SQL Lecture III

G51DBI - Databases and Interfaces

Yorgos Tzimiropoulos

yorgos.tzimiropoulos@nottingham.ac.uk

Overview of weeks 2-5

We will see how to translate English to Relational Algebra and SQL queries and vice versa

English: "Find all universities with > 20000 students"

Relational Algebra: $\pi_{uName}(\sigma_{enr > 20000}(University))$ **SQL**: Select uName From University Where

University.enr>20000

Theory is easy and simple

But a sequence of simple operations is not always so

obvious!

This Lecture

- **➢** Joins
 - · Cross, Inner, Natural, Outer
- ➤ ORDER BY to produce ordered output
- ➤ Aggregate functions
 - MIN, MAX, SUM, AVG, COUNT
- ➤ GROUP BY
- > HAVING
- **>** UNION
- ➤ Missing Information

Joins

- · JOINs can be used to combine tables in a SELECT query
 - of JOIN
 - · CROSS JOIN
 - · INNER JOIN
 - · NATURAL JOIN
 - · OUTER JOIN

- A CROSS JOIN B
 - · Returns all pairs of rows from A and B, the same as Cartesian Product
- Returns pairs of rows satisfying a condition
- A NATURAL JOIN B
 - · Returns pairs of rows with common values in identically named columns
- A OUTERJOIN B
 - Returns pairs of rows satisfying a condition (as INNER JOIN), BUT ALSO handles NULLS

CROSS JOIN

- SELECT * FROM A CROSS JOIN B
- Is the same as
- SELECT * FROM A, B
- · Usually best to use a WHERE clause to avoid huge result sets
 - · Without a WHERE clause, the number of rows produced will be equal to the number of rows in A multiplied by the number of rows in B.

CROSS JOIN

| Student | | |
|---------|------|--|
| ID | Name | |
| 123 | John | |
| 124 | Mary | |
| 125 | Mark | |
| 126 | Jane | |

Enrolment ID Code DBS 123 124 PRG 124 DBS 126 PRG

SELECT * FROM Student CROSS JOIN Enrolment

CROSS JOIN

Student Name

123 John 124 Mary 125 Mark 126 Jane

Enrolment

ID Code 123 DBS 124 PRG 124 DBS 126 PRG

SELECT * FROM Student CROSS JOIN

Enrolment

| ID | Name | ID | Code |
|-----|------|-----|------|
| 123 | John | 123 | DBS |
| 124 | Mary | 123 | DBS |
| 125 | Mark | 123 | DBS |
| 126 | Jane | 123 | DBS |
| 123 | John | 124 | PRG |
| 124 | Mary | 124 | PRG |
| 125 | Mark | 124 | PRG |
| 126 | Jane | 124 | PRG |
| 123 | John | 124 | DBS |
| 124 | Mary | 124 | DBS |

INNER JOIN

• INNER JOIN specifies a condition that pairs of rows must satisfy

· Can also use a USING clause that will output rows with equal values in the specified columns

SELECT *

FROM A INNER JOIN B ON condition

SELECT * FROM A INNER JOIN B USING (col1, col2)

• col1 and col2 must appear in both A and B

INNER JOIN

| Buyer | | |
|-------|---------|--|
| Name | Budget | |
| Smith | 100,000 | |
| Jones | 150,000 | |
| Green | 80,000 | |

SELECT * FROM

Buyer INNER JOIN Property ON Price <= Budget

| Troperty | | |
|-----------------|---------|--|
| Address | Price | |
| 15 High Street | 85,000 | |
| 12 Queen Street | 125,000 | |
| 87 Oak Lane | 175,000 | |

INNER JOIN

| Buyer | |
|-------|---------|
| Name | Budget |
| Smith | 100,000 |
| Jones | 150,000 |
| Green | 80,000 |

Property

| Address | Price |
|-----------------|---------|
| 15 High Street | 85,000 |
| 12 Queen Street | 125,000 |
| 87 Oak Lane | 175,000 |
| | |

SELECT * FROM

Buyer INNER JOIN Property ON Price <= Budget

| Name | Budget | Address | Price |
|-------|---------|-----------------|---------|
| Smith | 100,000 | 15 High Street | 85,000 |
| Jones | 150,000 | 15 High Street | 85,000 |
| Jones | 150,000 | 12 Queen Street | 125,000 |
| | | | |

INNER JOIN

Student

| Stauciit | |
|----------|------|
| ID | Name |
| 123 | John |
| 124 | Mary |
| 125 | Mark |
| 126 | Jane |

| Enrolment | |
|-----------|------|
| ID | Code |
| 123 | DBS |
| 124 | PRG |
| 124 | DBS |
| 126 | PRG |
| | |

SELECT * FROM

Student INNER JOIN Enrolment USING (ID)

INNER JOIN

Student ID Name 123 John Mary 124 125 Mark 126 Jane

Enrolment ID Code 123 DBS 124 PRG 124 DBS PRG 126

SELECT * FROM

Student INNER JOIN Enrolment USING (ID)

| ID | Name | Code |
|-----|------|------|
| 123 | John | DBS |
| 124 | Mary | PRG |
| 124 | Mary | DBS |
| 126 | Jane | PRG |

· A single ID row will be output representing the equal values from both Student.ID and Enrolment.ID

NATURAL JOIN

SELECT * FROM A NATURAL JOIN B

Is the same as

SELECT A.Col1, A.Col2, ..., A.Coln, [and all other columns except for B.Col1,...,B.Coln] FROM A, B WHERE A.Col1 = B.Col1

AND ... AND A.Coln = B.Coln • A NATURAL JOIN is effectively a special case of an INNER JOIN where the USING clause has specified all identically named columns

