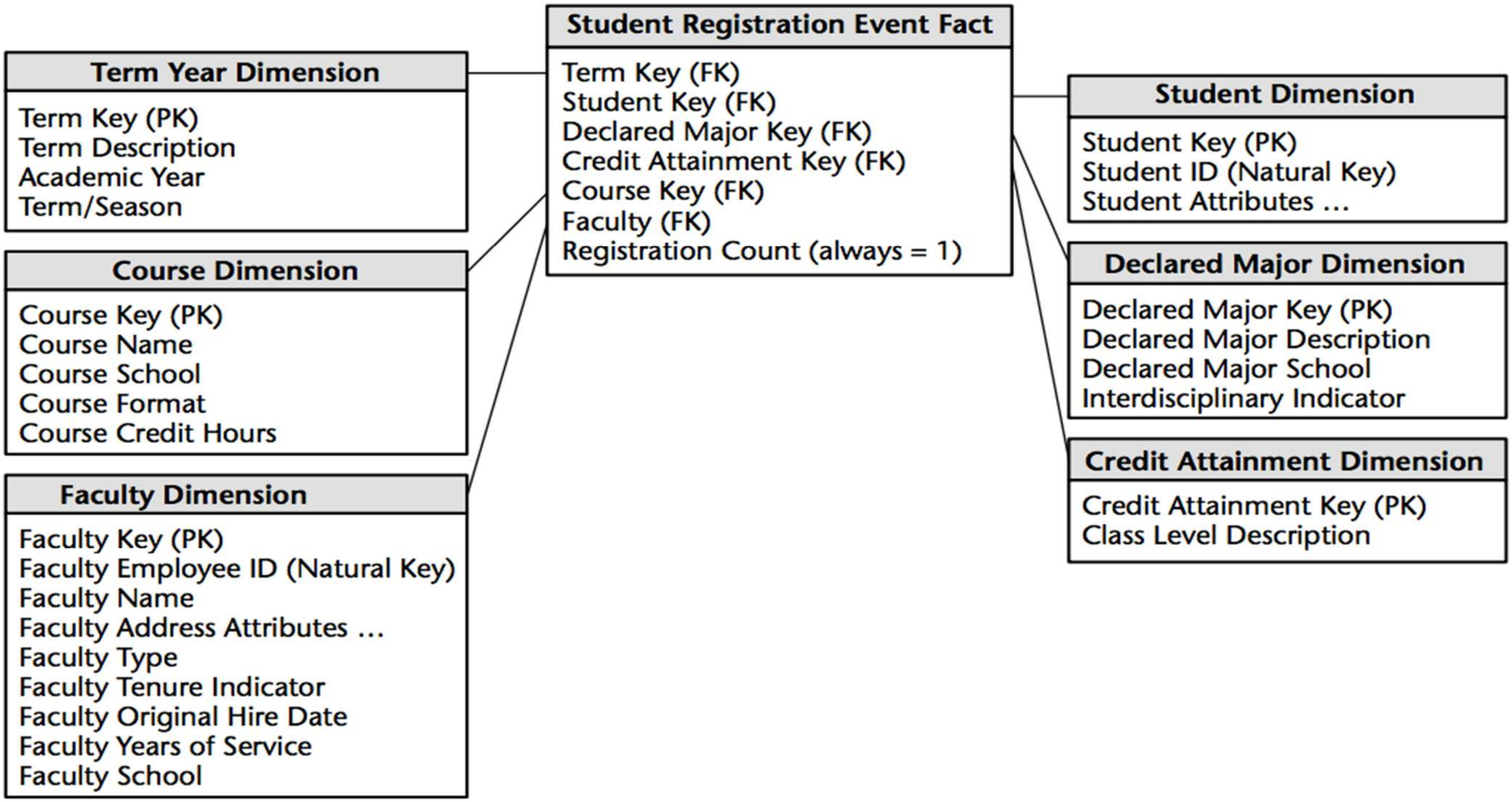
Measures in Fact Tables

- **Additive** facts are measurements in a fact table that can be added across all the dimensions. e.g., sales
- **Semi-additive** facts are numeric facts that can be added along some dimensions in a fact table but not others.
  - balance amounts are common semi-additive facts because they are additive across all dimensions except time.
- **Non-additive** facts cannot logically be added between rows.
  - Ratios and percentages
  - A good approach for non-additive facts is to store the fully additive components and later compute the final non-additive fact.

Factless Fact Tables

- **Factless fact table:** A fact table that has no facts but captures certain many-to-many relationships between the dimension keys. It’s most often used to represent events or to provide coverage information that does not appear in other fact tables.

Factless Fact Table Example



Source: *The Data Warehouse Toolkit* textbook

Multidimensional Data Model

- Multidimensional data can be stored in one of 3 ways (modes):
  - **ROLAP** (relational online analytical processing)
    - We access the data in a relational database and generate SQL queries to calculate information at the appropriate level when an end user requests it.
  - **MOLAP** (multidimensional online analytical processing)
    - Requires the pre-computation and storage of information in the cube — the operation known as processing. Most MOLAP solutions store such data in an optimized multidimensional array storage, rather than in a relational database.

MOLAP vs. ROLAP

	MOLAP	ROLAP
Data Compression	Can require up to 50% less disk space. A special technique is used for storing sparse cubes.	Requires more disk space
Query Performance	Fast query performance due to optimized storage, multidimensional indexing and caching	Not suitable when the model is heavy on calculations; this doesn't translate well into SQL.
Data latency	Data loading can be quite lengthy for large data volumes. This is usually remedied by doing only incremental processing.	As data always gets fetched from a relational source, data latency is small or none.
handling non-aggregatable facts	Tends to suffer from slow performance when dealing with textual descriptions	Better at handling textual descriptions

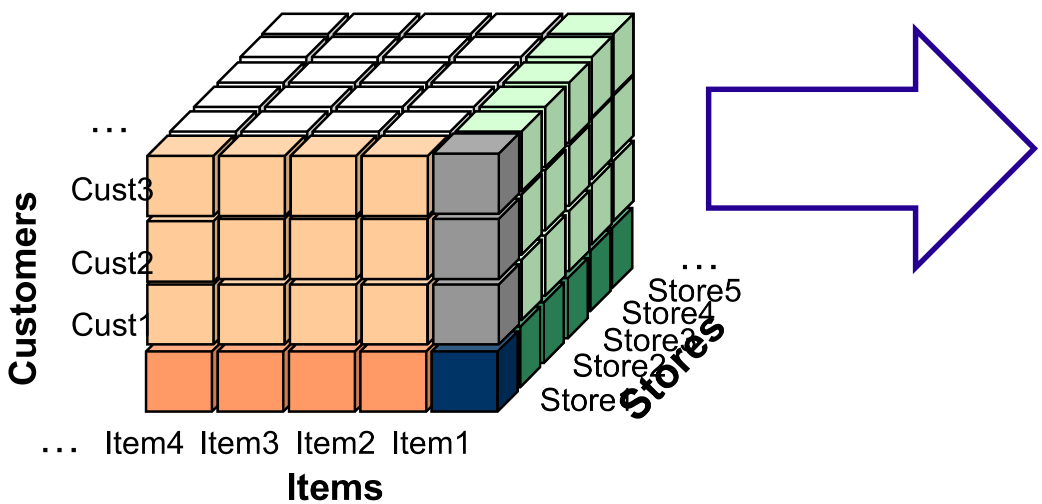
Which Storage Mode is Recommended?

- Almost always, choose MOLAP.
- Choose ROLAP if one or more of these are true:
  - There is a very large number of members in a dimension—typically hundreds of millions of members.
  - The dimension data is frequently changing.
  - You need real-time access to current data (as opposed to historical data).
  - You don’t want to duplicate data.
    - Reference: [Harinath, et *al.*, 2009]

- **HOLAP** (**hybrid online analytical processing**) is a combination of ROLAP and MOLAP, which allows storing part of the data in a MOLAP store and another part of the data in a ROLAP store, allowing us to exploit the advantages of each.

Great! Now let's talk about how that cube is stored a bit…

Representing a Cube in a Two-Dimensional Table



storeID	itemID	custID	Sum
store1	item1	cust1	10
store1	item1	Null	70
store1	Null	cust1	145
store1	Null	Null	325
Null	item1	cust1	10
Null	item1	Null	135
Null	Null	cust1	670
Null	Null	Null	3350

....

Add to the original cube: faces, edges, and corners … which are represented in the 2-D table using NULLs.

Finding Answers Quickly

- Large datasets and complex queries mean that we'll need improved querying capabilities
  - Materializing views
  - Finding Top N Queries
  - Using online aggregation

Queries over Views

Reminder: use a view as follows:

View

```
Create view TshirtSales AS
SELECT category, county, age, sales
FROM    AllSales F, Store S, Customer C, Item I
WHERE F.storeID = S.storeID AND F.custID = C.custID AND
            F.itemID = I.itemID ANDcategory  = 'Tshirt'
```

Query

```
SELECT category, county, age, SUM(sales)
From      TshirtSales
GROUP BY category, county, age;
```

Materializing Views

- Decision support activities require queries against complex view definitions to be computed quickly.
- Sophisticated optimization and evaluation techniques are not enough since OLAP queries are typically aggregate queries that require joining huge tables.
- Pre-computation is essential for interactive response times.
- A view whose tuples are stored in the database is said to be materialized.

Issues in View Materialization: Which views should we materialize?

