# More aspects of SQL

ISYS2120 Data and Information Management

Prof Alan Fekete
University of Sydney

Acknowledge: slides from Uwe Roehm and Alan Fekete, and from the materials associated with reference books (c) McGraw-Hill, Pearson

# Agenda

- *GROUP BY*
- Relational Algebra Operators
- NULL
- Nested subqueries
- “For every” queries

# Grouped aggregates

- A very common pattern in data analysis is to collect the information for each value of some combination of attributes, and report on an aggregate of summary for each case
  - In spreadsheets, this can be done with a pivot table
- Eg “Find the average sales *in each store*”, “*for each department*, give the number of employees”, ”*for each product and month*, show the number of items sold”

# Group-aggregates

Hypothetical biology dataset

| | Genus | Species | Region | | Weight |
|---|---|---|---|---|---|
| | | | | | |
| | Rattus | rattus | AUS | ABC | 216.5 |
| | Felis | catus | AUS | ABC | 3510 |
| | Rattus | rattus | USA | ABC | 249.5 |
| | Rattus | norvegicus | AUS | XYZ | 143.0 |
| | Mus | musculus | AUS | ABC | 85.3 |
| | Felis | catus | USA | XYZ | 3974 |

| Genus | Region | Avg(Weight) |
|---|---|---|
| Rattus | AUS | 179.75 |
| Rattus | USA | 249.5 |
| Felis | AUS | 3510 |
| Felis | USA | 3974 |
| Mus | AUS | 85.3 |

"*For each genus and region*, what is average weight of the corresponding Observations"

# 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:

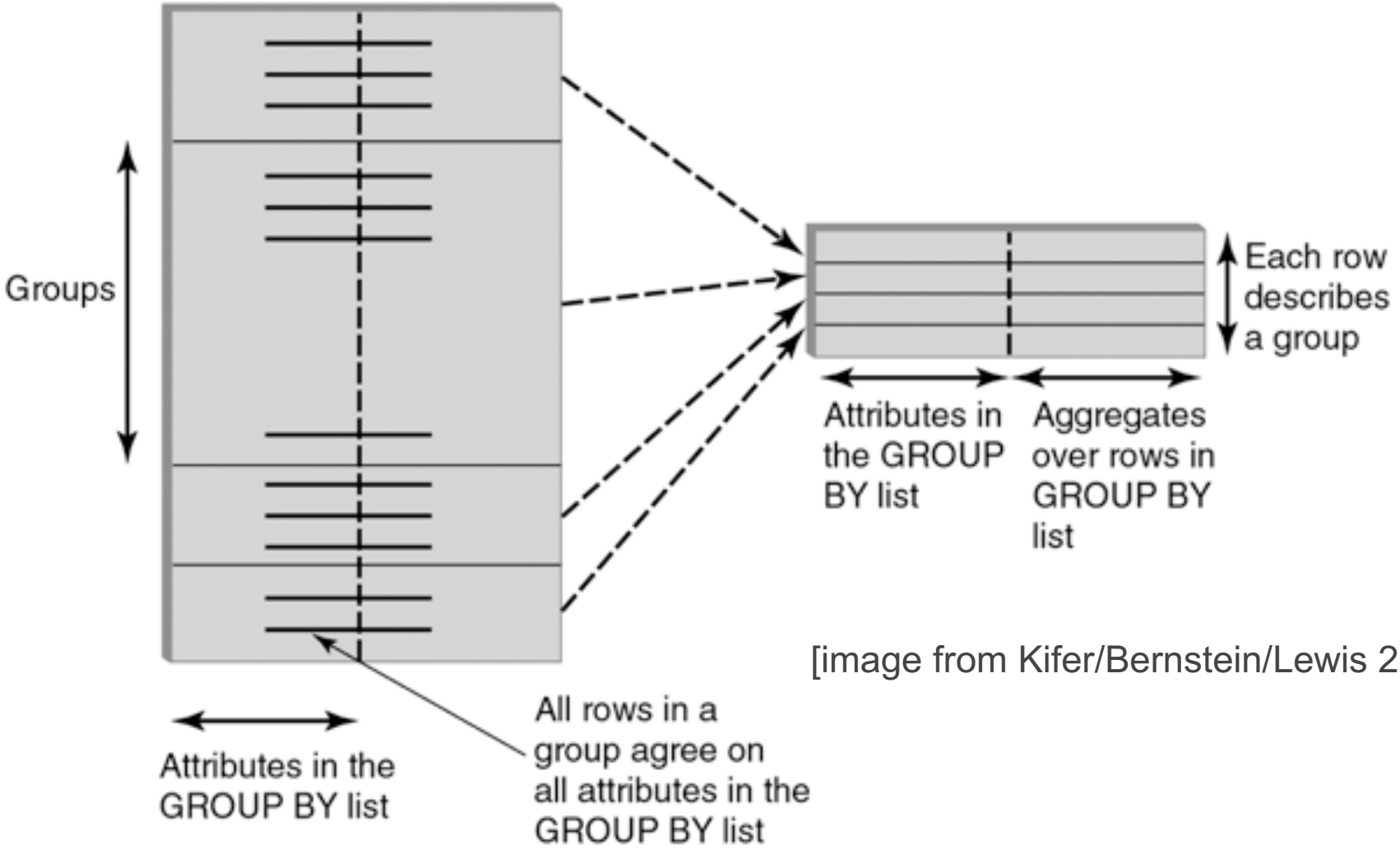
```
SELECT   [DISTINCT]   target-list
  FROM   relation-list
 WHERE   qualification
GROUP BY grouping-list
HAVING   group-qualification
```

