CS2102 Lecture 6
SQL (Part 3)

Aggregate Functions

- Aggregate function computes a single value from a set of tuples
- **Example**: Find the minimum, maximum, and average prices of pizzas sold by Corleone Corner

```
select   min (price), max (price), avg (price)
from     Sells
where    rname = 'Corleone Corner'
```

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

min	max	avg
19	25	22.6666666666666667

Aggregate Functions (cont.)

Query	Meaning
select min(A) **from** R	Minimum non-null value in A
select max(A) **from** R	Maximum non-null value in A
select avg(A) **from** R	Average of non-null values in A
select sum(A) **from** R	Sum of non-null values in A
select count(A) **from** R	Count number of non-null values in A
select count(*) **from** R	Count number of rows in R
select avg(**distinct** A) **from** R	Average of distinct non-null values in A
select sum(**distinct** A) **from** R	Sum of distinct non-null values in A
select count(**distinct** A) **from** R	Count number of distinct non-null values in A

Aggregate Functions (cont.)

- Let *R* be an empty relation
- Let *S* be a relation with cardinality = n where all values of A are null values

Query	Result
select min(A) **from** R	`null`
select max(A) **from** R	`null`
select avg(A) **from** R	`null`
select sum(A) **from** R	`null`
select count(A) **from** R	0
select count(*) **from** R	0

Query	Result
select min(A) **from** S	`null`
select max(A) **from** S	`null`
select avg(A) **from** S	`null`
select sum(A) **from** S	`null`
select count(A) **from** S	0
select count(*) **from** S	*n*

Usage of Aggregate Functions

- Aggregate functions can be used in different parts of SQL queries:
  - `SELECT` clause
  - `HAVING` clause (to be discussed later)
  - `ORDER BY` clause (to be discussed later)

Usage of Aggregate Functions (cont.)

Find the number of items ordered and the maximum order cost for an item

Orders

item	price	qty
A	2.50	100
B	4.00	100
C	7.50	100

count	max
3	750.00

select **count**(∗), **max**(price ∗ qty)
from Orders;

Usage of Aggregate Functions (cont.)

Find the most expensive pizzas and the restaurants that sell them (at the most expensive price)

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	25
Pizza King	Diavola	17
Pizza King	Hawaiian	21

pizza	rname
Hawaiian	Corleone Corner
Marinara	Mamma's Place

```
select   pizza, rname
from     Sells
where    price = (select max(price) from Sells);
```

GROUP BY Clause

For each restaurant that sells some pizza, find the minimum and maximum prices of its pizzas

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma’s Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

rname	min	max
Corleone Corner	19	25
Gambino Oven	16	16
Lorenzo Tavern	23	23
Mamma’s Place	22	22
Pizza King	17	21

GROUP BY Clause (cont.)

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

rname	min	max
Corleone Corner	19	25
Gambino Oven	16	16
Lorenzo Tavern	23	23
Mamma's Place	22	22
Pizza King	17	21

Conceptual processing steps:

1. Partition the tuples in `Sells` into groups based on `rname`
2. Compute `min(price)` and `max(price)` for each group
3. Output one tuple for each group

GROUP BY Clause (cont.)

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

rname	min	max
Corleone Corner	19	25
Gambino Oven	16	16
Lorenzo Tavern	23	23
Mamma's Place	22	22
Pizza King	17	21

```
select    rname, min(price), max(price)
from      Sells
group by  rname;
```

GROUP BY Clause (cont.)

Find the number of students for each (dept,year) combination. Show the output in ascending order of (dept,year).

Students

studentId	name	year	dept
12345	Alice	1	Maths
67890	Bob	2	CS
11123	Carol	4	Maths
20135	Dave	4	CS
20135	Eve	3	CS
18763	Fred	3	Maths
60031	George	1	Maths
87012	Hugh	2	CS
96410	Ivy	4	CS

dept	year	count
CS	2	2
CS	3	1
CS	4	2
Maths	1	2
Maths	3	1
Maths	4	1

select dept, year, **count**(∗) **from** Students
group by dept, year
order by dept, year;

GROUP BY Clause (cont.)

