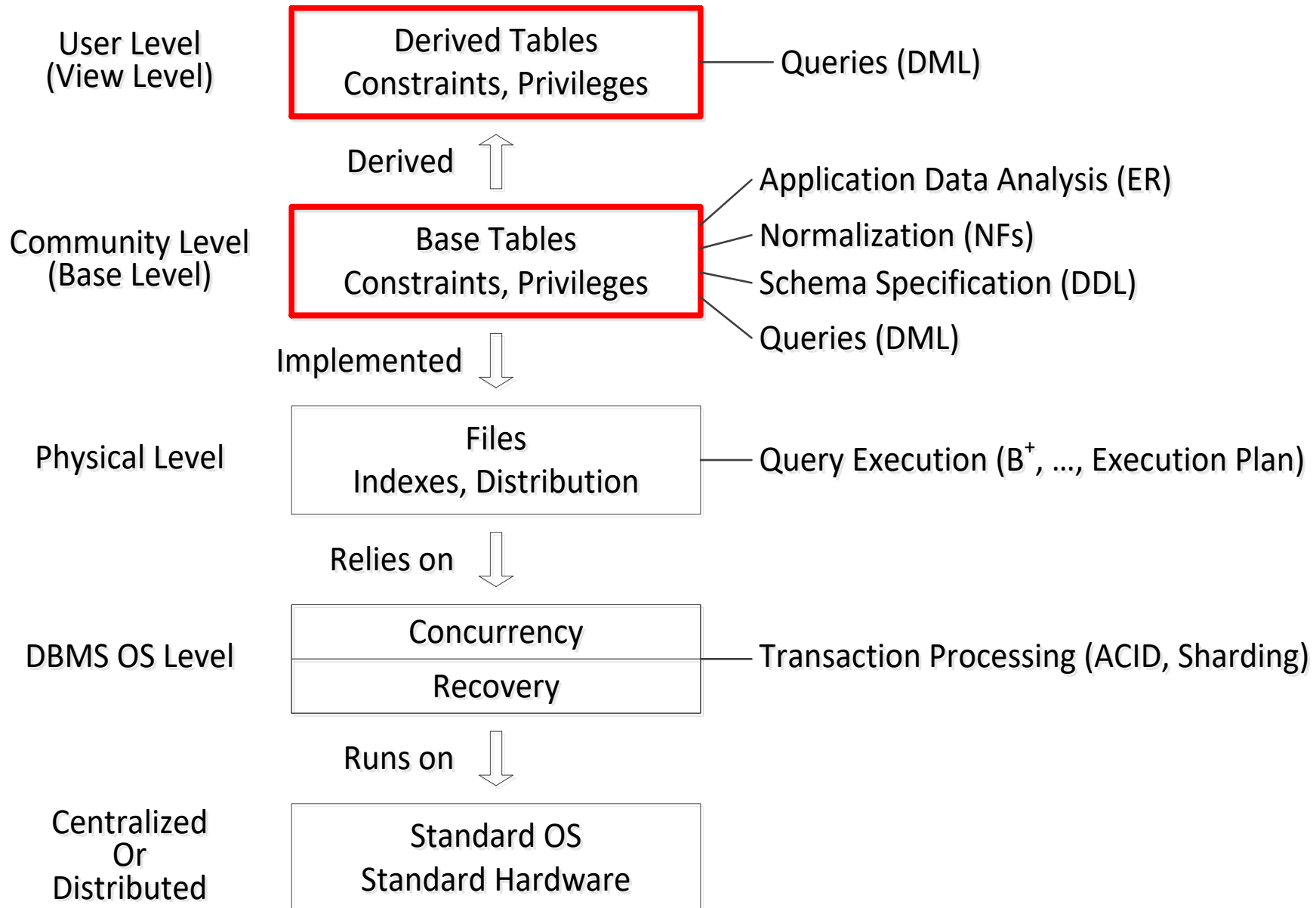


Unit 5
SQL: Data Manipulation Language
For Relational Databases

DML in Context



Introduction

SQL

- We study key features of the SQL standard for relational query/schema languages (more about schemas, that is specifying the structure of the database in the next unit)
- History:
 - SEQUEL by IBM implemented in a product (DB2)
- Standard's development
 - SQL-86
 - ...
 - SQL:2016
- Many commercial implementations “close” to one of the standards
 - With some parts missing and some parts added and some parts inconsistent.
- Very powerful but designed by a committee that agreed to practically everything proposed so many redundancies
- At its core relational algebra, which we will learn, with many additions

Our Focus

- We will focus on
 - As precise as feasible (here) description of ***the semantics of various operations***: some of them are rather surprising
 - Construction of ***simple and complex queries***, to show the full power of SQL
 - More concepts than you can get from any manual
- We will not focus on
 - Any specific system, though the class will work with MySQL
 - What you can get from a manual
 - The interaction through other languages with an SQL-based Database System

Our Focus

- The “companion” to SQL as DML is SQL as DDL (Data Definition and Control Language)
- We will cover it later
- When we cover DDL, we will include some material that properly belongs to DML as it is easier to go over it in the DDL context

More About Multisets

- The following two tables **are** equal, because:
 - They contain the same rows with the same multiplicity
 - The order of rows does not matter
 - The order of columns does not matter, as they are labeled

R	A	B
1	10	
2	20	
2	20	
2	20	

S	B	A
20	2	
20	2	
10	1	
20	2	

More About Multisets

- The following two tables (multisets) **are not** equal because:
 - There is a row that appears with different multiplicities in the two tables
 - Row (2,20) appears three times in R but two times in S

R	A	B
	1	10
	2	20
	2	20
	2	20

S	A	B
	1	10
	2	20
	2	20

- But as sets they would be equal!

Fundamental Operations In SQL

The Most Common Query Format (We Have Seen This Before)

- As we have seen, a very common expression in SQL is:

```
SELECT A1, A2, ...  
FROM R1, R2, ...  
WHERE F;
```



- In order of execution
 1. FROM: Single table or Cartesian product
 2. WHERE (optional): choose rows by condition (predicate)
 3. SELECT: choose columns by listing
- ***All three operations keep (do not remove) duplicates at any stage (unless specifically requested; more later)***

Set Operations

(Not All Of Them Always Implemented)

- UNION, *duplicates are removed*:

```
SELECT * FROM R  
UNION  
SELECT * FROM S;
```

R	A
	1
	1
	2
	3
	2

S	A
	1
	2
	2
	4
	2

Result	A
	1
	2
	3
	4

Set Operations

(Not All Of Them Always Implemented)

- MINUS, *duplicates are removed*:

```
SELECT * FROM R  
MINUS  
SELECT * FROM S;
```

R	A	S	A
	1		1
	1		2
	2		2
	3		4
	2		2

Result	A
	3

Set Operations

(Not All Of Them Always Implemented)

- INTERSECT, *duplicates are removed*:

```
SELECT * FROM R  
INTERSECT  
SELECT * FROM S;
```

R	A	S	A
	1		1
	1		2
	2		2
	3		4
	2		2

Result	A
	1
	2

Set Operations

(Not All Of Them Always Implemented)

