COMP9120

Week 6: Complex SQL

Semester 2, 2016

(Ramakrishnan/Gehrke – Chapter 5 & 4.2.5;

Kifer/Bernstein/Lewis – Chapter 5;

Ullman/Widom – Chapter 6.3&6.4)

Based on material by Dr. Bryn Jeffries
And Dr. Uwe Röhm

THE UNIVERSITY OF SYDNEY

Outline

- **Nested Subqueries**
- **Grouping**
- **Relational Division**
  - and how FOR ALL queries are efficiently expressed in SQL

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

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 'INFO2120'?

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='INFO2120' );
```

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 **Enrolled**

Processing a Correlated Subquery

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

1. First join the **`Student`** and **`Enrolled`** tables;
2. get the ***`uos_code`*** of the current tuple (start with 1st)
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) 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 & Exists Operators

- 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*

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

THE UNIVERSITY OF SYDNEY

In & Exists Operators

- 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 UOS U
                                ON (lecturer=empid)
                WHERE name = 'Einstein'  AND
                      U.uos_code = E.uos_code );
```

```
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 student.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** operator
  - tests whether a set is (not) empty   (**true** $\Leftrightarrow$ $R \neq \emptyset$) (**true** $\Leftrightarrow$ $R = \emptyset$)
- **unique** operator *(note: not supported by Oracle)*
  - tests whether a subquery has any duplicate tuples in its result
- **all** operator
  tests whether a predicate is true for the whole set
  F *comp* **ALL** $R \quad \Leftrightarrow \quad \forall\, t \in R : (F \text{ } comp \text{ } t)$
- **some** operator (any)
  tests whether some comparison holds for at least one set element
  F *comp* **SOME** $R \quad \Leftrightarrow \quad \exists\, t \in R : (F \text{ } comp \text{ } t)$
- **(not) in** operator
  - tests whether the result of a subquery includes (or excludes if using not) a value specified in the outer query

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 Student );
```

- 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* (forall)
- SQL Work-around:
  Search predicates of the form "for all" or " for every" can be formulated using the **not exists** operator
  - 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 );
```

Today's Agenda

- Nested Subqueries
- **Grouping**
- Relational Division
  - and how FOR ALL queries are efficiently expressed in SQL

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

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

company	amount
IBM	16500
DELL	16500
IBM	16500

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

company	amount
IBM	12000
DELL	4500

✓

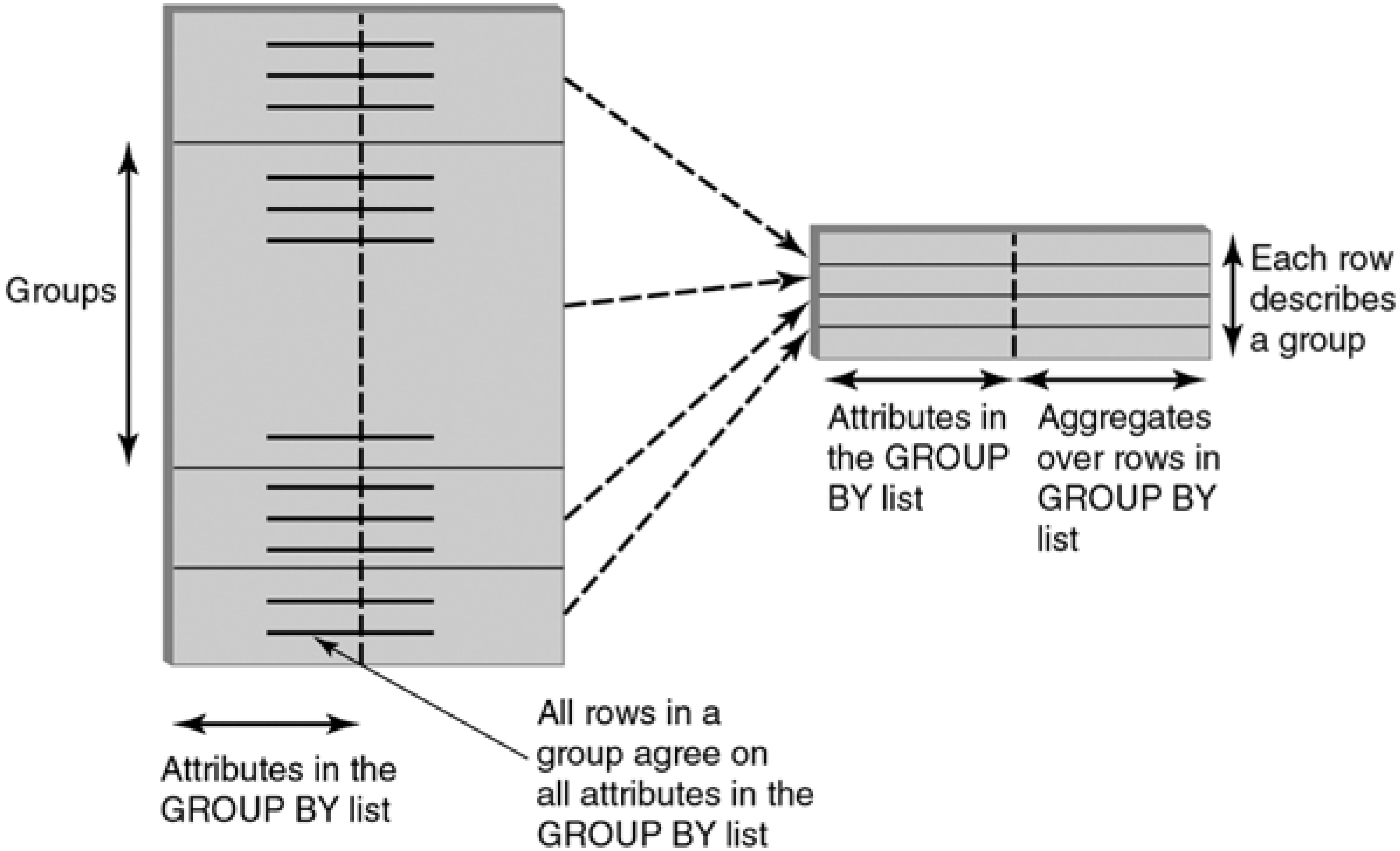
Queries with GROUP BY and HAVING

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