For each restaurant that sells some pizza, find its average pizza price. Show the restaurants in descending order of their average pizza price.

```
select     rname, avg(price) as avgPrice
from       Sells
group by   rname
order by   avgPrice desc;
```

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

rname	avgPrice
Lorenzo Tavern	23.0000000000000000
Corleone Corner	22.6666666666666667
Mamma's Place	22.0000000000000000
Pizza King	19.0000000000000000
Gambino Oven	16.0000000000000000

GROUP BY Clause (cont.)

Show all restaurants in descending order of their average pizza price. Exclude restaurants that do not sell any pizza.

```
select     rname
from       Sells
group by   rname
order by   avg(price) desc;
```

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

rname
Lorenzo Tavern
Corleone Corner
Mamma's Place
Pizza King
Gambino Oven

GROUP BY Clause: Properties

- In a query with "GROUP BY $a_1, a_2, \cdots, a_n$", two tuples t & t' belong to the same group if the following expression evaluates to `true`:

$$(t.a_1 \text{ IS NOT DISTINCT FROM } t'.a_1) \text{ AND } \cdots \text{ AND }$$
$$(t.a_n \text{ IS NOT DISTINCT FROM } t'.a_n)$$

- **Example**: Four groups in R if R is grouped by $\{$A, C$\}$

R

A	B	C
null	4	19
null	21	19
6	1	null
6	20	null
20	2	10
1	1	2
1	18	2

GROUP BY Clause: Properties (cont.)

- Each output tuple corresponds to one group
- For each column *A* in relation *R* that appears in the `SELECT` clause, one of the following conditions must hold:

    1. *A* appears in the `GROUP BY` clause,
    2. *A* appears in an aggregated expression in the `SELECT` clause (e.g., **min**(A)), or
    3. the primary ~~(or a candidate)~~ key of *R* appears in the `GROUP BY` clause

Not supported in PostgreSQL

- For this module, we will follow PostgreSQL's more restrictive group-by clause properties

GROUP BY Clause: Properties (cont.)

Students

studentId	name	year	dept
12345	Alice	1	Maths
11123	Carol	4	Maths
18763	Fred	3	Maths
60031	George	1	Maths
67890	Bob	2	CS
20135	Dave	4	CS
20135	Eve	3	CS
87012	Hugh	2	CS
96410	Ivy	4	CS

This query is invalid!

select dept, year, **count**($*$)
from Students
group by dept;

GROUP BY Clause: Properties (cont.)

For each restaurant that sells some pizza, find its name, area, and the average price of its pizzas

```
select    R.rname, R.area, avg(S.price)
from      Sells S, Restaurants R
where     S.rname = R.rname
group by  R.rname;
```

```
select    R.rname, R.area, avg(S.price)
from      Sells S, Restaurants R
where     S.rname = R.rname
group by  R.rname, R.area;
```

GROUP BY Clause: Properties (cont.)

The following query is valid in standard SQL but invalid in PostgreSQL

create table R (a **integer primary key**, b **unique not null**, c **integer**);
create table S (x **integer primary key**, y **integer**, a **references** R(a));

select c, **sum**(y)
from R **natural join** S
group by b;

GROUP BY Clause: Properties (cont.)

- If an aggregate function appears in the `SELECT` clause and there is no `GROUP BY` clause, then the `SELECT` clause must not contain any column that is not in an aggregated expression
- **Example**: The following query is invalid!

```
select    rname, min(price), max(price)
from      Sells
```

HAVING Clause

Find restaurants that sell pizzas with an average selling price of at least $22

Sells

rname	pizza	price
Corleone Corner	Diavola	24
Corleone Corner	Hawaiian	25
Corleone Corner	Margherita	19
Gambino Oven	Siciliana	16
Lorenzo Tavern	Funghi	23
Mamma's Place	Marinara	22
Pizza King	Diavola	17
Pizza King	Hawaiian	21

avg(price) = 22.67

avg(price) = 19

rname
Corleone Corner
Lorenzo Tavern
Mamma's Place

select rname
from Sells
group by rname
having **avg**(price) $>=$ 22;

HAVING Clause (cont.)