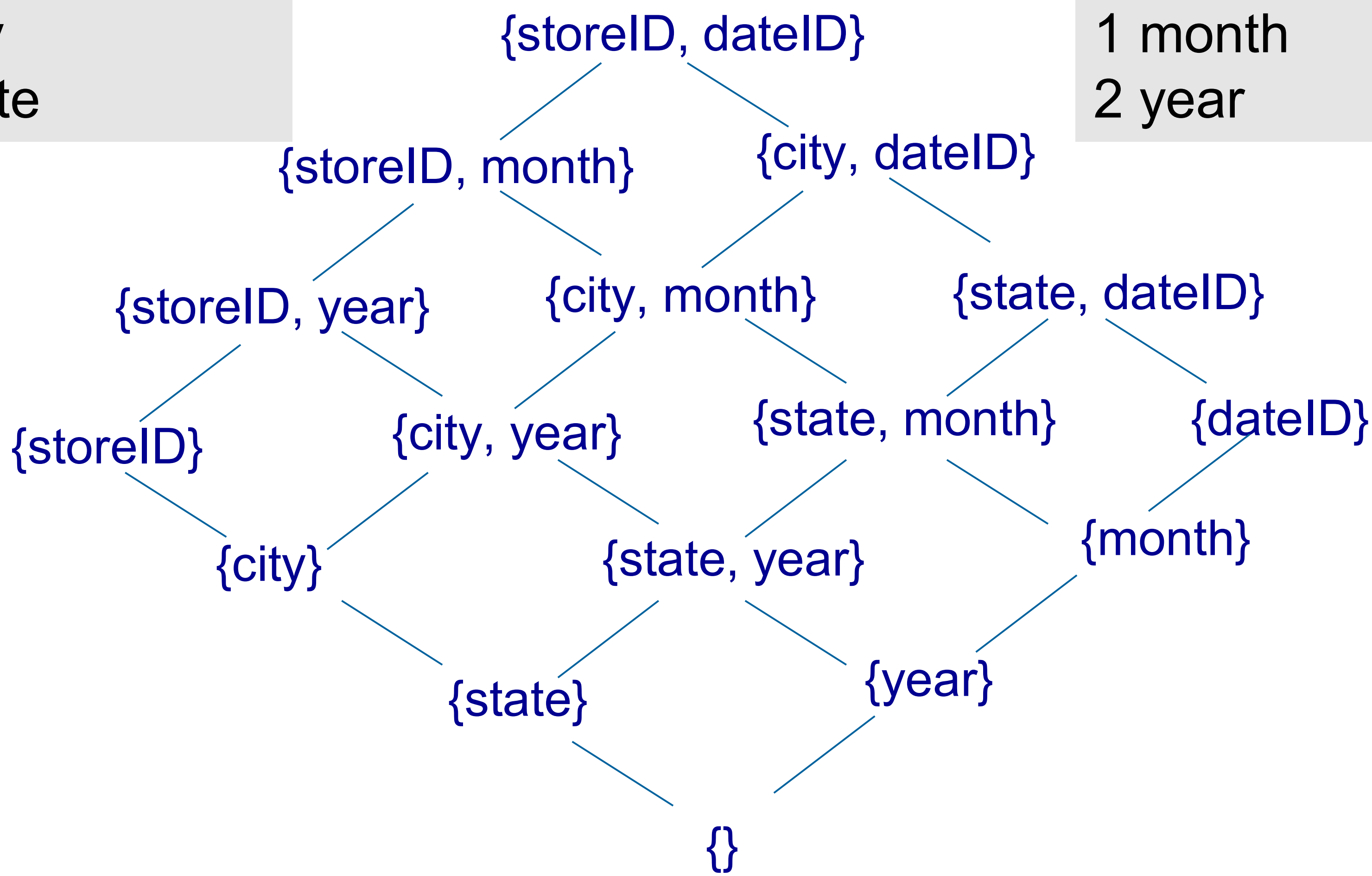
- Which views should we materialize?
- Based on size estimates for the views, suppose there is space for *k* views to be materialized. Which ones should we materialize?
- The goal is to materialize a small set of carefully chosen views to answer most of the queries quickly.
- **Fact:** Selecting *k* views to materialize such that the average time taken to evaluate all views of a lattice is minimized is a NP-hard problem.

The Exponential Explosion of Views

- Assume that we have two dimensions, each with a hierarchy

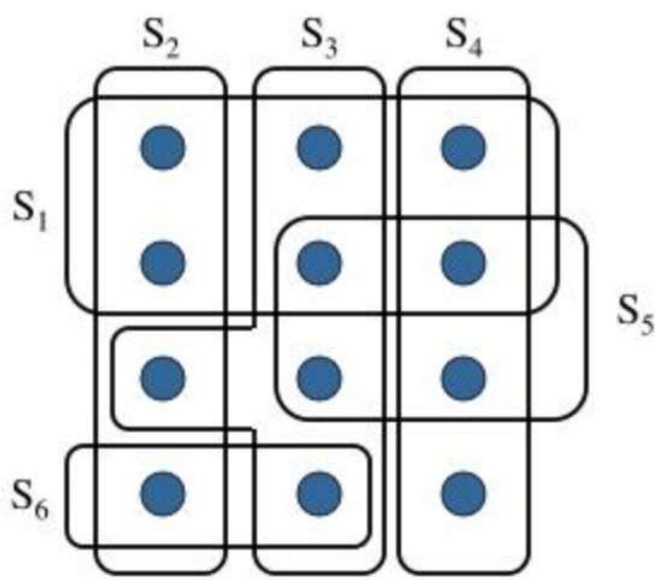
Store dimension
0 storeID
1 city
2 state

Calendar dimension
0 dateID
1 month
2 year



Maximum Coverage Problem Example

Given 12 ground facts/elements, 6 subsets, and a value for *k*, find the *k* subsets (e.g., *k* = 3) that between them cover as many ground elements as possible.



Difference between a NP-hard problem and a problem that you solve efficiently (polynomial time) is like the difference between solving a Sudoku puzzle vs. checking whether a given solution is valid.

Maximum Coverage problem has a structure similar to finding the top *k* views. We can find approximately optimal solutions quickly for both.

CPSC 304 – December 1 Administrative notes

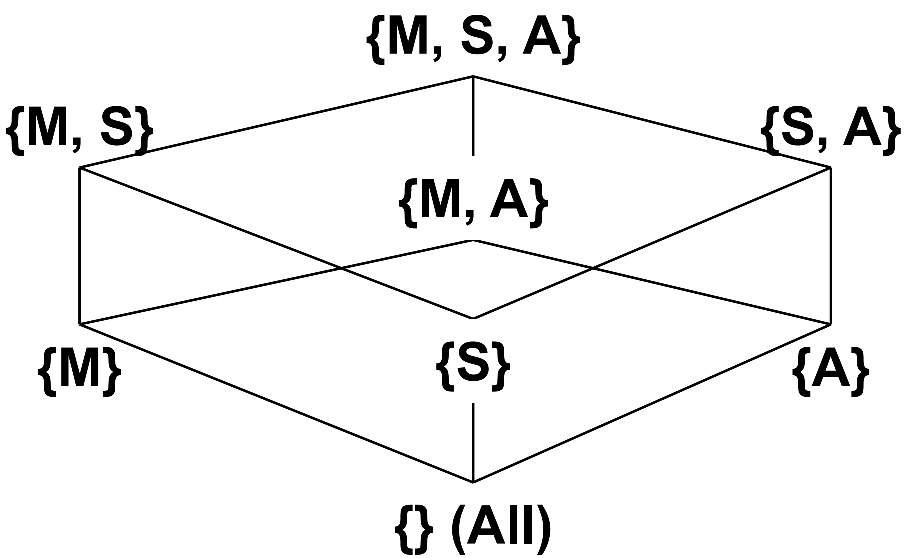
- Project Demos: November 28-December 2
- No tutorials this week
- **UNGRADED** In-class exercise released for data warehousing
- Final exam: Sunday, December 11 @3:30pm: Osborne A
  - Final exam office hours are listed on the office hour page. The page is still undergoing changes so check back regularly.

Now where were we...

- We'd been discussing data warehousing
- We'd said that it was usually visualized like a cube, and built using the cube operator
- But that cube is super big! We don't want to materialize all of it.
- How do we choose which parts to materialize?

Back to the student table: Student(snum,sname,major,standing,age)

- We can aggregate the student table based on major, standing, or age



Group by all 3 – any ordering gives you same # of tuples

```
SELECT major, standing, age, count(*)
FROM         student
GROUP BY major, standing, age
ORDER BY major, standing, age
MAJOR                     ST    AGE     COUNT(*)
------------------------- -- ---------- ----------
Accounting                JR     19              1
Animal Science            FR     18              1
Architecture              SR     22              1
Civil Engineering         SR     21              1
Computer Engineering      FR     18              1
Computer Engineering      SR     19              1
Computer Science          JR     18              1
Computer Science          JR     20              1
Computer Science          SO     17              1
Computer Science          SO     19              1
Economics                 JR     20              1
Education                 SR     21              1
Electrical Engineering    FR     17              2
English                   SR     21              1
Finance                   FR     18              2
History                   SR     20              1
Kinesiology               SO     19              1
Law                       JR     20              1
Mechanical Engineering    SO     19              1
Psychology                JR     20              1
Psychology                SO     18              1
Veterinary Medicine       SR     21              1
22 rows selected.
```

```
SELECT age, standing, major, count(*)
FROM student
GROUP BY age, standing, major
ORDER BY age, standing, major
 AGE       ST MAJOR                     COUNT(*)
---------- -- ------------------------- ----------
17         FR Electrical Engineering             2
17         SO Computer Science                   1
18         FR Animal Science                     1
18         FR Computer Engineering               1
18         FR Finance                            2
18         JR Computer Science                   1
18         SO Psychology                         1
19         JR Accounting                         1
19         SO Computer Science                   1
19         SO Kinesiology                        1
19         SO Mechanical Engineering             1
19         SR Computer Engineering               1
20         JR Computer Science                   1
20         JR Economics                          1
20         JR Law                                1
20         JR Psychology                         1
20         SR History                            1
21         SR Civil Engineering                  1
21         SR Education                          1
21         SR English                            1
21         SR Veterinary Medicine                1
22         SR Architecture                       1
22 rows selected.
```

Group by 2 option 1: Major, Standing

```
SELECT major, standing, count(*)
FROM student
GROUP BY major, standing
ORDER BY major, standing

MAJOR                     ST   COUNT(*)
------------------------- -- ----------
Accounting                JR           1
Animal Science            FR           1
Architecture              SR           1
Civil Engineering         SR           1
Computer Engineering      FR           1
Computer Engineering      SR           1
Computer Science          JR           2
Computer Science          SO           2
Economics                 JR           1
Education                 SR           1
Electrical Engineering    FR           2
English                   SR           1
Finance                   FR           2
History                   SR           1
Kinesiology               SO           1
Law                       JR           1
Mechanical Engineering    SO           1
Psychology                JR           1
Psychology                SO           1
Veterinary Medicine       SR           1

20 rows selected.
```

