


Extended Aggregation

- SQL-92 aggregation quite limited
 - 📌 Many useful aggregates are either very hard or impossible to specify
 - 📌 Data cube
 - 📌 Complex aggregates (median, variance)
 - 📌 binary aggregates (correlation, regression curves)
 - 📌 ranking queries ("assign each student a rank based on the total marks")
- SQL:1999 OLAP extensions provide a variety of aggregation functions to address above limitations
 - 📌 Supported by several databases, including Oracle and IBM DB2



Database System Concepts 4th Edition 22.1 ©Silberschatz, Korth, and Sudarshan

Extended Aggregation in SQL:1999


- The **cube** operation computes union of **group by**'s on every subset of the specified attributes
- 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.



Database System Concepts 4th Edition 22.2 ©Silberschatz, Korth, and Sudarshan


Extended Aggregation (Cont.)

- Relational representation of crosstab 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)
```
- 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, 'all', item-name)
```



Database System Concepts 4th Edition 22.3 ©Silberschatz, Korth, and Sudarshan

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*.



Database System Concepts 4th Edition 22.4 ©Silberschatz, Korth, and Sudarshan


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), ( ) }
```




Database System Concepts 4th Edition 22.5 ©Silberschatz, Korth, and Sudarshan

Ranking

- Ranking is done in conjunction with an order by specification.
- Given a relation *student-marks*(*student-id*, *marks*) find the rank of each student.


```
select student-id, rank() over (order by marks desc) as s-rank
from student-marks
```
- An extra **order by** clause is needed to get them in sorted order


```
select student-id, rank ( ) over (order by marks desc) as s-rank
from student-marks
order by s-rank
```
- Ranking may leave gaps: e.g. if 2 students have the same top mark, both have rank 1, and the next rank is 3
 - 📌 **dense_rank** does not leave gaps, so next dense rank would be 2



Database System Concepts 4th Edition 22.6 ©Silberschatz, Korth, and Sudarshan

Ranking (Cont.)

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


```
select student-id, section,
       rank ( ) over (partition by section order by marks desc)
       as sec-rank
from student-marks, student-section
where student-marks.student-id = student-section.student-id
order by section, sec-rank
```
- Multiple **rank** clauses can occur in a single **select** clause
- Ranking is done *after* applying **group by** clause/aggregation
- Exercises:
 - ☞ Find students with top n ranks
 - ☒ Many systems provide special (non-standard) syntax for "top-n" queries
 - ☞ Rank students by sum of their marks in different courses
 - ☒ given relation *student-course-marks(student-id, course, marks)*

Database System Concepts 4th Edition 22.7 ©Silberschatz, Korth and Sudarshan

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 student-id,
       rank ( ) over (order by marks desc nulls last) as s-rank
from student-marks
```

Database System Concepts 4th Edition 22.8 ©Silberschatz, Korth and Sudarshan

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 qual numbers of tuples. For instance, we can sort employees by salary, and use **ntile(3)** to find which range (bottom third, middle third, or top third) each employee is in, and compute the total salary earned by employees in each range:


```
select threentile, sum(salary)
from (
  select salary, ntile(3) over (order by salary) as threentile
  from employee) as s
group by threentile
```

Database System Concepts 4th Edition 22.9 ©Silberschatz, Korth and Sudarshan

Windowing

- E.g.: "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"
- Such *moving average* queries are used to smooth out random variations.
- In contrast to group by, the same tuple can exist in multiple windows
- **Window specification** in SQL:
 - ☞ Ordering of tuples, size of window for each tuple, aggregate function
 - ☞ E.g. given relation *sales(date, value)*

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

Database System Concepts 4th Edition 22.10 ©Silberschatz, Korth and Sudarshan

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
```

Database System Concepts 4th Edition 22.11 ©Silberschatz, Korth and Sudarshan