Find restaurants located in the ‘East’ area that sell pizzas with an average selling price higher than the minimum selling price at Pizza King

```
select    rname
from      Sells
where     rname in (
          select rname
          from   Restaurants
          where  area = 'East'
          )
group by  rname
having    avg(price) >
          (select  min(price)
          from     Sells
          where    rname = 'Pizza King');
```

HAVING Clause: Properties

- For each column *A* in relation *R* that appears in the `HAVING` clause, one of the following conditions must hold:

  1. *A* appears in the `GROUP BY` clause,
  2. *A* appears in an aggregated expression in the `HAVING` clause, or
  3. the primary (or a candidate) key of *R* appears in the `GROUP BY` clause

Students

studentId	name	year	dept
12345	Alice	1	Maths
11123	Carol	4	Maths
18763	Fred	3	Maths
60031	George	1	Maths
67890	Bob	2	CS
20135	Dave	4	CS
20135	Eve	3	CS
87012	Hugh	2	CS
96410	Ivy	4	CS

This query is invalid!

select dept, **count**($*$)
from Students
group by dept
having year = 3;

Conceptual Evaluation of Queries

select	**distinct** select-list
from	from-list
where	where-condition
group by	groupby-list
having	having-condition
order by	orderby-list
offset	offset-specification
limit	limit-specification

1. Compute the cross-product of the tables in **from-list**
2. Select the tuples in the cross-product that evaluate to *true* for the **where-condition**
3. Partition the selected tuples into groups using the **groupby-list**
4. Select the groups that evaluate to *true* for the **having-condition** condition
5. For each selected group, generate an output tuple by selecting/computing the attributes/expressions that appear in the **select-list**
6. Remove any duplicate output tuples
7. Sort the output tuples based on the **orderby-list**
8. Remove the appropriate output tuples based on the **offset-specification** & **limit-specification**

Table Expressions

Given the following database schema, find the courses where the total number of enrolled students is higher than that for the course named “Database Systems”. Output the cname and the total number of enrolled students for each selected course.

`Courses` (<u>cid</u>, cname, credits)

`Enrolls` (<u>sid, cid</u>, grade)

Assume that `cname` is a candidate key of `Courses`.

Table Expressions (cont.)

Find the courses where the total number of enrolled students is higher than that for the course named "Database Systems". Output the cname and the total number of enrolled students for each selected course.

```
select    C.cname, count(*) as numEnroll
from      Courses C, Enrolls E
where     C.cid = E.cid
group by  C.cid
having    count(*) >
          (select  count(*)
           from    Courses C, Enrolls E
           where   C.cid = E.cid
           and     C.cname = 'Database Systems');
```

Table Expressions (cont.)

Find the courses where the total number of enrolled students is higher than that for the course named “Database Systems”. Output the cname and the total number of enrolled students for each selected course.

```
select   cname, numEnroll
from     (select   C.cid, C.cname, count(∗) as numEnroll
          from     Courses C, Enrolls E
          where    C.cid = E.cid
          group by C.cid) as X
where    numEnroll >
         (select   count(*)
          from     Courses C, Enrolls E
          where    C.cid = E.cid
          and      C.cname = ’Database Systems’);
```

Common Table Expressions (CTEs)

Find the courses where the total number of enrolled students is higher than that for the course named “Database Systems”. Output the cname and the total number of enrolled students for each selected course.

```
with CourseEnroll as
    (select    C.cid, C.cname, count(*) as numEnroll
    from       Courses C, Enrolls E
    where      C.cid = E.cid
    group by   C.cid)
select   cname, numEnroll
from     CourseEnroll
where    numEnroll >
         (select   numEnroll
         from      CourseEnroll
         where     cname = 'Database Systems');
```

Common Table Expressions (CTEs)

```
with
        R1 as (Q1),
        R2 as (Q2),
        ...,
        Rn as (Qn)
select/insert/update/delete statement S;
```

- Each `Ri` is the name of a temporary relation defined by a query `Qi`
- `S` is a SQL statement that references Rn & possibly R1,R2,$\cdots$
- CTEs can be used for writing recursive queries (not covered)

Views

- A view defines a virtual relation that can be used for querying
- **Example**: Consider the following database schema:

`Courses`	(<u>courseId</u>, cname, credits, profId, lectureTime, quota)
`Profs`	(<u>profId</u>, pname, officeRoom, contactNum)
`Students`	(<u>studentId</u>, sname, email, birthDate)
`Enrollment`	(<u>courseId</u>, numUGrad, numPGrad, numExchange, numAudit)

