ISYS2120 – Data & Information Management

Week 4B: More SQL (Group By, NULL, Nested Subqueries)

Based on slides from Kifer/Bernstein/Lewis (2006) “Database Systems”
and from Ramakrishnan/Gehrke (2003) “Database Management Systems”,
and also including material from Fekete and Röhm.

Prof Alan Fekete

THE UNIVERSITY OF SYDNEY

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

This material has been reproduced and communicated to you by or on behalf of the University of Sydney pursuant to Part VB of the Copyright Act 1968 (**the Act**). The material in this communication may be subject to copyright under the Act. Any further copying or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice.

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:

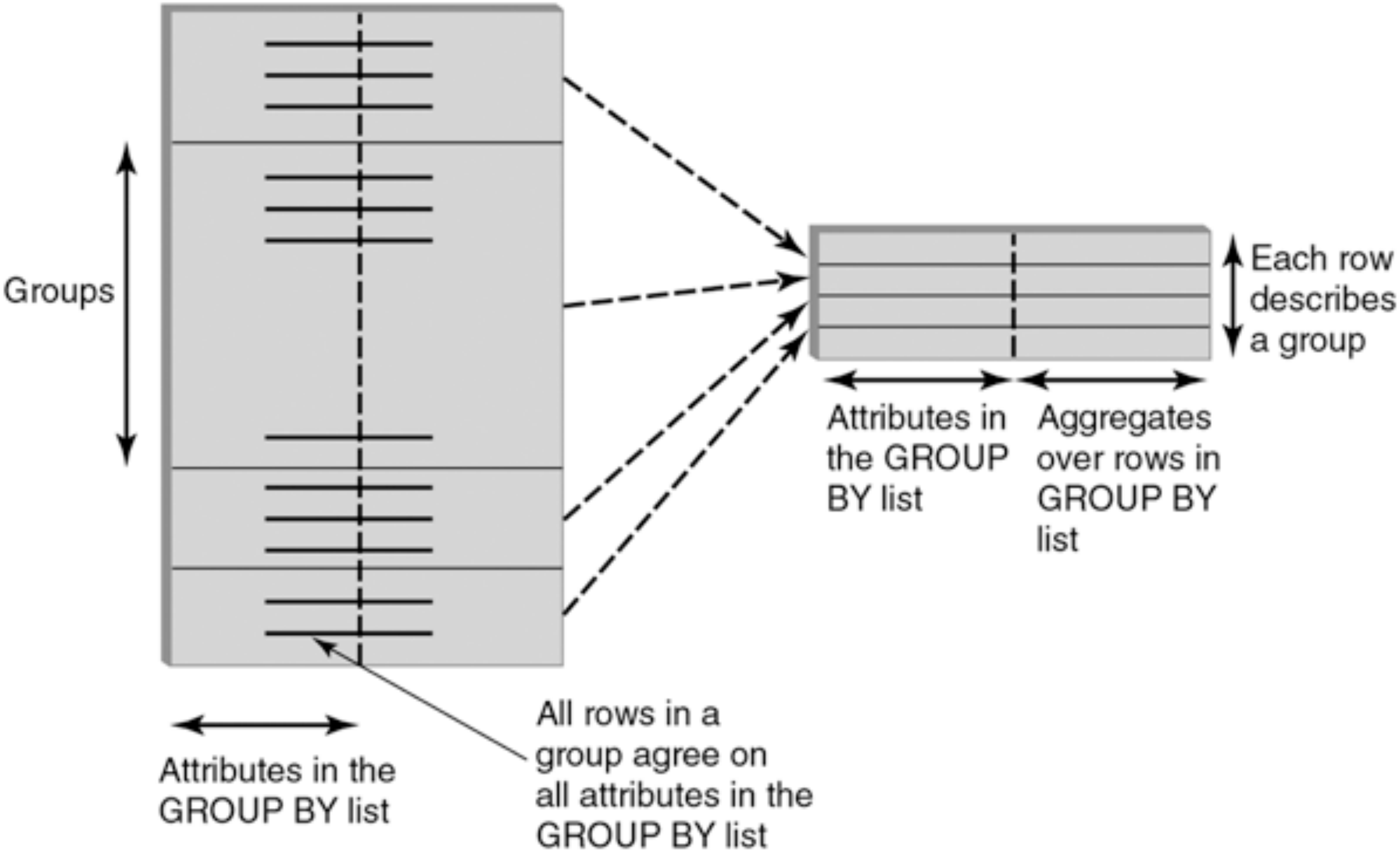
```
  SELECT   [DISTINCT]  target-list
    FROM   relation-list
   WHERE   qualification
GROUP BY   grouping-list
  HAVING   group-qualification
```

