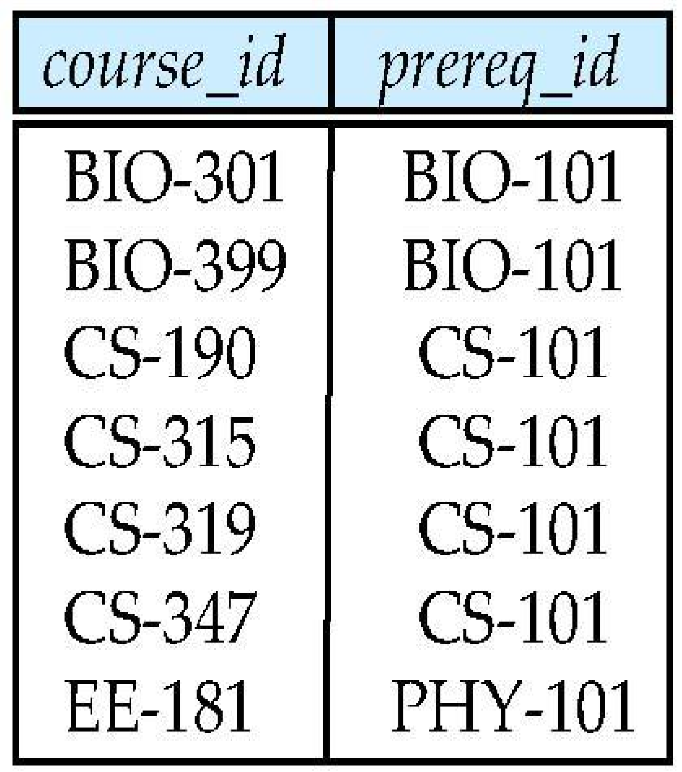
5.4 Recursive Queries

- 关系prereq的**传递闭包**(transitive closure)是一个包含**所有**（cid，pre）对的关系，pre是cid的一个*直接或间接*先修课程。
- 设**R**是集合A上的**二元关系**，R的**自反（对称、传递）闭包**是满足以下条件的关系R'：
- （i）R'是自反的（对称的、传递的）；
- （ii）$R' \supseteq R$;
- （iii）对于A上的**任何**自反（对称、传递）关系R"，若$R'' \supseteq R$，则有R"⊇R'。

5.4.1 用迭代来计算传递闭包

- 使用迭代：
  - 首先，找到CS-347的那些*直接*先修课程，
  - 然后，再找第一个集合中的所有课程的*先修课程*，如此类推。
  - 直到，某次循环中*没有新课程*加进来才停止



course_id	prereq_id
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

函数findAllPrereqs(cid)

- 这个函数以课程的course_id为参数（cid），计算该课程所有*直接或间接*的先修课程并返回它们组成的集合。
- 过程中用到了三个临时表：
- c_prereq:存储要返回的元组集合。
- new_c_prereq:存储在前一次迭代中找到的课程。
- temp:当对课程集合进行操作时用作临时存储。