- UNION ALL, *duplicates are not removed*:

```
SELECT * FROM R  
UNION ALL  
SELECT * FROM S;
```

R	A	S	A
	1		1
	1		2
	2		2
	3		4
	2		2

Result	A
	1
	1
	2
	3
	2
	1
	2
	2
	4
	2

- An element appears with the cardinality that in cardinality in R + cardinality in S

Set Operations

(Not All Of Them Always Implemented)

- MINUS ALL, *duplicates are not removed*:

```
SELECT * FROM R  
MINUS ALL  
SELECT * FROM S;
```



R	A	S	A	Result	A
	1		1		1
	1		2		3
	2		2		
	3		4		
	2		2		

- An element appears with the cardinality that is $\max(0, \text{cardinality in R} - \text{cardinality in S})$

Set Operations

(Not All of Them Always Implemented)

- INTERSECT ALL, *duplicates are not removed*:

```
SELECT * FROM R  
INTERSECT ALL  
SELECT * FROM S;
```

R	A	S	A	Result	A
	1		1		1
	1		2		2
	2		2		2
	3		4		
	2		2		

- An element appears with the cardinality that is $\min(\text{cardinality in R, cardinality in S})$

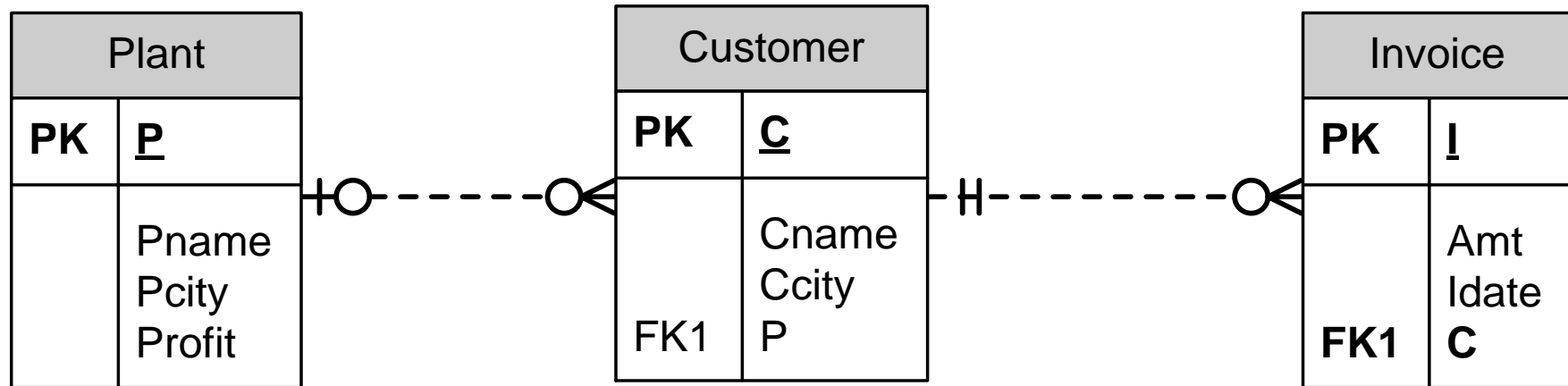
Our Sample Database

- We will describe the language by means of a toy database dealing with orders for a single product that are supplied to customers by plants
- It is chosen so that
 - It is small
 - Sufficiently rich to learn SQL
 - Therefore, a little artificial, but this does not matter
- Sample database: ***PlantCustomerInvoice.mdb*** in Microsoft Access Database Examples for this unit (optional, just if you want to follow along).

The Tables of Our Database

- **Plant(P,Pname,Pcity,Profit)**
 - This table describes the plants, identified by P. Each plant has a Pname, is in a Pcity, and makes certain Profit
- **Customer(C,Cname,Ccity,P)**
 - This table describes the customers, identified by C. Each customer has a Cname and lives in a Ccity. Each customer is assigned to a specific P, where the orders for the customers are fulfilled. This P is a foreign key referencing Plant
- **Invoice(I,Amt,Idate,C)**
 - This table describes the orders, identified by I. Each order is for some Amt (amount), is on a specific Idate, and placed by some C. This C is a foreign key referencing Customer. C must not be NULL
I could not use attribute “Date,” as it is a reserved keyword in SQL

The Tables of Our Database



Our Instance In Microsoft Access

Plant				
	P	Pname	Pcity	Profit
+	901	Alpha	Boston	\$45,000.00
+	902	Beta	Boston	\$56,000.00
+	903	Beta	Chicago	
+	904	Gamma	Chicago	\$51,000.00
+	905	Delta	Denver	\$48,000.00
+	906	Epsilon	Miami	\$51,000.00
+	907	Beta	Miami	\$65,000.00
+	908	Beta	Boston	\$51,000.00

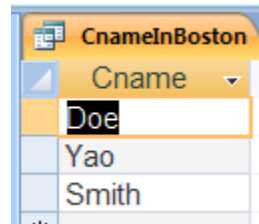
Customer				
	C	Cname	Ccity	P
+	1000	Doe	Boston	901
+	2000	Yao	Boston	902
+	3000	Doe	Chicago	903
+	4000	Doe	Seattle	
+	5000	Brown	Denver	903
+	6000	Smith	Seattle	907
+	7000	Yao	Chicago	904
+	8000	Smith	Denver	904
+	9000	Smith	Boston	903

Invoice				
	I	Amt	Idate	C
	501	30	2009-02-02	2000
	502	300	2009-02-03	3000
	503	200	2009-02-01	1000
	504	160	2009-02-03	1000
	505	150	2009-02-02	2000
	506	150	2009-02-02	4000
	507	200		2000
	508	20	2009-02-03	1000
	509	20		4000

Queries On A Single Table

- Find Cname for all customers who are located in Boston:

```
SELECT Cname  
FROM Customer  
WHERE Ccity = 'Boston';
```



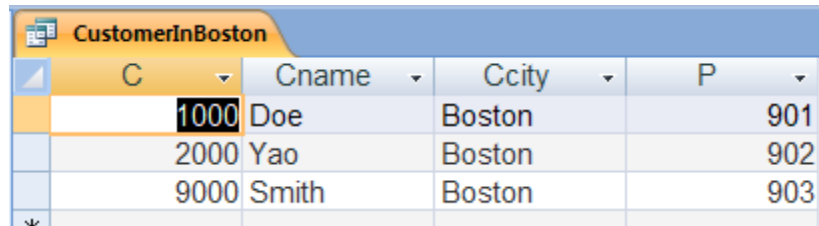
Cname
Doe
Yao
Smith

Queries On A Single Table

- Find full data on every customer located in Boston:

```
SELECT *  
FROM Customer  
WHERE Ccity = 'Boston';
```

- The asterisk, *, stands for the sequence of all the columns, in this case, C, Cname, Ccity, P

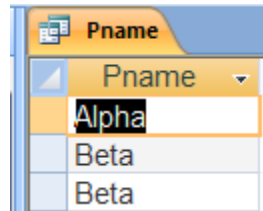


C	Cname	Ccity	P
1000	Doe	Boston	901
2000	Yao	Boston	902
9000	Smith	Boston	903

Queries On A Single Table

- Find Pname for all plants that are located in Boston:

```
SELECT Pname  
FROM Plant  
WHERE Pcity = 'Boston';
```



A screenshot of a database query result window. The window has a title bar with a small icon and the text 'Pname'. Below the title bar is a dropdown menu with 'Pname' selected. The main area of the window displays a list of plant names: 'Alpha', 'Beta', and 'Beta'. The 'Alpha' row is highlighted with a black background, and the 'Beta' row below it is highlighted with a light blue background.