- 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 outside of 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


Groups
Each row describes a group
Attributes in the GROUP BY list
Aggregates over rows in GROUP BY list
Attributes in the GROUP BY list
All rows in a group agree on all attributes in the GROUP BY list

[Kifer/Bernstein/Lewis 2006]

FIGURE 5.9 Effect of the GROUP BY clause.

Example:
Filtering Groups with HAVING Clause

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

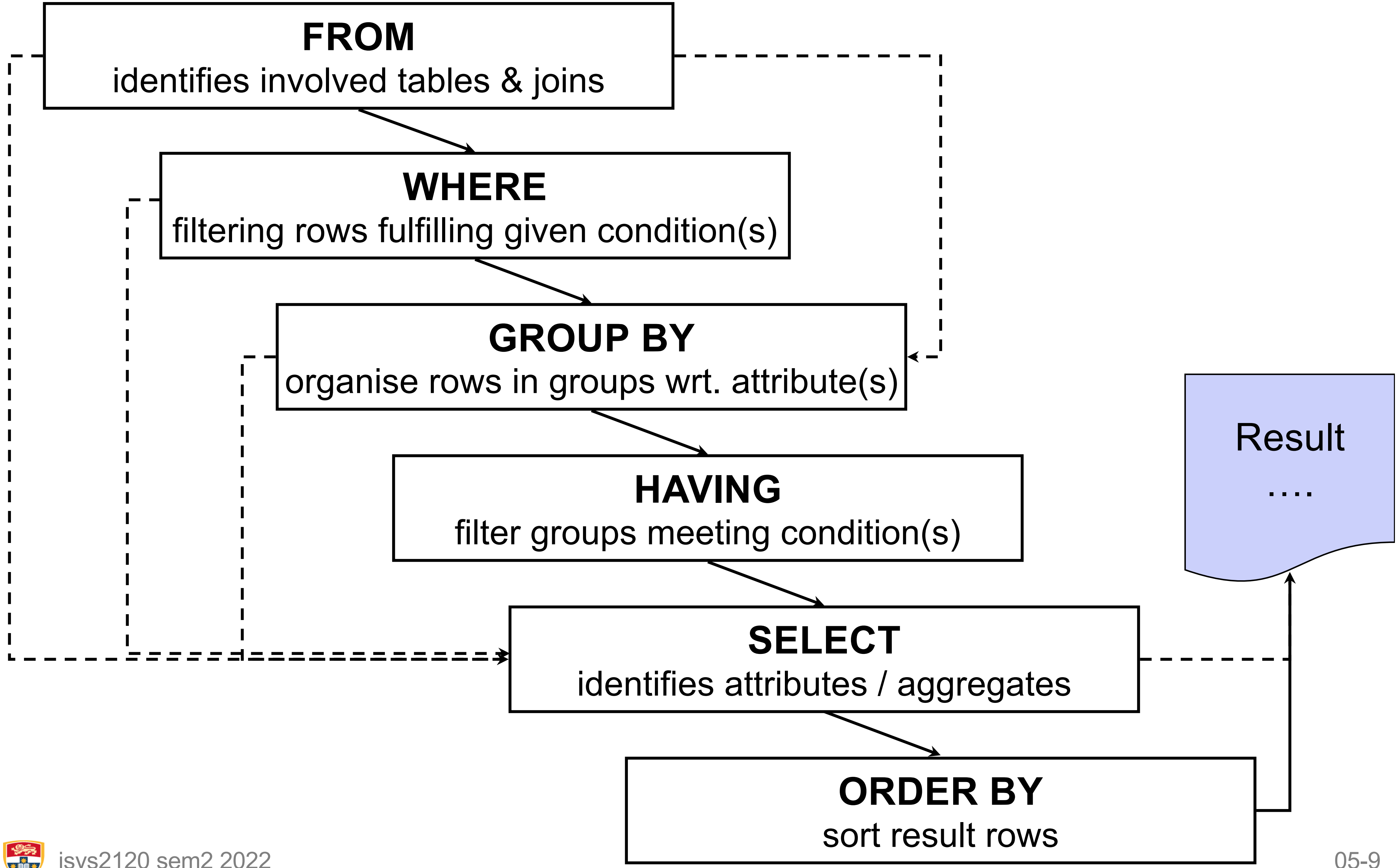
```
    SELECT uos_code as unit_of_study, AVG(mark)
      FROM Assessment
  GROUP BY uos_code
```