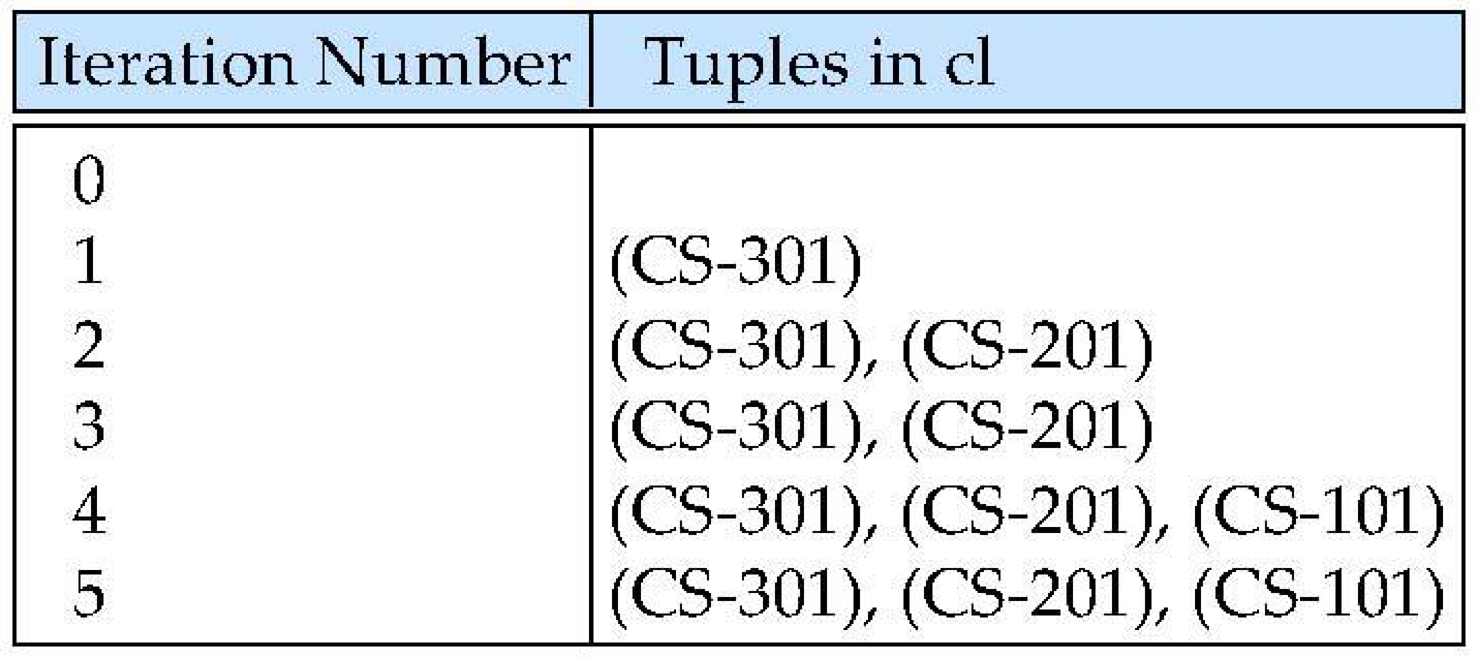
- SQL命令**create temporary table**来创建*临时表*;
- 这些表仅在执行查询的事务内部才可用，并随事务的完成而*被删除*。
- 如果findAllPrereqs(cid)的两个实例同时运行，每个实例都拥有**自己的**临时表副本;
- 假设它们**共享**一份副本，结果就会出错。
- except子句，保证即使在先修关系中存在**环**时（非正常情况），函数也能工作。

- **create function** *findAllPrereqs*(cid varchar(8))
- – – Finds all courses that are prerequisite (*directly or indirectly*) for cid
- **returns table** (course_id varchar(8))
- – – The relation prereq(course id, prereq id) specifies which course is *directly* a prerequisite for another course.
- **begin**
- **create temporary table** c_prereq (course_id varchar(8));
- – – table c prereq *stores* the set of courses to be *returned*

- **create temporary table** *new_c_prereq* (course_id varchar(8));
- – – table new_c_prereq contains courses found in the previous iteration
- **create temporary table** *temp* (course id varchar(8));
- – – table temp is used to store intermediate results
- **insert into** new_c_prereq
    - **select** prereq_id
    - **from** prereq
    - **where** course id = cid;

- **repeat**
    - insert into c_prereq
        - select course_id
        - from new_c_prereq;
    - insert into temp
        - (select prereq.prereq_id
        - from new_c_prereq, prereq
        - where new_c_prereq.course_id = prereq.course_id
        -)
        - except (

    - select course_id
    - from c_prereq
    -);
  - delete from new_c_prereq;
  - insert into new_c_prereq
    - select *
    - from temp;
  - delete from temp;
  - until not exists (select * from new_c_prereq)
  - **end repeat**;
  - **return** table c_prereq;
- **end**



Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

5.4.2 Recursion in SQL

- SQL:1999 permits **recursive** view definition
- Example: find which courses are a *prerequisite*, whether *directly* or *indirectly*, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as
(
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id =
prereq.course_id
    )
select *
from rec_prereq;
```

- This example **view**, *rec_prereq,* is called the *transitive closure* of the *prereq relation*
- 为了找到*指定*课程的先修课程，以CS-347为例，我们可以加入where子句“where rec_prereq．course_id=‘CS-347‘来修改*外层查询*。

- 用**递归**为某个指定课程，如CS-347，定义先修课程集合，方法如下。
- CS-347的（直接或间接的）先修课程是：
  - CS-347的先修课程。
  - 作为CS-347的（直接或间接的）*先修课程*的先修课程的课程。
- 注意，*第二条*是递归，因为它用CS-347的*先修课程*来定义CS-347的先修课程。

- 从SQL:1999开始，SQL标准中用**with recursive**子句来支持*有限形式*的**递归**，在递归中一个*视图*（或临时视图）用**自身**来表达自身。
- **with**子句用于定义一个临时视图，该视图的定义*只对*定义它的查询可用。
- **recursive**表示该视图是*递归的*。

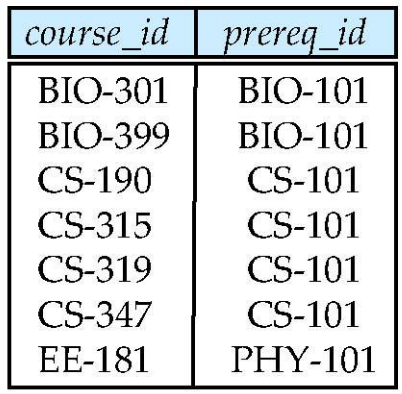
- 任何**递归视图**都必须被**定义**为*两个子查询*的**并**：
  - 一个*非递归*的基查询(base query)
  - 一个使用*递归视图*的递归查询(recursive query)。
- 持续重复*递归步骤*直至没有新的元组添加到视图关系中为止。
- 得到的视图关系实例，就称为递归视图定义的一个**不动点**（fixed point）。

The Power of Recursion

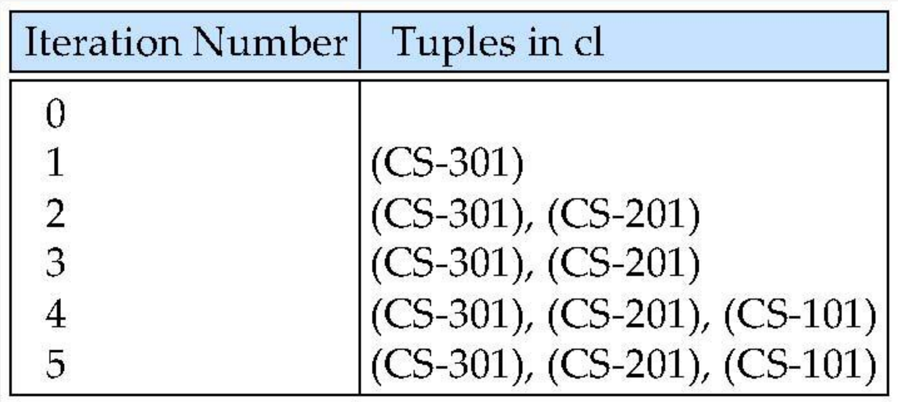
- Recursive views are required to be **monotonic**.
- That is, if we *add* tuples to *prereq* the view *rec_prereq* contains all of the tuples it contained before, plus possibly more
- that is, its result on a view relation instance V1 should be a *superset* of its result on a view relation instance V2 *if V1 is a superset of V2*.

- 特别指出，*递归查询*不能用于任何下列构造，因为它们会导致查询非单调：
  - 递归视图上的*聚集*。
  - 在使用递归视图的子查询上的not exists语句。
  - 右端使用递归视图的集合*差*(except)运算。
- SQL还允许使用**create recursive view**代替with recursive来创建*递归*定义的*永久视图*。

Example of Fixed-Point Computation



course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101



Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Advanced Aggregation Features

Ranking

- Ranking is done in conjunction with an order by specification.
- Suppose we are given a relation

  student_grades(ID, GPA)

  giving the grade-point average of each student
- Find the rank of each student.

  select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s_rank*
  from *student_grades*
- An extra **order by** clause is needed to get them in sorted order

  select *ID*, **rank**() **over** (**order by** *GPA* **desc) as** *s_rank*
  from *student_grades*
  order by *s_rank*
- Ranking may leave gaps: e.g. if 2 students have the same top GPA, both have rank 1, and the next rank is 3
  - **dense_rank** does not leave gaps, so next dense rank would be 2

Ranking

- Ranking can be done using basic SQL aggregation, but resultant query is very inefficient

```
 select ID, (1 + (select count(*)
                  from student_grades B
                  where B.GPA > A.GPA)) as s_rank
from student_grades A
order by s_rank;
```

Ranking (Cont.)

- Ranking can be done within partition of the data.
- "Find the rank of students within each department."

```
     select ID, dept_name,
          rank () over (partition by dept_name order by GPA
desc)
                    as dept_rank
    from dept_grades
    order by dept_name, dept_rank;
```

- Multiple **rank** clauses can occur in a single **select** clause.
- Ranking is done *after* applying **group by** clause/aggregation
- Can be used to find top-n results
    - More general than the **limit** *n* clause supported by many databases, since it allows top-n within each partition

Ranking (Cont.)

- Other ranking functions:
  - **percent_rank** (within partition, if partitioning is done)
  - **cume_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **row_number** (non-deterministic in presence of duplicates)
- SQL:1999 permits the user to specify **nulls first** or **nulls last**

select *ID*,
rank ( ) **over** (**order by** *GPA* **desc nulls last**) **as** *s_rank*
from *student_grades*

Ranking (Cont.)

- For a given constant *n*, the ranking the function *ntile*(*n*) takes the tuples in each partition in the specified order, and divides them into *n* buckets with equal numbers of tuples.
- E.g.,