 It can be written as $A \bowtie B$

and used in relational algebra expressions

Student (S) ID Name 123 John 124 Mary 125 Mark 126 Jane

Enrolment (E) ID Code 123 DBS 124 PRG 124 DBS 126 PRG

NATURAL JOIN

SELECT * FROM Student NATURAL JOIN Enrolment;

NATURAL JOIN

Student (S)

| Stadent (S) | | |
|-------------|------|--|
| ID | Name | |
| 123 | John | |
| 124 | Mary | |
| 125 | Mark | |
| 126 | Jane | |

| Enrolment (E) | | |
|---------------|----------------|--|
| ID Code | | |
| 23 | DBS | |
| 24 | PRG | |
| 24 | DBS | |
| 26 | PRG | |
| | 23 24 24 | |

SELECT * FROM

Student NATURAL JOIN Enrolment;

| ID | Name | Code |
|-----|------|------|
| 123 | John | DBS |
| 124 | Mary | PRG |
| 124 | Mary | DBS |
| 126 | Jane | PRG |

 $\pi_{Id, Name}$ ($\sigma_{(Code=DBS)}$ (S \bowtie E))

JOINs vs WHERE Clauses

- JOINs are not absolutely Yes necessary
 - · You can obtain the same results by selecting from multiple tables and using appropriate WHERE clauses
 - · Should you use JOINs?
- - They often lead to concise and elegant queries
 - · NATURAL JOINs are extremely common
- No
 - · Support for JOINs can vary between DBMSs
 - · Might be easier with subqueries

Outer Joins

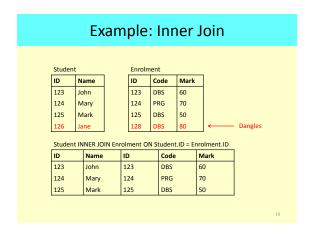
- When we take the join of two relations we match up tuples which share values
- · Some tuples have no match, and are 'lost'
- · These are called 'dangles'
- · Outer joins include dangles in the result and use NULLs to fill in the blanks
 - LEFT OUTER JOIN
 - RIGHT OUTER JOIN
 - FULL OUTER JOIN
- · Outer Joins use ON much like INNER JOIN

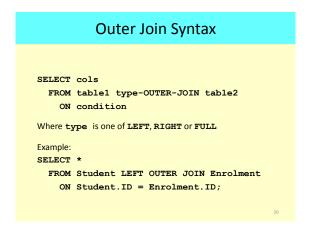
Example: Inner Join

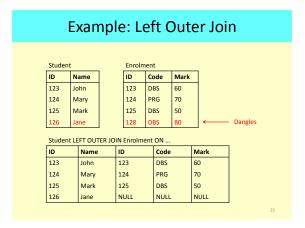
| Student | | | | |
|---------|------|--|--|--|
| ID | Name | | | |
| 123 | John | | | |
| 124 | Mary | | | |
| 125 | Mark | | | |
| 126 | Jane | | | |

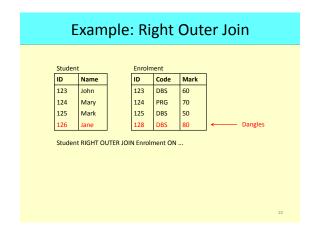
| Enrolment | | | | | | | |
|-----------|------|------|----------|--|--|--|--|
| ID | Code | Mark | | | | | |
| 123 | DBS | 60 | | | | | |
| 124 | PRG | 70 | | | | | |
| 125 | DBS | 50 | | | | | |
| 128 | DBS | 80 | ← | | | | |

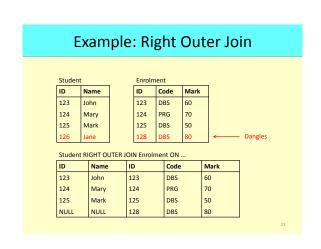
Student INNER JOIN Enrolment ON Student.ID = Enrolment.ID

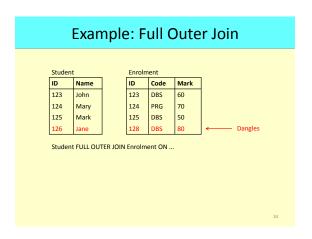












Example: Full Outer Join

 ID
 Name

 123
 John

 124
 Mary

 125
 Mark

 126
 Jane

| Enrolment | | | | | | |
|-----------|------|------|--|--|--|--|
| ID | Code | Mark | | | | |
| 123 | DBS | 60 | | | | |
| 124 | PRG | 70 | | | | |
| 125 | DBS | 50 | | | | |
| 128 | DBS | 80 | | | | |

Dangles

Student FULL OUTER JOIN Enrolment ON ..

| ID | Name | ID | Code | Mark |
|------|------|------|------|------|
| 123 | John | 123 | DBS | 60 |
| 124 | Mary | 124 | PRG | 70 |
| 125 | Mark | 125 | DBS | 50 |
| 126 | Jane | NULL | NULL | NULL |
| NULL | NULL | 128 | DBS | 80 |

Full Outer Join in MySQL

 Only Left and Right outer joins are supported in MySQL. If you really want a FULL outer join:

SELECT *

FROM Student FULL OUTER JOIN Enrolment
ON Student.ID = Enrolment.ID;

· Can be achieved using:

Full Outer Join in MySQL

 Only Left and Right outer joins are supported in MySQL. If you really want a FULL outer join:

SELECT *

FROM Student FULL OUTER JOIN Enrolment
ON Student.ID = Enrolment.ID;

Can be achieved using:

SELECT * FROM Student LEFT OUTER JOIN Enrolment ON Student.ID = Enrolment.ID UNION SELECT * FROM Student RIGHT OUTER JOIN

Enrolment ON Student.ID = Enrolment.ID;

Example

- Sometimes an outer join is the most practical approach. We may encounter NULL values, but may still wish to see the existing information
- For students graduating in absentia, find a list of all student IDs, names, addresses, phone numbers and their final degree classifications.