```
   SELECT   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



[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 course?

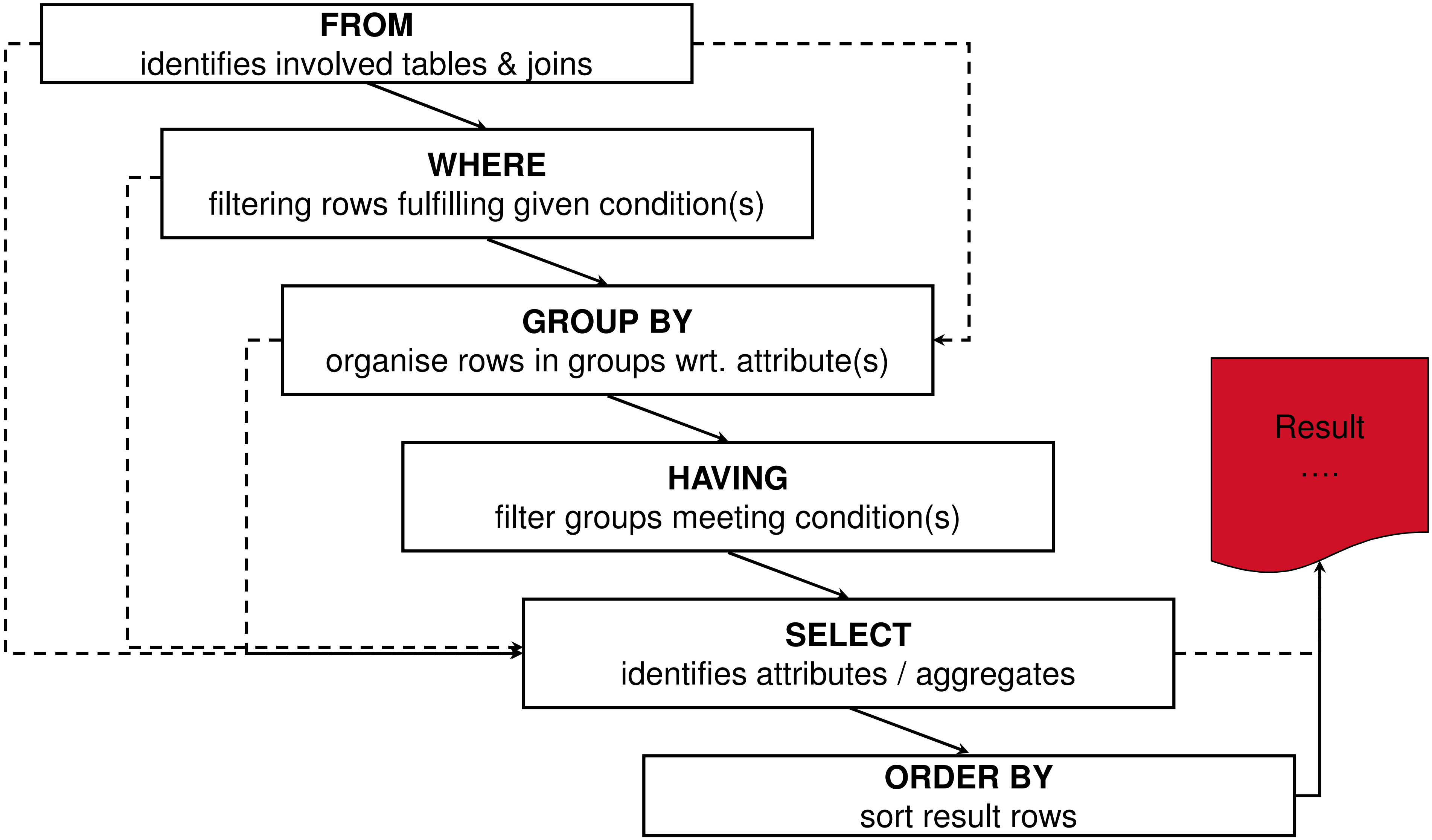
```
    SELECT uos_code as unit_of_study, AVG(mark)
      FROM Assessment
  GROUP BY uos_code;
```