- A *group* is a set of tuples where they have identical values, considering just the attributes in *grouping-list*.

# Warnings

- Note: Any attribute in **select** clause that is not within some aggregate function, must appear in the *grouping-list*
  - Intuitively, each answer tuple corresponds to a *group,* and these attributes must have a single value per group.
- Note: it is a common mistake to forget to show the grouping aggregate(s) in the SELECT clause
  - The reader won't be able to interprete the output: how would they know which group the aggregate is for?

# Group By Overview




[image from Kifer/Bernstein/Lewis 2006]

**FIGURE 5.9** Effect of the GROUP BY clause.

# Filtering Groups (HAVING clause)

- GROUP BY Example:
  - What was the average mark of each unit?

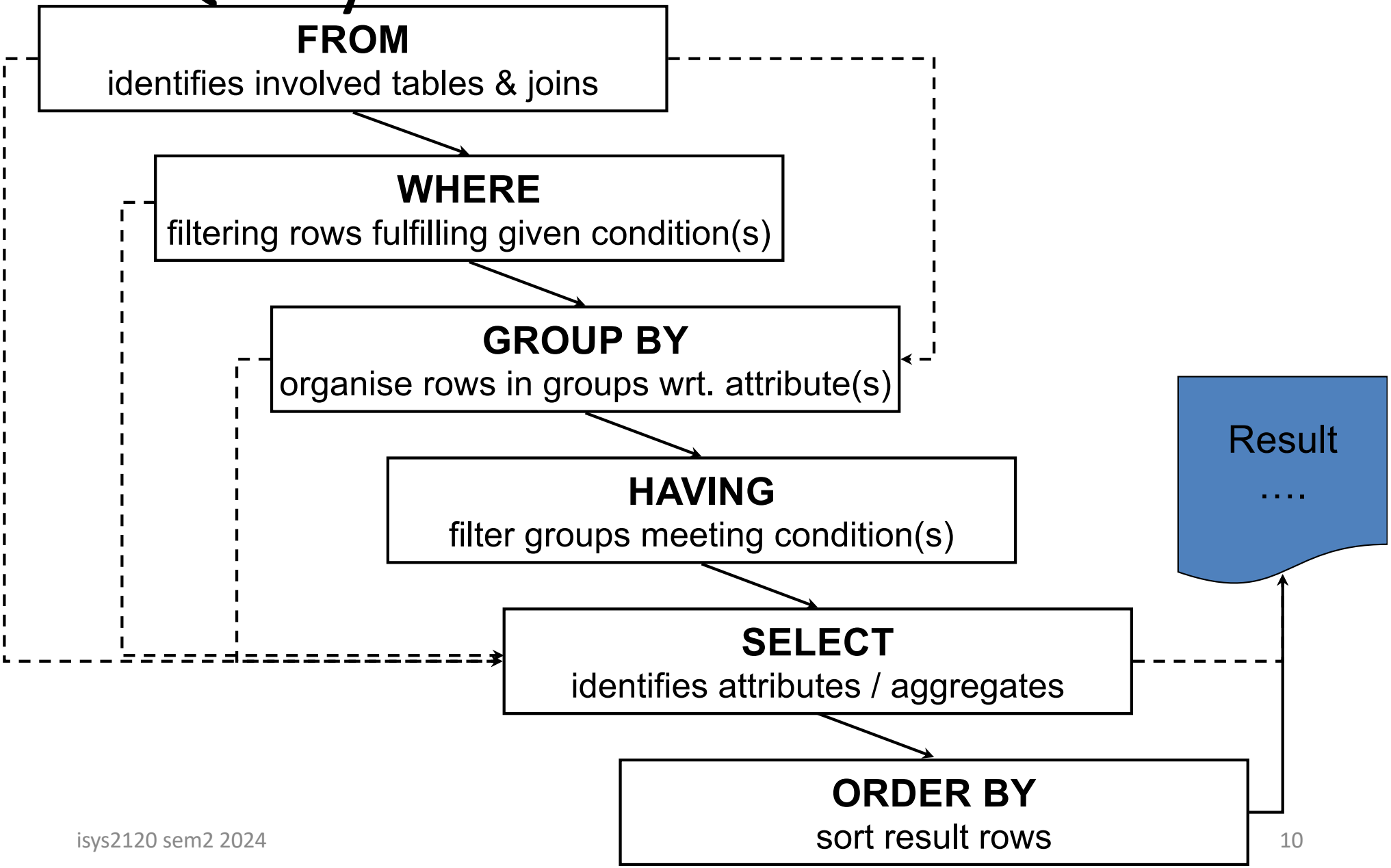
```
SELECT uos_code as unit_of_study, AVG(mark)
  FROM Assessment
GROUP BY uos_code
```