```
select ID, ntile(4) over (order by GPA desc) as quartile
    from student_grades;
```

Windowing

- Used to smooth out random variations.
- E.g., **moving average**: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
- **Window specification** in SQL:
  - Given relation *sales(date, value)*

```
select date, sum(value) over
  (order by date between rows 1 preceding and 1 following)
from sales
```

Windowing

- Examples of other window specifications:
    - **between rows unbounded preceding and current**
    - **rows unbounded preceding**
    - **range between** 10 **preceding and current row**
        - All rows with values between current row value –10 to current value
    - **range interval** 10 **day preceding**
        - Not including current row

Windowing (Cont.)

- Can do windowing within partitions
- E.g., Given a relation *transaction* (*account_number, date_time, value*), where value is positive for a deposit and negative for a withdrawal
  - "Find total balance of each account after each transaction on the account"

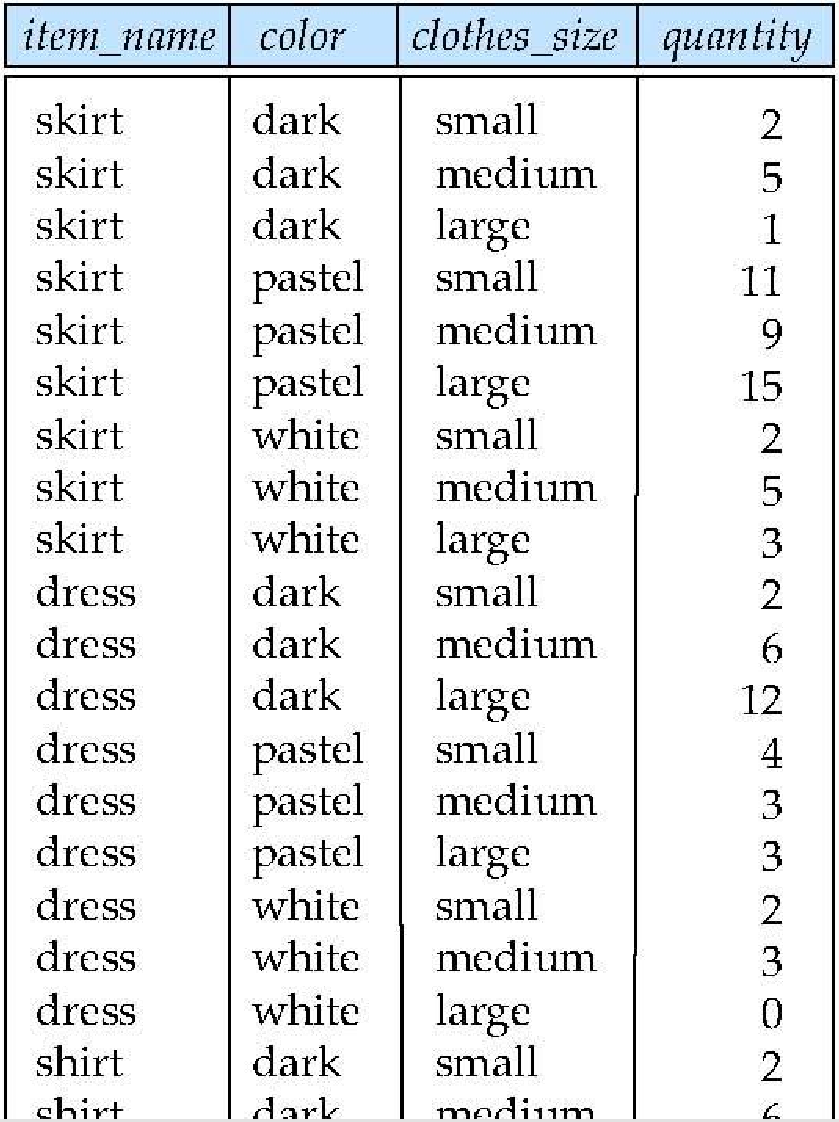
```
select account_number, date_time,
    sum (value) over
            (partition by account_number
             order by date_time
             rows unbounded preceding)
   as balance
from transaction
order by account_number, date_time
```

OLAP

Data Analysis and OLAP