Pname
Alpha
Beta
Beta

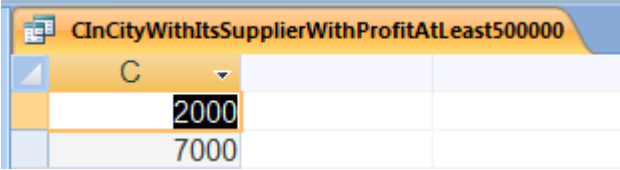
- Note that duplicates were not removed

Queries on a Single Table (Continued)

- Find every C who is supplied from a plant in the same city they are in and the plant's profit is at least 50000

```
SELECT C  
FROM Plant, Customer  
WHERE Plant.Pcity = Customer.Ccity  
AND Plant.P = Customer.P  
AND Profit >= 50000;
```

- Note that we need to “consult” two tables even though the answer is taken from a single table



C	
	2000
	7000

Queries on Two Tables And Renaming Columns and Tables

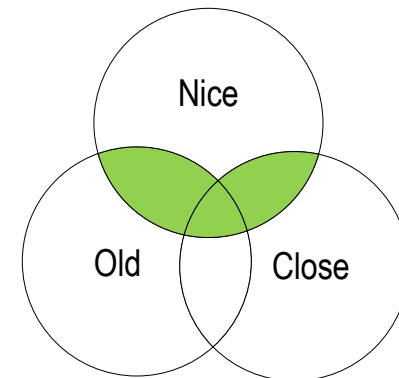
- We want to produce a table with the schema (Bigger,Smaller), where bigger and smaller are two P located in the same city and the Profit of the Bigger is bigger than that of the Smaller
 - Two (logical) copies of Plant were produced, the first one is First and the second one is Second .
 - The attributes of the result were renamed, so the columns of the answer are Bigger and Smaller

```
SELECT First.P AS Bigger, Second.P AS Smaller
FROM Plant AS First, Plant AS Second
WHERE First.City = Second.City AND First.Profit > Second.Profit;
```

PlantBiggerWithPlantSmaller	
Bigger	Smaller
908	901
902	901
907	906
902	908

An Interesting Query

- We have 3 relations:
 - Nice(City)
 - Old(City)
 - Close(City)
- **SELECT** *
FROM Nice, Old, Close
WHERE (Nice.City = Old.City)
 OR (Nice.City = Close.City);
- The result is indicated by green color



- But what if Old is empty? What is the result?

A Note About NULLs

- We will discuss NULLs later, but we can note something now
- There are two plants in Chicago, one of them has profit of NULL
- When the comparison for these two plants is attempted, the following two need to be compared:
 - \$51,000.00
 - NULL
- This comparison will return false for purposes of SELECT, but formally speaking the result is “UNKNOWN”

New Example Database Schema

- We will first cover the material using a new additional very natural example database suggested by Robert Soule
- Then we will review division using our running example database
- We will have two relations
 - Took(Person,Course). This relation records which Person Took which Course. For simplicity, we assume that every Person Took at least one Course.
 - Required(Course). This relation records which Course(s) are required for graduation. For simplicity, we assume that this relation is not empty.
- A Person Took, in general, both Required and not-Required Course(s)

An Example Instance

Took	Person	Course
	Marsha	OS
	Marsha	DB
	Marsha	Algebra
	Vijay	OS
	Vijay	DB
	Dong	OS
	Dong	Algebra
	Chris	Algebra
	Chris	English

Required	Course
	OS
	DB

- We will “run” some queries on the database and see the output
- We will remove duplicates to save space

Asking About Some

- **SELECT** Person
FROM Took, Required
WHERE Took.Course = Required.Course;
- Answer

	Person
	Marsha
	Vijay
	Dong

Asking About None

- `SELECT Person`
`FROM TOOK, REQUIRED`
`WHERE Took.Course <> Required.Course;`
- (“<>” means “not equal”)
- Answer (*wrong!*)

	Person
	Marsha
	Vijay
	Dong
	Chris

Asking About None

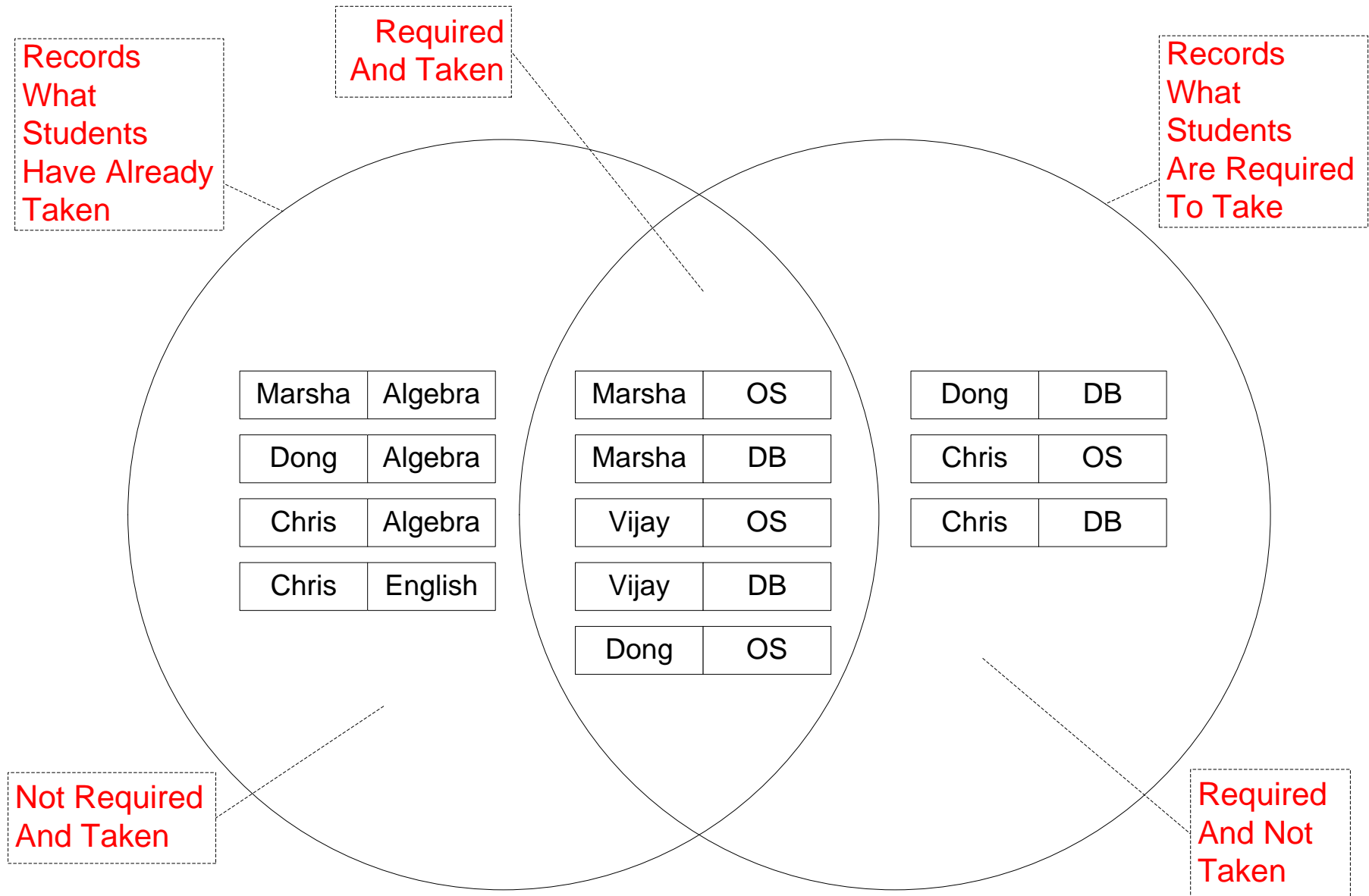
- `SELECT Person`
`FROM Took`
`MINUS`
`SELECT Person`
`FROM Took, Required`
`WHERE Took.Course = Required.Course;`
- Answer (correct)