- HAVING clause: can further filter groups to fulfil a predicate
  - Example: what is average mark in each unit where that average is more than 10

```
SELECT uos_code as unit_of_study, AVG(mark)
  FROM Assessment
GROUP BY uos_code
  HAVING AVG(mark) > 10
```

  - Note: Predicates in the **having** clause are applied after the formation of groups (and must be meaningful for a group as a whole, ie either a grouping attribute or an aggregate) whereas predicates in the **where** clause are applied to individual rows, before forming groups

# Query-Clause Evaluation Order



# Evaluation Example

- Find the average marks of 6-credit point courses with at least 2 results

```
  SELECT uos_code as unit_of_study, AVG(mark)
    FROM Assessment NATURAL JOIN UnitOfStudy
   WHERE credit_points = 6
GROUP BY uos_code
  HAVING COUNT(*) >= 2
```

1. Assessment and UnitOfStudy are joined

| uos_code | sid | emp_id | mark | title | cpts. | lecturer |
|---|---|---|---|---|---|---|
| COMP5138 | 1001 | 10500 | 60 | RDBMS | 6 | 10500 |
| COMP5138 | 1002 | 10500 | 55 | RDBMS | 6 | 10500 |
| COMP5138 | 1003 | 10500 | 78 | RDBMS | 6 | 10500 |
| COMP5138 | 1004 | 10500 | 93 | RDBMS | 6 | 10500 |
| ~~ISYS3207~~ | ~~1002~~ | ~~10500~~ | ~~67~~ | ~~IS Project~~ | ~~4~~ | ~~10500~~ |
| ~~ISYS3207~~ | ~~1004~~ | ~~10505~~ | ~~80~~ | ~~IS Project~~ | ~~4~~ | ~~10505~~ |
| SOFT3000 | 1001 | 10505 | 56 | C Prog. | 6 | 10505 |
| ~~INFO2120~~ | ~~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.

| uos_code | sid | emp_id | mark | title | cpts. | lecturer |
|---|---|---|---|---|---|---|
| COMP5138 | 1001 | 10500 | 60 | RDBMS | 6 | 10500 |
| COMP5138 | 1002 | 10500 | 55 | RDBMS | 6 | 10500 |
| COMP5138 | 1003 | 10500 | 78 | RDBMS | 6 | 10500 |
| COMP5138 | 1004 | 10500 | 93 | RDBMS | 6 | 10500 |
| ~~SOFT3000~~ | ~~1001~~ | ~~10505~~ | ~~56~~ | ~~C Prog.~~ | ~~6~~ | ~~10505~~ |
| INFO5990 | 1001 | 10505 | 67 | IT Practice | 6 | 10505 |
| … | … | … | …. | … | … | … |

4. Groups which fail the HAVING condition are discarded.

5. ONE answer tuple is generated per group

| uos_code | AVG(..) |
|---|---|
| COMP5138 | 56 |
| INFO5990 | 40.5 |

What happens if we have NULL values in grouping attributes?
See later this lecture

# Grouped query in FROM clause

- Find the average mark of assessments of those unit of studies where at least 10 students have been assessed.

```
select unit_of_study, avg_mark
from
    (select number AS unit_of_study,
            avg (mark) AS avg_mark,
            count(*) AS studentcount
     from assessment
     group by number ) RESULT
where studentcount > 10
```

- Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *result* in the **from** clause, and the attributes of *result* can be used directly in the **where** clause.