Example

| Student | | | | | | | |
|---------|------|------|------|------|--|--|--|
| ID | Name | alD | pID | Grad | | | |
| 123 | John | 12 | 22 | С | | | |
| 124 | Mary | 23 | 90 | Α | | | |
| 125 | Mark | 19 | NULL | Α | | | |
| 126 | Jane | 14 | 17 | С | | | |
| 127 | Sam | NULL | 101 | Α | | | |
| Degn | 90 | | | | | | |

| pID | pNumber | pMobile |
|-----|---------|-------------|
| 17 | 1111111 | 07856232411 |
| 22 | 2222222 | 07843223421 |
| 90 | 3333333 | 07155338654 |
| 101 | 4444444 | 07213559864 |

| Degree | | | | | |
|--------|----------------|--|--|--|--|
| ID | Classification | | | | |
| 123 | 1 | | | | |
| 124 | 2:1 | | | | |
| 125 | 2:2 | | | | |
| 126 | 2:1 | | | | |
| 127 | 3 | | | | |

| Address | | | | | | | | | |
|----------------|--|--|--|--|--|--|--|--|--|
| aStreet | aTown | aPostcode | | | | | | | |
| 5 Arnold Close | Nottingham | NG12 1DD | | | | | | | |
| 17 Derby Road | Nottingham | NG7 4FG | | | | | | | |
| 1 Main Street | Derby | DE1 5FS | | | | | | | |
| 7 Holly Avenue | Nottingham | NG6 7AR | | | | | | | |
| | aStreet 5 Arnold Close 17 Derby Road 1 Main Street | aStreet aTown 5 Arnold Close Nottingham 17 Derby Road Nottingham | | | | | | | |

Phone

Example: INNER JOINs

- An Inner Join with Student and Address will ignore Student 127, who doesn't have an address record
- An Inner Join with Student and Phone will ignore student 125, who doesn't have a phone record

Student

| ID | Name | aID | pID | Grad |
|-----|------|------|------|------|
| 123 | John | 12 | 22 | С |
| 124 | Mary | 23 | 90 | Α |
| 125 | Mark | 19 | NULL | Α |
| 126 | Jane | 14 | 17 | С |
| 127 | Sam | NULL | 101 | Α |

Example

SELECT ...

FROM Student LEFT OUTER JOIN Phone
ON Student.pID = Phone.pID
...

Student Phone

| ID | Name | alD | pID | Grad | pID | pNumber | pMobile |
|-----|------|------|------|------|------|---------|-------------|
| 123 | John | 12 | 22 | С | 22 | 222222 | 07843223421 |
| 124 | Mary | 23 | 90 | Α | 90 | 3333333 | 07155338654 |
| 125 | Mark | 19 | NULL | Α | NULL | NULL | NULL |
| 126 | Jane | 14 | 17 | С | 17 | 1111111 | 07856232411 |
| 127 | Sam | NULL | 101 | Α | 101 | 444444 | 07213559864 |

Example

SELECT ...

FROM Student LEFT OUTER JOIN Phone
ON Student.pID = Phone.pID

LEFT OUTER JOIN Address
ON Student.aID = Address.aID

...

| Stuc | lent | | | | Phone | | Address | | |
|------|------|------|------|------|---------|-------------|----------|-------|-----------|
| ID | Name | alD | pID | Grad | pNumber | pMobile | aStreet | aTown | aPostcode |
| 123 | John | 12 | 22 | С | 2222222 | 07843223421 | 5 Arnold | Notts | NG12 1DD |
| 124 | Mary | 23 | 90 | Α | 3333333 | 07155338654 | 7 Holly | Notts | NG6 7AR |
| 125 | Mark | 19 | NULL | Α | NULL | NULL | 1 Main | Derby | DE1 5FS |
| 126 | Jane | 14 | 17 | С | 1111111 | 07856232411 | 17 Derby | Notts | NG7 4FG |
| 127 | Sam | NULL | 101 | Α | 4444444 | 07213559864 | NULL | NULL | NULL |
| | | | | | | | | | 32 |

Example

SELECT ID, Name, aStreet, aTown, aPostcode, pNumber,
Classification

FROM Student LEFT OUTER JOIN Phone
ON Student.pID = Phone.pID

LEFT OUTER JOIN Address
ON Student.aID = Address.aID

INNER JOIN Degree ON Student.ID = Degree.ID

WHERE Grad = 'A';

Example

| ID | Name | aStreet | aTown | aPostcode | pNumber | Classification |
|-----|------|----------------|------------|-----------|---------|----------------|
| 124 | Mary | 7 Holly Avenue | Nottingham | NG6 7AR | 3333333 | 2:1 |
| 125 | Mark | 1 Main Street | Derby | DE1 5FS | NULL | 2:2 |
| 127 | Sam | NULL | NULL | NULL | 4444444 | 3 |

 The records for students 125 and 127 have been preserved despite missing information

Take home messages

Same results can be achieved in many ways

- 1. Subqueries can be used to simplify queries involving Cartesian Product
- 2. Subqueries better handle duplicates (compared to CP)
- Joins can be used as a more elegant way to writing queries involving Cartesian Product + WHERE Clause
- 4. Outer Join can handle NULLs

This Lecture

- More SQL SELECT
 - ORDER BY
- Aggregate functions
- GROUP BY and HAVING
- UNION
- Missing Information
 - Nulls and the Relational Model
 - Default Values

SQL SELECT Overview

SELECT

[DISTINCT | ALL] column-list FROM table-names [WHERE condition] [GROUP BY column-list] [HAVING condition] [ORDER BY column-list] ([] optional, | or)

