SQL SELECT III

G51DBS Database Systems

Jason Atkin

Last Lecture

- More SQL SELECT
  - Aliases
  - ‘Self-Joins’
  - Subqueries
  - IN, EXISTS, ANY, ALL
  - LIKE

Example 2 from last week

Student

sID	sName	sAddress	sYear
1	Smith	5 Arnold Close	2
2	Brooks	7 Holly Avenue	2
3	Anderson	15 Main Street	3
4	Evans	Flat 1a, High Street	2
5	Harrison	Newark Hall	1
6	Jones	Southwell Hall	1

Find a list of the names of any students who are enrolled on at least one module alongside 'Evans'

Module

mCode	mCredits	mTitle
G51DBS	10	Database Systems
G51PRG	20	Programming
G51IAI	10	Artificial Intelligence
G52ADS	10	Algorithms

Enrolment

sID	mCode
1	G52ADS
2	G52ADS
5	G51DBS
5	G51PRG
5	G51IAI
4	G52ADS
6	G51PRG
6	G51IAI

Student NATURAL JOIN Enrolment

Student NATURAL JOIN Enrolment

sID	sName	sAddress	sYear	mCode
1	Smith	5 Arnold Close	2	G52ADS
2	Brooks	7 Holly Avenue	2	G52ADS
~~3~~	~~Anderson~~	~~15 Main Street~~	~~3~~	
4	Evans	Flat 1a, High Street	2	G52ADS
5	Harrison	Newark Hall	1	G51DBS
5	Harrison	Newark Hall	1	G51PRG
5	Harrison	Newark Hall	1	G51IAI
6	Jones	Southwell Hall	1	G51PRG
6	Jones	Southwell Hall	1	G51IAI

Question did not say we could not use a join

Example 2: 2 joins and a sub-query

- Find a list of the names of any students who are enrolled on at least one module alongside 'Evans'

```
SELECT DISTINCT S1.sName, E1.mCode
FROM
Student S1 NATURAL JOIN Enrolment E1
WHERE E1.mCode IN
( SELECT E2.mCode FROM
Enrolment E2 NATURAL JOIN Student S2
WHERE S2.sName = 'Evans' );
```

Could also join the two sub-queries on **`E1.mCode = E2.mCode`**

Example 2: Three joins

- Find a list of the names of any students who are enrolled on at least one module alongside ‘Evans’

```
SELECT DISTINCT S1.sName, E1.mCode
FROM
(Student S1 NATURAL JOIN Enrolment E1)
INNER JOIN
(Student S2 NATURAL JOIN Enrolment E2)
USING (mCode)
WHERE S2.sName = 'Evans';
```

Cannot NATURAL JOIN the parts here! (sName, mCode etc would match)

Example 2: Sub queries, get names

```
SELECT DISTINCT S1.sName FROM Student S1
  WHERE S1.sID IN
    ( SELECT E1.sID FROM Enrolment E1
      WHERE E1.mCode IN
        ( SELECT E2.mCode FROM Enrolment E2
          WHERE E2.sID IN
            ( SELECT S2.sID FROM Student S2
              WHERE S2.sName = 'Evans') ) );
```

As long as we only want the student name, not the module id (mCode not in table)

Coursework Reminder

- Don’t forget the coursework
  - Create an ER diagram
  - Create the SQL to create the tables
  - Create the SQL to populate the tables
- **Deadline: 4pm Thursday 13th March**
- Hopefully you have had/will have had a tutorial with your tutor by then

This Lecture

- More SQL SELECT
  - ORDER BY
  - Aggregate functions
  - GROUP BY and HAVING
  - UNION
- Further reading
  - The Manga Guide to Databases, Chapter 4
  - Database Systems, Chapter 6

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

ORDER BY

- The ORDER BY clause sorts the results of a query
  - You can sort in ascending or descending order
  - Defaults to ascending if neither is specified
  - 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]
```

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	60
John	IAI	72

ORDER BY

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

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

Constants and Arithmetic

Constants and Arithmetic

- As well as columns, a SELECT statement can also be used to
  - Select constants
  - 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 1.175 * Price
  AS ‘Price inc. VAT’
  FROM Products
```

Aggregate Functions

- Aggregate functions compute summaries of data in a table
  - Most aggregate functions work on a single column of numerical data
  - **COUNT (*)** works on the entire row
- It is good 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 : guess the results…

Grades

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

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

Count
?

```
SELECT
  COUNT(Mark)
    AS Count
  FROM Grades
```

Count
?

```
SELECT
  COUNT(DISTINCT Mark)
    AS Count
  FROM Grades
```

Count
?