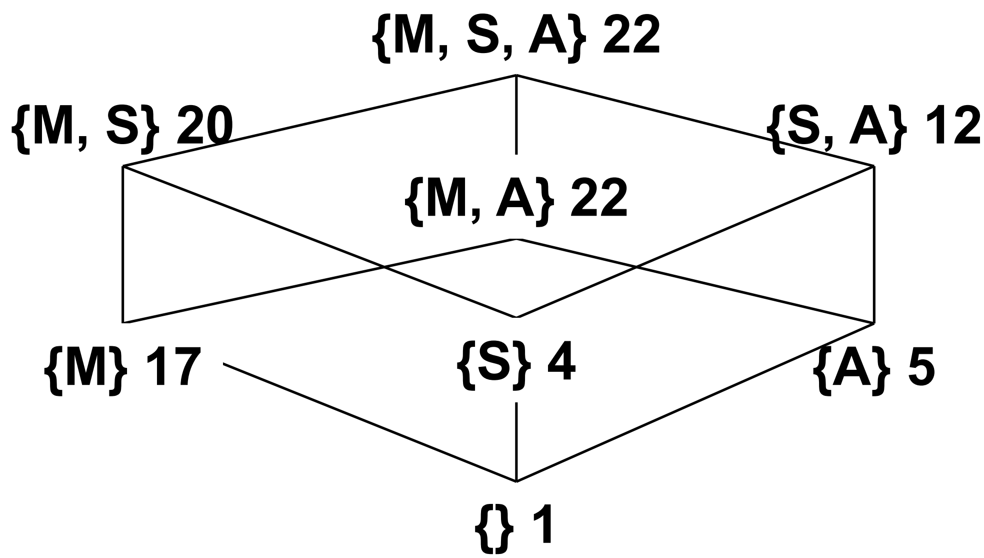
Group by 2 option 2: Standing, Age

```
SELECT standing, age, count(*)
FROM student
GROUP BY standing, age
ORDER BY standing, age

ST  AGE     COUNT(*)
-- ---------- ----------
FR   17          2
FR   18          4
JR   18          1
JR   19          1
JR   20          4
SO   17          1
SO   18          1
SO   19          3
SR   19          1
SR   20          1
SR   21          4
SR   22          1

12 rows selected.
```

Group by 2 option 3: Major, Age

```
SELECT major, age, count(*)
FROM student
GROUP BY major, age
ORDER BY major, age

MAJOR                            AGE        COUNT(*)
------------------------- ---------- ----------
Accounting                19                  1
Animal Science            18                  1
Architecture              22                  1
Civil Engineering         21                  1
Computer Engineering      18                  1
Computer Engineering      19                  1
Computer Science          17                  1
Computer Science          18                  1
Computer Science          19                  1
Computer Science          20                  1
Economics                 20                  1
Education                 21                  1
Electrical Engineering    17                  2
English                   21                  1
Finance                   18                  2
History                   20                  1
Kinesiology               19                  1
Law                       20                  1
Mechanical Engineering    19                  1
Psychology                18                  1
Psychology                20                  1
Veterinary Medicine       21                  1
22 rows selected.
```

In computing costs to query, we consider the # of tuples that you have to look at



The # is the # of tuples

Assuming all of the views were materialized, executing the query

```
SELECT standing, count(*)
FROM student
GROUP BY standing
```

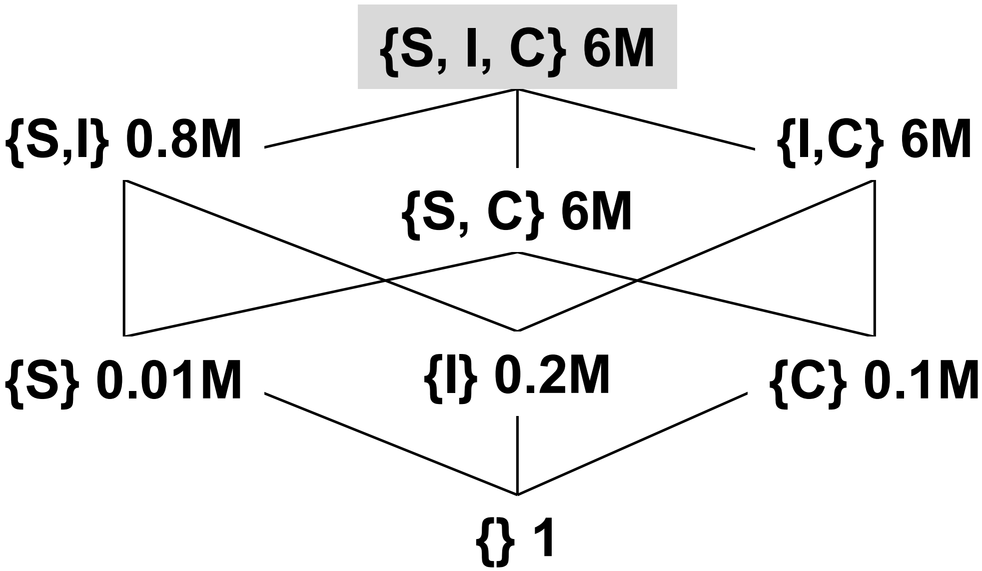
would cost 4 because that's the cheapest way to access the necessary tuples

We can also use {Major, Standing} to compute {Standing} for a cost of 20

```
SELECT major, standing, count(*)
FROM student
GROUP BY major, standing
ORDER BY major, standing

MAJOR                     ST   COUNT(*)
------------------------- -- ----------
Accounting                JR          1
Animal Science            FR          1
Architecture              SR          1
Civil Engineering         SR          1
Computer Engineering      FR          1
Computer Engineering      SR          1
Computer Science          JR          2
Computer Science          SO          2
Economics                 JR          1
Education                 SR          1
Electrical Engineering    FR          2
English                   SR          1
Finance                   FR          2
History                   SR          1
Kinesiology               SO          1
Law                       JR          1
Mechanical Engineering    SO          1
Psychology                JR          1
Psychology                SO          1
Veterinary Medicine       SR          1

20 rows selected.
```

```
SELECT standing, count(*)
FROM student
GROUP BY standing
ORDER BY standing

ST   COUNT(*)
-- ----------
SR          7
SO          5
FR          6
JR          6
```

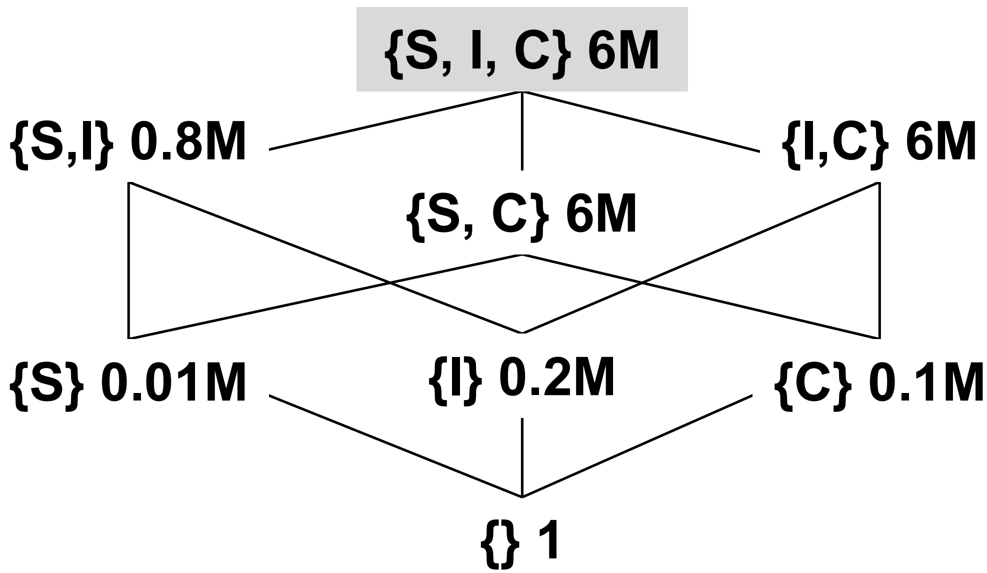
The HRU algorithm for materializing views



- The question is: which views can we materialize to answer queries the cheapest?
- Initially, only the top-most view is materialized

HRU [Harinarayan, Rajaraman, and Ullman, 1996]—SIGMOD Best Paper award—is a greedy algorithm that does not guarantee an optimal solution, though it usually produces a good solution. This solution is a good trade-off in terms of the space used and the average time to answer an OLAP query.

Benefit of Materializing a View



Intuitively, for each view under consideration, determine (1) if it can be used to answer a query and (2) if so, how much does it save?

Formally:

Define the benefit (savings) of view v relative to S as **B(v,S)**.

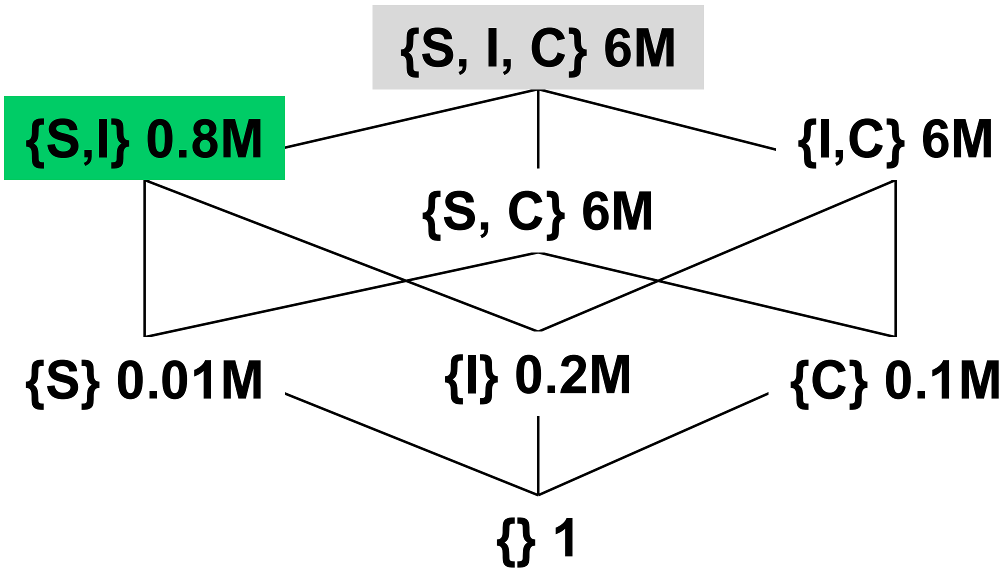
B(v, S) = 0
For each w ≦ v
  u = view of least cost in S such that w ≦ u
  if C(v) < C(u) then B_w = C(u) – C(v)
  else B_w = 0
  B(v,S) = B(v,S) + B_w
end

S = set of views selected for materialization

b ≦ a means b is a descendant of a (including itself) – b can be answered using only a (e.g., {S} ≦ {S,I}

C(v) = cost of view v, which we're approximating by its size

Benefit of Materializing a View



- The number associated with each node represents the number of rows in that view (in millions)
- Initial state has only the top most view materialized

Define the benefit (savings) of view v relative to S as **B(v,S)**.