ORDER BY

- · The ORDER BY clause sorts the results of a query
 - · You can sort in ascending (default) or descending
- Multiple columns can be
- · You cannot order by a column which isn't in the result

Grades

John DBS John IAI Mary DBS 60

James PR1 43 PR2 35

James Jane IAI

Name Code Mark

SELECT columns FROM tables WHERE condition ORDER BY cols [ASC | DESC]

Grades

Name Code Mark John DBS IAI 72 John DBS 60 Mary James PR1 43 James PR2 35

ORDER BY

SELECT * FROM Grades ORDER BY Mark;

ORDER BY

SELECT * FROM Grades ORDER BY Mark;

| Grades | | |
|--------|------|------|
| Name | Code | Mark |
| John | DBS | 56 |
| John | IAI | 72 |
| Mary | DBS | 60 |
| James | PR1 | 43 |
| James | PR2 | 35 |
| Jane | IAI | 54 |

| Name | Code | Mark |
|-------|------|------|
| James | PR2 | 35 |
| James | PR1 | 43 |
| Jane | IAI | 54 |
| John | DBS | 56 |

Mary DBS

John IAI

60

ORDER BY

SELECT * FROM Grades ORDER BY Code ASC,

| | | | Grades | | |
|------------|--|--|--------|------|-----|
| Mark DESC; | | | Name | Code | Ma |
| | | | John | DBS | 56 |
| | | | John | IAI | 72 |
| | | | Mary | DBS | 60 |
| | | | James | PR1 | 43 |
| | | | James | PR2 | 35 |
| | | | lane | IAI | 5/1 |

ORDER BY

SELECT * FROM Grades ORDER BY Code ASC, Mark DESC;

| Name | Code | Mark |
|-------|------|------|
| Mary | DBS | 60 |
| John | DBS | 56 |
| John | IAI | 72 |
| Jane | IAI | 54 |
| James | PR1 | 43 |
| James | PR2 | 35 |

Arithmetic

- · As well as columns, a SELECT statement can also be used to
 - Compute arithmetic expressions
 - · Evaluate functions
- · Often helpful to use an alias when dealing with expressions or functions

SELECT Mark / 100 FROM Grades;

SELECT Salary + Bonus FROM Employee;

SELECT 1.20 * Price AS 'Price inc. VAT' FROM Products;

Grades

John

John IAI 72

Mary DBS 60

James

Jane

James PR1

Name Code Mark

43

DBS

PR2 35

IAI 54

- Aggregate functions compute summaries of data in a table
 - Most aggregate functions (except COUNT (*)) work on a single column of numerical data
- · Again, it's best to use an alias to name the result

Aggregate Functions

- Aggregate functions . COUNT: The number of
 - . SUM: The sum of the
 - entries in the column · AVG: The average entry
 - in a column • MIN. MAX: The
 - minimum and maximum entries in a column

COUNT

SELECT

Grades

Name Code Mark John DBS 56 72 John IAI Mary DBS 60 James PR1 43 PR2 35 James

COUNT (*) AS Count FROM Grades;

SELECT COUNT (Code) AS Count

FROM Grades; SELECT

COUNT (DISTINCT Code) AS Count FROM Grades:

COUNT

Grades Name Code Mark DBS John John 72 Mary DBS 60 PR1 43 James PR2 35 James Jane 54

SELECT

COUNT(*) AS Count FROM Grades;

SELECT

COUNT (Code) AS Count FROM Grades;

SELECT

COUNT(DISTINCT Code) AS Count FROM Grades;

Count

Count

SUM, MIN/MAX and AVG

SELECT

SUM (Mark) AS Total FROM Grades:

SELECT

MAX(Mark) AS Best FROM Grades;

SELECT

AVG (Mark) AS Mean FROM Grades;

SUM, MIN/MAX and AVG

SELECT

Name Code Mark DBS 72 DBS 60 PR1 43 PR2 35

Grades

John

John

Mary

James

James

Jane

SUM(Mark) AS Total FROM Grades;

SELECT

MAX(Mark) AS Best FROM Grades;

SELECT

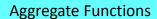
AVG (Mark) AS Mean FROM Grades;

Mean 53.33

Total

320

Best



 You can combine aggregate functions using arithmetic

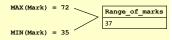
> Grades Name Code Mark John DBS 56 John 72 Mary DBS 60 James PR1 43 PR2 35 James Jane

SELECT

MAX(Mark) - MIN(Mark)

AS Range_of_marks

FROM Grades;



Example Modules Find John's average Code Title Credits mark, weighted by the DBS Database Systems credits of each module IAI Introduction to Al Grades SUM (Mark*Credits) / SUM (Credits) Code Mark AS 'Final Mark' John DBS 56 FROM Modules, Grades 72 John IAI WHERE Modules.Code=Grades.Code AND Grades.Name = 'John';

GROUP BY

- Sometimes we want to apply aggregate functions to groups of rows
- Example: find the average mark of each student individually
- The GROUP BY clause achieves this

SELECT cols1 FROM tables GROUP BY cols2;

GROUP BY

FROM tables
GROUP BY cols2;

- Every entry in 'cols1' should be in 'cols2', be a constant, or be an aggregate function
- You can have WHERE and ORDER BY clauses as well as a GROUP BY clause

GROUP BY

 Name
 Code
 Mark

 John
 DBS
 56

 John
 IAI
 72

 Mary
 DBS
 60

 James
 PR1
 43

 James
 PR2
 35

 Jane
 IAI
 54

SELECT Name,

AVG(Mark) AS Average

FROM Grades

GROUP BY Name;

GROUP BY

 Name
 Code
 Mark

 John
 DBS
 56

 John
 IAI
 72

 Mary
 DBS
 60

 James
 PR1
 43

 James
 PR2
 35

 Jane
 IAI
 54

SELECT Name,
AVG (Mark) AS Average
FROM Grades
GROUP BY Name;