- **Online Analytical Processing (OLAP)**
  - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called **multidimensional data**.
  - **Measure attributes**
    - measure some value
    - can be aggregated upon
    - e.g., the attribute *number* of the *sales* relation
  - **Dimension attributes**
    - define the dimensions on which measure attributes (or aggregates thereof) are viewed
    - e.g., attributes *item_name, color,* and *size* of the *sales* relation

Example sales relation



item_name	*color*	*clothes_size*	*quantity*
skirt	dark	small	2
skirt	dark	medium	5
skirt	dark	large	1
skirt	pastel	small	11
skirt	pastel	medium	9
skirt	pastel	large	15
skirt	white	small	2
skirt	white	medium	5
skirt	white	large	3
dress	dark	small	2
dress	dark	medium	6
dress	dark	large	12
dress	pastel	small	4
dress	pastel	medium	3
dress	pastel	large	3
dress	white	small	2
dress	white	medium	3
dress	white	large	0
shirt	dark	small	2
shirt	dark	medium	6
...	...	...	...
...	...	...	...

Cross Tabulation of *sales* by *item_name* and *color*

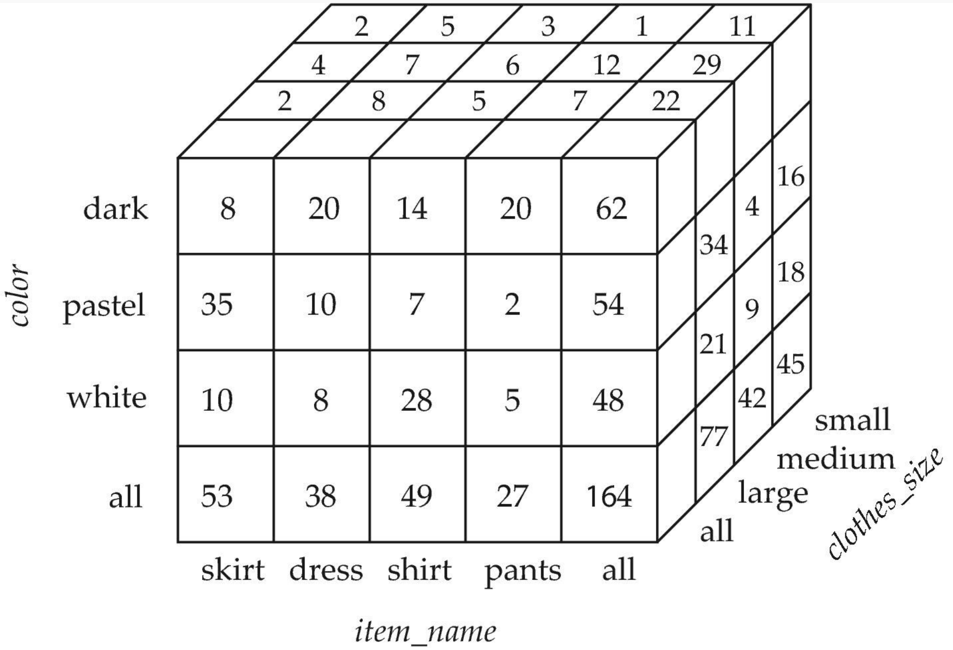
clothes_size **all**

		color			
		dark	pastel	white	total
item_name	skirt	8	35	10	53
	dress	20	10	5	35
	shirt	14	7	28	49
	pants	20	2	5	27
	total	62	54	48	164

- The table above is an example of a **cross-tabulation** (**cross-tab**), also referred to as a **pivot-table**.
  - Values for one of the dimension attributes form the row headers