	Person
	Chris

Asking About All

- Want people who took all required courses.
- Informal roadmap
 1. Find all Person(s): TempA(Person) (easy from Took)
 2. Find all “records” of them taking Courses required for them to graduate, whether they took them or not: TempB(Person,Course)
 3. Find all records that are “missing” listing what still needs to be Taken: TempC(Person,Course) = TempB – Took
 4. Find all Person(s) who are missing a Required Course: TempD (easy from TempC)
 5. Find all Person(s) who are not missing a Required Course: TempA – TempD

Helpful Venn Diagram



Find All (Person)

- TempA:

```
SELECT Person  
FROM Took;
```

TempA	Person
	Marsha
	Vijay
	Dong
	Chris

Find All (Person, Required Course)

- TempB: (Cross-product of person and course)

```
SELECT Person, Course  
FROM TempA, Required;
```

TempB	Person	Course
	Marsha	OS
	Marsha	DB
	Vijay	OS
	Vijay	DB
	Dong	OS
	Dong	DB
	Chris	OS
	Chris	DB

Find All (Person, Required Course Not Taken)

- TempC:

```
SELECT *  
FROM tempB  
MINUS  
SELECT *  
FROM Took;
```

TempC	Person	Course
	Dong	DB
	Chris	OS
	Chris	DB

Find All (Person Who Did Not Take A Required Course)

- TempD:

```
SELECT Person  
FROM TempC:
```

TempD	Person
	Dong
	Chris

Find All (Person Who Took All Required Courses)

- Answer:

```
SELECT *  
FROM TempA  
MINUS  
SELECT *  
FROM TempD:
```

Answer	Person
	Marsha
	Vijay

Asking About Some Versus Asking About All

- We first compute two tables
 - CnameInCcity(Ccity,Cname)
This table lists all the “valid” tuples of Ccity,Cname; it is convenient for us to list the city first
 - CnameInChicago(Cname)
This table lists the names of the customers located in Chicago.
- We then want to specify two queries
 - The first one is expressible by the **existential quantifier** (more about it later, if there is time)
 - The second one is expressible by the **universal quantifier** (more about it later, if there is time)

CnameInCcity

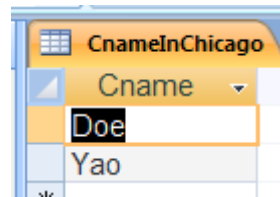
- **CREATE TABLE** CnameInCcity **AS**
SELECT Ccity, Cname
FROM Customer;

- This variant of the **SELECT** statement uses **INTO**, creates a new table, here CnameInCcity and populates it with the result of the query

CnameInCcity	
Ccity	Cname
Boston	Doe
Boston	Yao
Chicago	Doe
Seattle	Doe
Denver	Brown
Seattle	Smith
Chicago	Yao
Denver	Smith
Boston	Smith

CnameInChicago

- **CREATE TABLE** CnameInChicago AS
- **SELECT** Customer.Cname
FROM Customer
WHERE Ccity='Chicago';



Cname
Doe
Yao

Our Tables

- I have reproduced them, so they are larger and we can see them clearly

CnameInCcity	Ccity	Cname
	Boston	Doe
	Boston	Yao
	Boston	Smith
	Chicago	Doe
	Chicago	Yao
	Seattle	Doe
	Seattle	Smith
	Denver	Smith
	Denver	Brown

CnameInChicago	Cname
	Doe
	Yao

Asking About Some And About All

- List all cities, the set of whose Cnames, contains ***at least one*** Cname that is (also) in Chicago
 - This will be easy
-
- List all cities, the set of whose Cnames contains ***at least all*** the Cnames that are (also) in Chicago
 - This will be harder. What you see here is a method to do this using Boolean operations, but there are other methods using “outer joins” that would be more efficient. Go to Dennis lecture notes.

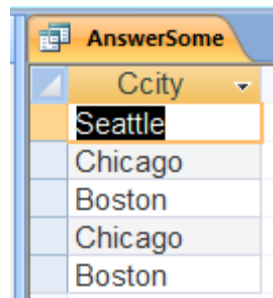
Asking About Some

- List all cities, the set of whose Cnames, contains **at least one** Cname that is (also) in Chicago

```
CREATE TABLE AnswerSome AS  
SELECT CnameInCcity.Ccity  
FROM CnameInChicago, CnameInCcity  
WHERE CnameInChicago.Cname = CnameInCcity.Cname;
```

AnswerSome	Ccity
	Boston
	Chicago
	Seattle

In Microsoft Access



Asking About All

- Want cities that have all Cnames in Chicago.
- The alternative would start with the left join (see <http://www.mysqltutorial.org/mysql-left-join.aspx>)

CREATE TABLE MightHaveNulls AS

```
SELECT CnameInCcity.Ccity, CnameInCcity.Cname  
FROM CnameInChicago LEFT JOIN CnameInCcity  
ON CnameInChicago.Cname = CnameInCcity.Cname;
```

Then delete any city with nulls in MightHaveNulls from AnswerSome.