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

```
    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 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

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

Today's Agenda

- Nested Subqueries
- Grouping
- **Relational Division**
  - and how FOR ALL SET queries are efficiently expressed in SQL

Motivating Problem

- How would you answer the following question in SQL?

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

‘For-All-Set’ Type Queries in SQL

- Some queries are hard to express with just the core RA 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 a 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 uosCode
  FROM UnitOfStudy
 WHERE uosCode LIKE 'INFO2%'
```

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

```
SELECT DISTINCT studId, uosCode
  FROM  Enrolled
 WHERE  uosCode 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*?

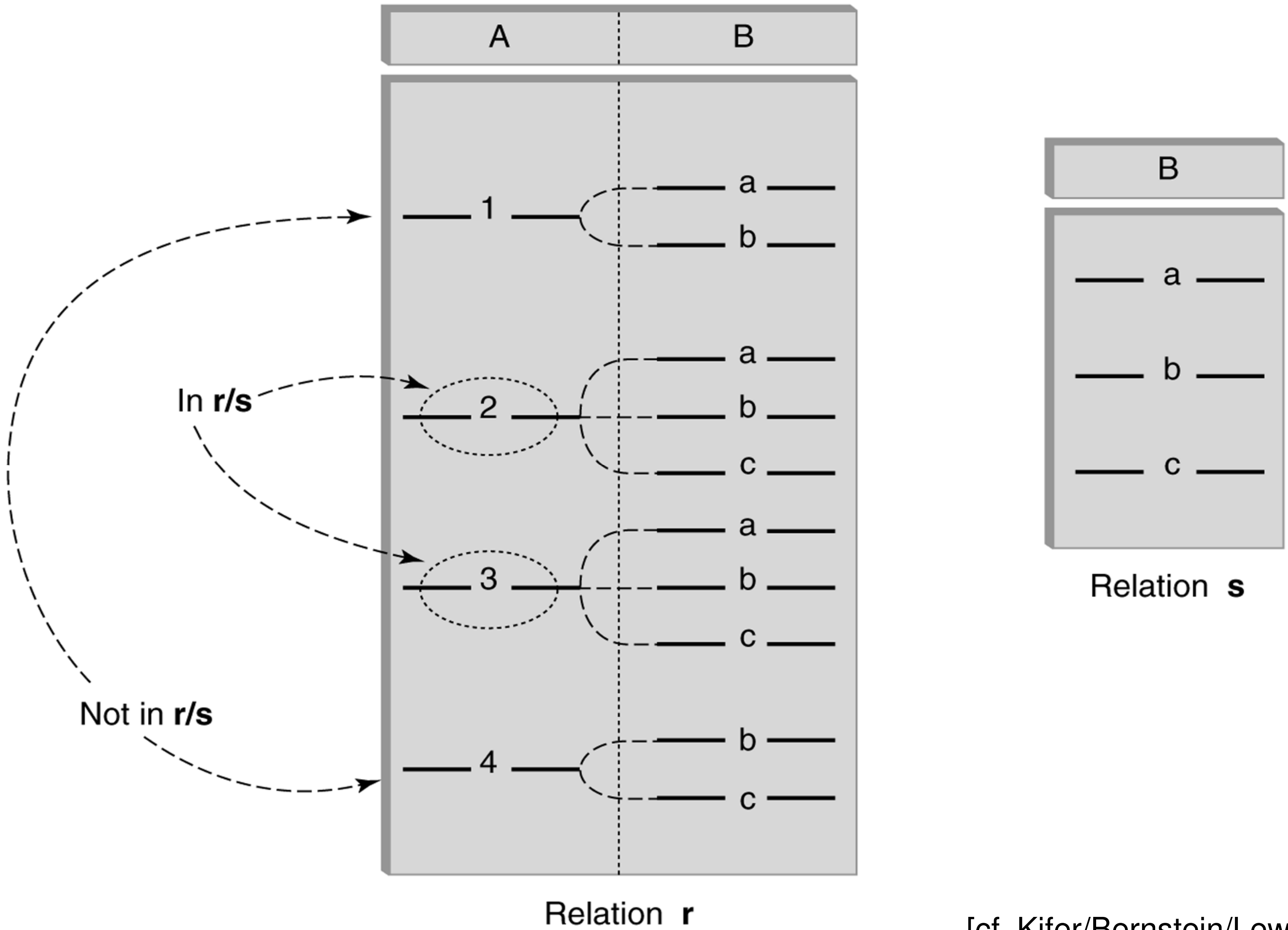
- Let's have a look at the foundations….

Relational Division

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

sno
s1

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

THE UNIVERSITY OF SYDNEY

SQL-Division Example (cont'd)

- "Write a 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

    $S = \pi_{uosCode}(\sigma_{uosCode\ LIKE\ 'INFO2\%'}(\text{UnitOfStudy}))$

- Candidate set (numerator)
  - Students who have taken **any** second year INFO subject.

    $R = \pi_{studId,\ uosCode}(\sigma_{uosCode\ LIKE\ 'INFO2\%'}(\text{Enrolled}))$

- 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 S.studId
 FROM Student S
WHERE NOT EXISTS(SELECT *
                   FROM UnitOfStudy U
                  WHERE uosCode LIKE 'INFO2%' AND NOT EXISTS(
                        SELECT E.studId FROM Enrolled E
                         WHERE E.studId = S.studId AND
                               E.uosCode=U.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 S.studId
  FROM Student s
 WHERE NOT EXISTS(SELECT uosCode
                   FROM UnitOfStudy U
                  WHERE uosCode LIKE 'INFO2%'
                EXCEPT
                   SELECT uosCode
                     FROM Enrolled E
                  WHERE E.studId = s.studId );
```