  - Values for another dimension attribute form the column headers
  - Other dimension attributes are listed on top
  - Values in individual cells are (aggregates of) the values of the dimension attributes that specify the cell.

Data Cube

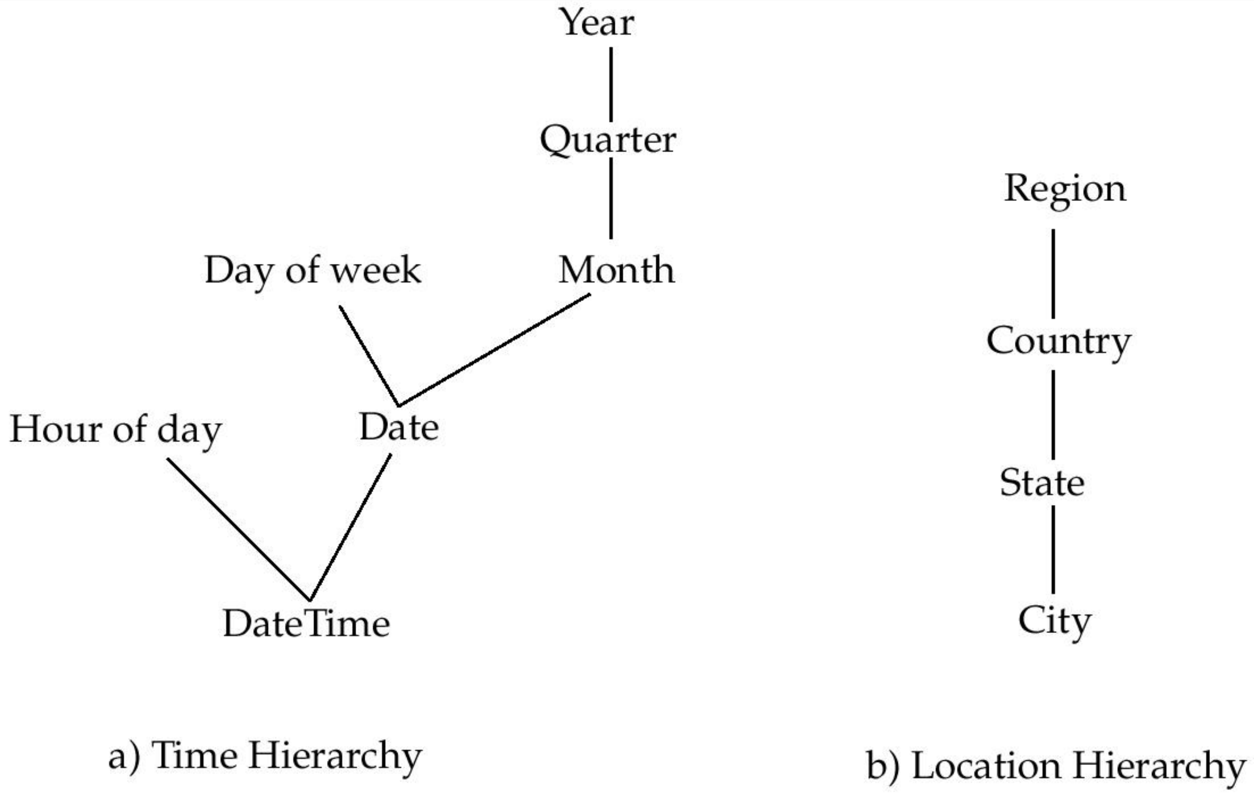
- A **data cube** is a multidimensional generalization of a cross-tab
- Can have *n* dimensions; we show 3 below
- Cross-tabs can be used as views on a data cube



	skirt	dress	shirt	pants	all
dark	8	20	14	20	62
pastel	35	10	7	2	54
white	10	8	28	5	48
all	53	38	49	27	164

Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail
  - E.g., the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year

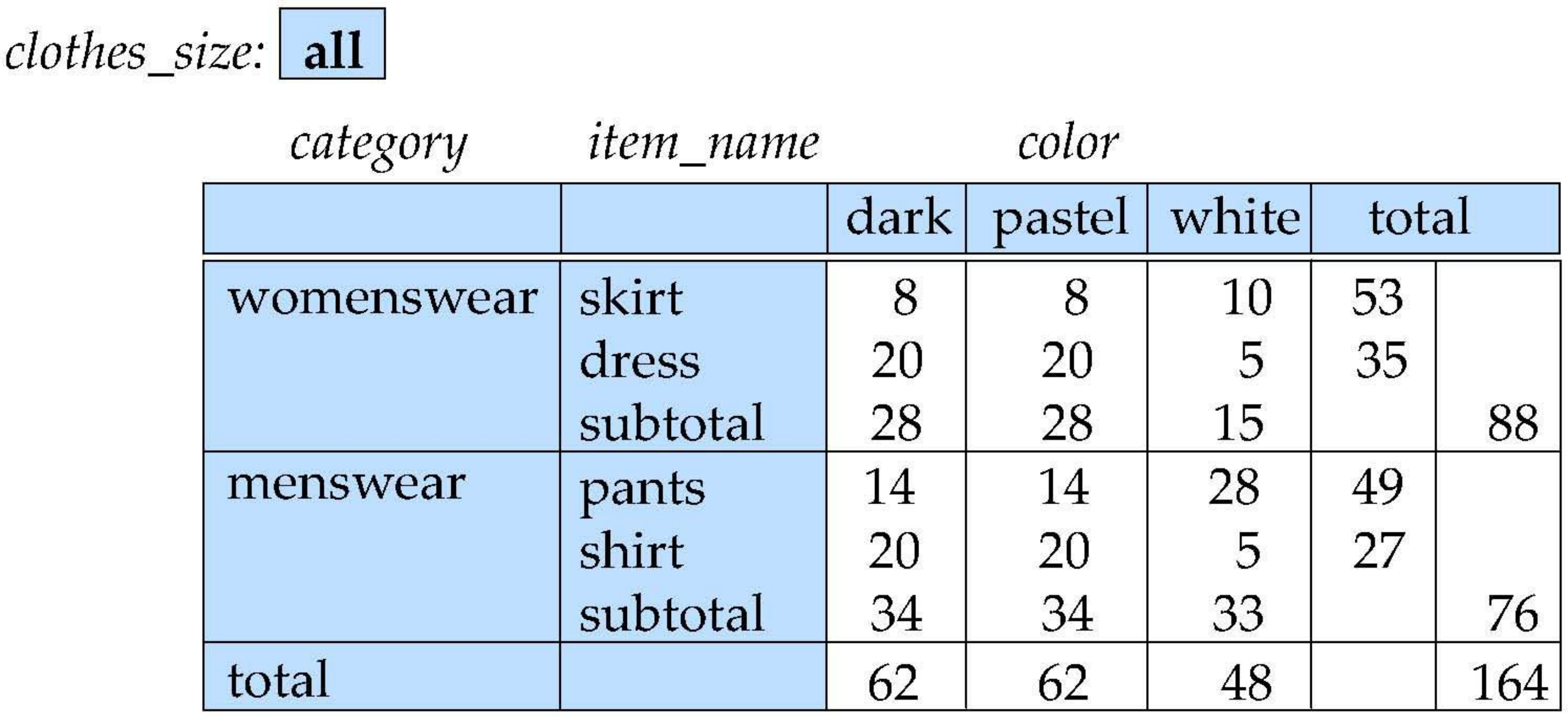


a) Time Hierarchy

b) Location Hierarchy

Cross Tabulation With Hierarchy

- Cross-tabs can be easily extended to deal with hierarchies
  - Can drill down or roll up on a hierarchy

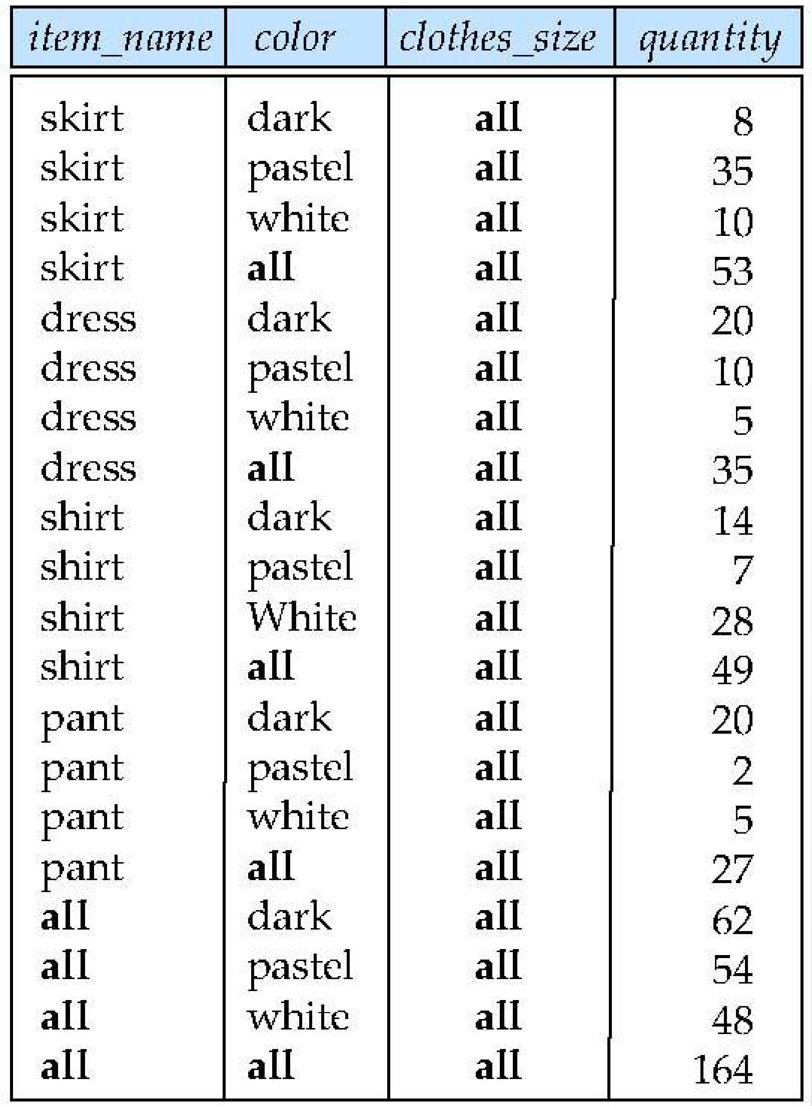


clothes_size: **all**

category	*item_name*	*color*				
		dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

Relational Representation of Cross-tabs

- Cross-tabs can be represented as relations
  - We use the value **all** is used to represent aggregates.
  - The SQL standard actually uses null values in place of **all** despite confusion with regular null values.



