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!

2

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

3

Joins

- JOINs can be used to combine tables in a SELECT query
  - There are numerous types 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

A INNER JOIN B
- 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

4

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**.

5

CROSS JOIN

Student

ID	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
```

6

CROSS JOIN

Student

ID	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

7

INNER JOIN

- **INNER JOIN** specifies a condition that pairs of rows must satisfy

```
SELECT *
  FROM A INNER JOIN B
  ON condition
```

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

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

- **col1** and **col2** must appear in both **A** and **B**

8

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
```

9

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

10

INNER JOIN

Student

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)
```

11

INNER JOIN

Student

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)
```

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

12

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

NATURAL JOIN

Student (S)

ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment (E)

ID	Code
123	DBS
124	PRG
124	DBS
126	PRG

```
SELECT * FROM
  Student NATURAL JOIN
  Enrolment;
```


NATURAL JOIN

Student (S)

ID	Name
123	John
124	Mary
125	Mark
126	Jane

Enrolment (E)

ID	Code
123	DBS
124	PRG
124	DBS
126	PRG

```
SELECT * FROM
  Student NATURAL JOIN
  Enrolment;
```

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

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

JOINs vs WHERE Clauses

- JOINs are not absolutely necessary
  - You can obtain the same results by selecting from multiple tables and using appropriate WHERE clauses
  - Should you use JOINs?
- Yes
  - 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 sub-queries

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

Dangles

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

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

Dangles

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

ID	Name	ID	Code	Mark
123	John	123	DBS	60
124	Mary	124	PRG	70
125	Mark	125	DBS	50

Outer Join Syntax

```
SELECT cols
  FROM table1 type-OUTER-JOIN table2
    ON condition
```

Where **type** is one of **LEFT**, **RIGHT** or **FULL**

Example:

```
SELECT *
  FROM Student LEFT OUTER JOIN Enrolment
    ON Student.ID = Enrolment.ID;
```


Example: Left Outer 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

Dangles

Student LEFT 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

Example: Right Outer 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

Dangles

Student RIGHT OUTER JOIN Enrolment ON ...

Example: Right Outer 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

Dangles

Student RIGHT OUTER JOIN Enrolment ON ...

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

Example: Full Outer 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

Dangles

Student FULL OUTER JOIN Enrolment ON ...

Example: Full Outer 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

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

25

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:

26

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;
```

27

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.

28

Example

Student

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

Phone

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

aID	aStreet	aTown	aPostcode
12	5 Arnold Close	Nottingham	NG12 1DD
14	17 Derby Road	Nottingham	NG7 4FG
19	1 Main Street	Derby	DE1 5FS
23	7 Holly Avenue	Nottingham	NG6 7AR

29

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	C
124	Mary	23	90	A
125	Mark	19	NULL	A
126	Jane	14	17	C
127	Sam	NULL	101	A

30

Example

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

Student / Phone

ID	Name	aID	pID	Grad	pID	pNumber	pMobile
123	John	12	22	C	22	2222222	07843223421
124	Mary	23	90	A	90	3333333	07155338654
125	Mark	19	NULL	A	NULL	NULL	NULL
126	Jane	14	17	C	17	1111111	07856232411
127	Sam	NULL	101	A	101	4444444	07213559864

Example

```
SELECT ...
  FROM Student LEFT OUTER JOIN Phone
     ON Student.pID = Phone.pID
  LEFT OUTER JOIN Address
     ON Student.aID = Address.aID
  ...
```

Student / Phone / Address

ID	Name	aID	pID	Grad	pNumber	pMobile	aStreet	aTown	aPostcode
123	John	12	22	C	2222222	07843223421	5 Arnold...	Notts	NG12 1DD
124	Mary	23	90	A	3333333	07155338654	7 Holly...	Notts	NG6 7AR
125	Mark	19	NULL	A	NULL	NULL	1 Main...	Derby	DE1 5FS
126	Jane	14	17	C	1111111	07856232411	17 Derby...	Notts	NG7 4FG
127	Sam	NULL	101	A	4444444	07213559864	NULL	NULL	NULL

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)
3. 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*)

37

ORDER BY

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

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

38

ORDER BY

Grades

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

```
SELECT * FROM Grades
  ORDER BY Mark;
```

39

ORDER BY

Grades

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

```
SELECT * FROM Grades
  ORDER BY Mark;
```

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

40

ORDER BY

Grades

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

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

41

ORDER BY

Grades

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

```
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

42

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;
```


Aggregate Functions

- 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
  - **COUNT**: The number of rows
  - **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

Grades

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

```
SELECT
  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
John	DBS	56
John	IAI	72
Mary	DBS	60
James	PR1	43
James	PR2	35
Jane	IAI	54

```
SELECT
  COUNT(*) AS Count
  FROM Grades;
```

Count
6

```
SELECT
  COUNT(Code)
    AS Count
  FROM Grades;
```

Count
6

```
SELECT
  COUNT(DISTINCT Code)
    AS Count
  FROM Grades;
```

Count
4

SUM, MIN/MAX and AVG

Grades

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

```
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

Grades

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