COUNT

Grades

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

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

Count
7

7 rows

```
SELECT
  COUNT(Mark)
     AS Count
  FROM Grades
```

Count
6

6 non-NULL marks

```
SELECT
  COUNT(DISTINCT Mark)
     AS Count
  FROM Grades
```

Count
5

5 different
non-NULL marks

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

Note: Work on ONE column. Ignore NULL values.

Aggregate Functions

- You can combine aggregate functions using arithmetic

```
SELECT
  MAX(Mark) – MIN(Mark)
    AS Range
  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
37

Example

Modules

Code	Title	Credits
DBS	Database Systems	10
GRP	Group Project	20

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

Grades

Name	Code	Mark
John	DBS	56
John	IAI	72

```
SELECT
      ???

FROM ???

WHERE ???
```

Example

Modules

Code	Title	Credits
DBS	Database Systems	10
GRP	Group Project	20

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

Grades

Name	Code	Mark
John	DBS	56
John	IAI	72

```
SELECT
      ???

FROM Modules, Grades

WHERE Modules.Code = Grades.Code
  AND Grades.Name = ‘John’
```

Example

Modules

Code	Title	Credits
DBS	Database Systems	10
GRP	Group Project	20

Grades

Name	Code	Mark
John	DBS	56
John	IAI	72

- 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

GROUP BY

- Sometimes we want to apply aggregate functions to groups of rows

- Example, find the average mark of each student individually rather than all together

- The GROUP BY clause achieves this

```
SELECT <cols1>
 FROM <tables>
 GROUP BY <cols2>
```

GROUP BY

```
SELECT <cols1>
  FROM <tables>
  GROUP BY <cols2>
```

- You can have **WHERE** and **ORDER BY** clauses as well as a **GROUP BY** clause

- Every entry in <cols1> should be
  - in <cols2>
  - *or* be a constant
  - *or* be an aggregate function

i.e. how to combine
The rows returned

1. Tables are joined
2. **WHERE** clauses remove rows
3. **GROUP BY** clauses combine remaining rows together

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
  - Same results, but **possibly** produced in a different order

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 Rules

- GROUP BY works slightly differently in MySQL than in other DBMSs.
- **Usually**, every column you name in your output must either be one of the GROUP BY columns or an aggregate function…
- For example:

```
SELECT
  ID, Name, AVG(Mark)
FROM Grades
GROUP BY ID, Name
```

Grades

Name	ID	Code	Mark
John	1	DBS	56
John	1	IAI	72
Mary	2	DBS	60
James	3	PR1	43
James	3	PR2	35
Jane	4	IAI	54
Jane	4	DBS	32
Jane	4	PR1	73
Bill	5	DBS	54
Bill	5	IAI	63
Bill	5	PR1	12

GROUP BY Rules

- In MySQL, ***for convenience***, you are **allowed** to break this rule.
- You do not have to select a SELECT a column even if it's in your GROUP BY clause
- You can SELECT columns which are not in your GROUP BY clause
- **Despite this, you should follow the ISO standard where possible**
  - Avoids problems if you use a different DBMS in the future
- **This can lead to peculiar output when multiple values are options for one row**
- If there were multiple options for the same row, MySQL would be free to choose a random one for each name
  - "The server is free to choose any value from each group, so unless they are the same, the values chosen are indeterminate." Source: MySQL online documentation

HAVING

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:
  1. Tables are joined
  2. **`WHERE`** clauses
  3. **`GROUP BY`** clauses and aggregates
  4. Column selection
  5. **`HAVING`** clauses
  6. **`ORDER BY`**

UNION

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 data types**

UNION

Grades

Name	Code	Mark
Jane	IAI	52
John	DBS	56
John	IAI	72
James	PR1	43
James	PR2	35
Mary	DBS	60

- **Question:**
  Find, in a single query, the average mark for each student and the average mark overall

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, i.e. 'Name' and 'Average'

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
Jane	52
John	64
James	39
Mary	60
Total	53

Combining these things…

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

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
  - Join the results using a UNION