Name Average
John 64
Mary 60
James 39
Jane 54

GROUP BY

Sales

| Month | Department | Value |
|-------|------------|-------|
| March | Fiction | 20 |
| March | Travel | 30 |
| March | Technical | 40 |
| April | Fiction | 10 |
| April | Fiction | 30 |
| April | Travel | 25 |
| April | Fiction | 20 |
| May | Fiction | 20 |
| May | Travel | 50 |

- · Find the total value of the sales for each department in each month
 - Can group by Month then Department or Department then Month

GROUP BY

SELECT Month, Department, SUM(Value) AS Total FROM Sales

GROUP BY Month, Department;

| Month | Department | Total |
|-------|------------|-------|
| April | Fiction | 60 |
| April | Travel | 25 |
| March | Fiction | 20 |
| March | Technical | 40 |
| March | Travel | 30 |
| May | Fiction | 20 |
| May | Technical | 50 |

SELECT Month, Department, SUM(Value) AS Total FROM Sales GROUP BY Department, Month;

| Month | Department | Total |
|-------|------------|-------|
| April | Fiction | 60 |
| March | Fiction | 20 |
| May | Fiction | 20 |
| March | Technical | 40 |
| May | Technical | 50 |
| April | Travel | 25 |
| March | Travel | 30 |

GROUP BY

Sales

| Month | Department | Value |
|-------|------------|-------|
| March | Fiction | 20 |
| March | Travel | 30 |
| March | Technical | 40 |
| April | Fiction | 10 |
| April | Fiction | 30 |
| April | Travel | 25 |
| April | Fiction | 20 |
| May | Fiction | 20 |
| May | Travel | 50 |

- · Find the total value of the sales for each department in each month
- Can group by Month then Department or Department then Month
- · Same results, but produced in a different order

GROUP BY Rules

- GROUP BY works slightly differently in MySQL than in other DBMSs.
- Usually, every column you name in your SELECT statement must also appear in your **GROUP BY clause. Apart** from those in Aggregate functions.
- For example:

SELECT ID, Name, AVG (Mark) FROM Students GROUP BY ID, Name;

GROUP BY Rules

- In MySQL, for convenience, you are allowed to break this rule.
- You are allowed to **GROUP BY a column** that won't appear in the output table
- Despite this, you should follow the ISO standard where possible
 - · Avoids problems if you use a different DBMS in the future
- Can lead to peculiar output where multiple values get output as one

GROUP BY Rules

• The MySQL extension means you do not need to GROUP BY every column you're SELECTing. It also means you don't have to SELECT a column even if it's in your GROUP BY clause:

SELECT artName, AVG(cdPrice) FROM Artist NATURAL JOIN CD GROUP BY artID;

GROUP BY Rules

• Be careful though, relaxed rules means you might get peculiar output if you're not careful:

SELECT artID, cdTitle, AVG(cdPrice) FROM Artist NATURAL JOIN CD GROUP BY artID;

GROUP BY Rules

 Be careful though, relaxed rules means you might get peculiar output if you're not careful:

SELECT artID, cdTitle, AVG(cdPrice) FROM Artist NATURAL JOIN CD GROUP BY artID;

| artID | cdTitle | AVG(cdPrice) |
|-------|-----------------------------|--------------|
| 1 | Black Holes and Revelations | 10.99 |
| 2 | Ninja Tuna | 9.99 |
| 3 | For Lack of a Better Name | 9.99 |
| 4 | Version | 11.99 |
| 5 | Record Collection | 12.99 |
| 6 | Merriweather Post Pavilion | 12.99 |
| 7 | Only By The Night | 11.49 |
| 8 | Hands All Over | 11.99 |

GROUP BY Rules

 Be careful though, relaxed rules means you might get peculiar output if you're not careful:

SELECT artID, cdTitle, AVG(cdPrice) FROM Artist NATURAL JOIN CD GROUP BY artID;

| artID | cdTitle | AVG(cdPrice) | |
|-------|-----------------------------|--------------|--------|
| 1 | Black Holes and Revelations | 10.99 | R |
| 2 | Ninja Tuna | 9.99 | |
| 3 | For Lack of a Better Name | 9.99 | |
| 4 | Version | 11.99 | WRONG! |
| 5 | Record Collection | 12.99 | / |
| 6 | Merriweather Post Pavilion | 12.99 | |
| 7 | Only By The Night | 11.49 | 2 |
| 8 | Hands All Over | 11.99 | 63 |

GROUP BY Rules

What's the best way? Instead of:
 SELECT artName, AVG(cdPrice)
 FROM Artist NATURAL JOIN CD
 GROUP BY artID;

Try:

SELECT artName, Average
FROM (SELECT artID, artName,
AVG(cdPrice) AS Average
FROM Artist NATURAL JOIN CD
GROUP BY artID) AS SubTable;

HAVING

- HAVING is like a WHERE clause, except that it only applies to the results of a GROUP BY query
- It can be used to select groups which satisfy a given condition

SELECT Name,
AVG (Mark) AS Average
FROM Grades
GROUP BY Name
HAVING AVG (Mark) >= 40;

HAVING

- HAVING is like a WHERE clause, except that it only applies to the results of a GROUP BY query
- It can be used to select groups which satisfy a given condition

SELECT Name,
AVG (Mark) AS Average
FROM Grades
GROUP BY Name
HAVING AVG (Mark) >= 40;

| Name | Average |
|------|---------|
| John | 64 |
| Mary | 60 |
| lano | E4 |

WHERE and HAVING

- WHERE refers to the rows of tables, so cannot make use of aggregate functions
- HAVING refers to the groups of rows, and so cannot use columns which are not in the GROUP BY or an aggregate function
- · Think of a query being processed as follows:
 - · Tables are joined
 - where clauses
 - · GROUP BY clauses and aggregates
 - · Column selection
 - HAVING clauses
 - · ORDER BY