- 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 whereas predicates in the **where** clause are applied to individual rows, before forming groups

Query-Clause Evaluation Order



FROM
identifies involved tables & joins

WHERE
filtering rows fulfilling given condition(s)

GROUP BY
organise rows in groups wrt. attribute(s)

HAVING
filter groups meeting condition(s)

SELECT
identifies attributes / aggregates

ORDER BY
sort result rows

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

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

NULL Values

- 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"*!
- 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
```

- Consequence: Three-valued logic
  - The result of any arithmetic expression involving null is null
    - e.g. 5 + null returns null
  - However, (most) aggregate functions simply ignore nulls

NULL Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
  - e.g. 5 < *null* or *null* <> *null* or *null* = *null*

- Three-valued logic using the truth value *unknown*:
  - OR: (*unknown* **or** *true*) = *true*, (*unknown* **or** *false*) = *unknown*
    (*unknown* **or** *unknown*) = *unknown*
  - AND: (*true* **and** *unknown*) = *unknown*, (*false* **and** *unknown*) = *false*, (*unknown* **and** *unknown*) = *unknown*
  - NOT: (**not** *unknown*) = *unknown*

- 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'
    ignores all students without a grade so far

NULL Values and Aggregation

- Aggregate functions except **count(*)** ignore null values on the aggregated attributes
  - result is null if there is no non-null amount

- Examples:
  - Average mark of all assignments

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

  - Number of all assignments

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

More 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**

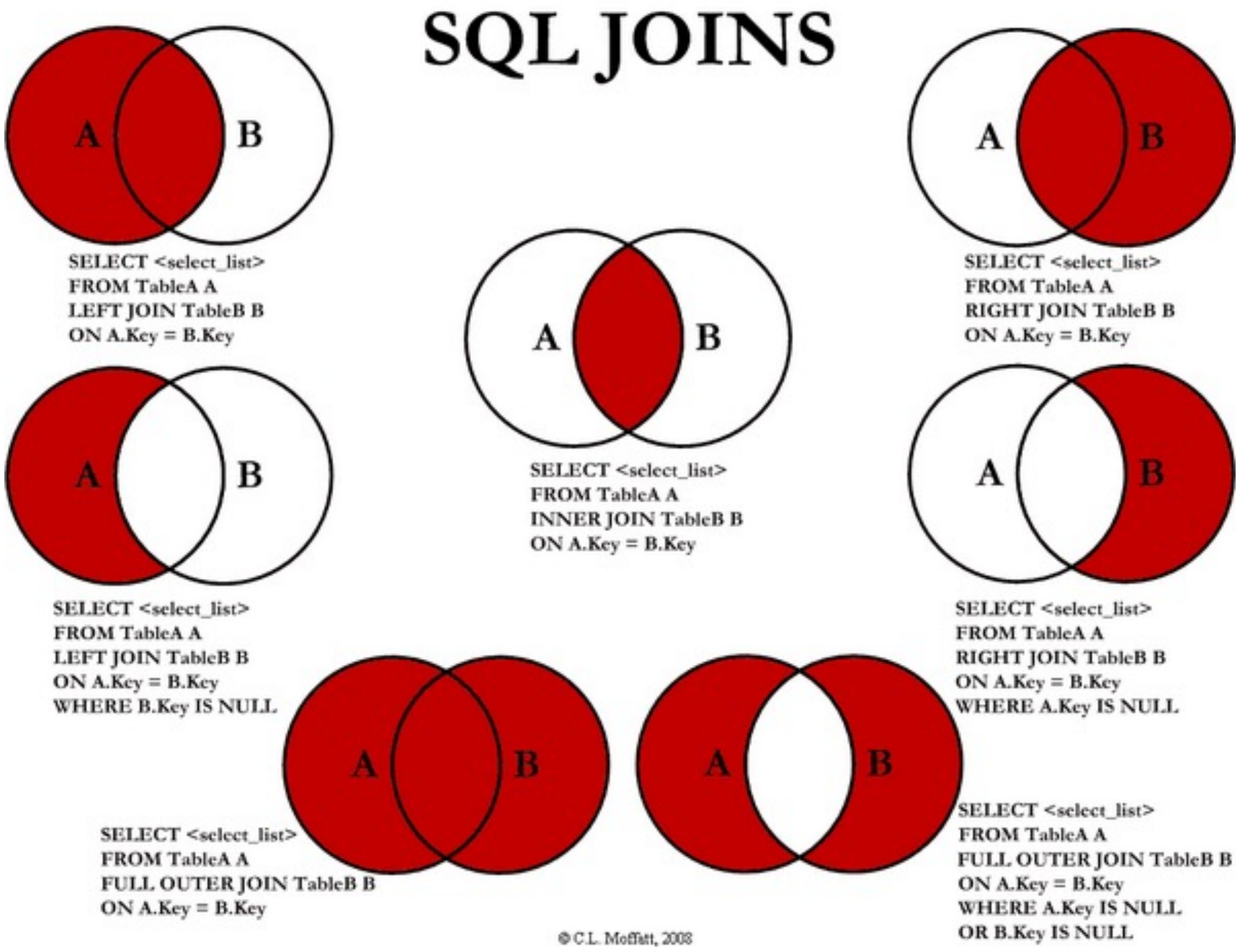
- Join Conditions:
  - **natural**
  - **on** *<join condition>*
  - **using** *<attribute list>*

e.g: *Student* **inner join** *Enrolled* **using** *(sid)*

inner join result						
sid	name	birthdate	country	sid2	uos_code	grade
112	*'A'*	*01.01.84*	*India*	*112*	*SOFT1*	*P*
200	*'B'*	*31.5.79*	*China*	*200*	*COMP2*	*C*

e.g : *Student* **left outer join** *Enrolled* **using** *(sid)*

left outer join *result*						
sid	name	birthdate	country	sid2	uos_code	grade
112	*'A'*	*01.01.84*	*India*	*112*	*SOFT1*	*P*
200	*'B'*	*31.5.79*	*China*	*200*	*COMP2*	*C*
210	*'C'*	*29.02.82*	*Australia*	*null*	*null*	*null*



SQL JOINS

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```