# Agenda

- GROUP BY
- *Relational Algebra Operators*
- NULL
- Nested subqueries
- “For every” queries

# Recall: SQL Join Operators

- SQL offers join operators to directly do joining without putting a WHERE clause to express the match-up of values.
  - R **natural join** S
    - Put together rows, one from each table, in which the same-named columns have same values (each same-named attribute appears once only)
  - R **inner join** S **on** <join condition>
  - R **inner join** S **using** (<list of attributes>)
  - Note that the keyword **inner** can be left out, it is the default **join**; however either **on** or **using** are needed with inner join
- These additional operations are typically used as expressions in the from clause
  - List all students and in which courses they enrolled.

```
select name, uos_code, semester
  from Student natural join Enrolled
```

  - Who is teaching "ISYS2120"?

```
select name
  from UnitOfStudy join Academic on lecturer=empid
 where uos_code='ISYS2120'
```

# Set Operations

- The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
  - Some platforms (eg Oracle) use MINUS instead of EXCEPT
  - Some platforms do not support all these
- The set operation can be performed at top level on results of queries, or in FROM clause
- As in RA, the operands need to have same structure (column names in same order, with same data types)

# Set Operations and Duplicates

- Each of the set operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

  Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:
  - $m + n$ times in *r* **union all** *s*
  - $\min(m,n)$ times in *r* **intersect all** *s*
  - $\max(0, m - n)$ times in *r* **except all** *s*

# Examples of Set Operations

- Find all customers who have an account (ie they deposit), a loan (ie they borrow), or both:
  - **(select** *customer_name* **from** *depositor*)
    **union**
    **(select** *customer_name* **from** *borrower)*

- Find all customers who have both an account and a loan
  - (**select** *customer_name* **from** *depositor*)
    **intersect**
    **(select** *customer_name* **from** *borrower)*

- Find all customers who have an account but no loan
  - (**select** *customer_name* **from** *depositor*)
    **except**
    **(select** *customer_name* **from** *borrower)*

# Agenda

- GROUP BY
- Relational Algebra Operators
- *NULL*
- Nested subqueries
- “For every” queries

# NULL Values

- Recall: It is possible for tuples to have a null value, denoted by `null`, for some of their attributes
  - Integral part of SQL to handle missing / unknown information
  - **null** signifies that a value *does not exist*, it does *not mean "0" or "blank"*!
  - Schema might constrain a column to prevent NULL values
- The predicate `is null` can be used to check for null values
  - e.g. Find students which enrolled in a course without a grade so far.

```
SELECT sid
        FROM Enrolled
 WHERE grade IS NULL
```

  - Warning: do NOT try t use equality or inequality test eg grade = NULL

# Computing with Nulls

- We describe the rules as three-valued logic
  - Conditions can evaluate to any of: UNKNOWN, TRUE, or FALSE
- *Arithmetic expression*: x *op y* (where *op* is +, –, *, etc.) has value NULL when either x is NULL or y is NULL (or both)
- *Boolean Comparison*: *x op y* (where *op* is <, >, <>, =, etc.) has value UNKNOWN when either x or y is NULL
  - WHERE T.*cost* > T.*price* -- when T.price is NULL, this condition is UNKNOWN
  - WHERE (T. *price*/T.*cost*) > 2 -- when T.price is NULL, this condition is UNKNOWN
- Boolean Operator: x AND y, x OR y, NOT x has the value UNKNOWN when either x is UNKNOWN or y is UNKNOWN, or both
- Tuple is only accepted by where clause predicate when it evaluates to true (not included when it evaluates to false, or to unknown)

  e.g: select sid from enrolled where grade <> 'DI'
  does not return a student for which grade is NULL
- *Aggregates*: COUNT(*) counts any row with NULL just like any other value; other aggregates ignore NULLs

# NULL Values and Aggregation

- COUNT(*) counts any row with NULL just like any other value; other aggregates ignore NULLs
  - result is null if there is no non-null value to aggregate

