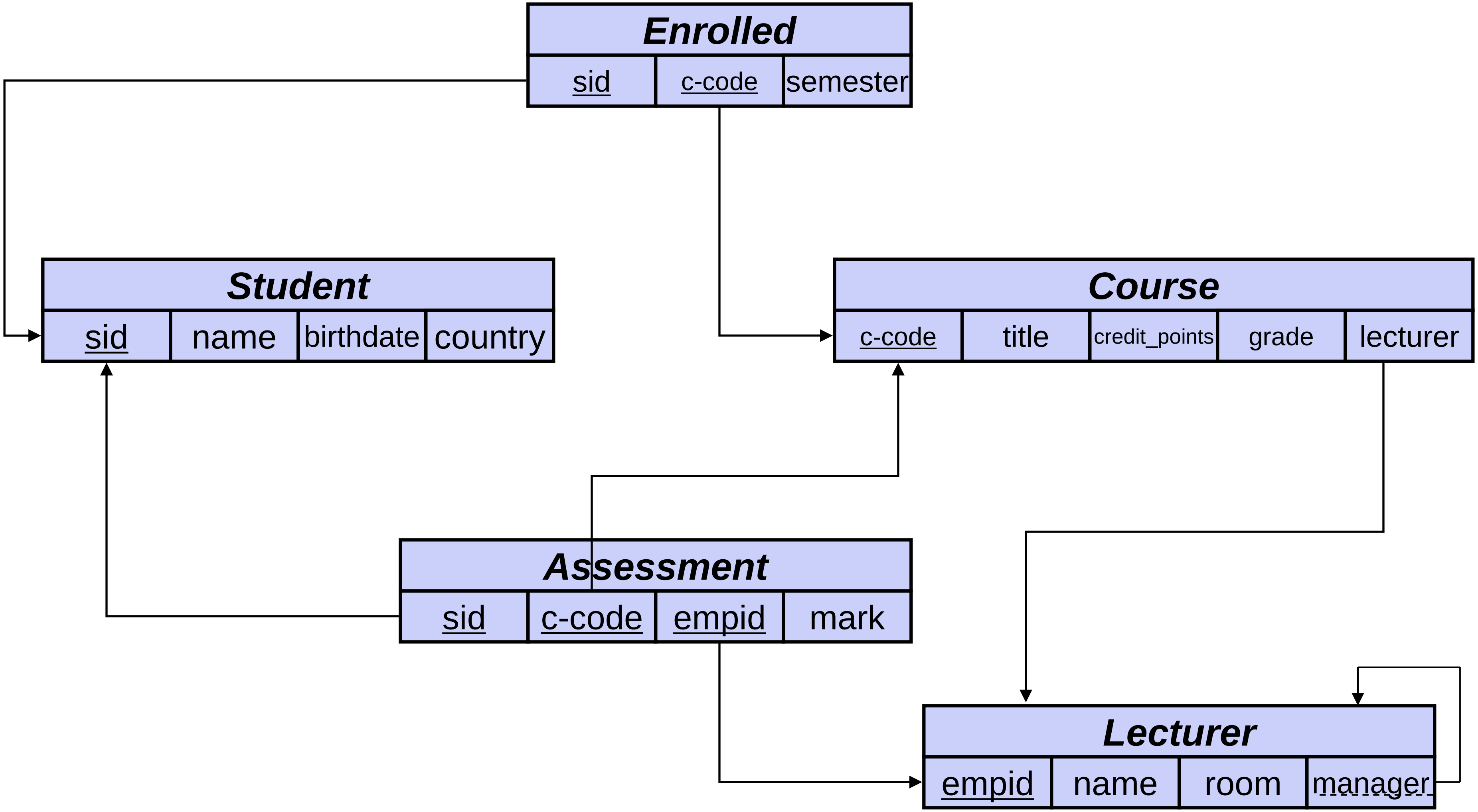
SQL Part 2

Nested Subqueries & Grouping

Peter Scheuermann

Running Example - Database Schema



Nested Subqueries

- SQL provides a mechanism for the nesting of **subqueries**.

