
# Introduction to Data Management

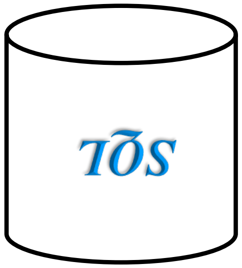
## *** The “Flipped” Edition ***

## Lecture #24

## (Advanced SQL Analytics)

Instructor: Mike Carey

mjcarey@ics.uci.edu



# Announcements

- Homework info:
  - HW #7 (*Physical DB Design*) is the 2nd to last HW
    - Due ”today”, Mon, Nov 22 (or late on Tue) at 6PM
  - HW #8 (*NoSQL and Analytics*) is the last HW!
    - Due Wed, Dec 1 (or late on Thu) at 6PM

- Thanksgiving scheduling oddities:
  - *No in-person class* on Wed Nov 24 in honor of the Thanksgiving holiday (traffic, etc.)
  - *No discussion sessions* this week, also due to the Thanksgiving holiday
  - *No in-person class* on Mon Nov 29 due to a non-reschedulable medical appt (sorry!)

# Are We There Yet?

## Topic Coverage and Exam Schedule

### Syllabus

| Topic | Reading (Required!) |
|---|---|
| Databases and DB Systems | Ch. 1 |
| Entity-Relationship (E-R) Data Model | Ch. 6.1-6.5, 6.8-6.9 |
| Relational Data Model | Ch. 2.1-2.4, 3.1-3.2 |
| E-R to Relational Translation | Ch. 6.6-6.7 |
| Relational Design Theory | Ch. 7.1-7.4.2 |
| ***Midterm Exam 1*** | ***Fri, Oct 22*** (during lecture time) |
| Relational Algebra | Ch. 2.5-2.7 |
| Relational Calculus | Wikipedia: Tuple relational calculus |
| SQL Basics (SPJ and Nested Queries) | Ch. 3.3-3.5 |
| SQL Analytics: Aggregation, Nulls, and Outer Joins | Ch. 3.6-3.9, 4.1 |
| Advanced SQL: Constraints, Triggers, Views, and Security | Ch. 4.2, 4.4-4.5, 4.7 |
| ***Midterm Exam 2*** | ***Mon, Nov 15*** (during lecture time) |
| Storage | Ch. 12.1-12.4, 12.6-12.7 |
| Indexing | Ch. 14.1-14.4, 14.5 |
| Physical DB Design | Ch. 14.6-14.7, 15.1-15.3, 15.5.3 |
| Semistructured Data Management (*a.k.a.* NoSQL) | Ch. 8.1, AsterixDB SQL++ Primer, Couchbase SQL++ Book |
| Data Science 1: Advanced SQL Analytics | Ch. 5.5, 11.3 |
| Data Science 2: Notebooks, Dataframes, and Python/Pandas | Lecture notes and Jupyter notebook |
| Basics of Transactions | Ch. 4.3, Ch. 17 |
| ***Endterm Exam*** | ***Fri, Dec 3*** (during lecture time) |

# Ranking in SQL

- ***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 the results 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 will have rank 1, and the next rank will be 3
  - **dense_rank** does not leave gaps, so next the dense rank would be 2

# Ranking

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

1 + the number of students with a higher GPA than student *A*

```
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;
```

(**Note:** This is a classic example where the query optimizer is *not* going to understand the *intent* of your query w/o more help... ☺)

# Ranking (Cont.)

- Ranking can also be done within partitions 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 **over** clauses can occur in a single **select** clause)
- (Ranking (**over**) is done *after* applying **group by** clause/aggregation)
- Can be used to find top-n results

```
select * from
    (select ID, rank() over (order by GPA desc) as s_rank
     from student_grades)
where s_rank >= 3;
```

- More general than the **limit** *n* clause supported by many databases, since it also allows top-n ***within each partition***

# Ranking (Cont.)

- PostgreSQL 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*

- Other ranking functions (which you can explore further on your own):
  - **row_number** (non-deterministic in presence of duplicates
  - **cume_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **percent_rank** (within partition, if partitioning is done)

- 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;*

... then annotate each row

organize the data ...