Skip to slide 75

Roadmap

1. TempA = (all cities)
2. TempB = (all cities, all customers); for every city all the customers in the database, **not only** the customers in this city
3. TempC = TempB – CnameInCcity = (all cities, customers that should be in the cities to make them good but are not there); in other words, for each Ccity a Cname that it does not have but needs to have to be a “good” City
4. TempD = (all bad cities)
5. AnswerAll = TempA – TempD = (all good cities)

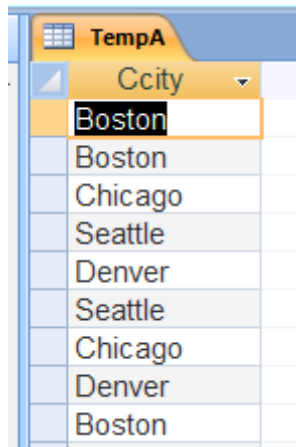
Asking About All

- `SELECT Ccity INTO TempA
FROM CnameInCcity;`
- Set of all cities in which there could be customers

TempA	Ccity
	Boston
	Chicago
	Seattle
	Denver

In Microsoft Access

- Note duplicates: nothing surprising about this, as duplicates are not removed



A screenshot of a Microsoft Access table named 'TempA'. The table has a single column labeled 'Ccity'. The data in the table is as follows:

Ccity
Boston
Boston
Chicago
Seattle
Denver
Seattle
Chicago
Denver
Boston

Asking About All

- **SELECT** Ccity, Cname **INTO** tempB
FROM TempA, CnameInChicago;
- Set of all pairs of the form (Ccity,Cname); in fact a Cartesian product of **all** cities with **all** desired Cnames (**not only** cities that **have all** desired Cnames)

tempB	Ccity	Cname
	Boston	Doe
	Boston	Yao
	Chicago	Doe
	Chicago	Yao
	Seattle	Doe
	Seattle	Yao
	Denver	Doe
	Denver	Yao

In Microsoft Access

tempB	
Ccity	Cname
Boston	Doe
Boston	Yao
Boston	Doe
Boston	Yao
Chicago	Doe
Chicago	Yao
Seattle	Doe
Seattle	Yao
Denver	Doe
Denver	Yao
Seattle	Doe
Seattle	Yao
Chicago	Doe
Chicago	Yao
Denver	Doe
Denver	Yao
Boston	Doe
Boston	Yao

Asking About All (Not Real Microsoft Access SQL Syntax)

- `SELECT * INTO tempC
FROM (SELECT *
FROM tempB)
MINUS
(SELECT *
FROM CnameInCcity);`



- Set of all pairs of the form (Ccity,Cname), such that the Ccity does not have the Cname; this is a “bad” Ccity with a proof why it is bad

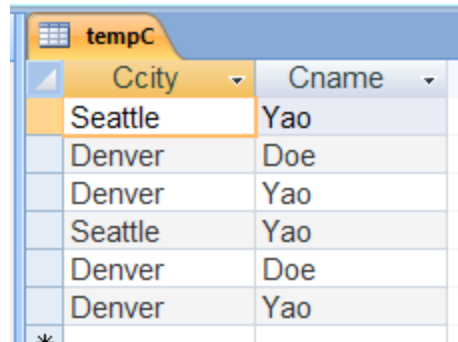
tempC	Ccity	Cname
	Seattle	Yao
	Denver	Doe
	Denver	Yao

Microsoft Access Has To Do This Differently We Will Understand This Later

- **SELECT * INTO tempC**
FROM tempB
WHERE NOT EXISTS
(SELECT *
FROM CnameInCcity
WHERE tempB.Ccity = CnameInCcity.Ccity AND
tempB.Cname = CnameInCcity.Cname);
- Set of all pairs of the form (Ccity,Cname), such that the Ccity does not have the Cname; this is a “bad” Ccity with a proof why it is bad

tempC	Ccity	Cname
	Seattle	Yao
	Denver	Doe
	Denver	Yao

In Microsoft Access



A screenshot of a Microsoft Access table named 'tempC'. The table has two columns: 'Ccity' and 'Cname'. The data is as follows:

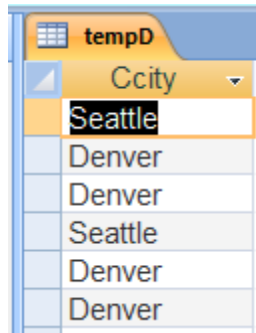
Ccity	Cname
Seattle	Yao
Denver	Doe
Denver	Yao
Seattle	Yao
Denver	Doe
Denver	Yao

Asking About All

- `SELECT Ccity`
`FROM tempC`
`INTO tempD;`
- Set of all “bad” Cities, that is cities that lack at least one Cname in CnameInChicago

tempD	Ccity
	Seattle
	Denver
	Denver

In Microsoft Access



A screenshot of a Microsoft Access table named 'tempD'. The table has a single column labeled 'Ccity'. The data in the table is as follows:

Ccity
Seattle
Denver
Denver
Seattle
Denver
Denver

Asking About All (Not Real Microsoft Access SQL Syntax)

- **SELECT * INTO AnswerAll**
FROM (SELECT *
FROM tempA)
MINUS
(SELECT *
FROM tempD);
- Set of all “good” cities, that is cities that are not “bad”

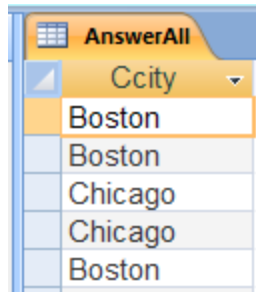
AnswerAll	Ccity
	Boston
	Chicago

Microsoft Access Has To Do This Differently We Will Understand This Later

- `SELECT * INTO AnswerAll
FROM tempA
WHERE NOT EXISTS
(SELECT *
FROM tempD
WHERE tempD.Ccity = tempA.Ccity);`
- Set of all “good” cities, that is cities that are not “bad”

AnswerAll	Ccity
	Boston
	Chicago

In Microsoft Access



A screenshot of a Microsoft Access table named 'AnswerAll'. The table has a column named 'Ccity' which is a dropdown menu. The dropdown is currently open, showing a list of city names: Boston, Boston, Chicago, Chicago, and Boston. The first 'Boston' entry is highlighted with an orange background.

Ccity
Boston
Boston
Chicago
Chicago
Boston

Nulls And Duplicates


NULLS And Duplicates

- We now move to look at some aspects of SQL, which were not applicable to our relational algebra model
- We will use, for this purposes simpler example databases and then will return to our PlantCustomerInvoice.mdb database

NULLs

- Each domain is augmented with a **NULL**
- NULL, intuitively stands for one of the following
 - Value unknown
 - Value not permitted to be known (to some of us)
 - Value not applicable
- Semantics of NULLs is very complicated, I will touch on the most important aspects
- ***There are two variants***
 - ***For most of SQL DML***
 - ***For SQL DDL and some SQL DML***
- But the core is common

NULLs

- We start with a SELECT statement
- **SELECT ...**
FROM ...
WHERE condition 
- As we know:
 - Each tuple is tested against the condition
 - If the condition on the tuple is TRUE, then it is passed to SELECT
- What happens if the condition is, say “**x = 5**”, with x being a column name?
 - It may happen that some current value in column x is NULL, what do we do?
- What happens if the condition is, say “**x = 5 OR x <> 5**”, with x being a column name?
 - No matter what the value of x is, even if x is NULL, this should evaluate to TRUE? Or should it? It does not!
- We use a new logic