- A **subquery** is a **select-from-where** expression that is nested within another query.
  - In a condition of the WHERE clause
  - As a "table" of the FROM clause
  - Within the HAVING clause

- A common use of subqueries is to perform tests for *set membership*, *set comparisons*, and *set cardinality*.

Example: Nested Queries

- Find the names of students who have enrolled in 'EECS495'?

The IN operator will test to see if the SID value of a row is included in the list returned from the subquery

```
SELECT name
  FROM Student
 WHERE sid IN ( SELECT sid
                  FROM Enrolled
                 WHERE c_code='EECS495)
```

Subquery is embedded in parentheses. In this case it returns a list that will be used in the WHERE clause of the outer query

- Which students have the same name as a lecturer?

```
SELECT sid, name
  FROM Student
 WHERE name IN ( SELECT name
                   FROM Lecturer )
```

Correlated vs. Non-correlated Subqueries

- **Noncorrelated subqueries**:
  - Do not depend on data from the outer query
  - Execute once for the entire outer query
- **Correlated subqueries:**
  - Make use of data from the outer query
  - Execute once for each row of the outer query
  - Can use the EXISTS operator

Processing a Noncorrelated Subquery

```
SELECT name
  FROM Student
 WHERE sid     IN  ( SELECT DISTINCT sid
                       FROM Enrolled );
```

1. The subquery executes first and returns as intermediate result all student IDs from the **`Enrolled`** table

No reference to data in outer query, so subquery executes once only

2. The outer query executes on the results of the subquery and returns the searched student names

These are only the students that have IDs in the **`Enrolled`** table

In vs. Exists Function

- The comparison operator **`IN`** compares a value *v* with a set (or multi-set) of values *V*, and evaluates to **true** if *v* is one of the elements in *V*
  - A query written with nested SELECT... FROM... WHERE... blocks and using the = or IN comparison operators can *always* be expressed as a single block query.

- **`EXISTS`** is used to check whether the result of a correlated nested query is empty (contains no tuples) or not

Correlated Nested Queries

- The inner subquery does not have to be completely independent of the outer query
  - Example:
    Find all students who have enrolled in lectures given by ‘Einstein’.

```
select distinct name
  from Student s, Enrolled e
 where s.sid = e.sid and
       exists ( select  *
                  from Lecturers, Course c
                 where  name = ‘Einstein’  and
                        lecturer = c.empid and
                c.c_code=e.c_code )
```

Subquery refers to `Enrolled`

Processing a Correlated Subquery

SID	NAME	BIRTHDATE	COUNTRY	C-CODE	SEMESTER
200300456	Henry	01-JAN-82	India	COMP5138	2005-S2
200300456	Henry	01-JAN-82	India	ELEC1007	2005-S2
200400500	Liu	04-APR-80	China	COMP5235	2005-S1
200400500	LIu	04-APR-80	China	ELEC1007	2005-S1

1. First join the **`Student`** and **`Enrolled`** tables;

2. get the ***`c_code`*** of the 1. tuple

3. Evaluate the subquery for the current ***`c_code`*** to check whether it is taught by Einstein

Subquery refers to outer-query data, so executes once for each row of outer query

C-CODE	TITLE	CPTS	LECTURER	EMPID	NAME	ROOM
COMP5138	RDBMS	6	1	1	Peter Chen	G12
INFO2120	RDBMS	6	1	1	Peter Chen	G12
ISYS3207	IS Project	4	2	2	Albert Einstein	Heaven
ELEC1007	Introduction to Physics	6	2	2	Albert Einstein	Heaven

4. If yes, include in result.

5. Loop to step (2) until whole outer query is checked.

Note: only the students that enrolled in a course taught by Albert Einstein will be included in the final results