```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

© C.L. Moffatt, 2008

http://dieswaytoofast.blogspot.com.au/2013/05/sql-joins-visualized.html

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'?

```
SELECT name
  FROM Student
 WHERE sid IN ( SELECT sid
                  FROM Enrolled
                 WHERE uos_code='ISYS2120' )
```

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

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

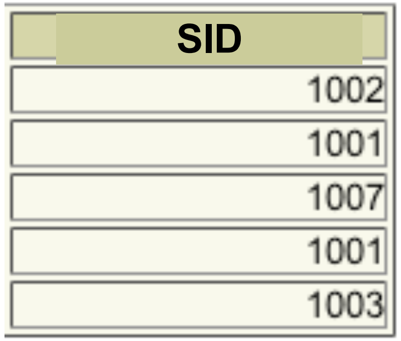
- **Noncorrelated subqueries:**
  - Do not depend on data from the outer query
  - Execute once for the entire outer query

- **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 );
```

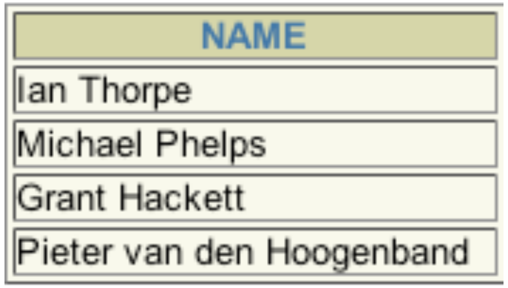
1. The subquery executes first and returns as intermediate result all student IDs from the **`Enrolled`** table



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’.

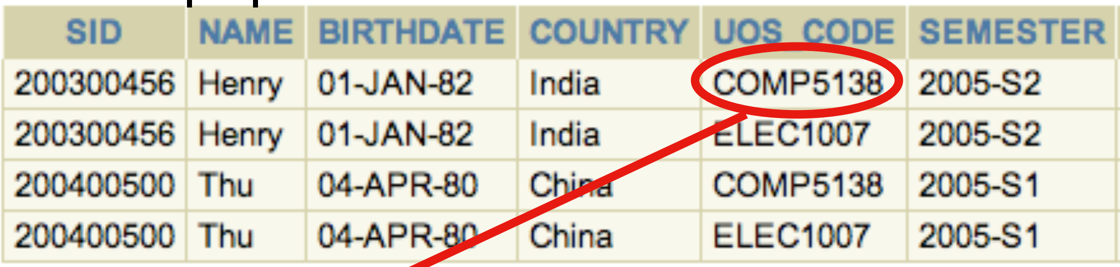
```
SELECT DISTINCT name
  FROM Student, Enrolled e
 WHERE Student.sid = e.sid AND
       EXISTS ( SELECT  1
                  FROM  Lecturers, UnitofStudy u
                 WHERE  name = ‘Einstein’ AND
                        lecturer = empid  AND
                        u.uos_code=e.uos_code )
```

Subquery refers to **e**

Processing a Correlated Subquery

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

Student |><| enrolled

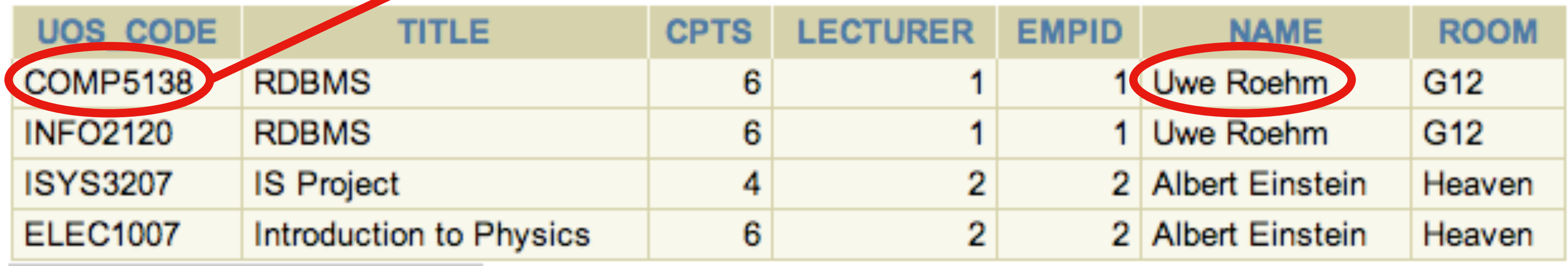


SID	NAME	BIRTHDATE	COUNTRY	UOS_CODE	SEMESTER
200300456	Henry	01-JAN-82	India	COMP5138	2005-S2
200300456	Henry	01-JAN-82	India	ELEC1007	2005-S2
200400500	Thu	04-APR-80	China	COMP5138	2005-S1
200400500	Thu	04-APR-80	China	ELEC1007	2005-S1

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

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

4. If yes, include in result.

5. Loop to step (2) on the next tuple, 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

- 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

```
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)
```

without a subquery

```
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
```

Set Comparison Operators in SQL

- (**not**) **exists** clause
  - tests whether a set is (not) empty (**true** $\Leftrightarrow$ $R \neq \emptyset$) (**true** $\Leftrightarrow$ $R = \emptyset$)
- **unique** clause *(note: not supported by Oracle or PostgreSQL)*
  - tests whether a subquery has any duplicate tuples in its result
- **all** clause
  - tests whether a predicate is true for the whole set

    F *comp* **ALL** $R \quad \Leftrightarrow \quad \forall\, t \in R : (F \textit{ comp } t)$
- **some** clause (any)
  - tests whether some comparison holds for at least one set element

    F *comp* **SOME** $R \quad \Leftrightarrow \quad \exists\, t \in R : (F \textit{ comp } t)$

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 marks.