UNION

- **EXCEPT**
- · These treat the tables as sets and are the usual set operators of union, intersection and difference
- · We'll be concentrating on UNION

• UNION, INTERSECT and • They all combine the results from two select statements

- UNION
- UNION, INTERSECT and **EXCEPT**
 - · These treat the tables as sets and are the usual set operators of union, intersection and difference
- · We'll be concentrating on UNION
- They all combine the results from two select statements
- The results of the two selects should have the same columns and corresponding data types

UNION

Grades

| Grades | | |
|--------|------|------|
| Name | Code | Mark |
| Jane | IAI | 54 |
| John | DBS | 56 |
| John | IAI | 72 |
| James | PR1 | 43 |
| James | PR2 | 35 |
| Mary | DBS | 60 |
| | | |

 Find, in a single query, the average mark for each student and the average mark overall

UNION

· The average for each student:

GROUP BY Name;

SELECT Name, AVG (Mark) AS Average FROM Grades

The average overall

SELECT

'Total' AS Name, AVG (Mark) AS Average FROM Grades:

• Note - this has the same columns as average by student

UNION

SELECT Name AVG (Mark) AS Average FROM Grades GROUP BY Name

UNION

SELECT

'Total' AS Name, AVG (Mark) AS Average FROM Grades;

Name Average James 39 Jane John 64 Mary 60 Total 53.3333

Final SELECT Example

- Examiners' reports
 - · We want a list of students and their average mark
 - · For first and second years the average is for that year
 - · For finalists it is 40% of the second year plus 60% of the final year averages

- We want the results
 - Sorted by year (desc) then average mark (high to low) then last name, first name and finally ID
 - · To take into account of the number of credits each module is worth
 - · Produced by a single query

Example Output

| Year | Student.ID | Last | First | AverageMark |
|------|------------|----------|------------|-------------|
| 3 | 11014456 | Andrews | John | 81 |
| 3 | 11013891 | Smith | Mary | 78 |
| 3 | 11014012 | Brown | Amy | 76 |
| 3 | 11013204 | Jones | Steven | 76 |
| 3 | 11014919 | Robinson | Paul | 74 |
| 2 | 34012703 | Equacte | Robert | 73 |
| 7 | 11027871 | Green | TVTICTIOCI | |
| 1 | 11024298 | Hall | David | 43 |
| 1 | 11024826 | Wood | James | 40 |
| | | | | |

Clarke

Wilson

Taylor

Williams

Sarah

Paul

Matthew 34

Tables for the Example Student ID First Last Year Grade ID Code Mark YearTaken Module Title Credits Code

Getting Started

- · Finalists should be treated differently to other years
 - · Write one SELECT for the finalists
 - · Write a second SELECT for the first and second
 - · Merge the results using a UNION

UNION

QUERY FOR OTHERS

QUERY FOR FINALISTS

Table Joins

 Both subqueries need information from all the tables

11027621

11024978

11026563

11027625

- · The student ID, name and year
- The marks for each module and the year
- The number of credits for each module

- This is a natural join operation
 - · But because we're practicing, we're going to use a standard CROSS JOIN and WHERE clause
 - · Exercise: repeat the query using natural join

The Query So Far

SELECT some-information FROM Student, Module, Grade WHERE Student.ID = Grade.ID AND Module.Code = Grade.Code AND student-is-in-third-year

UNION

SELECT some-information FROM Student, Module, Grade WHERE Student.ID = Grade.ID AND Module.Code = Grade.Code AND student-is-in-first-or-second-year;

Information for Finalists

- · We must retrieve
 - Computed average mark, weighted 40-60 across years 2 and 3
 - First year marks must be ignored
 - The ID, Name and Year are needed as they are used for ordering
- · The average is difficult
 - We don't have any statements to separate years 2 and 3 easily
 - We can exploit the fact that 40 = 20 * 2 and 60 = 20 * 3, so YearTaken and the weighting have the same relationship

The Query So Far

SELECT some-information
FROM Student, Module, Grade
WHERE Student.ID = Grade.ID
AND Module.Code = Grade.Code
AND student-is-in-third-year

UNION

. . .

Information for Finalists

Information for Others

- Other students are easier than finalists
 - We just need their average marks where YearTaken and Year are the same
 - As before, we need ID, Name and Year for ordering

The Query So Far

UNION

SELECT some-information
FROM Student, Module, Grade
WHERE Student.ID = Grade.ID
AND Module.Code = Grade.Code
AND student-is-in-first-or-second-year;

Information for Others

SELECT Year, Student.ID, Last, First,
SUM(Mark*Credits)/120 AS AverageMark
FROM Student, Module, Grade
WHERE Student.ID = Grade.ID
AND Module.Code = Grade.Code
AND YearTaken = Year
AND Year IN (1,2)
GROUP BY Year, Student.ID, First, Last

The Final Query

SELECT Year, Student.ID, Last, First,
SUM(((20*YearTaken)/100)*Mark*Credits)/120 AS AverageMark
FROM Student, Module, Grade
NHERE Student.ID = Grade.ID AND Module.Code = Grade.Code
AND YearTaken IN (2,3)
AND Year = 3
GROUP BY Year, Student.ID, Last, First
UNION

SELECT Year, Student.ID, Last, First, SUM(Mark*Credits)/120 AS AverageMark
FROM Student, Module, Grade
WHERE Student.ID = Grade.ID AND Module.Code = Grade.Code
AND YearTaken = Year
AND Year IN (1,2)
GROUP BY Year, Student.ID, Last, First
GROUP BY Year, Student.ID, Last, First
GROUP BY Year, Student.ID, Last, First