NULLs

- We abbreviate:
 - T for TRUE
 - F for FALSE
 - U for UNKNOWN
- Standard 2-valued logic

	NOT
F	T
T	F

	OR	F	T
F	F	F	T
T	T	T	T

	AND	F	T
F	F	F	F
T	F	F	T

- New 3-valued logic

	NOT
F	T
U	U
T	F

	OR	F	U	T
F	F	F	U	T
U	U	U	U	T
T	T	T	T	T

	AND	F	U	T
F	F	F	F	F
U	F	F	U	U
T	F	F	U	T

- U is “between” F and T, “metathink” as being “maybe T or maybe F”

NULLs

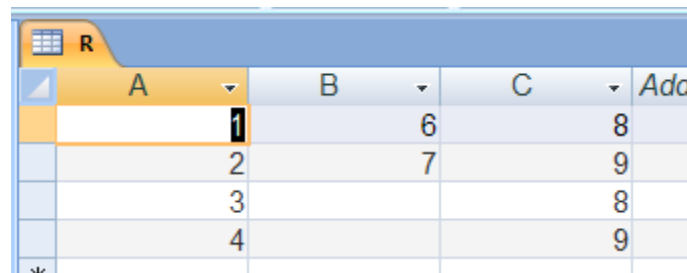
- Something to aid intuition
- Think
 - **NOT**(x) as $1 - x$
 - x **OR** y as $\max(x,y)$
 - x **AND** y as $\min(x,y)$
- Then for 2-valued logic
 - FALSE is 0
 - TRUE is 1
- Then for 3-valued logic
 - FALSE is 0
 - UNKNOWN is 0.5
 - TRUE is 1

NULLs

- Back to a SELECT statement
- SELECT ...
FROM ...
WHERE condition
- As we know, each tuple is tested against the condition. Then, these are the rules
 - If the condition on the tuple is TRUE, then it is passed to SELECT
 - If the condition on the tuple is UNKNOWN, then it is not passed to SELECT
 - If the condition on the tuple is FALSE, then it is not passed to SELECT
- ***In the context of SQL SELECT queries, UNKNOWN behaves exactly the same as FALSE***
- So why introduce it the concept of UNKNOWN? Because it will behave differently, exactly the same as TRUE, in the context of SQL DDL and SQL DML INSERT, as we will see later

NULLs

- We will use a simple Microsoft Access database: ***Nulls.mdb*** in Extras
- It has only one table



A screenshot of a Microsoft Access table named 'R'. The table has four columns: 'A', 'B', 'C', and 'Add'. The first row has values 1, 6, 8, and an empty cell. The second row has values 2, 7, 9, and an empty cell. The third row has values 3, an empty cell, 8, and an empty cell. The fourth row has values 4, an empty cell, 9, and an empty cell. The table is displayed in a grid view with a blue header bar.

A	B	C	Add
1	6	8	
2	7	9	
3		8	
4		9	

NULLs

- *Any comparison in which one side is NULL is UNKNOWN*

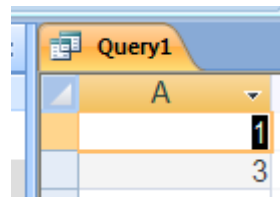
R	A	B	C
	1	6	8
	2	7	9
	3	NULL	8
	4	NULL	9

- `SELECT A`
`FROM R`
`WHERE B = 6 OR C = 8;`

- We get:

	A
	1
	3

In Microsoft Access



A screenshot of a Microsoft Access query result grid. The grid has a header row with the letter 'A' and a data row with the number '3'. The header row is highlighted in orange, and the data row is highlighted in light blue. The grid is titled 'Query1' in the top left corner.

A
3

NULLs

- **Any comparison in which one side is NULL is UNKNOWN**

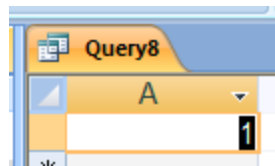
R	A	B	C
	1	6	8
	2	7	9
	3	NULL	8
	4	NULL	9

- **SELECT A**
FROM R
WHERE B = 6 AND C = 8;

- We get:

	A
	1

In Microsoft Access



NULLs

- *Any comparison in which one side is NULL is UNKNOWN*

R	A	B	C
	1	6	8
	2	7	9
	3	NULL	8
	4	NULL	9

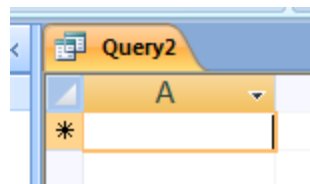
- `SELECT A`
`FROM R`
`WHERE B = NULL;`

- We get:

	A
--	---

which is an empty table

In Microsoft Access



NULLs

- *Any comparison in which one side is NULL is UNKNOWN*

R	A	B	C
	1	6	8
	2	7	9
	3	NULL	8
	4	NULL	9

- `SELECT A`
`FROM R`
`WHERE B <> NULL;`

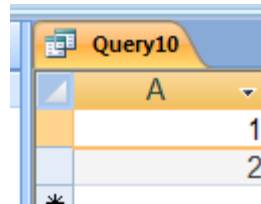
- We get:

	A
--	---

which is an empty table

In Microsoft Access

- But note what Access did, *which is wrong*:



A screenshot of a Microsoft Access query result window titled 'Query10'. The window displays a table with a single column labeled 'A'. The table contains two rows of data: the first row has the value '1' and the second row has the value '2'. The table is shown in a compact view with a light blue header and a light yellow body.

A
1
2

- This is actually astounding!

NULLs

- Any comparison in which one side is NULL is UNKNOWN

R	A	B	C
	1	6	8
	2	7	9
	3	NULL	8
	4	NULL	9

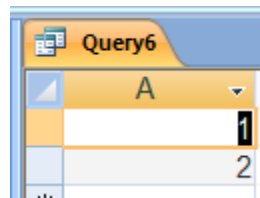
- SELECT A
FROM R
WHERE B = B;

- We get:

A
1
2

- Because, going row by row:
 - 6 = 6 is TRUE
 - 7 = 7 is TRUE
 - NULL = NULL is UNKNOWN
 - NULL = NULL is UNKNOWN

In Microsoft Access



Query6	
A	
	1
	2

NULLs

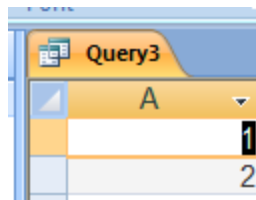
- *A new keyword made of three words: IS NOT NULL*

R	A	B	C
	1	6	8
	2	7	9
	3	NULL	8
	4	NULL	9

- `SELECT A`
`FROM R`
`WHERE B IS NOT NULL;`
- We get:

	A
	1
	2

In Microsoft Access



A screenshot of a Microsoft Access query result grid. The grid has a header row with a blue background and a column labeled 'A'. Below the header, there are two data rows. The first data row has a white background and the number '1' in the 'A' column. The second data row has a light gray background and the number '2' in the 'A' column.