```
SELECT S.sid
   FROM Student S
  WHERE S.mark >= ALL ( SELECT mark
                         FROM Assessment
                        WHERE uos_code='ISYS2120' )
```

- Find students which never repeated any subjects.

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

Examples: Set Comparison (cont’ d)

- 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
  - Example:
    Find courses where all enrolled student already 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 )
```

Motivating Problem

- 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.”

Many Ways to Write this Query…

“Finds the actors (by ID) who played in the film ***VELVET TERMINATOR***.”

```
SELECT actor_id
  FROM Film_Actor a, Film f
 WHERE a.film_id = f.film_id
   AND f.title = 'VELVET TERMINATOR'
```

```
SELECT aid AS actor_id
  FROM (SELECT f.film_id AS f1,  a.film_id AS f2,
               actor_id  AS aid, f.title   AS title
          FROM Film f, Film_Actor a)
 WHERE f1=f2 AND title = 'VELVET TERMINATOR'
```

```
SELECT actor_id
  FROM (SELECT actor_id, title
          FROM Film_Actor NATURAL JOIN Film)
 WHERE title = 'VELVET TERMINATOR'
```

```
SELECT actor_id
  FROM Film_Actor
 WHERE film_id IN (SELECT film_id
                     FROM Film
                    WHERE title = 'VELVET TERMINATOR')
```

‘For-All-Set’ Type Queries in SQL

- Some queries are hard to express with just the core Rational Algebra operators and joins; e.g.
  - 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*.

SQL-Division Example

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

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

```
SELECT uos_code
  FROM UnitOfStudy
 WHERE uos_code LIKE ‘INFO2%’
```

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

```
SELECT DISTINCT sid, uos_code
  FROM  Enrolled
WHERE  uos_code LIKE ‘INFO2%’
```

SQL-Division Example (cont'd)

- So far so good.
- But how do we find students in the *candidate set* that have a match for every entry in the *base set*?
- Let's have a look at the foundations….