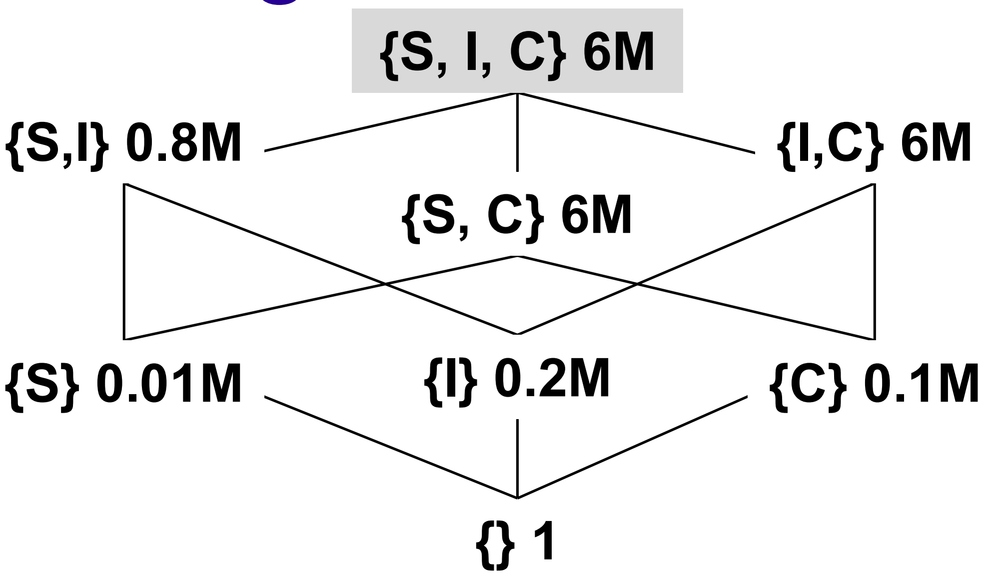
B(v, S) = 0
For each w ≦ v
    u = view of least cost in S such that w ≦ u
    if C(v) < C(u) then B_w = C(u) – C(v)
    else B_w = 0
    B(v,S) = B(v,S) + B_w
end

Example

S = {**{S, I, C}**}, **v = {S, I}**

$B_{\{S, I\}}$ **= 5.2 M**
$B_{\{S\}}$ **= 5.2 M**
$B_{\{I\}}$ **= 5.2 M**
$B_{\{\}}$ **= 5.2 M**
B(v,S) = 5.2M *4

Finding the Best *k* Views to Materialize



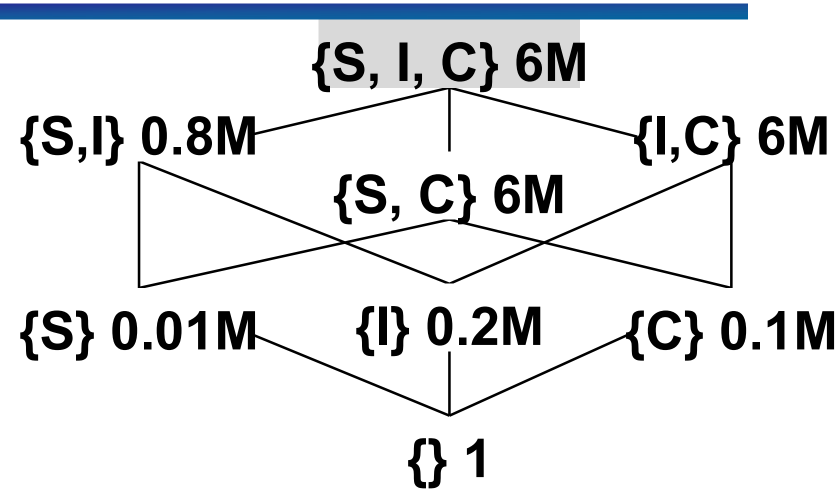
- The number associated with each node represents the number of rows in that view (often in millions)
- Initial state has only the top most view materialized

A greedy algorithm for finding the best *k* views to materialize

```
S = {top view}
for i=1 to k do begin
    select v ⊄ S such that B(v,S) is maximized
    S = S union {v}
end
```

HRU Algorithm Example. Pick the best 2 views beyond {S,I,C} to materialize: Round 1

{S,I} offers biggest benefit: materialize it.

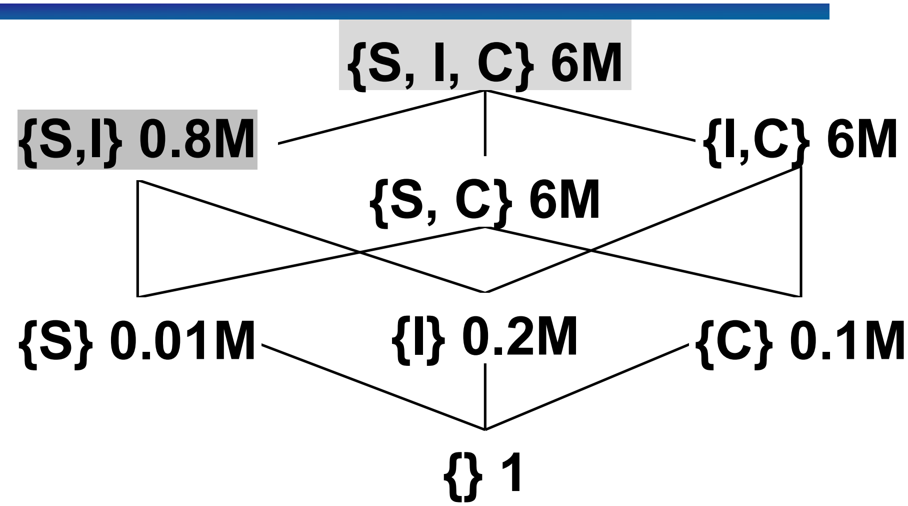


View	1st choice	Explanation
{S, I}	**(6-0.8)M *4 = 20.8M**	Can impact {S,I}, {S}, {I}, {}. Benefit is over {S,I,C}
{S, C}	(6-6) *4 = 0	Can impact {S,C}, {S},{C},{}. Benefit from {S,I,C} is zero
{I, C}	(6-6) *4 = 0	Can impact {I,C}, {I},{C},{}. Benefit from {S,I,C} is zero
{S}	(6-0.01) M*2 = 11.98M	Can impact {S}, {}. Benefit is over {S,I,C}
{I}	(6-0.2) M*2 = 11.6M	Can impact {I}, {}. Benefit is over {S,I,C}
{C}	(6-0.1) M*2 = 11.8M	Can impact {C}, {}. Benefit is over {S,I,C}
{}	6M – 1	Can impact {}. Benefit is over {S,I,C}

HRU Algorithm Example. Pick the best 2 views beyond {S,I,C} to materialize: Round 2

{S,I} is already materialized.

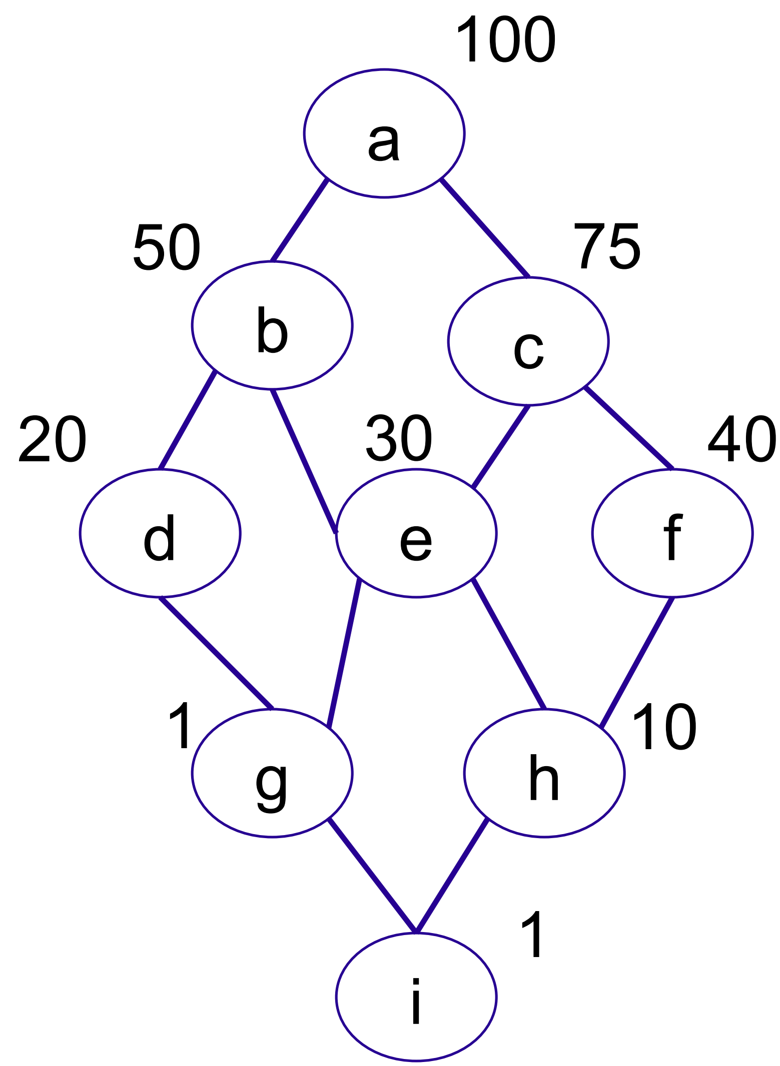
{C} offers biggest benefit: materialize it



View	2^{nd} choice	Explanation
{S, I}		Already materialized. Not an option
{S, C}	(6-6) *2 = 0	Can impact {S,C}, {S},{C},{}. Benefit of 0 from {S,I,C} for {S,C},{C}. {S,I} is cheaper for {S},{}
{I, C}	(6-6) *2 = 0	Same reasoning as {S,C}
{S}	(0.8-0.01)M*2 = 1.58M	Can impact {S}, {}. Benefit is over {S,I}
{I}	(0.8-0.2)M*2 = 1.2M	Can impact {I}, {}. Benefit is over {S,I}
{C}	**(6-0.1)M + (0.8-0.1)M = 6.6M**	Can impact {C}, {}. Benefit over {S,I,C} for {C}, {S,I} for {}
{}	0.8M - 1	Can impact {}. Benefit is over {S,I}