- Examples:
  - Average mark of the assessments

```
SELECT AVG (mark)    -- ignores tuples with nulls
  FROM Assessment
```

  - Number of the assessments

```
SELECT COUNT (*)      -- counts all tuples (only with *)
  FROM Assessment
```

# NULL values and GROUP BY

- When forming groups, a group is made for all the rows which have NULL in a particular grouping attribute, and the same values for the other grouping attributes
  - That is, NULL is treated as a legitimate value in grouping, and these NULLs are treated together (even though they do not show as “equal to” one another in conditions)
- This is a special feature in the SQL standard

# Nulls and Join Operators

- The usual join (also called inner join) only includes pairs of tuples that match on the join attribute
  - Consider R join S. What if no match is found for a tuple of R?
    - It won’t appear in the result at all
- SQL has outer join operators as well
    - They include rows from one source relation without any match, with null values from the other source relation

# Example Scenario

- Relation *loan*

| *loan-number* | *branch-name* | *amount* |
|---|---|---|
| L-170 | Downtown | 3000 |
| L-230 | Redwood | 4000 |
| L-260 | Perryridge | 1700 |

- Relation borrower

| *customer-name* | *loan-number* |
|---|---|
| Jones | L-170 |
| Smith | L-230 |
| Hayes | L-155 |

- Note: *borrower* information missing for L-260 and *loan* information missing for L-155

# Example

- *loan* **inner join** *borrower* **on** *loan.loan-number = borrower.loan-number*

| *loan-number* | *branch-name* | *amount* | *customer-name* | *loan-number* |
|---|---|---|---|---|
| L-170 | Downtown | 3000 | Jones | L-170 |
| L-230 | Redwood | 4000 | Smith | L-230 |

- *loan* left outer join *borrower* on *loan.loan-number = borrower.loan-number*

| *loan-number* | *branch-name* | *amount* | *customer-name* | *loan-number* |
|---|---|---|---|---|
| L-170 | Downtown | 3000 | Jones | L-170 |
| L-230 | Redwood | 4000 | Smith | L-230 |
| L-260 | Perryridge | 1700 | *null* | *null* |

# Example

- *loan* **natural inner join** *borrower*

| *loan-number* | *branch-name* | *amount* | *customer-name* |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |

- *loan* natural right outer join *borrower*

| *loan-number* | *branch-name* | *amount* | *customer-name* |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |
| L-155 | null | null | Hayes |

# Example

- *loan* **full outer join** *borrower* **using** *(loan-number)*

| *loan-number* | *branch-name* | *amount* | *customer-name* |
|---|---|---|---|
| L-170 | Downtown | 3000 | Jones |
| L-230 | Redwood | 4000 | Smith |
| L-260 | Perryridge | 1700 | *null* |
| L-155 | null | null | Hayes |

# Summary: Join Operators

- Available join types:
  - **inner join**
  - **A left outer join B**
    - For an A tuple with no matching B tuple, include it with null in B columns
  - **right outer join**
  - **full outer join**

- Join Conditions:
  - **natural**
  - **on** <*join condition*>
  - **using** <*attribute list*>

- Using Join operators
  - At top level of query
  - In FROM clause
  - In subquery (se later this lecture)

# Agenda

- GROUP BY
- Relational Algebra Operators
- NULL
- *Nested subqueries*
- “For every” queries

# Nested Subqueries

- SQL provides a mechanism for the nesting of **subqueries** helping in the formulation of complex queries

- 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 'ISYS2120'?

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 uos_code='ISYS2120')
```

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. Noncorrelated 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

| SID |
|---|
| 1002 |
| 1001 |
| 1007 |
| 1001 |
| 1003 |

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

| NAME |
|---|
| Ian Thorpe |
| Michael Phelps |
| Grant Hackett |
| Pieter van den Hoogenband |

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

# Correlated Nested Queries

- With correlated nested queries, the inner subquery depends on the outer query
  - Example:
    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  1
          FROM  Lecturers, UnitofStudy u
         WHERE  name = ‘Einstein’ AND
                lecturer = empid  AND
                   u.uos_code=e.uos_code )
```

