<table>
<tr><th colspan="3">Subquery expressions</th></tr>
<tr><td>Scoping rules</td><td colspan="2">- Queries with subquery expressions: nested queries
- Subquery expression: inner query nested within an outer query
- Scoping rules for table alias (i.e. tuple variable):
▪ A tuple variable declared in a subquery/query Q can be used only in Q and any subquery nested in Q
▪ If a tuple variable is declared both locally in a subquery Q as well as in an outer query, the local declaration applies in Q</td></tr>
<tr><td>EXISTS subquery

(exists)</td><td colspan="2">- Returns true if the result of the subquery is non-empty; otherwise, empty – false
- Not concerned about the contents of the result</td></tr>
<tr><td></td><td><pre>select distinct cname
from Likes L
where exists (
      select 1
      from Sells S
      where S.rname = 'Corleone Corner'
      and S.pizza = L.pizza
      );

-- OR/equivalent to –

select distinct L.cname
from Likes L inner join Sells S
     on S.pizza = L.pizza
where S.rname = 'Corleone Corner' ;</pre></td><td>- select 1 is commonly used with exists
- It can be anything,
select 0, select 2 etc.
- In the example, so long some customer likes some pizza, select it
- The Sells in the inner query only applies there (scoping rules)
- The where clause returns true/false</td></tr>
<tr><td>NOT EXISTS subquery

(not exists)</td><td><pre>select cname
from Customers C
where not exists (
      select 1
      from Likes L natural join Sells S
      where S.rname = 'Corleone Corner'
      and L.cname = C.cname
      );</pre></td><td>Question:
Find distinct customers who does not like any pizza sold by ‘Corleone Corner’</td></tr>
<tr><td>IN subquery

(in)</td><td colspan="2">- Subquery must return exactly one column
- Returns false if result of subquery is empty; otherwise non-empty – returns the result of the Boolean expression:
((v = v1) or (v = v2) or … or (v = vn))
where v is the result of expression
{v1, v2, … vn} is the result of the subquery</td></tr>
<tr><td></td><td><pre>select distinct cname
from Likes
where pizza in (
      select pizza
      from Sells
      where rname = 'Corleone Corner'
      );</pre></td><td>Question:
Find distinct customers who like some pizza sold by ‘Corleone Corner’</td></tr>
<tr><td></td><td colspan="2">- Other way of using: expression IN (value1, value2, … valuen)
where values are in the same domain as the attribute</td></tr>
<tr><td></td><td><pre>select distinct pizza from Contains
where ingredient
in ('ham' , 'seafood');</pre></td><td>Question:
Find pizzas that contain ham or seafood</td></tr>
</table>

<table>
<tr><td rowspan="3">(Non-)Correlated nested queries</td><td colspan="2">- Correlated nested query: a nested query with a subquery that references a tuple variable declared in an outer query</td></tr>
<tr><td><pre>
select cname
from Likes L
where exists (
      select 1
      from Sells S
      where S.rname = 'Corleone Corner'
      and S.pizza = L.pizza
      );
</pre></td><td>Example of a correlated nested query – inner query references outside query</td></tr>
<tr><td><pre>
select distinct cname
from Likes
where pizza in (
      select pizza
      from Sells
      where rname = 'Corleone Corner'
      );
</pre></td><td>Example of a non-correlated nest query</td></tr>
<tr><td rowspan="3">ANY/SOME subquery

(any)</td><td colspan="2">- Generalisation of in subquery
- Expression operator any(subquery) where subquery must return exactly one column
- Returns false if result of subquery is empty; otherwise if non-empty, returns the result of the Boolean expression
((v op v1) or (v op v2) or … or (v op vn))
where v denotes the result of the expression
{v1, v2, … vn} denotes the result of the subquery
op denotes the operator (which can be customised)
- In the in subquery, op is =</td></tr>
<tr><td><pre>
select distinct rname
from Sells
where rname <> 'Corleone Corner'
and price > any (
    select price
    from Sells
    where rname = 'Corleone Corner'
);
</pre></td><td>Question:
Find distinct restaurants that sell some pizza P1 that is more expensive than some pizza P2 sold by “Corleone Corner”. P1 and P2 are not necessarily the same pizza. Exclude “Corleone Corner” from the query result

- The subquery references the price of any pizza sold by “Corleone Corner”
- The two price are non-correlated</td></tr>
<tr><td>ALL subquery