Take home messages

- 1. Joins as alternatives to Cross Product
- 2. ORDER BY to produce ordered output
- 3. Aggregate functions
 - a. MIN, MAX, SUM, AVG, COUNT
- 4. GROUP BY
- 5. HAVING
- 6. UNION between SELECT statements
 - a. Usual set rules apply
- 7. Same results can be achieved in many ways

Missing Information

Missing Information

- Sometimes we don't know what value an entry in a relation should have
 - We know that there is a value, but don't know what it is
 - There is no value at all that makes any sense
- Two main methods have been proposed to deal with this
 - NULLs can be used as markers to show that information is missing
 - A default value can be used to represent the missing value

NULLs

- NULL is a placeholder for missing or unknown value of an attribute. It is not itself a value.
- Codd proposed to distinguish two types of NULLs:
 - A-marks: data Applicable but not known (for example, someone's age)
 - I-marks: data is Inapplicable (telephone number for someone who does not have a telephone, or spouse's name for someone who is not married)

Problems with NULLs

- Problems extending relational algebra operations to NULLs:
 - Selection operation: if we check tuples for "Mark > 40" and for some tuple Mark is NULL, do we include it?
- Comparing tuples in two relations: are two tuples <John, NULL> and <John, NULL> the same or not?
- Additional problems for SQL:
 - NULLs treated as duplicates?
 - Inclusion of NULLs in count, sum, average? If yes, how?
 - · Arithmetic operations behaviour with argument NULL?

Theoretical Solutions

- Use three-valued logic instead of classical twovalued logic to evaluate conditions.
- This is the idea behind testing conditions in WHERE clause of SQL SELECT: only tuples where the condition evaluates to true are returned.
- When there are no NULLs around, conditions evaluate to true or false, but if a null is involved, a condition might evaluate to the third value ('undefined', or 'unknown').

3-valued logic

 If the condition involves a boolean combination, we evaluate it as follows:

| а | b | a OR b | a AND b | a == b |
|---------|---------|---------|---------|---------|
| True | True | True | True | True |
| True | False | True | False | False |
| True | Unknown | True | Unknown | Unknown |
| False | True | True | False | False |
| False | False | False | False | True |
| False | Unknown | Unknown | False | Unknown |
| Unknown | True | True | Unknown | Unknown |
| Unknown | False | Unknown | False | Unknown |
| Unknown | Unknown | Unknown | Unknown | Unknown |

SQL NULLs in Conditions

SELECT *
FROM Employee
Where Salary > 15,000;

Name Salary
John 25,000

John 25,000 Mark 15,000 Anne 20,000 Chris NULL

SQL NULLs in Conditions

SELECT *
FROM Employee
Where Salary > 15,000;

Name Salary
John 25,000
Mark 15,000
Anne 20,000
Chris NULL

Salary > 15,000
 evaluates to 'unknown'
 on the last tuple – not
 included

Name Salary
John 25,000
Anne 20,000

SQL NULLs in Conditions

SELECT *
FROM Employee
Where Salary > 15,000
OR Name = 'Chris';

 Employee

 Name
 Salary

 John
 25,000

 Mark
 15,000

 Anne
 20,000

 Chris
 NULL

SQL NULLs in Conditions

SELECT *
FROM Employee
Where Salary > 15,000
OR Name = 'Chris';

Name Salary
John 25,000
Mark 15,000
Anne 20,000
Chris NULL

 Salary > 15,000 OR Name = 'Chris' is essentially Unknown OR TRUE on the last tuple

a b a OR b
Unknown True True

SQL NULLs in Conditions

SELECT *
FROM Employee

Where Salary > 15,000 OR Name = 'Chris';

Employee

| Name | Salary |
|-------|--------|
| John | 25,000 |
| Mark | 15,000 |
| Anne | 20,000 |
| Chris | NULL |

 Salary > 15,000 OR Name = 'Chris' is essentially Unknown OR TRUE on the last tuple

| Name | Salary |
|-------|--------|
| John | 25,000 |
| Anne | 20,000 |
| Chris | NULL |

SQL NULLs in Arithmetic

SELECT

Name,
Salary * 0.05 AS Bonus
FROM Employee;

Employee

| Name | Salary | |
|-------|--------|--|
| John | 25,000 | |
| Mark | 15,000 | |
| Anne | 20,000 | |
| Chris | NULL | |

SQL NULLs in Arithmetic

SELECT Name,

Salary * 0.05 AS Bonus FROM Employee;

Employee

| Name | Salary |
|-------|--------|
| John | 25,000 |
| Mark | 15,000 |
| Anne | 20,000 |
| Chris | NULL |

 Arithmetic operations applied to NULLs result in NULLS

| Name | Bonus | |
|-------|-------|--|
| John | 1,250 | |
| Mark | 750 | |
| Anne | 1,000 | |
| Chris | NULL | |

SQL NULLs in Aggregation

SELECT

AVG(Salary) AS Average,

COUNT(Salary) AS Count,

SUM(Salary) AS Sum

FROM Employee;

Employee

| Name | Salary |
|-------|--------|
| John | 25,000 |
| Mark | 15,000 |
| Anne | 20,000 |
| Chris | NULL |

SQL NULLs in Aggregation

SELECT

AVG(Salary) AS Average, COUNT(Salary) AS Count, SUM(Salary) AS Sum

FROM Employee;

 Name
 Salary

 John
 25,000

 Mark
 15,000

 Anne
 20,000

 Chris
 NULL

- Average = 20,000
- Count = 3
- Sum = 60,000
- Using COUNT(*) would give 4

SQL NULLs in GROUP BY

SELECT Salary,

COUNT (Name) AS Count FROM Employee

GROUP BY Salary;