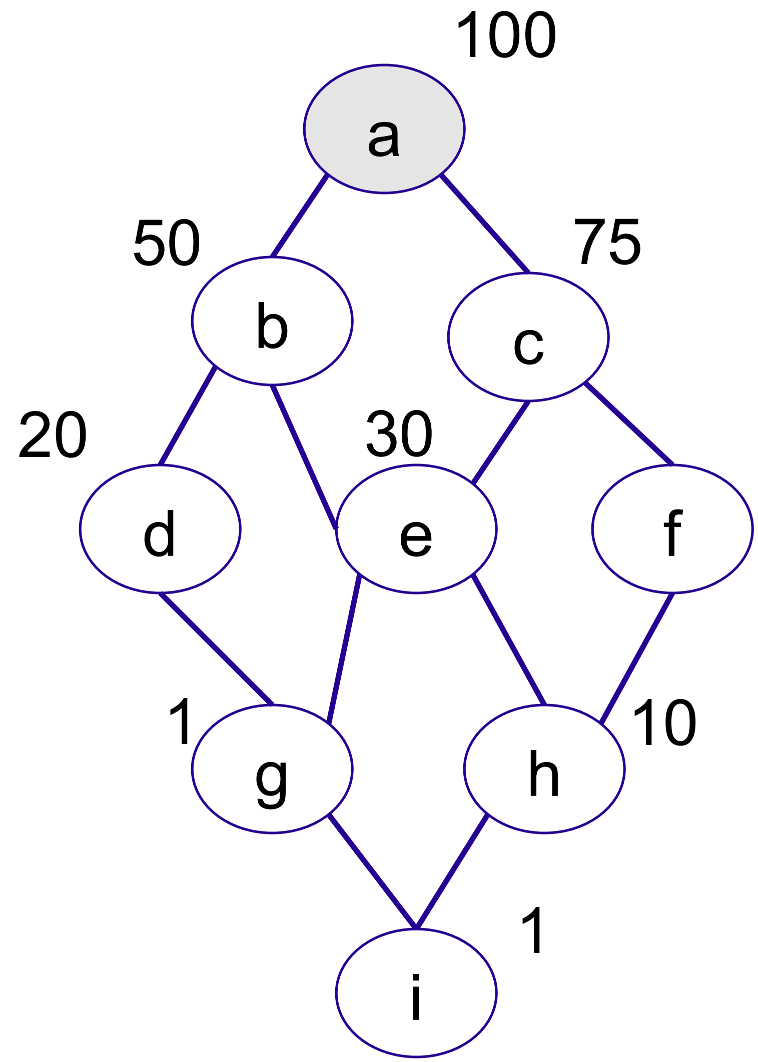
In-class Exercise

Assuming 'a' is already materialized, what are the best 3 other views that we should materialize? Begin by picking the "best" view to materialize.



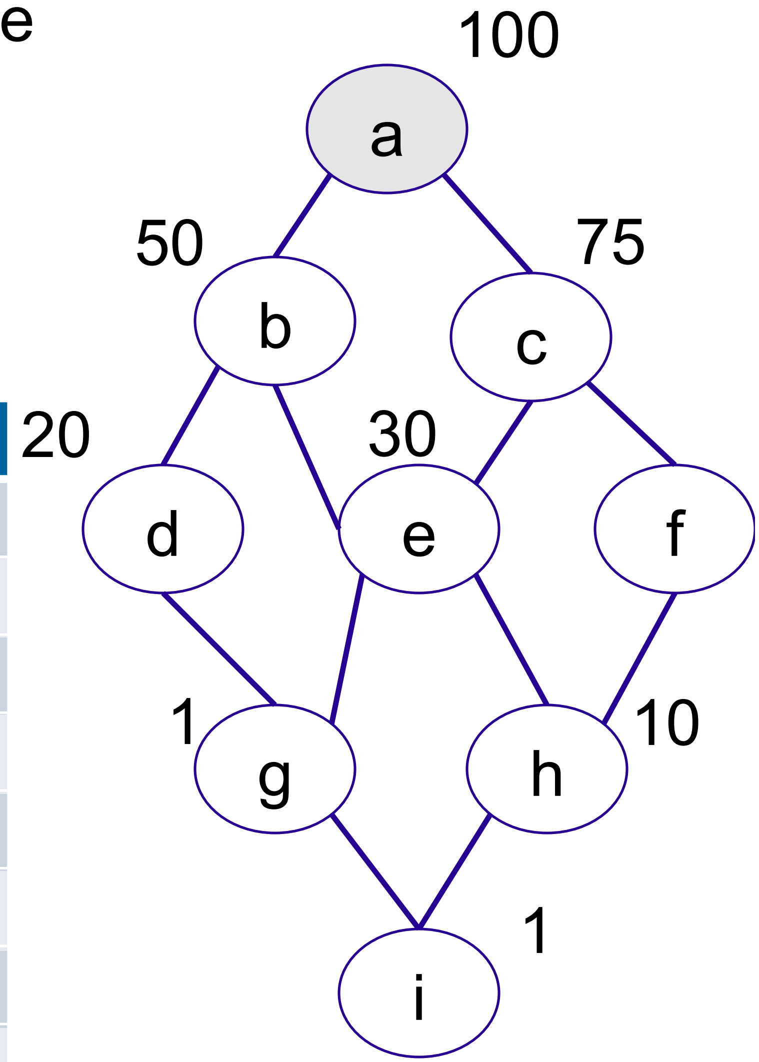
A note on workload

- Thus far, we've been assuming that the workload is evenly distributed.
- If it's not, then multiplying by the workload expected at each spot in the lattice will give you a more precise answer
- For example, assume that queries at g and h each make up 20% of the workload, and the remainder of the nodes make 10% each



Taking workload into account

- Assume queries at g and h each make up 20% of the workload, and the remainder of the nodes make 10% each

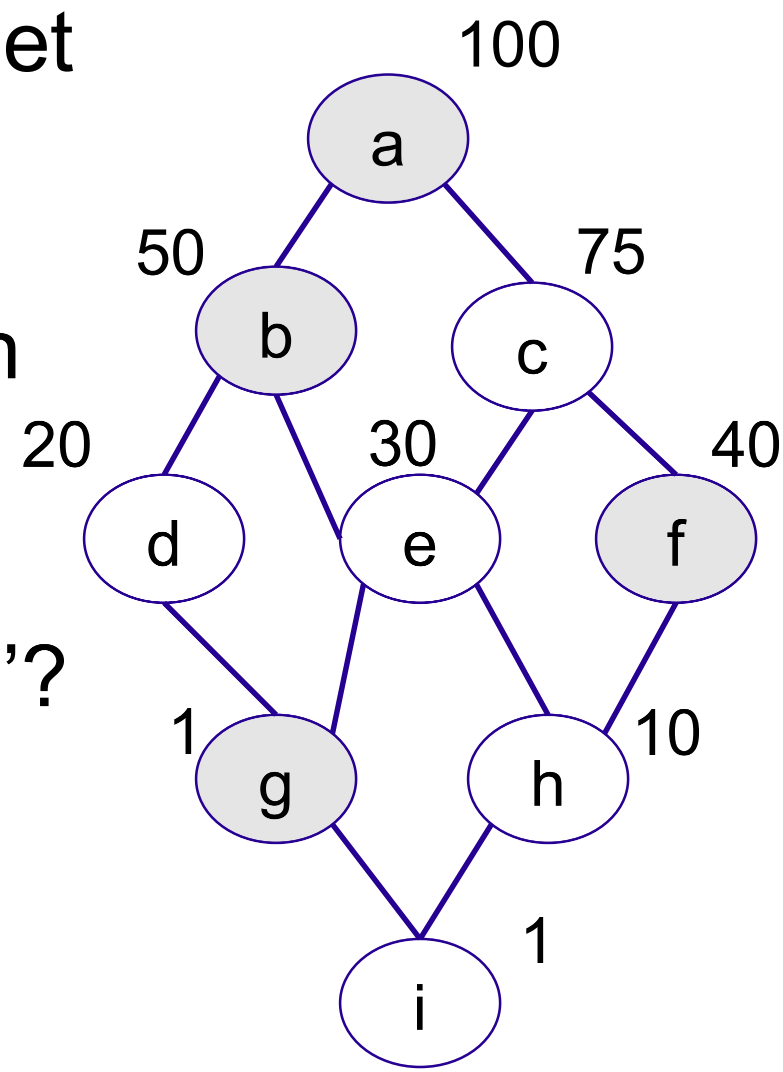


Original: With workload:

View	1^{st} choice	View	1^{st} choice
b	50*6=300	b	.2*2*50+.1*4*50=40
c	25*6=150	c	.2*2*25+.1*4*25=20
d	80*3=240	d	.2*1*80+.1*2*80=32
e	70*4=280	e	.2*2*70+.1*2*70=42
f	60*3=180	f	.2*1*60+.1*2*60=24
g	99*2=198	g	.2*1*99+.1*1*99=29.7
h	90*2=180	h	.2*1*90+.1*1*90=27
i	99	i	.1*99=9.9

Using the Materialized Views

- Once we have chosen a set of views, we need to consider how they can be used to answer queries on other views.
- What is the best way to answer queries on view ‘h’?



Issues in View Materialization: Incremental recomputing

- How do we maintain views incrementally without recomputing them from scratch?
  - Two steps
    - Modify the changes to the view when the data changes
    - Apply only those changes to the materialized view
- There may be challenges in refreshing, especially if the base tables are distributed across multiple locations

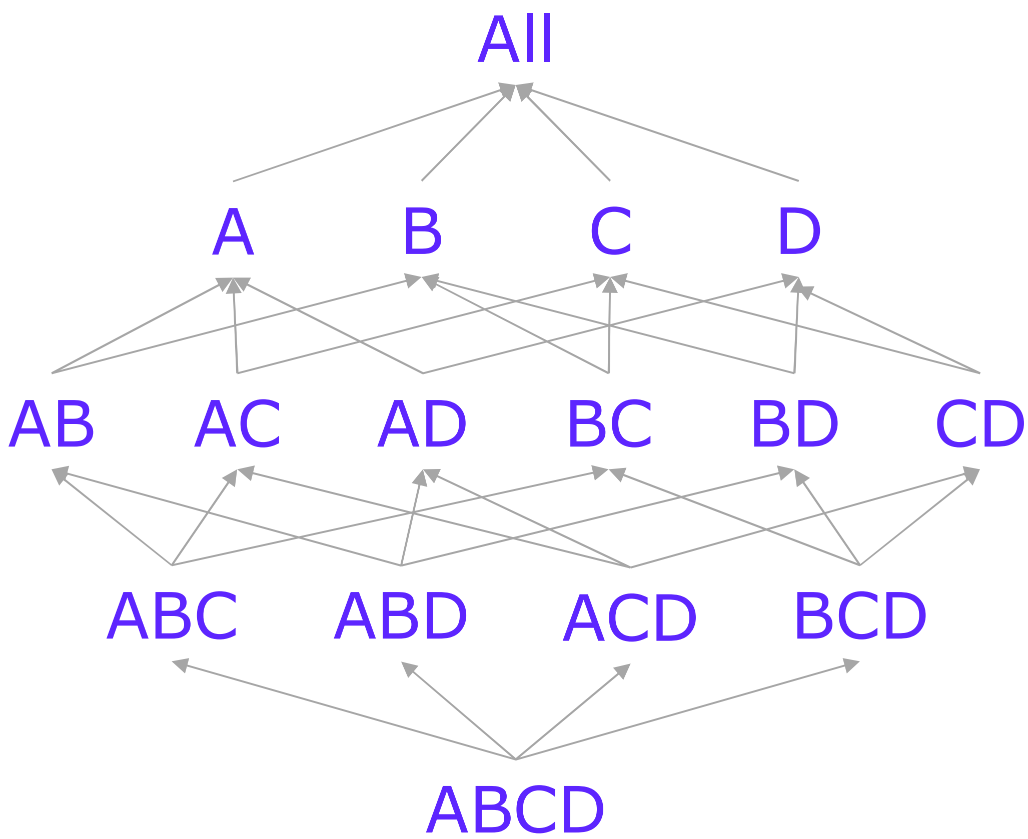
Issues in View Materialization: Handling updates