# Windowing in SQL

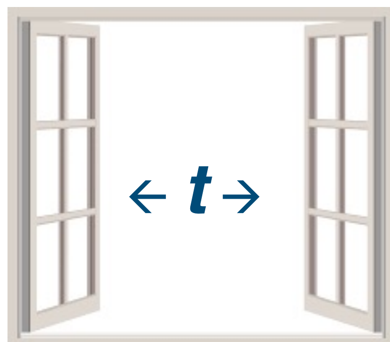
- Makes it possible to annotate tuples based on some related context
  - E.g., everything up to the current tuple, or nearby tuples, or ...
- One use case: *smooth out random variations*
  - E.g., **moving average**: "Given the sales values for each date, calculate for each date the average of the sales over that day, the previous day, and the next day"
- **Window specification** in SQL:
  - Given a relation ***daily_sales(date, value)***:

```
select date, avg(value) over
    (order by date rows between 1 preceding and 1 following) as daily
from daily_sales
```

*current row +/- 1 row*

# Windowing

- Some more examples of other window specifications:
  - **rows between unbounded preceding and current**
  - **rows unbounded preceding** (more succint)
  - **range between** 10 **preceding and current row** (see docs... ☺)
    - All rows with values between current row value-10 to current value




- Another example – running total of daily sales

  **select** *date*, ***sum***(*value*) **over**
  (**order by** *date* **rows unbounded preceding**) as total
  **from** *daily_sales*

# 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 (running total) ***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*

organize the data by accounts

start summing from the beginning

PostgreSQL

# 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
- 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 *quantity* of a *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 a *sales* relation
- Let's first look at these concepts in general – then look at modeling them in a SQL context..

# 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 Tab of *sales* by *item_name* and *color*

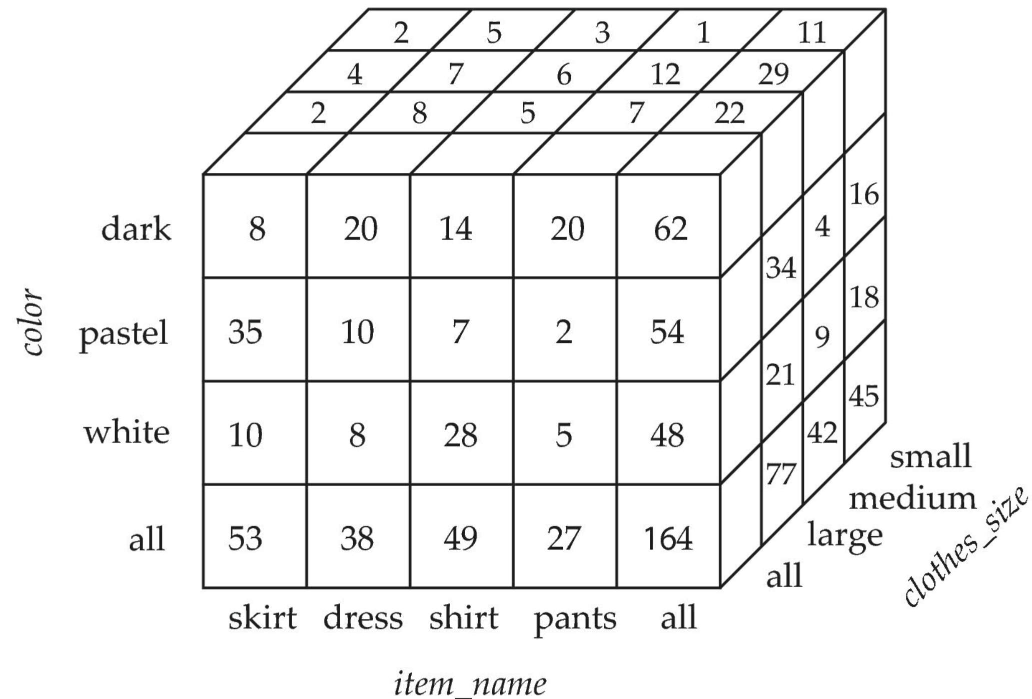
*clothes_size* **all**

| | | *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 sometimes 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 (non-tabulated) dimension attributes are listed on top
  - Values in the individual cells are (aggregates of) the values of the dimension attributes that specify the cell