```
<QUERY FOR FINALISTS>

UNION

<QUERY FOR OTHERS>
```

Table Joins

- Both subqueries need information from all of 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 an obvious natural join operation
  - Because we're practicing, we're going to use a standard CROSS JOIN and WHERE clause

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**

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

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
```

Example Output

Year	Student.ID	Last	First	AverageMark
3	11014456	Andrews	John	81
3	11013891	Smith	Mary	78
3	11014012	Jones	Steven	76
3	11013204	Brown	Amy	76
3	11014919	Robinson	Paul	74
3	[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

Next Lecture

- Missing Information
  - Nulls and the Relational Model
  - Outer Joins
  - Default Values
- Further reading
  - The Manga Guide to Databases, Chapter 2
  - Database Systems, Chapter 4

Missing Information

G51DBS Database Systems

Jason Atkin

jaa@cs.nott.ac.uk

This Lecture

- Missing Information
  - Nulls and the Relational Model
  - Outer Joins
  - Default Values
- Further reading
  - The Manga Guide to Databases, Chapter 2
  - Database Systems, Chapter 4

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 kinds 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 with extending relational algebra operations to NULLs:
  - Defining selection operation:
    if we check tuples for some property like Mark > 40
    and for some tuple, Mark is NULL, do we include it?
  - Comparing tuples in two relations: are the two tuples <John,NULL> and <John,NULL> the same or not?
- Additional problems for SQL:
  - Do we treat NULLs as duplicates?
  - Do we include them in count, sum, average and if yes, how?
  - How do arithmetic operations behave when an argument is NULL?

Theoretical Solutions

- Use three-valued logic instead of classical two-valued logic to evaluate conditions
- 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')
- This is the idea behind testing conditions in the WHERE clause of SQL SELECT: only tuples where the condition evaluates to true are returned
  - i.e. NOT 'false' **OR** 'undefined'

3-valued logic

- Results if condition involves a boolean combination:

a	b	a OR b	a AND b	a == b
True	True	True	True	True
True	False	True	False	False
False	True	True	False	False
False	False	False	False	True
True	Unknown	True	Unknown	Unknown
Unknown	True	True	Unknown	Unknown
False	Unknown	Unknown	False	Unknown
Unknown	False	Unknown	False	Unknown
Unknown	Unknown	Unknown	Unknown	Unknown

SQL NULLs in Conditions

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

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

Employee

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

Name	Salary
John	25,000
Anne	20,000

SQL NULLs in Conditions

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

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

Employee

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

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

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

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

- 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;
```

- NULLs are treated as equivalents in GROUP BY clauses

Employee

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

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

Outer Joins

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’ with an inner join
- These are called ‘dangles’
- Joins do not normally include 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 like INNER JOIN
- Note: Cannot use ‘USING’ version

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 ...

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 ...

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:

```
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
- Example: 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	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

Example: INNER JOINs

- An Inner Join with Student and Address will ignore Student 127, who doesn't have an address 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

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

Example: INNER JOINs

- 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

Phone

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

Example

```sql
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';
```

Student

ID	Name	aID	pID	Grad

Phone

pID	pNumber	pMobile

Degree

ID	Classification

Address

aID	aStreet	aTown	aPostcode

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

Default Values

Default Values

- Default values are an alternative to the use of NULLs
- If a value is not known then a particular **placeholder value** (the default) is used
- These are actual values, so don't need 3-value logic (3VL) etc.
- 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

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 a sensible value, so cannot appear by accident as a real value
- 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 a string such as 'unknown' in a column of integers

Splitting Tables

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
  - Defaults rely on having an ‘invalid’ value to use
- 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	Name
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 for these things

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 CHAR(50)
       NOT NULL,
  Salary INT
       DEFAULT 10000
       NOT NULL,
  Phone CHAR(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?

- Method to use usually depends on 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 be (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**
  - Many additional issues that will be covered next lectures

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

Next Lecture

- Normalisation
  - Data Redundancy
  - Functional Dependencies
  - Normal Forms
  - First, Second and Third Normal Forms
- Further reading
  - The Manga Guide to Databases, Chapter 3
  - Database Systems, Chapter 14