item_name	*color*	*clothes_size*	*quantity*
skirt	dark	**all**	8
skirt	pastel	**all**	35
skirt	white	**all**	10
skirt	**all**	**all**	53
dress	dark	**all**	20
dress	pastel	**all**	10
dress	white	**all**	5
dress	**all**	**all**	35
shirt	dark	**all**	14
shirt	pastel	**all**	7
shirt	White	**all**	28
shirt	**all**	**all**	49
pant	dark	**all**	20
pant	pastel	**all**	2
pant	white	**all**	5
pant	**all**	**all**	27
all	dark	**all**	62
all	pastel	**all**	54
all	white	**all**	48
all	**all**	**all**	164

Extended Aggregation to Support OLAP

- The **cube** operation computes union of **group by**′ s on every subset of the specified attributes
- Example relation for this section
  sales(*item_name, color, clothes_size, quantity*)
- E.g. consider the query

  select *item_name, color, size,* **sum**(*number*)
  from *sales*
  group by cube(*item_name, color, size*)

  This computes the union of eight different groupings of the *sales* relation:

  { (*item_name, color, size*), (*item_name, color*),
  (*item_name, size*), (*color, size*),
  (*item_name*), (*color*),
  (*size*), ( ) }

  where ( ) denotes an empty **group by** list.
- For each grouping, the result contains the null value for attributes not present in the grouping.

Online Analytical Processing Operations

- Relational representation of cross-tab that we saw earlier, but with *null* in place of **all**, can be computed by

