MISSING INFORMATION

Introduction to Database Systems

Mahdi Akhi
Sharif University of Technology

IN THIS LECTURE

- Missing Information
  - **`NULL`**s and the relational model
  - **`OUTER JOIN`**s
  - Default values
- For more information
  - Not really covered by Connolly and Begg
  - Some information in Chapter 3.3, 5, and 6 Ullman and Widom 6.1.5, 6.1.6, 6.3.8

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
  - **NULL**s 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 **NULL**s:
  - Defining selection operation: if we check tuples for some property like Mark > 40 and for some tuple Mark is **NULL**, do we include it?
  - Defining intersection or difference of two relations: are two tuples <John,NULL> and <John,NULL> the same or not?
- Additional problems for SQL: do we treat **NULL**s 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 SOLUTION

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

3-VALUED LOGIC

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

x	y	x AND y	x OR y	NOT x
true	true	true	true	false
true	unknown	unknown	true	false
true	false	false	true	false
un	true	un	true	un
un	un	un	un	un
un	false	false	un	un
false	true	false	true	true
false	un	false	un	true
false	false	false	false	true

3-VALUED LOGIC

false=0, true=1, unknown=1/2, NOT(x)=1-x, AND(x,y) = min(x,y),
OR(x,y) =max(x,y):

x	y	x AND y	x OR y	NOT x
true	true	true	true	false
true	unknown	unknown	true	false
true	false	false	true	false
un	true	un	true	un
un	un	un	un	un
un	false	false	un	un
false	true	false	true	true
false	un	false	un	true
false	false	false	false	true

THEORETICAL SOLUTIONS 2

- Use variables instead of **`NULL`**s to represent unknown values.
- Different unknown values correspond to different variables
- When we apply operations such as selection to tables with variables, variables may acquire side conditions (constraints), for example x > 40 if x was unknown value of Mark and we include it in result of selection Mark > 40.
- This works out fine, but has high computational complexity and is not used in practice.
- More on conditional tables: Abiteboul, Hull, Vianu, Foundations of Databases.

SQL SOLUTION: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

SQL SOLUTION:NULLS IN CONDITIONS

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

Salary > 15,000 OR Name = 'Chris' evaluates to 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 SOLUTION: ARITHMETIC

```
SELECT
Salary*1.1 AS NewSalary
FROM Employee
```

Employee

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

- Arithmetic operations applied to **NULL**s result in **NULL**s

NewSalary
27,500
16,500
22,000
NULL

SQL SOLUTION: AGGREGATES

```
SELECT
   AVG(Salary) AS Avg,
   COUNT(Salary) AS Num,
   SUM(Salary) AS Sum
FROM Employee
```

Employee

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

- Avg = 20,000
- Num = 3
- Sum = 60,000
- **`SELECT COUNT(*)...`** gives a result of 4

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 **`NULL`**s to fill in the blanks
  - Left outer join
  - Right outer join
  - Full outer 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

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

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

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

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

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

OUTER JOIN SYNTAX IN ORACLE

```
SELECT <cols>
  FROM <table1> <type> OUTER JOIN <table2>
    ON <condition>
```

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

Example:

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

DEFAULT VALUES

- Default values are an alternative to the use of **`NULL`**s
  - If a value is not known a particular placeholder value - the default - is used
- These are actual values, so don't need 3VL etc.
- Default values can have more meaning than **`NULL`**s
  - 'none'
  - 'unknown'
  - 'not supplied'
  - 'not applicable'

DEFAULT VALUE EXAMPLE

Parts

ID	Name	Wgt	Qty
1	Nut	10	20
2	Bolt	15	-1
3	Nail	3	100
4	Pin	-1	30
5	???	20	20
6	Screw	-1	-1
7	Brace	150	0

- Default values are
  - ??? for Name
  - -1 for Wgt and Qty
- -1 is used for Wgt and Qty as it is not sensible otherwise so won't appear by accident, but what about

```
UPDATE Parts
   SET Qty = Qty + 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

- **`NULL`**s and **defaults** both try to fill entries with missing data
  - **`NULL`**s 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 **`NULL`**s are in separate tables
  - Entries that would be **`NULL`** are not present in these tables

SPLITTING TABLES EXAMPLE

Parts

ID	Name	Wgt	Qty
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

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

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

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

PROBLEMS WITH SPLITTING TABLES

- Splitting tables has its own problems
  - We might introduce many extra tables
  - Information gets spread out over the database
  - Queries become more complex and require many joins
- We can recover the original table, but
  - We need to do an outer join to do so
  - This introduces **`NULL`**s, which brings in all the associated problems again

SQL SUPPORT

- SQL allows both **NULL**s 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 **NULL**s

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

SQL SUPPORT

- SQL allows you to insert **NULL**s

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

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

- You can also check for **NULL**s

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

```
SELECT Name FROM
  Employee WHERE Phone
  IS NOT NULL
```

WHICH METHOD TO USE?

- Often a matter of personal choice, but
  - 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
- **NULL**s 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

- There will be many entries with missing information
- 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 **`NULL`**s
  - Publisher, director, and producer are all optional
  - It is unlikely we'll ever use them in computation

END

Thanks to Dr. Mohammad Tanhaei, Assistant Prof. at Ilam University

NEXT LECTURE

- Normalisation to **3NF**
  - Data redundancy
  - Functional dependencies
  - Normal forms
    - First, Second and Third Normal Forms
- For more information
  - Connolly and Begg chapter 13
  - Ullman and Widom 1.1.4 (2nd edition), more in 3rd edition (3.5).