```
SELECT
  SUM(Mark) AS Total
  FROM Grades;
```

Total
320

```
SELECT
  MAX(Mark) AS Best
  FROM Grades;
```

Best
72

```
SELECT
  AVG(Mark) AS Mean
  FROM Grades;
```

Mean
53.33

Aggregate Functions

- You can combine aggregate functions using arithmetic

```
SELECT
  MAX(Mark) - MIN(Mark)
    AS Range_of_marks
  FROM Grades;
```

Grades

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

```
MAX(Mark) = 72
MIN(Mark) = 35
```

Range_of_marks
37

Example

Modules

Code	Title	Credits
DBS	Database Systems	10
IAI	Introduction to AI	20
PRG	Programming	10

Grades

Name	Code	Mark
John	DBS	56
John	IAI	72
Mary	DBS	60

- Find John's average mark, weighted by the credits of each module

```
SELECT
  SUM(Mark*Credits)
    / SUM (Credits)
  AS 'Final Mark'
 FROM Modules, Grades
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

```
SELECT cols1
 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

Grades

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

Grades

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
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

WRONG!

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
Jane	54

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`

67

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

68

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

69

UNION

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

70

UNION

- The average for each student:

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

- The average overall

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

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

71

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	54
John	64
Mary	60
Total	53.3333

72

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
[illegible]	[illegible]	Edwards	Robert	73
		⋮		
1	11027871	Green	Michael	[illegible]
1	11024298	Hall	David	43
1	11024826	Wood	James	40
1	11027621	Clarke	Stewart	39
1	11024978	Wilson	Sarah	36
1	11026563	Taylor	Matthew	34
1	11027625	Williams	Paul	31

Tables for the Example

Student

ID	First	Last	Year

Grade

ID	Code	Mark	YearTaken

Module

Code	Title	Credits

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 years
  - Merge the results using a UNION

```
QUERY FOR FINALISTS

UNION

QUERY FOR OTHERS
```


Table Joins

- Both subqueries need information from all the tables
  - The student ID, name and year
  - The marks for each module and the year taken
  - 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

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


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
WHERE 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

ORDER BY Year desc, AverageMark desc, Last, First, ID;
```


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 two-valued 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').

91

3-valued logic

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

a	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

92

SQL NULLs in Conditions

```
SELECT *
 FROM Employee
 Where Salary > 15,000;
```

Employee

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

93

SQL NULLs in Conditions

```
SELECT *
 FROM Employee
 Where Salary > 15,000;
```

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

Employee

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

Name	Salary
John	25,000
Anne	20,000

94

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

95

SQL NULLs in Conditions

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

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

Employee

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

a	b	a OR b
Unknown	True	True

96

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

97

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

98

SQL NULLs in Arithmetic

```
SELECT
 Name,
 Salary * 0.05 AS Bonus
 FROM Employee;
```

- Arithmetic operations applied to NULLs result in NULLS

Employee

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

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

99

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

100

SQL NULLs in Aggregation

```
SELECT
 AVG(Salary) AS Average,
 COUNT(Salary) AS Count,
 SUM(Salary) AS Sum
 FROM Employee;
```

- Average = 20,000
- Count = 3
- Sum = 60,000

- Using COUNT(*) would give 4

Employee

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

101

SQL NULLs in GROUP BY

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

Employee

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

102

SQL NULLs in GROUP BY

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

Employee

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

103

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

104

SQL NULLs in ORDER BY

```
SELECT *
FROM Employee
ORDER BY Salary;
```

- NULLs are considered and reported in ORDER BY clauses

Employee

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

Employee

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

105

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

106

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

107

Default Value Example

Parts

ID	Name	Weight	Quantity
1	Nut	10	20
2	Bolt	15	-1
3	Nail	3	100
4	Pin	-1	30
5	Unknown	20	20
6	Screw	-1	-1
7	Brace	150	0

- 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
```

108

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

Parts

ID	Name	Weight	Quantity
1	Nut	10	20
2	Bolt	15	NULL
3	Nail	3	100
4	Pin	NULL	30
5	NULL	20	20
6	Screw	NULL	NULL
7	Brace	150	0

→

Names

ID	Name
1	Nut
2	Bolt
3	Nail
4	Pin
6	Screw
7	Brace

Weights

ID	Weight
1	10
2	15
3	3
5	20
7	150

Quantities

ID	Quantity
1	20
3	100
4	30
5	20
7	0

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 NULLs

```
INSERT INTO Employee
 VALUES ('John',
        12000, NULL);

UPDATE Employee
 SET Phone = NULL
 WHERE Name = 'Mark';
```

- You can also check for NULLs

```
SELECT Name FROM
 Employee WHERE
 Phone IS NULL;

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

Items

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

Items

ID	Title	Price	Shipping

Books

ID	Author	Publisher

CDs

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??