- *Strategy for implementing division in SQL*:
  - Find the candidate set R
    - in our example: all 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 studId
  FROM Student JOIN Enrolled USING studId
 WHERE uosCode LIKE 'INFO2%'   -- count only 2ndyear INFO units taught
 GROUP BY studId
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 Equality in SQL

- A similar issue is **comparing two sets for equality** in SQL
- Problem: There is no set-equality 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)

Approaches

- As this is a quite common problem in querying, a lot of different approaches are known
  - E.g. based on set theory:

$$R = S \Leftrightarrow R - S = \emptyset \wedge S - R = \emptyset$$

  - That's however typically one of the slower version when done in SQL
  - Better just using counting again similar to our last optimisation for division
    - But then be careful to have the correct filter condition in place

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

You should now be able to…

- **…formulate even complex SQL Queries**
  - Including multiple joins with correct join conditions
  - correlated and non-correlated subqueries
  - Grouping and Having conditions
  - Relational division and full set comparisons
- …transform SQL queries between different forms
  - E.g.
    - correlated queries and join queries
  - …know how SQL relates to the relational algebra

References

- Kifer/Bernstein/Lewis (2nd edition)
  - Chapter 5; Chapter 4.2.5
- Ramakrishnan/Gehrke (3rd edition - the ‘Cow’ book)
  - Chapter 5
- Ullman/Widom (3rd edition)
  - Sections 6.3 and 6.4
- Silberschatz/Korth/Sudarshan (5th edition - ‘sailing boat’)
  - Sections 3.1-3.6
- Elmasri/Navathe (5th edition)
  - Sections 8.4 and 8.5.1

Next Lecture

- Schema Normalization
  - Functional Dependencies
  - Schema Normal Forms
  - Schema Normalization

- Readings:
  - Kifer/Bernstein/Lewis book, Chapter 6
  - Ramakrishnan/Gehrke (Cow book), Chapter 19
  - Ullman/Widom, Chapter 3 (up-to 3.5)