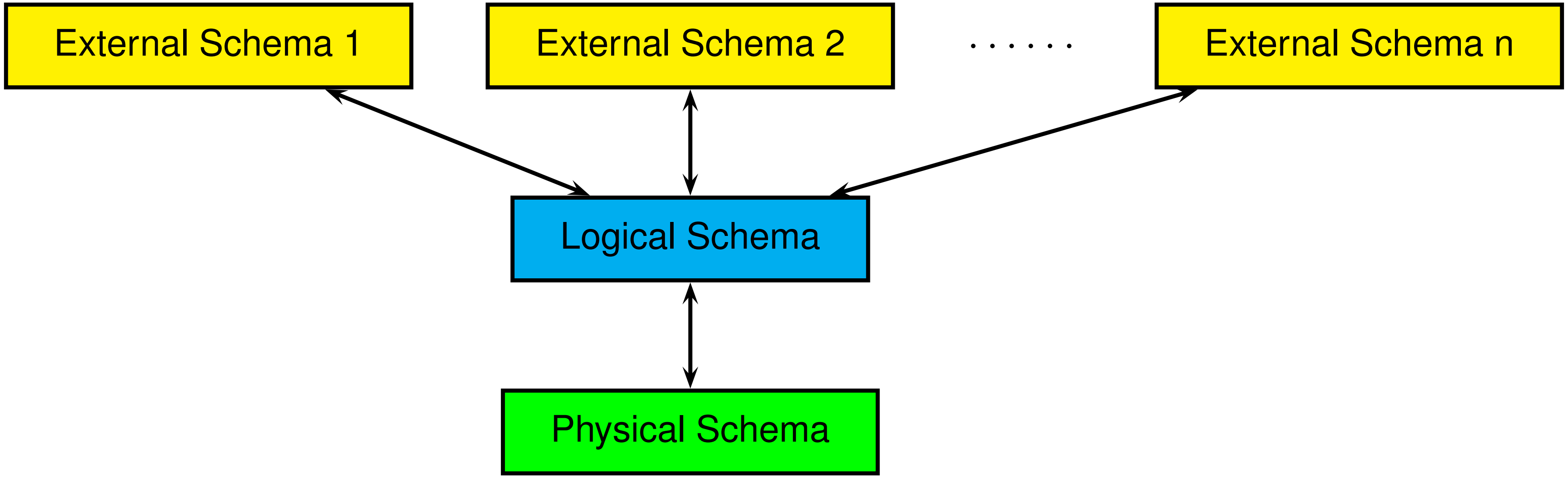
```
create view CourseInfo as
    select C.cname, P.pname, C.lectureTime,
    E.numUGrad+E.numPGrad+E.numExchange+E.numAudit as numEnrolled
    from   Courses C, Profs P, Enrollment E
    where  C.profId = P.profId
    and    C.courseId = E.courseId;
```

Views (cont.)

```
create view CourseInfo as
    select C.cname, P.pname, C.lectureTime,
    E.numUGrad+E.numPGrad+E.numExchange+E.numAudit as numEnrolled
    from   Courses C, Profs P, Enrollment E
    where  C.profId = P.profId
    and    C.courseId = E.courseId;
```

```
create view CourseInfo (cname, pname, lectureTime, numEnrolled) as
    select C.cname, P.pname, C.lectureTime,
           E.numUGrad + E.numPGrad + E.numExchange + E.numAudit
    from   Courses C, Profs P, Enrollment E
    where  C.profId = P.profId
    and    C.courseId = E.courseId;
```

Views: Providing Logical Data Independence



- Logical Schema - logical structure of data in DBMS
- Physical Schema - how the data described by logical schema is physically organized in DBMS
- External Schema - A customized view of logical schema
- Logical (Physical) Data independence: Insulate users/applications from changes to logical (physical) schema

Conditional Expressions: CASE

Scores

name	marks
Alice	92
Bob	63
Carol	58
Dave	47

name	grade
Alice	A
Bob	B
Carol	C
Dave	D

```
select name, case
        when marks >= 70 then 'A'
        when marks >= 60 then 'B'
        when marks >= 50 then 'C'
        else 'D'
end as grade
from   Scores;
```

Conditional Expressions: CASE (cont.)

case
  when $\mathit{condition}_1$ **then** result_1
  . . .
  when $\mathit{condition}_n$ **then** result_n
  else result_0
end

case `expression`
  when value_1 **then** result_1
  . . .
  when value_n **then** result_n
  else result_0
end

Conditional Expressions: COALESCE

Tests

name	first	second	third
Alice	pass	null	null
Bob	fail	pass	null
Carol	fail	fail	pass
Dave	fail	fail	fail
Eve	fail	fail	null

name	result
Alice	pass
Bob	pass
Carol	pass
Dave	fail
Eve	fail

```
select name, case
        when (first = 'pass') or (second = 'pass')
              or (third = 'pass') then 'pass'
        else 'fail'
end as result
from   Tests;
```

Conditional Expressions: COALESCE (cont.)