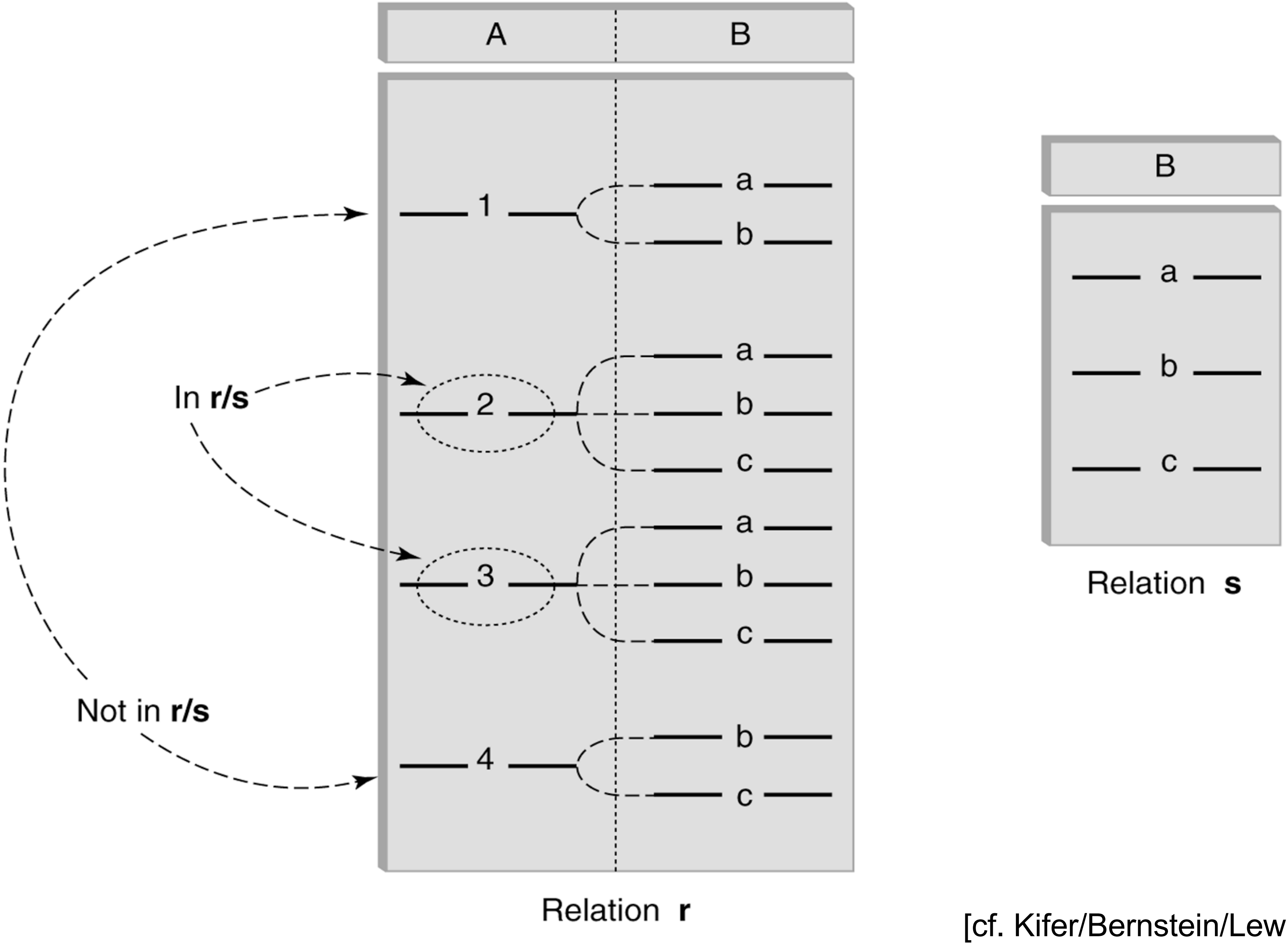
Relational Division

- *Query type*: Find the items in a set that are related to *all* tuples in another set
  - **Note:** This can be seen as the inverse of the cross product (x) …

- Relational Algebra: Division operator (R / S)
  - We call the base set (S) the *divisor* (or *denominator)*
  - and the candidate set (R) the *dividend* (or *numerator)*

- **Definition:** Relational Division
  - $R\ (a_1, \ldots a_n, b_1, \ldots b_m)$
  - $S\ (b_1 \ldots b_m)$
  - *R/S*, with attributes $a_1, \ldots a_n$, is the set of all tuples <*a*> such that for <u>every</u> tuple <*b*> in *S*, there is an <*a,b*> tuple in *R*
    - R/S := $\left\{ \langle a \rangle \mid \forall \langle b \rangle \in S : \exists \langle a,b \rangle \in R \right\}$

Visualisation of Division



[cf. Kifer/Bernstein/Lewis, Figure 5.6]

Examples of Division A/B

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

R

Example 1

pno
p2

S1

sno
s1
s2
s3
s4

R/S1

Example 2

pno
p2
p4

S2

sno
s1
s4

R/S2

Example 3

pno
p1
p2
p4

S3

R/S3

[cf. Ramakrishnan/Gehrke]

Expressing R/S Using Basic Operators

- Division is not an essential operator; just a useful shorthand.
  - (This is also true of joins, but joins are so common that systems implement joins specially)
  - Division can be expressed in terms of projection, set difference, and cross-product

- *Idea*: For *R/S*, compute all *a* values that are not \`disqualified' by some *b* value in *S*.
  - *a* value is *disqualified* if by attaching *b* value from *S*, we obtain an *ab* tuple that is not in *R*.

Disqualified *a* values: $\pi_a((\pi_a(R) \times S) - R)$

R/S: $\pi_a(R) -$ all disqualified tuples

SQL-Division Example (cont'd)

- "Write an SQL query that finds the student(s) who have enrolled in ***all*** second year INFO subjects."