```
        select item_name, color, sum(number)
        from sales
        group by cube(item_name, color)
```

- The function **grouping()** can be applied on an attribute
    - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

```
select item_name, color, size, sum(number),
        grouping(item_name) as item_name_flag,
        grouping(color) as color_flag,
        grouping(size) as size_flag,
from sales
group by cube(item_name, color, size)
```

Online Analytical Processing Operations

- Can use the function **decode()** in the **select** clause to replace such nulls by a value such as **all**
  - E.g., replace *item_name* in first query by

    decode(**grouping**(item_*name*), 1, ʻallʼ , *item_name*)

Extended Aggregation (Cont.)

- The **rollup** construct generates union on every prefix of specified list of attributes
- E.g.,

  select *item_name*, *color*, *size*, **sum**(*number*)
  from *sales*
  group by rollup(*item_name, color, size*)

  Generates union of four groupings:

  { (*item_name, color, size*), (*item_name, color*), (*item_name*), ( ) }
- Rollup can be used to generate aggregates at multiple levels of a hierarchy.
- E.g., suppose table *itemcategory*(*item_name, category*) gives the category of each item. Then

  select *category, item_name*, **sum**(*number*)
  from *sales, itemcategory*
  where *sales.item_name = itemcategory.item_name*
  group by rollup(*category, item_name*)

  would give a hierarchical summary by *item_name* and by *category.*

Extended Aggregation (Cont.)

- Multiple rollups and cubes can be used in a single group by clause
  - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

select *item_name, color, size*, **sum**(*number*)
from *sales*
group by rollup(*item_name*), **rollup**(*color, size*)

generates the groupings

{item_name, ()} X {(color, size), (color), ()}

= { (*item_name, color, size*), (*item_name, color*), (*item_name*),
(*color, size*), (*color*), ( ) }

Online Analytical Processing Operations

- **Pivoting:** changing the dimensions used in a cross-tab is called
- **Slicing:** creating a cross-tab for fixed values only
  - Sometimes called **dicing**, particularly when values for multiple dimensions are fixed.
- **Rollup:** moving from finer-granularity data to a coarser granularity
- **Drill down:** The opposite operation - that of moving from coarser-granularity data to finer-granularity data

OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as **multidimensional OLAP (MOLAP)** systems.
- OLAP implementations using only relational database features are called **relational OLAP (ROLAP)** systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called **hybrid OLAP (HOLAP)** systems.

OLAP Implementation (Cont.)

- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
  - Space and time requirements for doing so can be very high
    - 2^n combinations of **group by**
  - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
      - For all but a few "non-decomposable" aggregates such as *median*
      - is cheaper than computing it from scratch
- Several optimizations available for computing multiple aggregates
  - Can compute aggregate on (*item_name, color*) from an aggregate on (*item_name, color, size*)
  - Can compute aggregates on (*item_name, color, size*),

End of Chapter 5