Tests

name	first	second	third
Alice	pass	null	null
Bob	fail	pass	null
Carol	fail	fail	pass
Dave	fail	fail	fail
Eve	fail	fail	null

name	result
Alice	pass
Bob	pass
Carol	pass
Dave	fail
Eve	fail

```
select name, coalesce(third,second,first) as result
from   Tests;
```

- **coalesce** returns the first non-null value in its arguments
- Null is returned if all arguments are null

Conditional Expressions: NULLIF

Tests

name	result
Alice	absent
Bob	fail
Carol	pass
Dave	absent
Eve	pass

name	status
Alice	null
Bob	fail
Carol	pass
Dave	null
Eve	pass

select name, **nullif**(result,'absent') as status
from Tests;

- **nullif** ($value_1$, $value_2$)
- Returns *null* if $value_1$ is equal to $value_2$; otherwise returns $value_1$

Pattern Matching with LIKE Operator

Find customer names ending with “e” that consists of at least four characters

select cname **from** Customers **where** cname **like** ’_ _ _%e’;

Customers

cname	area
Homer	West
Lisa	South
Maggie	East
Moe	Central
Ralph	Central
Willie	North

cname
Maggie
Willie

- Underscore _ matches any single character
- Percent % matches any sequence of 0 or more characters
- “string **not like** pattern” is equivalent to “**not** (string **like** pattern)”
- For more advanced regular expressions, use **similar to** operator

Queries with Universal Quantification

- **Example**: Find the names of all students who have enrolled in all the courses offered by CS department

Courses (courseId, name, dept)
Students (studentId, name, birthDate)
Enrolls (sid, cid, grade)

Queries with Universal Quantification (cont.)

- Let R denote the set of all students who have enrolled in all the courses offered by CS department
- Let $\overline{R}$ = Students − R
- $\overline{R}$ = the set of all students who have not enrolled in all the courses offered by CS department
- A student $s \in \overline{R}$ iff there exists some CS course c such that s has not enrolled in c
- Given a studentId x, let F(x) = set of courseIds of CS courses that are not enrolled by student with studentId x
- $\overline{R} = \{s \in \text{Students} \mid F(\text{s.studentId}) \neq \emptyset\}$

Queries with Universal Quantification (cont.)

- $\overline{R} = \{s \in \text{Students} \mid F(\text{s.studentId}) \neq \emptyset\}$
- $\overline{R}$ can be computed by the following pseudo SQL query:

  select s.studentId **from** Students **where exists** (F(s.studentId))

- R can be computed by the following pseudo SQL query:

  select s.studentId **from** Students **where not exists** (F(s.studentId))

Queries with Universal Quantification (cont.)

```
--F(x): set of courseIds of CS courses that are not enrolled
--by student with studentId x
select courseId
from   Courses C
where  dept = 'CS'
and    not exists (
       select 1
       from   Enrolls E
       where  E.cid = C.courseId
       and    E.sid = x
);
```

Queries with Universal Quantification (cont.)

```
--Names of students who have enrolled in all CS Courses
select name
from   Students S
where not exists (
        select courseId
        from   Courses C
        where dept = 'CS'
        and not exists (
             select 1
             from   Enrolls E
             where E.cid = C.courseId
             and   E.sid = S.studentId
        )
);
```

Summary

- Conceptual evaluation of queries

select	**distinct** select-list
from	from-list
where	where-condition
group by	groupby-list
having	having-condition
order by	orderby-list
limit	limit-specification
offset	offset-specification

- Non-scalar subqueries can be used in `FROM`, `WHERE`, and `HAVING` clauses
- Aggregate functions can be used in `SELECT`, `HAVING`, and `ORDER BY` clauses
- SQL Reference: `https://www.postgresql.org/docs/current/index.html`