A
1
2

NULLs

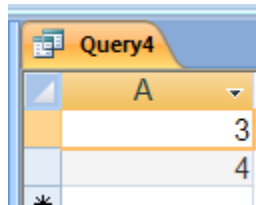
- *A new keyword made of two words: IS NULL*

R	A	B	C
	1	6	8
	2	7	9
	3	NULL	8
	4	NULL	9

- `SELECT A`
`FROM R`
`WHERE B IS NULL;`
- We get:

	A
	3
	4

In Microsoft Access



Query4	
A	
	3
	4

NULLs

- We have not discussed arithmetic operations yet, but will later
- If one of the operands is NULL, the result is NULL (some minor exceptions), so:
 - $5 + \text{NULL} = \text{NULL}$
 - $0 * \text{NULL} = \text{NULL}$
 - $\text{NULL} / 0 = \text{NULL}$

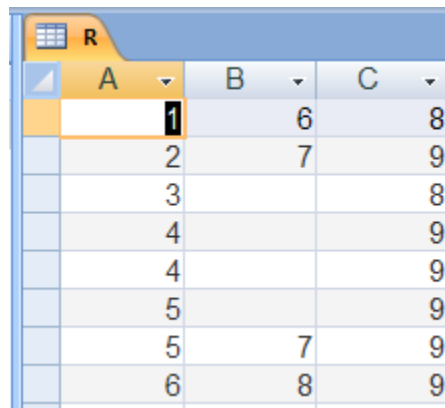
NULLs

- All NULLs are duplicates of each other (even though it is UNKNOWN whether they are equal to each other)*
- We will understand what the implications of this are once we look a little closer at duplicates and aggregates

- * This is not my fault!

Duplicates

- Standard SELECT FROM WHERE statement does not remove duplicates at any stage of its execution
- Standard UNION, EXCEPT, INTERSECT remove duplicates
- UNION ALL, EXCEPT ALL, INTERSECT ALL do not remove duplicates with rather interesting semantics
- We will just go over some of these here, using database ***Nulls+Duplicates.mdb*** in Extras
- It has one table



A	B	C
1	6	8
2	7	9
3		8
4		9
4		9
5		9
5	7	9
6	8	9

Duplicates

- SELECT B, C
FROM R
WHERE A < 6;

R		
A	B	C
1	6	8
2	7	9
3		8
4		9
4		9
5		9
5	7	9
6	8	9

Query1	
B	C
6	8
7	9
	8
	9
	9
	9
7	9

Duplicates

- `SELECT DISTINCT B, C`
`FROM R`
`WHERE A < 6;`
- New keyword `DISTINCT` removes duplicates from the result (all NULLs are duplicates of each other)

A	B	C
1	6	8
2	7	9
3		8
4		9
4		9
5		9
5	7	9
6	8	9

B	C
	8
	9
6	8
7	9

Removing Duplicate Rows From A Table

- **SELECT DISTINCT ***
FROM R;
- This can be used to remove duplicate rows (later need to rename the result so it is called R; minor syntax issue)

R			
A	B	C	
1	6	8	
2	7	9	
3		8	
4		9	
4		9	
5		9	
5	7	9	
6	8	9	

Query3			
A	B	C	
1	6	8	
2	7	9	
3		8	
4		9	
5		9	
5	7	9	
6	8	9	

Aggregation

Aggregation



- It is possible to perform aggregate functions on tables
- The standard aggregate operators are:
 - **SUM**; computes the sum; NULLs are ignored
 - **AVG**; computes the average; NULLs are ignored
 - **MAX**; computes the maximum; NULLs are ignored
 - **MIN**; computes the minimum; NULLs are ignored
 - **COUNT**; computes the count (the number of); NULLs are ignored, but exception below



Aggregation

- It is sometimes important to specify whether duplicates should or should not be removed before the appropriate aggregate operator is applied
- Modifiers to aggregate operators
 - ***DISTINCT*** (remove duplicates)
 - ***COUNT*** can also have * specified, to count the number of tuples, without removing duplicates, here NULLs are not ignored, example of this later
- Microsoft Access does not support DISTINCT in aggregatons.

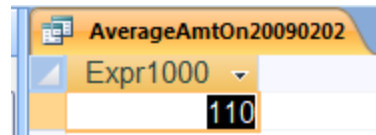
Queries With Aggregates

- Find the average Amt in Invoice, taking into account only orders from February 2, 2009

```
SELECT AVG(Amt)
FROM Invoice
WHERE Idate = #2009-02-02#;
```

- Note that we must not remove duplicates before computing the average of all the values of Amt, to get the right answer
- Note that we had to assume that there are no duplicate rows in Invoice; we know how to clean up a table
- Note syntax for date
- Skip to page 125.

In Microsoft Access



Queries With Aggregates

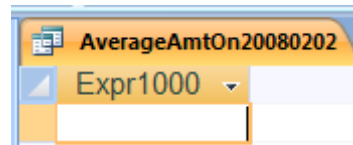
- Find the average Amt in Invoice, taking into account only DISTINCT amount values in orders from February 2, 2009
 - `SELECT AVG(DISTINCT Amt)`
`FROM Invoice`
`WHERE Idate = #2009-02-02#;`
- Cannot run this on Microsoft Access
 - Should return: 60

Queries With Aggregates

- Find the average Amt in Invoice, taking into account only orders from February 2, 2008

```
SELECT AVG(Amt)
FROM Invoice
WHERE Idate = #2008-02-02#;
```

In Microsoft Access



Queries With Aggregates

- Find the number of different values of Amt in Invoice, taking into account only orders from February 2, 2009

```
SELECT COUNT(DISTINCT Amt)
FROM Invoice
WHERE Idate = #2009-02-02#;
```

- Here we had to remove duplicates, to get the right answer
- Cannot run on Microsoft Access

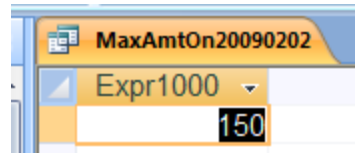
Queries With Aggregates

- Find the largest Amt in Invoice, taking into account only orders from February 2, 2009

```
SELECT MAX(Amt)
FROM Invoice
WHERE Idate = #2009-02-02#;
```

- Does not matter if we remove duplicates or not

In Microsoft Access



Queries With Aggregates

- Find the smallest Amt in Invoice, taking into account only orders from February 2, 2009

```
SELECT MIN(Amt)
FROM Invoice
WHERE Idate = #2009-02-02#;
```

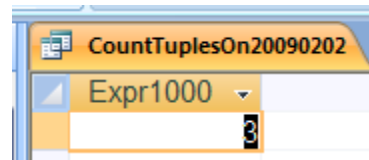
- Does not matter if we remove duplicates or not

Queries With Aggregates

- Find the number of tuples in Invoice, taking into account only orders from February 2, 2009

```
SELECT COUNT(*)  
FROM Invoice  
WHERE ldate = #2009-02-02#;
```

In Microsoft Access

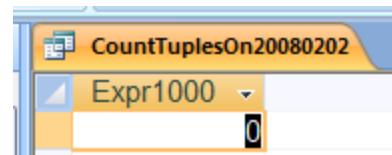


Queries With Aggregates

- Find the number of tuples in Invoice, taking into account only orders from February 2, 2008

```
SELECT COUNT(*)  
FROM Invoice  
WHERE ldate = #2008-02-02#;
```