- Base set (our denominator)
  - **All** second year INFO subjects
    $S = \pi_{\mathit{uosCode}} (\sigma_{\mathit{uosCode\ LIKE\ 'INFO2\%'}} (\mathrm{UnitOfStudy}))$

- Candidate set (numerator)
  - Students who have taken **any** second year INFO subject.
    $R = \pi_{\mathit{studId,\ uosCode}} (\sigma_{\mathit{uosCode\ LIKE\ 'INFO2\%'}} (\mathrm{Enrolled}))$

- Result is *numerator/denominator* (R/S)

Division in SQL

- *Strategy for implementing division in SQL*:
  - Recall definition of division:
    $R/S := \{\langle a\rangle \mid \forall\langle b\rangle \in S : \exists\langle a,b\rangle \in R\}$
- Core problem: no universal quantification in SQL
  - Hence need to reformulate: $\{\langle a\rangle \mid \neg\exists\langle b\rangle \in S : \neg\exists\langle a,b\rangle \in R\}$
  - This we can express in SQL:

```
SELECT DISTINCT S.name
  FROM Student S
 WHERE NOT EXISTS(SELECT *
                   FROM UnitOfStudy S
                  WHERE uosCode LIKE 'INFO2%' AND NOT EXISTS(
                         SELECT 1 FROM Enrolled E
                          WHERE E.studId = S.studId AND
                                E.uosCode=S.uosCode )
       )
```

Division in SQL - optimized

- The previous example is not very elegant and hard to understand
- So let’s further simplify our mathematical expression for division:

$$\neg\exists\langle b\rangle \in S : \neg\exists\langle a,b\rangle \in R \;\Rightarrow\; S \subseteq \pi_{<b>}(R)$$

- Idea:
  Use set-difference to test whether S is a subset of R, i.e.
  output tuples where S - $\pi_{<b>}(R)$ is empty

```
SELECT name
  FROM Student S
 WHERE NOT EXISTS(SELECT uosCode
                    FROM UnitOfStudy S
                   WHERE uosCode LIKE ‘INFO2%’
               EXCEPT
                  SELECT uosCode
                    FROM Enrolled E
                   WHERE E.studId = S.studId )
```

Division in SQL

- *Strategy for implementing division in SQL*:
  - Find the candidate set R
    - in our example: all 2nd year INFO subjects that were taken by a particular student, *s*
  - Find the base set S
    - in the example: all 2nd year INFO subjects
  - Output *s* if $S \supseteq R$, or, equivalently, if $R – S$ is empty

Division in SQL – further optimized

- Further optimization: Just compare the counts!
  - Rationale: If the two sets R and S are equal, they have the same cardinality

    Formally: $S \subseteq \pi_{<b>}(R) \quad \Rightarrow \quad \left|\pi_{<b>}(R)\right| \geq |S|$

```
SELECT name
  FROM Student JOIN Enrolled USING studId
 WHERE uosCode LIKE ‘INFO2%’   -- count only 2ndyear INFO units taught
 GROUP BY name
HAVING COUNT(*) = ( SELECT COUNT(*)
                      FROM UnitOfStudy
                     WHERE uosCode LIKE ‘INFO2%’ )
```

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

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

Similar Problem: Set Comparison in SQL

- A similar issue is **comparing two sets for equality** in SQL

- Problem: There is no set-comparison operator in SQL…
  - … WHERE (SELECT bla FROM…) = (SELECT blubb FROM…) does not work
  - We only can check for
    - empty set (NOT EXISTS (*set*)), and
    - set membership (*value* IN *set*)
  - And do some core set operations
    - set union (set_A UNION set_B)
    - set intersection (set_A INTERSECT set_B)
    - set difference (set_A EXCEPT set_B) (use MINUS in Oracle)

References

- Kifer/Bernstein/Lewis (2nd edition – 2006)
  - Chapter 5.1
    one section on RA that covers everything as discussed here in the lecture
- Ramakrishnan/Gehrke (3rd edition - the ‘Cow’ book (2003))
  - Chapter 4.2
    one compact section on RA, including a discussion of relational division
- Ullman/Widom (3rd edition – 2008)
  - Chapter 2.4
    a nice and gentle introduction to the basic RA operations,
    leaves out relational division though
  - Chapters 5.1 and 5.2
    goes beyond what we cover here in the lecture by extending RA to bags and also introduces grouping, aggregation and sorting operators