Subquery refers to **e**

# Processing a Correlated Subquery

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

2. get the ***`uos_code`*** of the 1st tuple

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

4. If yes, include in result.

5. Loop to step (2) on the next tuple, until whole outer query is checked.

**Student |><| enrolled**

| SID | NAME | BIRTHDATE | COUNTRY | UOS_CODE | SEMESTER |
|---|---|---|---|---|---|
| 200300456 | Henry | 01-JAN-82 | India | COMP5138 | 2005-S2 |
| 200300456 | Henry | 01-JAN-82 | India | ELEC1007 | 2005-S2 |
| 200400500 | Thu | 04-APR-80 | China | COMP5138 | 2005-S1 |
| 200400500 | Thu | 04-APR-80 | China | ELEC1007 | 2005-S1 |

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

| UOS_CODE | TITLE | CPTS | LECTURER | EMPID | NAME | ROOM |
|---|---|---|---|---|---|---|
| COMP5138 | RDBMS | 6 | 1 | 1 | Uwe Roehm | G12 |
| INFO2120 | RDBMS | 6 | 1 | 1 | Uwe Roehm | G12 |
| ISYS3207 | IS Project | 4 | 2 | 2 | Albert Einstein | Heaven |
| ELEC1007 | Introduction to Physics | 6 | 2 | 2 | Albert Einstein | Heaven |

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

# Testing if a subquery EXISTS

- The boolean expression EXISTS (subquery) can appear in a WHERE clause
  - Usually the subquery is correlated, so it returns different relation for each row of the outer query
  - Since it doesn't matter what values are returned, usually the subquery is "select * from …"
- The expression EXISTS(R) is true when the subquery returns a non-empty relation (one or more rows)
  - It is FALSE when the subquery returns empty (ie no rows)
- Warning: the expression EXISTS(R) is boolean, so it can't be compared with a number or string, by "=" or other comparison operator

# EXISTS example

- Find branches located in Sydney where there is an account whose balance is over 10000

```
select distinct branch_name
        from branch
       where branch_city = 'Sydney'
              and exists (select *
                          from  account
                          where  balance > 10000
                          and account.branch_name = branch.branch_name
                          )
```

Equivalent query without exists (using a simple join)

```
select distinct branch_name
        from branch, account
       where branch_city = 'Sydney'
              and balance > 10000
             and account.branch_name = branch.branch_name
```

# NOT EXISTS example

- Find branches located in Sydney where there isn't an account whose balance is over 10000

```
select distinct branch_name
        from branch
       where branch_city = 'Sydney'
             and not exists (select *
                              from  account
                             where  balance > 10000
                                    and account.branch_name = branch.branch_name
                             )
-- this is not the same as asking for branches in Sydney with an account for which
   NOT(balance >10000)
```

# 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

# In vs. Exists Function

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

```
SELECT distinct name
 FROM Student JOIN Enrolled E USING (sid)
 WHERE EXISTS ( SELECT *
                FROM Lecturer JOIN UnitOfStudy U
                                    ON (lecturer=empid)
                WHERE name = 'Einstein'
                      AND U.uos_code = E.uos_code )
```

**Query using IN** **without a subquery**

```
SELECT distinct name
 FROM  Student
WHERE Student.sid IN
 (SELECT e.sid
    FROM Enrolled e, Lecturer, UOS u
  WHERE name = 'Einstein'
     AND lecturer = empid
     AND u.uos_code = e.uos_code)
```

```
SELECT distinct students.name
 FROM Student,Enrolled e,Lecturer,UOS u
WHERE Student.sid = e.sid
   AND lecturer.name = 'Einstein'
   AND lecturer = empid
   AND u.uos_code = e.uos_code
```

# Dealing with multi-row subqueries

- When the subquery returns several rows, we need to be careful how it is used in the outer-query
  - We can ask if “v IN (a set of values)”
  - We can ask if “EXISTS (a set of values)”
  - It can be confusing when we ask if “v > (a set of values)” or if “v = (a set of values)”
  - SQL provides two ways to numerically compare a value to a set of values
  - We can compare the value to ALL of the ones in the set, or to SOME one from the set

# Set Comparison

- **all** clause
  - tests whether a comparison is true for the whole set
    *F comp* **all** (R) means ($\forall\ t \in R$ : (*F comp t*))
  - If R is empty, this comparison is true!
- **some** clause (also called **any**)
  - tests whether a comparison holds for at least one set element
    *F comp* **some** (R) means ($\exists\ t \in R$ : (*F* comp *t*))
  - If R is empty, this comparison is false!
- Here
  - *comp* can be: =, <, >, <=, >= or <>
  - *F* is a constant value or an attribute
  - *R* is a one-column relation [maybe a single value eg from aggregation]
- v < ALL (SELECT x FROM …) is another way to say v < (SELECT min(x) FROM …)