In vs. Exists Function

- Find all students who have enrolled in lectures given by ‘Einstein’.

```
select distinct name
  from Student, Enrolled e
 where Student.sid = e.sid  and
       exists ( select *
                  from  Lecturer, Course c
                 where  name = 'Einstein'  and
                        lecturer = empid    and
                        c.c_code = e.c_code )
```

select distinct *name*

from *Student*

where *Student.sid in*

*(***select** *e.sid*

from *Enrolled e, Lecturer, Course c*

where *name = ‘Einstein’*

and *lecturer = empid*

and *c.c_code = e.c_code)*

select distinct *students.name*

from *Student, Enrolled e, Lecturer, Course c*

where *Student.sid = e.sid*

and *lecturer.name = ‘Einstein’*

and *lecturer = empid*

and *c.c_code = e.c_code*

Set Comparison

- **all** clause
  - tests whether a predicate is true for the whole set
    F <comp> all R $\Leftrightarrow \forall\, t \in R$: (F <comp> t)
- **some** clause (any)
  - tests whether some comparison holds for at least one set element
    F <comp> some R $\Leftrightarrow \exists\, t \in R$: (F <comp> t)
- (**not**) **exists** clause
  - tests whether a set is (not) empty (R $\Leftrightarrow$ R $\neq \emptyset$) (R $\Leftrightarrow$ R = $\emptyset$)
- **unique** clause
  - tests whether a subquery has any duplicate tuples in its result
- where
  - <comp> can be: $<, \leq, >, \geq, =, \neq$
  - F is a fixed value or an attribute
  - R is a relation

Examples: Set Comparison

- Find the students with highest grades in EECS213

```
SELECT S.sid
  FROM Student S
 WHERE S.grade >= ALL ( SELECT grade
                         FROM Enrolled
                        WHERE c_code='EECS213')
```

- Find students which enrolled in just one course.

```
SELECT sid, name
  FROM Student
 WHERE unique(SELECT *
                FROM Enrolled
               WHERE Enrolled.sid = Student.sid)
```

Examples: Set Comparison (cont'd)

- Search predicates of the form "for all" or " for every" can be formulated using the **not exists** clause
  - Example:
    Find courses where all enrolled student already have a grade.

```
SELECT c_code
  FROM Course C
 WHERE NOT EXISTS
      ( SELECT *
          FROM Enrolled E,
         WHERE E.c_code=C.c_code
               and grade is null )
```

Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several *groups* of tuples.
- Example: Find company and total amount of sales

Sales Table

company	amount
IBM	5500
DELL	4500
IBM	6500

```
SELECT Company, SUM(Amount)
  FROM Sales
```

company	amount
IBM	16500
DELL	16500
IBM	16500

```
SELECT Company, SUM(Amount)
  FROM Sales
 GROUP BY Company
```

company	amount
IBM	12000
DELL	4500

Queries with GROUP BY and HAVING

- In SQL, we can "partition" a relation into *groups* according to the value(s) of one or more attributes:

```
   SELECT   [DISTINCT]   target-list
     FROM   relation-list
    WHERE   qualification
 GROUP BY   grouping-list
   HAVING   group-qualification
```

- A *group* is a set of tuples that have the same value for all attributes in *grouping-list*.
- Note: Attributes in **select** clause outside of aggregate functions must appear in the *grouping-list*
  - Intuitively, each answer tuple corresponds to a *group,* and these attributes must have a single value per group.