(all)</td><td colspan="2">- Expression operator all(subquery) where subquery must return exactly one column
- Returns true if result of subquery is empty; otherwise if non-empty, returns
((v op v1) and (v op v2) and … and (v op vn))
where v denotes the result of the expression
{v1, v2, … vn} denotes the result of the subquery
op denotes the operator</td></tr>
<tr><td><pre>
select rname, pizza, price
from Sells S1
where price >= all (
      select S2.price
      from Sells S2
      where S2.rname = S1.rname
);
</pre></td><td>Question:
For each restaurant, find the name and the price of its most expensive pizzas. Exclude restaurants that do not sell any pizza</td></tr>
</table>

UNIQUE subquery

(unique)

- Returns *false* if the result subquery contains at least two distinct records t1 and t2 such that *"t1 = t2"* evaluates to *true;* otherwise, *true*
- *"t1 = t2"* is evaluated as *"(t1.a1 = t2.a1) and … and (t1.an = t2.an)"* where {*a1 … an*} are the attributes in the schema of the subquery result
- *unique* subqueries are not widely supported (depends on the database)

```
select distinct pizza
from Sells S
where unique(
      select R.area
      from Restaurants R natural join
           Sells S2
      where S2.pizza = S.pizza
      );
```

Question:
Find distinct pizzas that are sold by at most one restaurant in each area; exclude pizzas that are not sold by any restaurant
(i.e. unique pizzas only sold by one restaurant in the area)

Scalar subqueries

- A *scalar subquery* is a subquery that returns at most one tuple with one column
- If the subquery's result is empty, its return value is *null*
- It can be used as a scalar expression (i.e. used anywhere where a single value is needed)

```
select R.rname, R.area, S.price
from Sells S, Restaurants R
where S.pizza = 'Funghi'
and S.rname = R.rname;

-- OR/equivalent to –

select rname,
       (select R.area
       from Restaurants R
       where R.rname = S.rname), price
from Sells
where pizza = 'Funghi' ;
```

Question:
For each restaurant that sells Funghi, finds its name, area and selling price

In the subquery, restaurant area is added (since every restaurant has only 1 possible area)

Non-scalar subqueries

Non-scalar subqueries can be used in different parts of SQL queries:

- *where* clause
- *from* clause
- *having* clause

Can be used in *where, from, having* clauses

Database modifications with subqueries

```
-- Table Students
create table Students (
   studentId        integer,
   name             varchar(100),
   birthdate        integer,
   year             integer,
   primary key (studentId)
);
```

```
-- Table Enrolls
create table Enrolls (
   sid         integer
               references Students,
   cid         integer
               references Courses
   grade       char(2)
   primary key (sid, cid)
);
```

```
-- Enroll all first-year students in the course 101
insert into Enrolls(sid, cid)
   select studentId, 101
   from Students
   where year = 1;
```

Aggregate functions

- Aggregate functions compute a single value from a set of tuples
- They cannot be used directly in a *where* clause
- Can be an expression involving more than one attribute

Can be used in *select, having, order by* clauses

List of aggregate functions:

Query	Meaning
`select min(A) from R`	Minimum value in A
`select max(A) from R`	Maximum value in A
`select avg(A) from R`	Average of values in A
`select sum(A) from R`	Sum of values in A
`select count(A) from R`	Count number of non-null values in A (Special case of *count(*)*)
`select count(*) from R`	Count number of rows in R (* refers to number of rows in a table – including *null* - with respect to all attributes in a set)
`select avg(distinct A) from R`	Average of distinct values in A
`select sum(distinct A) from R`	Sum of distinct values in A
`select count(distinct A) from R`	Count number of distinct non-null values in A

Ignores *null* values if present

Special case 1:
Let R be an empty relation

- *count* → 0
- *min, max, avg* → *null*

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

Special case 2:
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`	n

Usage of aggregate functions

- Can be used in different parts of SQL queries:
  - *select* clause
  - *having* clause
  - *order by* clause

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

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

- Expression *(price*qty)* is the order cost

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

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

max(price) is a scalar subquery

<table>
<tr><td>ORDER BY clause

(order by)</td><td colspan="2">Sorts rows</td></tr>
<tr><td></td><td><pre>
select *
from Restaurants natural join Sells
order by area asc, price desc;

select *
from Restaurants natural join Sells
order by area, price desc;
</pre></td><td>Question:
For each restaurant that sells some pizza, find its name, area, and the pizza it sells together with their prices. Show the output in ascending order of the area, followed by in descending order of the price</td></tr>
<tr><td>LIMIT clause