Except for some issues with null values

- w < SOME (SELECT x FROM …) is another way to say w < (SELECT max(x) FROM …)

# Example: Set Comparison

- Find the student(s) with highest mark in tasks of ISYS2120

```
SELECT A1.sid
  FROM Assessment A1
 WHERE A1.uos_code= 'ISYS2120' AND
       A1.mark >= ALL (SELECT A2.mark
                        FROM Assessment A2
                       WHERE A2.uos_code= 'ISYS2120' )
```

# Agenda

- GROUP BY
- Relational Algebra Operators
- NULL
- Nested subqueries
- *“For every” queries*

# ‘For every’ Queries

- These queries can be done in Relational Algebra using the relational division operator
  - Find students who have taken *all* the core units of study,
  - Find suppliers who supply *all* the red parts,
  - Find customers who have ordered *all* items from a given line of products etc.

- These queries check whether or not a *candidate data* is related to each of the values of a given *base set*.

# For every in SQL

- SQL does not directly support *universal quantification* (for all)
- SQL Work-around:
  Search predicates of the form "for all" or " for every" can be formulated using the **not exists** clause on a negated condition
  - Example: "Find courses where all enrolled student already have a grade" convert to "Find courses where there is not an enrolled student who does not have a grade"

```
 SELECT uos_code
FROM UnitOfStudy U
  WHERE NOT EXISTS
   ( SELECT *
     FROM Enrolled E
    WHERE E.uos_code=U.uos_code
          and grade is null )
```

# Example

- How would you answer the following question in SQL?

"Write an SQL query that finds the student(s) that have taken *every* ISYS subject in second year."

# SQL-Division Example

- "Write an SQL query that finds the student(s) that have
  taken *every* ISYS subject in second year."

- What is our base set?
  - **All** second year ISYS subjects
  - In SQL:

```
SELECT uos_code
FROM UnitOfStudy
  WHERE uos_code LIKE 'ISYS2%'
```

- What is our candidate set?
  - Student who have enrolled in **any** second year ISYS subject.
  - In SQL:

```
SELECT DISTINCT sid, uos_code
  FROM  Enrolled
WHERE  uos_code LIKE 'ISYS2%'
```

# Division in SQL

- *Strategy for implementing division in SQL*:
- Reformulate as “not exists not”
  - Eg Find the students for whom there is not an ISYS subject in second year, that the student did not take
  - This we can express in SQL:

```
SELECT DISTINCT S.name
  FROM Student S
 WHERE NOT EXISTS(SELECT *
                    FROM UnitOfStudy U
                   WHERE U.uosCode LIKE 'ISYS2%'
                       AND NOT EXISTS(
                                 SELECT 1
                                   FROM Enrolled E
                                 WHERE E.studId = S.studId
                                   AND E.uosCode=U.uosCode
                                     )
                    )
```

# Division in logic

- Mathematically, we can rewrite

$$\left\{ \langle a \rangle \mid \forall \langle b \rangle \in S : \exists \langle a,b \rangle \in R \right\}$$

as

$$\left\{ \langle a \rangle \mid \neg \exists \langle b \rangle \in S : \neg \exists \langle a,b \rangle \in R \right\}$$

# Division in SQL alternative

- Reformulate as “not exists a set-difference”
  - Eg Find the students for whom there is not an ISYS subject in second year, other than ones the student did take

```
SELECT name
  FROM Student S
 WHERE NOT EXISTS(SELECT uosCode
                    FROM UnitOfStudy
                   WHERE uosCode LIKE 'ISYS2%'
                  EXCEPT
                  SELECT E.uosCode
                    FROM Enrolled E
                   WHERE E.studId = S.studId )
```

# Division in SQL – another way

- Just compare the counts!
  - Find students for which the number of second year ISYS subjects that they take is equal to the total number of second year ISYS subjects

```
SELECT name
  FROM Student JOIN Enrolled USING studId
 WHERE uosCode LIKE 'ISYS2%'  -- count only 2ndyear ISYS units taught
 GROUP BY name
HAVING COUNT(*) = ( SELECT COUNT(*)
                      FROM UnitOfStudy
                     WHERE uosCode LIKE 'ISYS2%' )
```

Important that we filter in both the outer grouping and the inner sub-query for 2nd year ISYS!
Otherwise you compare the wrong counts!

This query above will fail if a student has repeated any subject.
Brainteaser: How would you fix that?