Group By Overview

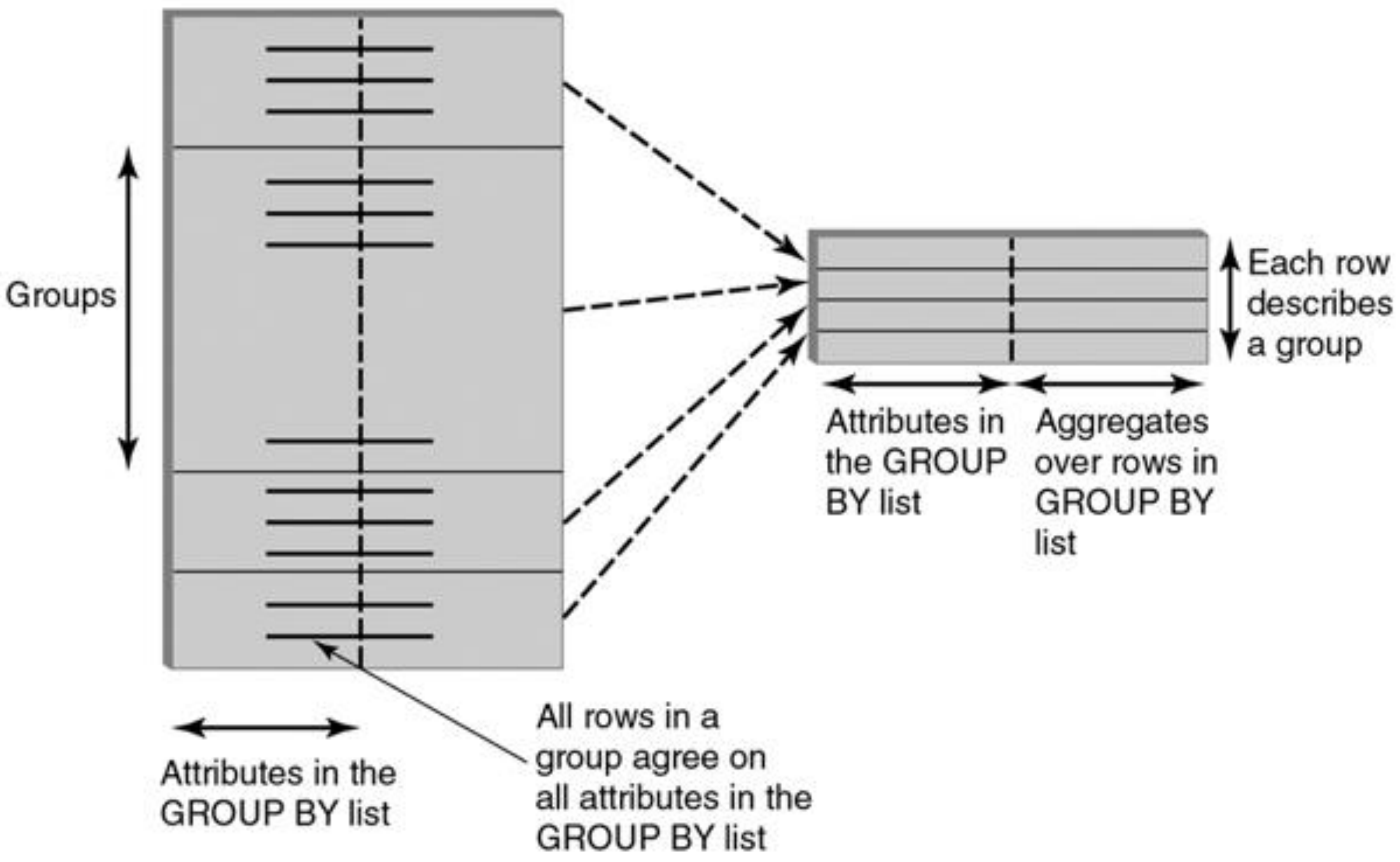


FIGURE 5.9 Effect of the GROUP BY clause.

Example:
Filtering Groups with HAVING Clause

- GROUP BY Example:
  - What was the average grade of each course?

```
    SELECT c_code as unit_of_study, AVG(grade)
      FROM Enrolled
  GROUP BY c_code
```

- HAVING clause: can further filter groups to fulfil a predicate
  - Example:

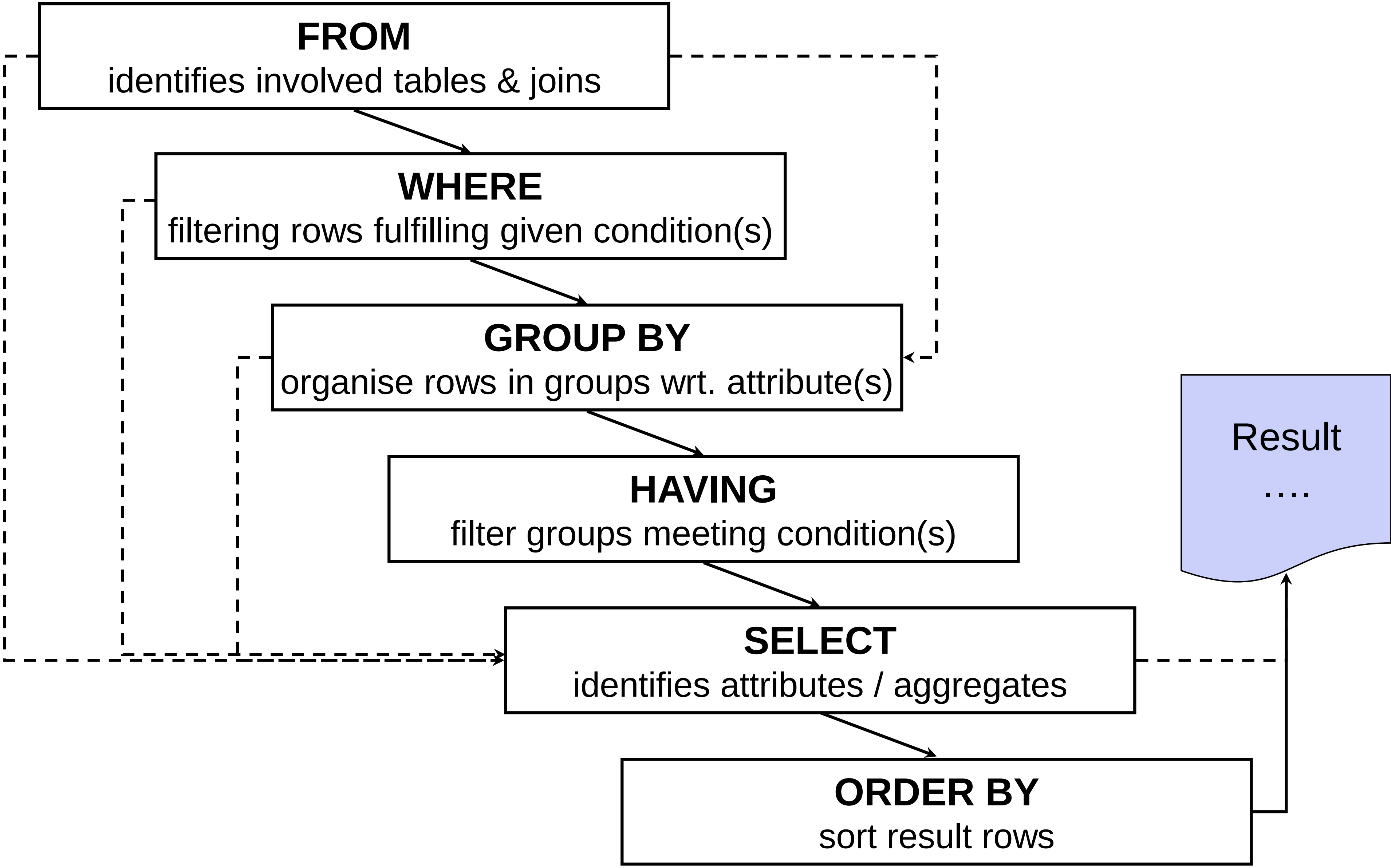
```
    SELECT c_code as unit_of_study, AVG(grade)
      FROM Enrolled
  GROUP BY c_code
    HAVING AVG(grade) > 85
```

  - Note: Predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Examples of invalid queries

- SELECT ~~SID~~, age FROM Student GROUP BY age;
  - Recall there is one output row per group
  - There can be multiple SID values per group
- SELECT ~~SID~~, MAX(GPA) FROM Student;
  - Recall there is only one group for an aggregate query with no GROUP BY clause
  - There can be multiple SID values
  - Wishful thinking (that the output SID value is the one associated with the highest GPA) does NOT work
  - ☞ Another way of writing the max GPA query?