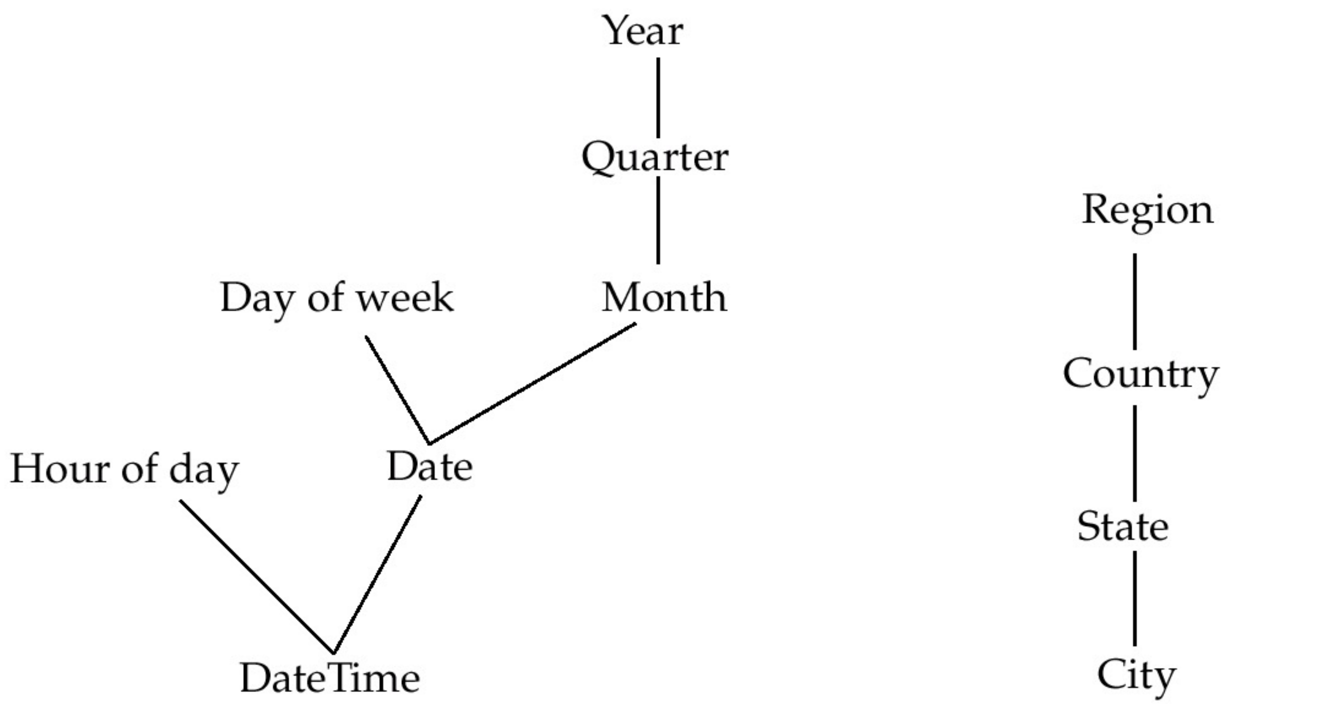
# Data Cube

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



# Hierarchies on Dimensions

- **Hierarchy** on dimension attributes: lets the dimensions 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 extended to deal with hierarchies
  - Can drill down or roll up along 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 a Cross-Tab

- Cross-tabs can be represented as relations
  - The value **all** is used here to represent aggregates
  - SQL actually uses ***null*** values instead of **all** despite confusion with regular null values (sigh!) – necessary to work with all data types

| *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 |

# OLAP Extensions to SQL Aggregation

- 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, clothes_size,* **sum**(*quantity*)
  **from** *sales*
  **group by cube**(*item_name, color, clothes_size*)
  **order by** *item_name, color,* clothes_*size*

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

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

  (called **grouping sets**)

  where ( ) denotes an empty **group by** list.
- For each grouping, the result contains null values for those attributes that are not present in the grouping (kind of an "outer union")

# Online Analytical Processing Operations

- The relational representation of the cross-tab that we saw earlier, but with *null* in place of **all** (thank you, SQL ☺) can be computed using the query:

  ```
  select item_name, color, sum(quantity)
  from sales
  group by cube(item_name, color)
  order by 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. (Helpful to identify the nulls introduced by OLAP operations.)

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

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

    coalesce(*item_name*, 'all') **AS** item_name, ...

# Extended Aggregation (Cont.)

- The **rollup** construct generates union on every ***prefix*** of a specified list of attributes – probably the most commonly used form of OLAP grouping
- E.g.,

  **select** *item_name*, *color*, clothes_*size*, **sum**(*quantity*)
  **from** *sales*
  **group by rollup**(*item_name, color, clothes_size*)

  - Generates union of ***four*** groupings:

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

- Rollup is usually used to generate aggregates at multiple levels of a hierarchy
  - E.g., suppose a second table *item_category*(*item_name, category*) records the higher-level category of each item in the sales table (e.g., clothing, electronics, hardware, entertainment ...). Then

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

  would provide a hierarchical summary by *item_name* and by *category*

# Summary

- SQL has been extended with language features to support **OLAP** – short for ***on-line analytical processing*** – right in the database server
  - Used for BI (business intelligence) / reporting
  - Used for Data Science (for data exploration)
- We've just taken a whirlwind tour of some of the key features
  - Ranking (based on some ORDER BY of the data)
  - Windowing (based on optional PARTITION BY + ORDER BY)
  - CUBE and ROLLUP (and GROUPING SETS)
- These features are part of the extended SQL standard can be found in many database systems
  - Initially for systems aimed at data warehousing use cases
  - But most RDBMSs how support them (including PostgreSQL)
  - They are also available in SQL++!  (As you will soon see... ☺)