Sam

Chris

Imployee

Name Salary

John 25,000

Mark 15,000

Anne 20,000

Jack NULL

20,000 NULL

SQL NULLs in GROUP BY

SELECT
Salary,
COUNT(Name) AS Count
FROM Employee
GROUP BY Salary;

 Imployee

 Name
 Salary

 John
 25,000

 Mark
 15,000

 Anne
 20,000

 Jack
 NULL

 Sam
 20,000

 Chris
 NULL

 NULLs are treated as equivalents in GROUP BY clauses

| Salary | Count |
|--------|-------|
| NULL | 2 |
| 15,000 | 1 |
| 20,000 | 2 |
| 25,000 | 1 |
| | |

SQL NULLs in ORDER BY

SELECT *
FROM Employee
ORDER BY Salary;

| Employee | | |
|----------|-------|--------|
| | Name | Salary |
| | John | 25,000 |
| | Mark | 15,000 |
| | Anne | 20,000 |
| | Jack | NULL |
| | Sam | 20,000 |
| | Chris | NULL |

SQL NULLs in ORDER BY

SELECT *
FROM Employee
ORDER BY Salary;

 Employee

 Name
 Salary

 John
 25,000

 Mark
 15,000

 Anne
 20,000

 Jack
 NULL

 Sam
 20,000

 Chris
 NULL

 NULLs are considered and reported in ORDER BY clauses

Employee

Name Salary

Chris NULL

Jack NULL

Mark 15,000

Anne 20,000

Sam 20,000

John 25,000

Missing Information

- Sometimes we don't know what value an entry in a relation should have
 - We know that there is a value, but don't know what it is
 - There is no value at al that makes any sense
- Two main methods have been proposed to deal with this
 - NULLs can be used as markers to show that information is missing
 - A default value can be used to represent the missing value

Default Values

- Default values are an alternative to the use of NULLs
 - If a value is not known a particular placeholder value - the default - is used
- These are actual values.
- Default values can have more meaning than NULLs
 - 'none'
- 'unknown'
- · 'not supplied'
- · 'not applicable'
- Not all defaults represent missing information. It depends on the situation

Default Value Example

Parts Name Weight Quantity Nut Bolt 15 3 Nail 100 Pin 30 Unknow 20 20 Screw 150 Brace

- Default values are
 - "Unknown" for Name
 - -1 for Weight and Quantity
- -1 is used for Wgt and Qty as it is not sensible otherwise
- There are still problems: UPDATE Parts SET Quantity =

Quantity + 5

Problems With Default Values

- · Since defaults are real values
 - · They can be updated like any other value
 - · You need to use a value that won't appear in any other circumstances
 - · They might not be interpreted properly
- · Also, within SQL defaults must be of the same type as the column
 - · You can't have have a string such as 'unknown' in a column of integers

Splitting Tables

- · NULLs and defaults both try to fill entries with missing data
- · NULLs mark the data as missing
- · Defaults give some indication as to what sort of missing information we are dealing with
- · Often you can remove entries that have missing data
- · You can split the table up so that columns which might have NULLs are in separate tables
- Entries that would be NULL are not present in these tables

Splitting Tables Example Weights ID Weight 10 Parts 15 ID Name Weight ID Name Nut 10 Nut 20 Bolt 15 NULL Bolt 150 100 Nail Nail Pin NULL 30 Quantities Pin NULL 20 20 ID Quantity Screw NULL NULL Screw 20 Brace 150 100 30 20

Problems with Splitting Tables

- Splitting tables has other problems
 - · Could introduce many more tables
 - · Information gets spread out over the database
 - · Queries become more complex and require many joins
- · We can recover the original table, but
 - · Requires Outer Joins
 - · Reintroduces the NULL values, which means we're back to the original problem

SQL Support

- · SQL allows both NULLs and defaults:
 - A table to hold data on employees
 - · All employees have a name
 - · All employees have a salary (default 10000)
 - · Some employees have phone numbers, if not we use NULLs

CREATE TABLE Employee Name VARCHAR (50) NOT NULL, Salary INT DEFAULT 10000 NOT NULL, Phone VARCHAR (15) NULL);

SQL Support

• SQL allows you to insert • You can also check for **NULLs**

NULLs

INSERT INTO Employee VALUES ('John', 12000, NULL); SELECT Name FROM Employee WHERE Phone IS NULL;

UPDATE Employee SET Phone = NULL WHERE Name = 'Mark'; SELECT Name FROM Employee WHERE

Phone IS NOT NULL;

Which Method to Use?

- Most often dependent on the scenario
 - · Default values should not be used when they might be confused with 'real' values
 - Splitting tables shouldn't be used too much or you'll have lots of tables
- NULLs can (and often are) used where the other approaches seem inappropriate
- · You don't have to always use the same method you can mix and match as needed

Example

- · For an online store we have a variety of products - books, CDs, and DVDs
 - · All items have a title, price, and id (their catalogue number)
 - · Any item might have an extra shipping cost, but some don't

- There is also some data specific to each type
 - · Books must have an author and might have a publisher
 - · CDs must have an artist
 - · DVDs might have a producer or director

Example

• We could put all the data in one table

ID Title Price Shipping Author Publisher Artist Producer Director

- Every row will have missing information
- We are storing three types of thing in one table

Example

- It is probably best to split the three types into separate tables
 - · We'll have a main Items table
 - · Also have Books, CDs, and DVDs tables with FKs to the Items table

ID Title Price Shipping Books ID Author Publisher ID Artist

DVDs ID Producer Director

Example

- Each of these tables might still have some missing information
 - · Shipping cost in items could have a default value of 0
 - · This should not disrupt computations
 - · If no value is given, shipping is free

- · Other columns could allow NULLs
 - · Publisher, director, and producer are all optional
 - · It is unlikely we'll ever use them in computation

Thanks for your attention!

Any questions??