Query-Clause Evaluation Order



Evaluation Example

- Find the average grades of 3-credit point courses with at least 2 students registered

```
SELECT c_code as unit_of_study, AVG(grade)
FROM Enrollment NATURAL JOIN Course
WHERE credit_points >= 3
GROUP BY c_code
HAVING COUNT(*) > 2
```

1. Enrollment and Course are joined

c_code	sid	emp_id	grade	title	cpts.	lecturer
COMP513	1001	10500	60	RDBMS	3	10500
COMP513	1002	10500	55	RDBMS	3	10500
COMP513	1003	10500	78	RDBMS	3	10500
COMP316	1004	10500	93	RDBMS	3	10500
~~ISYS327~~	~~1002~~	~~10500~~	~~67~~	~~IS Project~~	~~1~~	~~10500~~
~~ISYS327~~	~~1004~~	~~10505~~	~~80~~	~~IS Project~~	~~2~~	~~10505~~
~~SOFT300~~	~~1001~~	~~10505~~	~~56~~	~~C Prog.~~	~~2~~	~~10505~~
INFO212	1005	10500	63	DBS 1	4	10500
…	…	…	….	…	…	…

2. Tuples that fail the WHERE condition are discarded

Evaluation Example (cont'd)

3. Remaining tuples are partitioned into groups by the value of attributes in the grouping-list.

c_code	sid	emp_id	grade	title	cpts.	lecturer
COMP513	1001	10500	60	RDBMS	3	10500
COMP513	1002	10500	55	RDBMS	3	10500
COMP513	1003	10500	78	RDBMS	3	10500
COMP316	1004	10500	93	RDBMS	3	10500
INFO5990	1001	10505	67	IT Practice	4	10505
…	…	…	….	…	…	…

4. Groups which fail the HAVING condition are discarded.

5. ONE answer tuple is generated per group

c_code	AVG(..)
COMP5133	61
INFO5990	82

Question: What happens if we have NULL values in grouping attributes?

SQL set and bag operations

- UNION, EXCEPT, INTERSECT
  - Set semantics
    - Duplicates in input tables, if any, are first eliminated
  - Exactly like set [, ¡, and \ in relational algebra
- UNION ALL, EXCEPT ALL, INTERSECT ALL
  - Bag semantics
  - Think of each row as having an implicit count (the number of times it appears in the table)
  - Bag union: sum up the counts from two tables
  - Bag difference: proper-subtract the two counts
  - Bag intersection: take the minimum of the two counts

Examples of bag operations

Bag1

fruit
apple
apple
orange e

Bag2

fruit
apple
orange
orange

Bag1 UNION ALL Bag2

fruit
apple
apple
orange
apple
orange
orange

Bag1 EXCEPT ALL Bag2

fruit
apple

Bag1 INTERSECT ALL Bag2

fruit
apple
orange

Expressiveness and Limitations of SQL

- SQL is relational complete
  - SQL has more expressiveness than relational algebra (due to, e.g., arithmetic expressions, aggregate functions, GROUP BY and HAVING clauses)

- SQL is not "Turing complete"
  - Not everything, which is computable, can be expressed using SQL
  - Examples:
    - Variance of grades in enrolments ?
    - Given a database with direct flights, calculate all possible flight connections between two cities?
      - => SQL-92 does not support recursion
  - "SQL is neither structured, nor a language" (anonymous)

Examples of set versus bag operations

- *Enroll(SID, CID), ClubMember(club, SID)*

  - (SELECT SID FROM ClubMember)
    EXCEPT
    (SELECT SID FROM Enroll);
    - SID's of students who are in clubs but not taking any classes

  - (SELECT SID FROM ClubMember)
    EXCEPT ALL
    (SELECT SID FROM Enroll);
    - SID's of students who are in more clubs than classes

Recursion in SQL:1999

- SQL:1999 permits recursive view definition
- E.g. query to find all flight-connections:

```
with recursive connections (start,dest ) as
(
        select departure, destination
        from   flights
              union
        select f1.start, f2.destination
        from connections f1, flights f2
        where f1.dest = f2.departure )
select *
  from connections
```

You should now be able to…

- …**formulate even complex SQL Queries**
  - Including multiple joins with correct join conditions
  - correlated and noncorrrelated subqueries
  - Grouping and Having conditions

- …transform SQL queries between different forms
  - E.g.
    - correlated queries and join queries
    - Implicit and explicit natural join queries

- …know the principle expressiveness of SQL
  - and how it relates to the relational algebra