- How should we refresh and maintain a materialized view when an underlying table is modified?
- Maintenance policy: Controls when we refresh
  - Immediate: As part of the transaction that modifies the underlying data tables
    - \+ Materialized view is always consistent
    - \- Updates are slow
  - Deferred: Some time later, in a separate transaction
    - \- View is inconsistent for a while
    - \+ Can scale to maintain many views without slowing updates

Deferred Maintenance

- Three flavors:
  - Lazy: Delay refresh until next query on view; then refresh before answering the query.
    - This approach slows down queries rather than updates, in contrast to immediate maintenance.
  - Periodic (Snapshot): Refresh periodically. Queries possibly answered using outdated version of view tuples. Also widely used in asynchronous replication in distributed databases
  - Event-based (Forced): e.g., Refresh after a fixed number of updates to underlying data tables

That's how to materialize *some* of the views. What if we want *all* views?



Given:

- A cube to materialize
- For each view in the cube:
  - Cost to materialize the view if the inputs are already sorted (A)
  - Cost to materialize the view if the inputs are NOT already sorted (S)
  - Need completely different algorithm!

The difference between which views to materialize and how to materialize all views.

- Previously, we looked at how to choose which views to materialize
  - There we were optimizing what is the cost to query the cube
- Now we are materializing all elements, we're just figuring out the cheapest way to do it
  - Querying the cube will cost the same no matter what order – everything is materialized
  - The ordering of the attributes at each node in the lattice matters!

Let's revisit our student example. Consider group by {standing, age}

```
SELECT standing, age, count(*)
FROM student
GROUP BY standing, age
ORDER BY standing, age

ST                AGE        COUNT(*)
-- ---------- ----------
FR                 17               2
FR                 18               4
JR                 18               1
JR                 19               1
JR                 20               4
SO                 17               1
SO                 18               1
SO                 19               3
SR                 19               1
SR                 20               1
SR                 21               4
SR                 22               1

12 rows selected.
```

```
SELECT age, standing, count(*)
FROM student
GROUP BY age, standing
ORDER BY age, standing

 AGE         ST COUNT(*)
---------- -- ----------
17          FR         2
17          SO         1
18          FR         4
18          JR         1
18          SO         1
19          JR         1
19          SO         3
19          SR         1
20          JR         4
20          SR         1
21          SR         4
22          SR         1

12 rows selected.
```

There are two ways to compute this – {standing, age}, and {age, standing}. Both have the same # of tuples and the same information. But the ordering makes a difference on which of age or standing is easier to compute next

Let's revisit our student example. Consider group by {standing, age}

```
SELECT standing, age, count(*)
FROM student
GROUP BY standing, age
ORDER BY standing, age

ST  AGE      COUNT(*)
-- ---------- ----------
FR   17             2
FR   18             4
JR   18             1
JR   19             1
JR   20             4
SO   17             1
SO   18             1
SO   19             3
SR   19             1
SR   20             1
SR   21             4
SR   22             1

12 rows selected.
```

```
SELECT age, standing, count(*)
FROM student
GROUP BY age, standing
ORDER BY age, standing

 AGE         ST COUNT(*)
---------- -- ----------
17          FR        2
17          SO        1
18          FR        4
18          JR        1
18          SO        1
19          JR        1
19          SO        3
19          SR        1
20          JR        4
20          SR        1
21          SR        4
22          SR        1

12 rows selected.
```

If I compute {standing, age}, I can pipeline the computation of standing – I can just send the tuples on without having to resort. Super fast! But to compute the answers for standing from {age, standing} I have to resort.

Let's revisit our student example. Consider group by {standing, age}

```
SELECT standing, age, count(*)
FROM student
GROUP BY standing, age
ORDER BY standing, age

ST  AGE      COUNT(*)
-- ---------- ----------
FR    17           2
FR    18           4
JR    18           1
JR    19           1
JR    20           4
SO    17           1
SO    18           1
SO    19           3
SR    19           1
SR    20           1
SR    21           4
SR    22           1

12 rows selected.
```

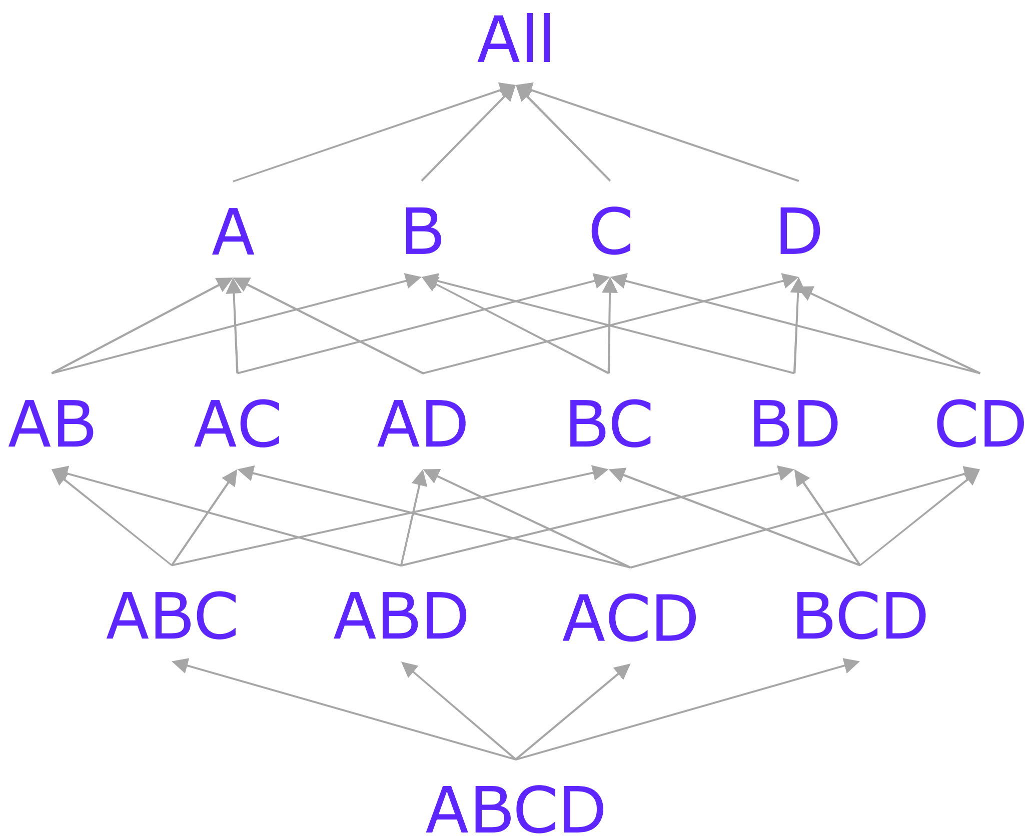
```
SELECT age, standing, count(*)
FROM student
GROUP BY age, standing
ORDER BY age, standing

 AGE        ST COUNT(*)
---------- -- ----------
17          FR        2
17          SO        1
18          FR        4
18          JR        1
18          SO        1
19          JR        1
19          SO        3
19          SR        1
20          JR        4
20          SR        1
21          SR        4
22          SR        1

12 rows selected.
```

Similarly, if I compute {age,standing}, I can pipeline the computation of age – I can just send the tuples on without having to resort. Super fast! But to compute the answers for age from {standing, age}, I have to resort.

Administrative Notes
April 6, 2021

- Upcoming due dates:
  - April 5-9: Project milestone 5
  - April 9: Tutorial 8 (SQL Server)
    - Start early; there is a lot of configuration required
    - Go to tutorial (any tutorial) if you are having issues
  - April 13: Project milestone 6
- Office hours with Jessica Thursday after class to talk about the exam. We will be using a waiting room for this office hour so that you can talk in private.
- Posted office hour schedule will end on the last day of classes. We will have a new schedule for the final exam period.