# Schema for some exercises

*students*

| sid | name | birthdate | country |
|---|---|---|---|
| *int* | *varchar* | *date* | *varchar* |

*courses*

| uoscode | title | credit_points | lecturer |
|---|---|---|---|
| *varchar* | *varchar* | *int* | *int* |

*enrolled*

| sid | uoscode | grade |
|---|---|---|
| *int* | *int* | *char* |

*assessment*

| sid | uoscode | empid | mark |
|---|---|---|---|
| *int* | *varchar* | *int* | *int* |

*lecturers*

| empid | name | room |
|---|---|---|
| *int* | *varchar* | *varchar* |

# Exercises

- How many courses are there with the lowest credit_point value, among the courses lectured by Alan Fekete?
- Find the name of student(s) whose mark on an assessment in a course is above the average mark for assessments in that course
- Find the cases of a student and a course, where the student is enrolled in the course and has not received any assessments in that course
- Find the lecturer, if any, who is teaching every course that is worth 12 credit_points
- Find the students, if any, who are enrolled in every course that is worth 12 credit_points

# Exercises

- How many courses are there with the lowest credit_point value, among the courses lectured by Alan Fekete?

```
SELECT COUNT(DISTINCT uoscode)
FROM courses
WHERE lecturer IN (SELECT empid
                   FROM lecturers
                   WHERE name = 'Alan Fekete')
    and credit_points = (SELECT min(credit_points)
                         FROM courses
                         WHERE lecturer IN (SELECT empid
                                            FROM lecturers
                                            WHERE name = 'Alan Fekete'
                                            )
                         )
```

DISTINCT isn't actually needed,
since uoscode is primary key of courses

# Exercises

- Find the name of student(s) whose mark on an assessment in a course is above the average mark for assessments in that course

```
SELECT name
FROM students, assessment a1
WHERE students.sid = a1.sid
        and a1.mark > (SELECT avg(a2.mark)
                        FROM assessment a2
                      WHERE a2.uoscode = a1.usocode)
```

# Exercises

- Find the cases of a student and a course, where the student is enrolled in the course and has not received any assessments in that course

```
SELECT sid, uoscode
FROM enrolled
WHERE NOT EXISTS (SELECT *
                  FROM assessment a
                  WHERE a.uoscode = enrolled.uoscode
                        and a.sid = enrolled.sid)
```

Alternative formulation

```
SELECT sid, uoscode
FROM students, courses
WHERE EXISTS (SELECT *
              FROM enrolled
              WHERE enrolled.sid=students.sid and enrolled.uoscode = courses.uoscode)
      and NOT EXISTS (SELECT *
                      FROM assessment a
                      WHERE a.sid = students.sid and a.uoscode = courses.uoscode)
```

# Exercises

- Find the lecturer, if any, who is teaching every course that is worth 12 credit_points

Convert to: Find the lecturer, if any, where there is not a 12 credit_point course that the lecturer does not not teach

```
SELECT *
FROM lecturers
WHERE NOT EXISTS (SELECT *
                  FROM courses
                  WHERE credit_points = 12
                        and courses.lecturer <> lecturers.empid)
```

Note the way we make use of the fact that there is a single lecturer in a course

# Exercises

- Find the students, if any, who are enrolled in every course that is worth 12 credit_points

Convert to: Find the students, if any, where there is not a 12 credit_point course for which there is not an enrollment for that student in that course

```
SELECT *
FROM students s
WHERE NOT EXISTS (SELECT *
                  FROM courses c
                  WHERE credit_points = 12
                        and NOT EXISTS (SELECT *
                                        FROM enrolled e
                                        WHERE e.usocode = c.uoscode
                                              and e.sid = s.sid
                                        )
                  )
```

# References

- Silberschatz/Korth/Sudarshan(7ed)
  - Chapter 3.5-3.8, 4.1

Also

- Kifer/Bernstein/Lewis(complete version, 2ed)
  - Chapter 5.2.2-5.2.8, 5.2.10
- Ramakrishnam/Gehrke(3ed)
  - Chapter 5.3-5.6
- Garcia-Molina/Ullman/Widom(complete book, 2ed)
  - Chapter 6.1.6-6.1.7, 6.2.5, 6.3, 6.4

# Summary

- GROUP BY
- Relational Algebra Operators
  - UNION, INTERSECT, EXCEPT
- NULL
  - Use in SQL computations
- Nested subqueries
  - Correlated, uncorrelated
- “For every” queries