(limit)</td><td colspan="2">- Used when they are too many results
- Similar to head() in dplyr</td></tr>
<tr><td></td><td><pre>
-- Shows only the top 3 records
-- If less than 3 available, just shows what's there
select pizza, rname, price
from Sells
order by price desc
limit 3;
</pre></td><td>Question:
Find the top three most expensive pizzas. Show the pizza name, the name of the restaurant that sells it, and its selling price for each output tuple and show the output in descending order of price

** Note that if output is not order-ed by, there will be a different output every time (since the database is non-deterministic)</td></tr>
<tr><td>GROUP BY clause

(group by)</td><td colspan="2">Conceptual processing steps:
1) Partition tuples in relation into groups based on a set of attributes or an attribute
2) Computes aggregate function (if any) in select clause or where clause
3) Outputs one tuple for each group</td></tr>
<tr><td></td><td><pre>
select rname, min(price), max(price)
from Sells
group by rname;
</pre></td><td>Question:
For each restaurant that sells some pizza, find the minimum and maximum prices of its pizzas

- group by rname logically partitions rows into groups by rname
- There is no where clause because it is not a selection condition
- If there is a where clause, that is done first to filter out the needed rows (followed by group by)</td></tr>
<tr><td></td><td><pre>
-- Counting the number of students/group
select dept, year, count(*)
from Students
group by dept, year
-- To do sorting
order by dept, year;
</pre></td><td>Question:
Find the number of students for each (dept, year) combination. Show the output in ascending order of (dept, year)</td></tr>
</table>

<table>
<tr><td></td><td>

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

</td><td>Question:
For each restaurant that sells some pizza, find its average pizza price. Show the restaurants in descending order of their average pizza price</td></tr>
<tr><td>GROUP BY properties</td><td colspan="2">- In a query with "GROUP BY a1, a2, …, an", two tuples t and t' belong to the same group if the following expression evaluates to true:
(t.a1 IS NOT DISTINCT FROM t'.a1) AND …
AND (t.an IS NOT DISTINCT FROM t'.an)
- null is regarded as NOT distinct (i.e. the same)
- Each output tuple corresponds to one group (i.e. one tuple/group)
- For each column A in relation R that appears in select clause, one of the following conditions must hold/ Attributes in select clause must satisfy at least one of these conditions:
▪ (1) A appears in the group by clause
(i.e. all tuples have same value for A)
▪ (2) A appears in an aggregated expression in the select clause
(E.g. min(A))
▪ (3) The primary (or candidate) key of R appears in the group by clause
- **Output needs to be related to all of the tuples in the group</td></tr>
<tr><td></td><td>

```
-- Satisfies 3rd condition
select R.rname, R.area, avg(S.price)
from Sells S, Restaurants R
where S.rname = R.rname
-- Does not need to group by R.area
-- since it is a key of restaurant name
group by R.rname;

-- OR/equivalent to –

-- Satisfies 2nd condition
select R.rname, R.area, avg(S.price)
from Sells S, Restaurants R
where S.rname = R.rname
group by R.rname, R.area;
```

</td><td>Question:
For each restaurant that sells some pizza, find its name, area, and the average price of its pizzas</td></tr>
<tr><td></td><td>

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

</td><td>INVALID QUERY</td></tr>
<tr><td></td><td>

```
-- Calculates for all the tuples in the
table
select rname, min(price), max(price)
from Sells;
```

</td><td>INVALID QUERY – because 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 an aggregated expression</td></tr>
</table>

HAVING clause

(having)

- Eliminates groups of tuples
- 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
    (i.e. a common attribute that should be shared by all tuples in the group)
  - (2) *A* appears in an aggregated expression in the *having* clause
  - (3) The primary (or candidate) key of *R* appears in the *group by* clause

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

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

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

Question:
Find restaurants that sell pizzas with an average selling price higher than the minimum selling price at Pizza King

Example of a more complex Boolean expression

```
select dept, count(*)
from Students
group by dept
-- Ambiguous
-- Need all year = 3?
-- or at least one year = 3 is enough?
having year = 3;
```

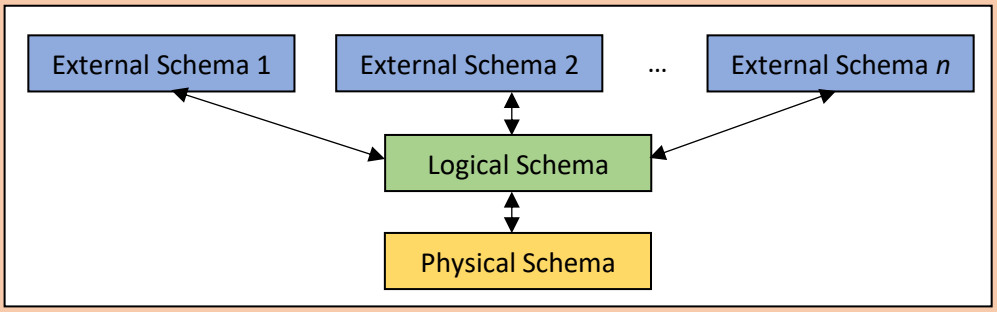
INVALID QUERY

VIEWS clause

(create view)