The PipeSort Algorithm [Agarwal et al., VLDB 1996]



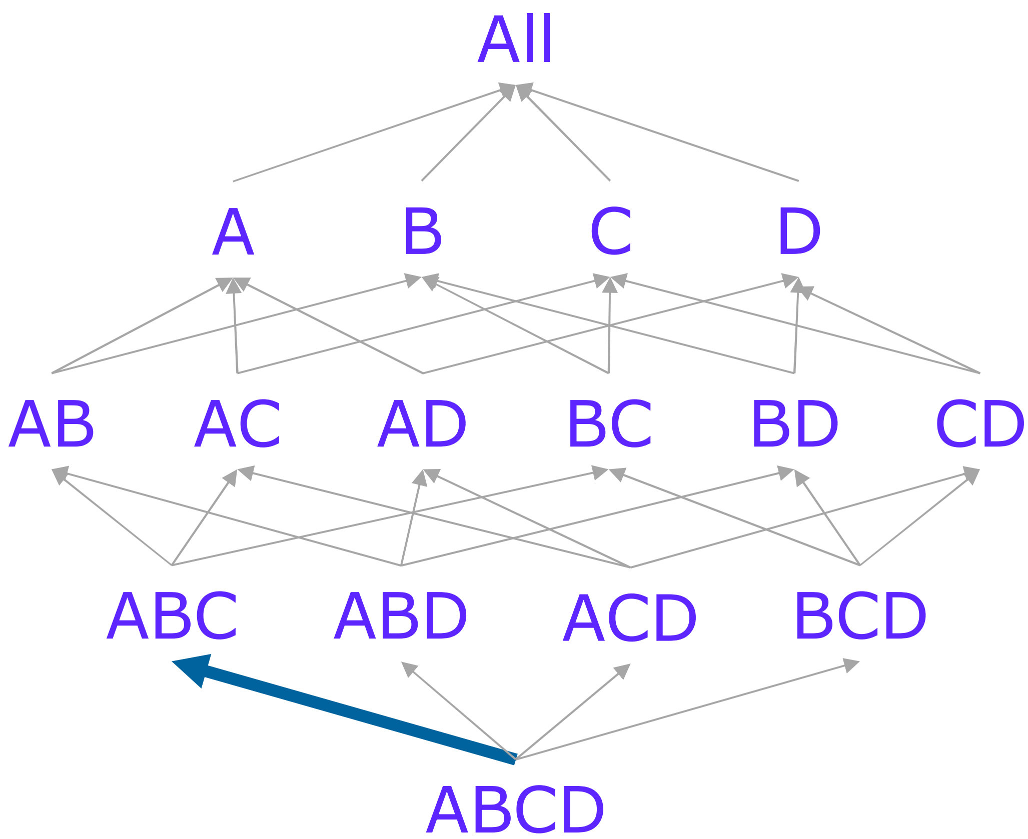
Output:

- Exactly which views to materialize in which order

Key ideas:

- Pipelining the computation is cheaper
- Ordering matters to the pipelining but NOT the cube.
- Compute each view once
- Greedy works well

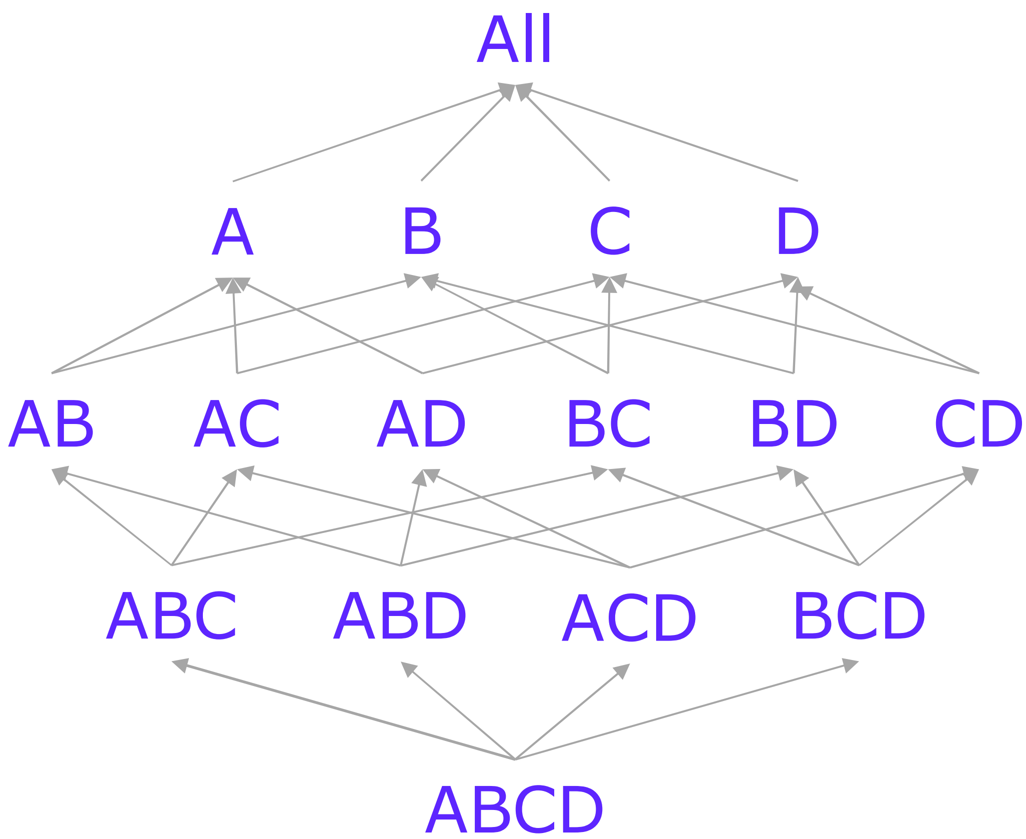
Example of sorting issues



Take the current ordering of the views

- Given that we have computed ABCD:
  - Computing ABC can be pipelined (i.e., cheap)
  - Computing ABD, ACD, and BCD cannot be pipelined (i.e., less cheap)

Pipesort



Level 0

Level 1

Level 2

Level 3

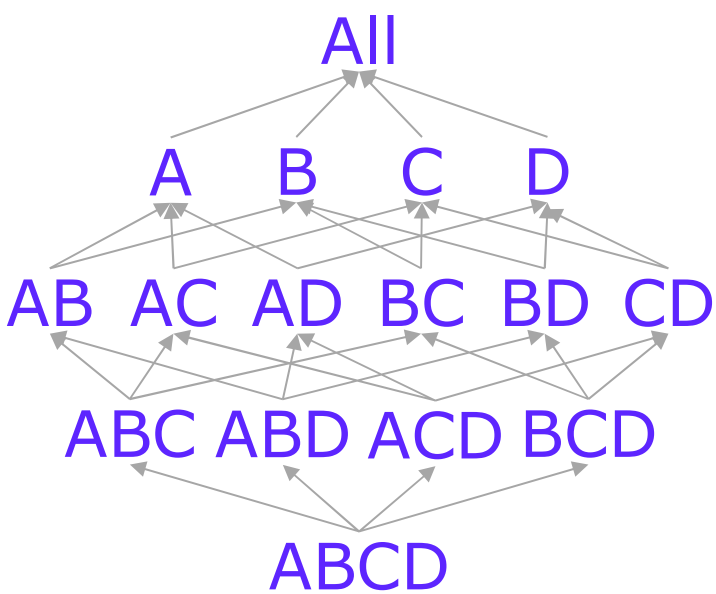
Level 4

Start from Level 0 and move down.

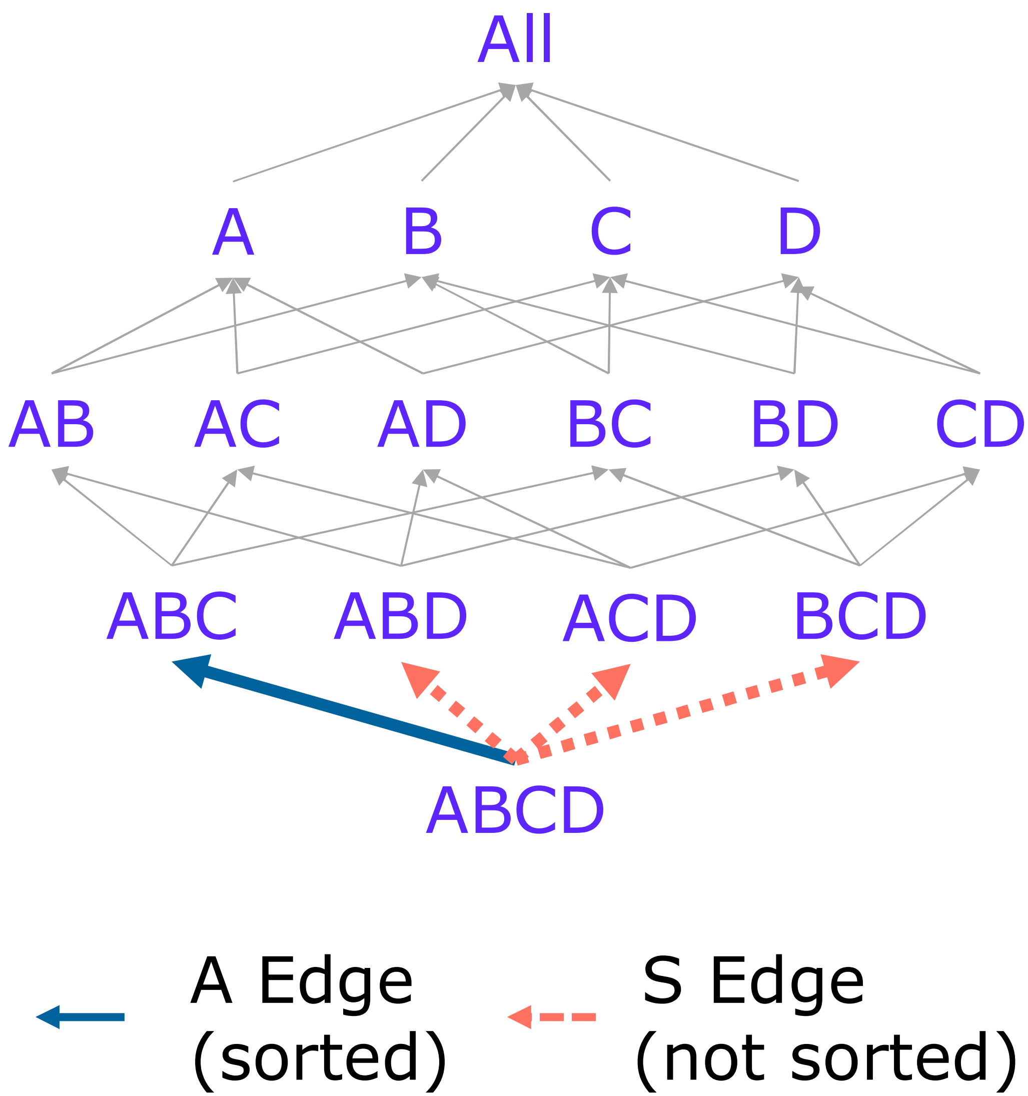
Find the best way to construct level k from level k+1.

Ordering Matters!

- ABC means the tuples are ordered by A, B, and then C
- The ordering of the view has been determined during the previous round

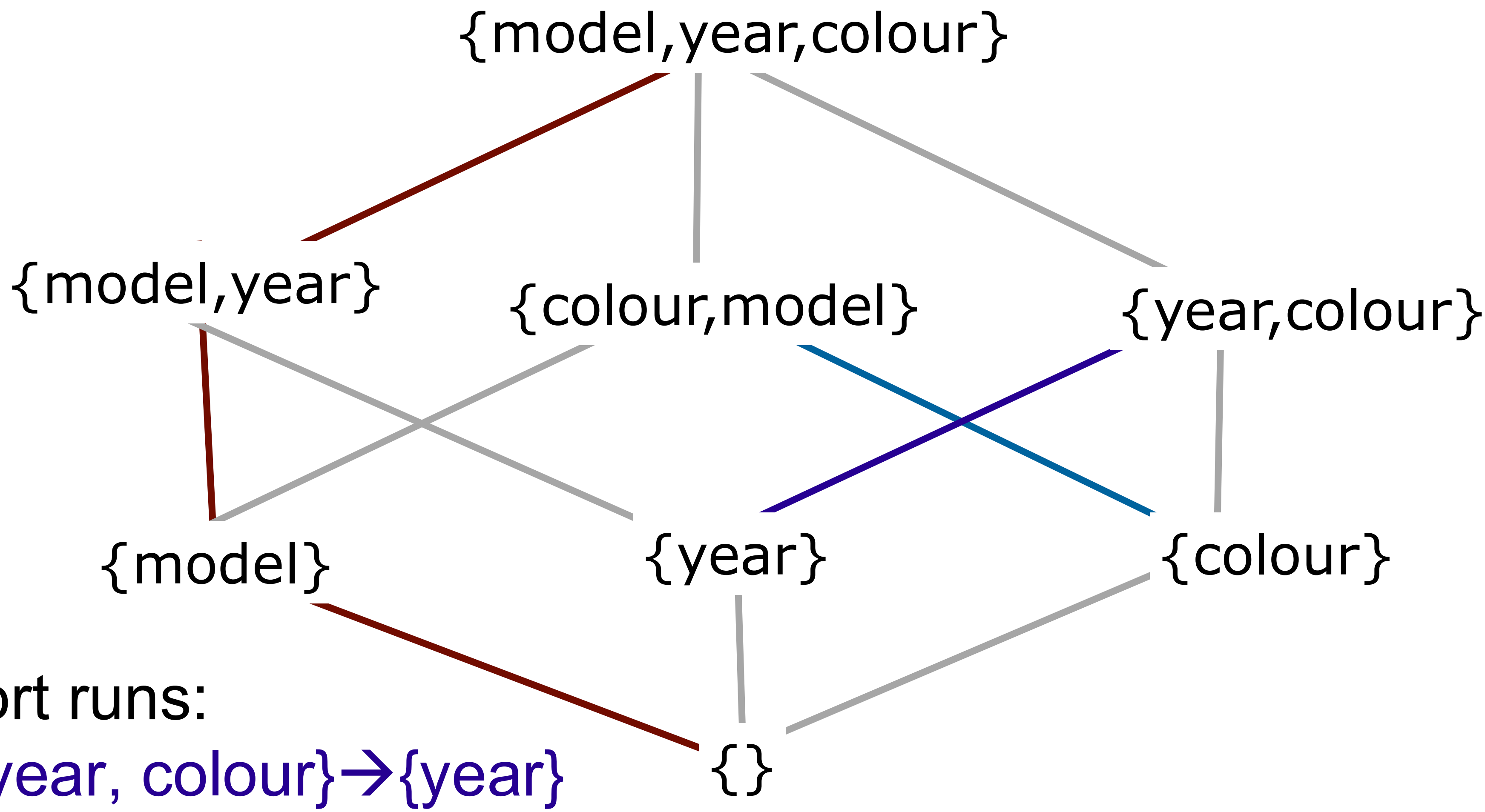


Greedy works really well



- At each level, look at the costs to compute the next level
- Use A edges if already sorted
- Use S edges if not already sorted
- Each view can be the origin of **one** A edge
- Each view can be the origin of as many S edges as necessary
- Greedy: minimize per-level cost

If orderings are fixed, you can compute sort runs without considering costs



3 sort runs:

{year, colour}→{year}

{colour,model}→{colour}

{model, year, colour}→{model,year}→ {model}→{}

Top N Queries

- For complex queries, users like to get an approximate answer quickly and keep refining their query.
- Top N Queries: If you want to find the 10 (or so) cheapest items, it would be nice if the DBMS could avoid computing the costs of *all* items before sorting to determine the 10 cheapest.
  - Idea: Guess a cost c such that the 10 cheapest items all cost less than c, and that not too many more cost less; then, add the selection "cost < c" and evaluate the query.
    - If the guess is right, great; we avoid computation for items that cost more than c.
    - If the guess is wrong, then we need to reset the selection and re-compute the query.

Top N Queries

Some DBMSs (e.g., DB2) offer special features for this.

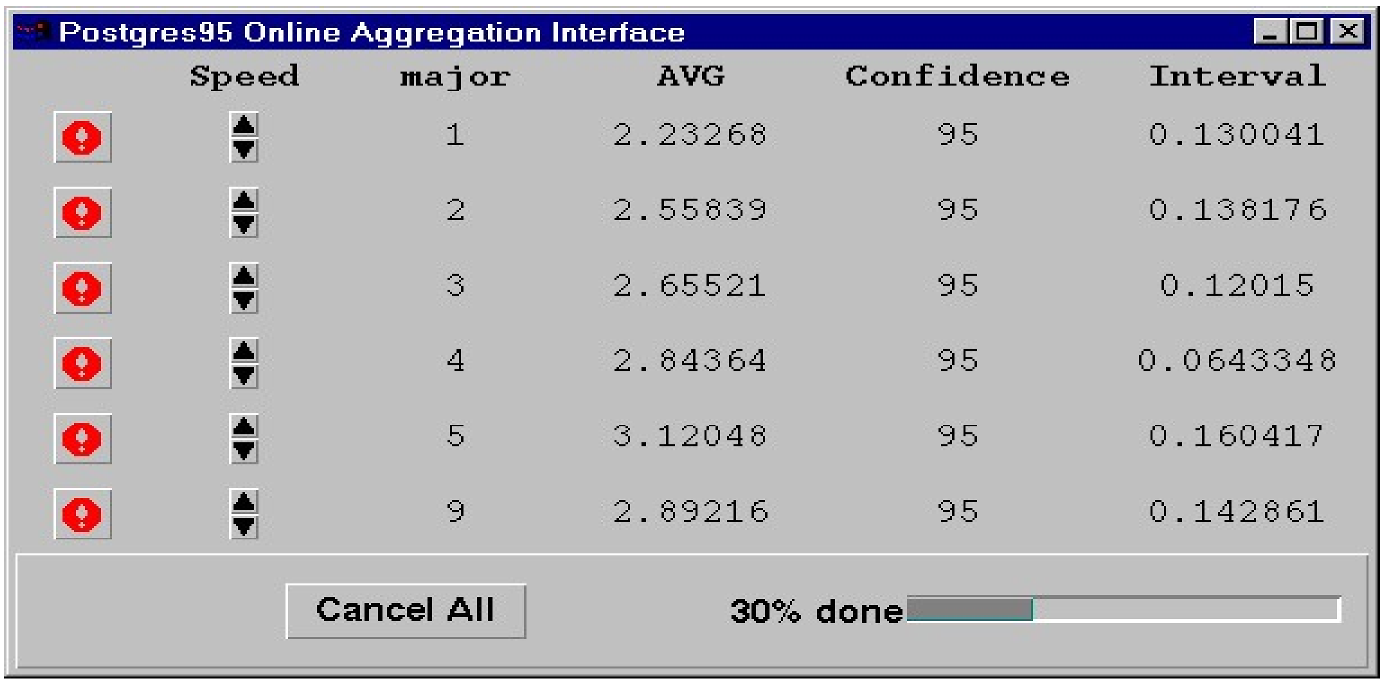
```
SELECT      P.pid, P.pname, S.sales
FROM        AllSales S, Products P
WHERE       S.pid=P.pid AND S.locid=1 AND S.timeid=3
ORDER BY S.sales DESC
OPTIMIZE FOR 10 ROWS;
```

```
SELECT      P.pid, P.pname, S.sales
FROM        AllSales 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;
```

- "OPTIMIZE FOR" construct is not in SQL:1999, but supported in DB2 or Oracle 9i
- Cut-off value c is chosen by the optimizer

Online Aggregation

- Online Aggregation: Consider an aggregate query. We can provide the user with some information before the exact average is computed.
  - e.g., Can show the current “running average” as the computation proceeds:



Speed	major	AVG	Confidence	Interval
	1	2.23268	95	0.130041
	2	2.55839	95	0.138176
	3	2.65521	95	0.12015
	4	2.84364	95	0.0643348
	5	3.12048	95	0.160417
	9	2.89216	95	0.142861

That's how to *build* an OLAP system

- But how do you use it?

Multidimensional Expressions (MDX)

- Multidimensional Expressions (MDX) is a query language for OLAP databases, much like SQL is a query language for relational databases.
- Like SQL, MDX has SELECT, FROM, and WHERE clauses (and others).
- Sample MDX syntax:

```
SELECT <content> ON COLUMNS,
       <content> ON ROWS,
       <content> ON PAGES
FROM   <name_of_cube>
WHERE
```

- You will be running MDX queries in tutorial.

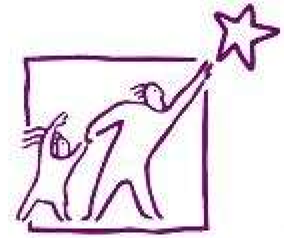
Multidimensional Expressions (MDX)

- SQL returns query results in the form of a 2-dimensional table; therefore, the SELECT clause defines the column layout. MDX, however, returns query results in the form of a *k*-dimensional sub-cube. The SELECT clause defines the *k* axes.
- The 3 default axes are named as follows:
    - Axis 0 = “columns” (or just write “axis(0)”)
    - Axis 1 = “rows”
    - Axis 2 = “pages”
  - The ON keyword specifies the axes.
    - e.g., SELECT (…) ON COLUMNS

Multidimensional Expressions (MDX)

- The queries can be simple or complex.
- MDX allows you to restrict analysis/calculations to a particular sub-cube of the overall data cube. This is useful for slice and dice operations.
  - e.g., You may wish to focus only the set of T-shirts that were sold to 18 year olds.
- MDX queries can be optimized, similar in spirit to the way SQL queries are optimized; but, the process is more complex.

Learning Goals Revisited



- Compare and contrast OLAP and OLTP processing (e.g., focus, clients, amount of data, abstraction levels, concurrency, and accuracy).
- Explain the ETL tasks (i.e., extract, transform, load) for data warehouses.
- Explain the differences between a star schema design and a snowflake design for a data warehouse, including potential tradeoffs in performance.
- Argue for the value of a data cube in terms of:
  - The type of data in the cube (numeric, categorical, temporal, counts, sums)
  - The goals of OLAP (e.g., summarization, abstractions), and
  - The operations that can be performed (drill-down, roll-up, slicing/dicing).
- Estimate the complexity of a data cube, in terms of the number of equivalent aggregation queries.