- A *view* defines a virtual relation that can be used for querying
- Macro definition of *view*: no records physically stored
- Can serve as a form of security mechanism: dictate what is exposed to the user

- Can provide logical data dependence
- 3 levels of abstraction:



- Logical schema – logical structure of data in DBMS (might change over time)
(Views can shield users from application changes in logical schema)
- Physical schema – how the data described by logical schema is **physically organised** in DBMS
- External schema – a customised view of logical schema
- Logical/Physical Data independence: Insulate users/application from changes to logical (physical) schema (SQL queries operate on logical schema)

```
create view CourseInfo(cname, pname, lectureTime, numStudent) as
   select C.cname, P.pname, C.lectureTime,
          (select count(*) from Enrolls E where E.cid = C.courseId)
   from Courses C natural join Profs P;
```

Conceptual evaluation of queries (** order of precedence)

Clauses

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

1. Compute the **cross-product** of the tables in from-list
   (For simplicity, assume that giant table only has table names, no *join* expression)
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 limit-specification

Queries with universal quantification

When a *for all* quantifier is needed

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

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

Tables provided

- Let R: set of all students enrolled in all the CS courses
- Let R' = Student – R
  where R': set of all students not enrolled in all CS courses
- A student ($s \in R'$) $\Leftrightarrow$ ($\exists$ some CS course *c* s.t. s is not enrolled in *c*)
- Given a *studentId x*, let F(x) = set of all courseIds of CS courses that are not enrolled by student with *studentId x*
- Thus R' = { $s \in$ Students | F(s.studentId) $\neq \emptyset$ }
- R' can be computed by the following pseudo SQL query:
  select *s.studentId* ***from*** *Students S* ***where exists*** *(F(s.studentId))*
- R can be computed by:
  select *s.studentId* ***from*** *Students* ***where not exists*** *(F(s.studentId))*

```
select coursed
from Course C
where dept = 'CS'
and not exists (
        select 1
        from Enrolls E
        where E.cid = C.courseId
        and E.sid = x);
```

```
-- F(x): set of courseIds
of CS courses that are not
enrolled by student with
studentId x
```

	select name from Student S -- Negates where not exists (select coursed from Courses C where dept = CS and not exists (select 1 from Enrolls E where E.cid = C.courseId and E.sid = S.studentId));	-- Names of students who have enrolled in all CS courses
Table expressions		
Example	Courses(cid, cname, credits) Enrolls(sid, cid, grade) -- Assume cname is a candidate key of Courses	Tables provided
	select C.cname, (select count(*) from Enrolls E where E.cid = C.cid) as numEnroll from Courses C natural join Enrolls E group by C.cid having count(*) > (select count(*) from Courses C natural join Enrolls E where C.cname = 'Database Systems'); -- OR/equivalent to – select cname, numEnroll from (select C.cid, C.cname, count(*) as numEnroll from Courses C natural join Enrolls E group by C.cid) as X where numEnroll > (select count(*) from Courses C natural join Enrolls E where C.cname = 'Database Systems');	Question: Find the courses where the total number of enrolled students is higher than that for the course name "Database Systems". Output the cname and the total number of enrolled students for each selected course - **Improved complexity:** In the 2nd attempt, the number of times of aggregation (*count(*)*) is cut from 3 to 2

Common table expressions (CTE) - **even improved complexity

- Advantage: able to refer to a table in other parts of the query
- Able to make use of already computed aggregated expressions/expressions for other queries
- Can be used to write recursive queries (independent learning)

Format

```
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* and possibly *R1*, *R2*, …

Example from above

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

CTEs are used especially when there is a subquery used in the *from* clause

Conditional expressions

CASE expression

(case…
when…then…
else…
end)

```
case
    when condition1 then result1
    ...
    when conditionn then resultn
    else result0
end
```

General form 1

```
case expression
    when value1 then result1
    ...
    when valuen then resultn
    else result0
end
```

General form 2

Example

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;
```

Others

Other conditional expressions: *coalesce* and *nullif* functions