In Microsoft Access



Queries With Aggregates

- If the
FROM ...
WHERE ...
part produces an empty table then:
 - SELECT COUNT (*)
returns 0
 - SELECT COUNT
returns 0
 - SELECT MAX
returns NULL
 - SELECT MIN
returns NULL
 - SELECT AVG
returns NULL
 - SELECT SUM
returns NULL

Queries With Aggregates

- If the
FROM ...
WHERE ...
part produces an empty table **then**:

SELECT SUM.... returns NULL

- This violates standard mathematics, for instance

$$\sum \{i \mid i \text{ is prime and } 32 \leq i \leq 36\} = 0$$

and not undefined or NULL

Queries With Aggregates

- Assume I own all the plants
- How much money I made (or actually invoiced) on February 2, 2009?
- Let's use a nice title for the column (just to practice)

```
SELECT SUM(Amt) AS Billed20090202  
FROM Invoice  
WHERE Idate = #2009-02-02#;
```

- Logically, it makes sense that we get 330

In Microsoft Access

SumAmtOn20090202	
Billed20090	
	330

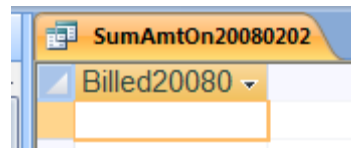
Queries With Aggregates

- Assume I own all the plants
- How much money I made (or actually invoiced) on February 2, 2008?
- Let's use a nice title for the column (just to practice)

```
SELECT SUM(Amt) AS Billed20080202  
FROM Invoice  
WHERE Idate = #2008-02-02#;
```

- Logically (and mathematically, following standard laws of mathematics), it makes sense that we get 0
- But we get NULL

In Microsoft Access



Queries With Aggregates

- In some applications it may make sense
- For example, if a student has not taken any classes, perhaps the right GPA is NULL
- Even in Mathematics, we would be computing number of points divided by number of courses, $0/0$, which is undefined

Queries With Aggregates

- It is possible to have quite a sophisticated query, which shows the importance of this construct:
- (Completely) ignoring all orders placed by C = 3000, list for each Idate the sum of all orders placed, if the average order placed was larger than 100

```
SELECT Idate, SUM(Amt)
FROM Invoice
WHERE C <> 3000
GROUP BY Idate
HAVING AVG(Amt) > 100;
```



- The order of execution is:
 1. FROM
 2. WHERE
 3. GROUP
 4. HAVING
 5. SELECT
- We will trace this example to see how this works

Queries With Aggregates

- To make a smaller table, I put only the day (one digit) instead of the full date, which the database actually has
- So, instead of 2009-02-02 I just write 2
- No problem, as everything in the table is in the range 2009-02-01 to 2009-02-03

Queries With Aggregates

Invoice	I	Amt	Idate	C
	501	30	2	2000
	502	300	3	3000
	503	200	1	1000
	504	160	3	1000
	505	150	2	2000
	506	150	2	4000
	507	200	NULL	2000
	508	20	3	1000
	509	20	NULL	4000

- After FROM, no change, we do not have Cartesian product in the example

I	Amt	Idate	C
501	30	2	2000
502	300	3	3000
503	200	1	1000
504	160	3	1000
505	150	2	2000
506	150	2	4000
507	200	NULL	2000
508	20	3	1000
509	20	NULL	4000

Queries With Aggregates

I	Amt	Idate	C
501	30	2	2000
502	300	3	3000
503	200	1	1000
504	160	3	1000
505	150	2	2000
506	150	2	4000
507	200	NULL	2000
508	20	3	1000
509	20	NULL	4000

- After WHERE C <> 3000

I	Amt	Idate	C
501	30	2	2000
503	200	1	1000
504	160	3	1000
505	150	2	2000
506	150	2	4000
507	200	NULL	2000
508	20	3	1000
509	20	NULL	4000

Queries With Aggregates

I	Amt	Idate	C
501	30	2	2000
503	200	1	1000
504	160	3	1000
505	150	2	2000
506	150	2	4000
507	200	NULL	2000
508	20	3	1000
509	20	NULL	4000

- After GROUP BY Idate

I	Amt	Idate	C
501	30	2	2000
505	150	2	2000
506	150	2	4000
503	200	1	1000
504	160	3	1000
508	20	3	1000
507	200	NULL	2000
509	20	NULL	4000

Queries With Aggregates

I	Amt	Idate	C
501	30	2	2000
505	150	2	2000
506	150	2	4000
503	200	1	1000
504	160	3	1000
508	20	3	1000
507	200	NULL	2000
509	20	NULL	4000

- We have 4 groups, corresponding to the dates: 2, 1, 3, NULL
- We compute for ourselves the average order for each group, the group condition

Idate	AVG(Amt)
2	110
1	200
3	90
NULL	110

- Groups for dates 2, 1, NULL satisfy the “group” condition

Queries With Aggregates

I	Amt	Idate	C
501	30	2	2000
505	150	2	2000
506	150	2	4000
503	200	1	1000
504	160	3	1000
508	20	3	1000
507	200	NULL	2000
509	20	NULL	4000

- Groups for dates 2, 1, NULL satisfy the “group” condition, so after `HAVING AVG(Amt) > 100`

I	Amt	Idate	C
501	30	2	2000
505	150	2	2000
506	150	2	4000
503	200	1	1000
507	200	NULL	2000
509	20	NULL	4000

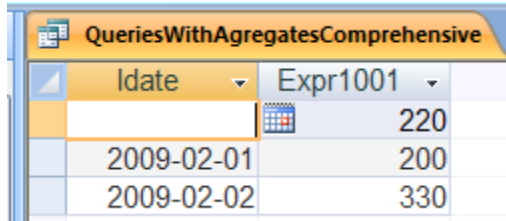
Queries With Aggregates

I	Amt	Idate	C
501	30	2	2000
505	150	2	2000
506	150	2	4000
503	200	1	1000
507	200	NULL	2000
509	20	NULL	4000

- The SELECT statement “understands” that it must work on group, not tuple level

Idate	SUM(Amt)
2	330
1	200
NULL	220

In Microsoft Access



The screenshot shows a table with two columns: 'Idate' and 'Expr1001'. The 'Idate' column contains three rows: a blank row, '2009-02-01', and '2009-02-02'. The 'Expr1001' column contains the values 220, 200, and 330 respectively. The table is sorted by the 'Idate' column, with the blank row at the top.

Idate	Expr1001
	220
2009-02-01	200
2009-02-02	330

- Note that Microsoft Access sorted on the Idate column (with NULL “smaller than” everything else)
- This is fine, as any order in a table is permitted, including sorting rows
- Actually, not a bad idea, but I did differently to show you that sorting is not a requirement, just implementation choice

Queries With Aggregates

- Not necessary to have the WHERE clause, if all tuples should be considered for the GROUP BY operation
- Not necessary to have the HAVING clause, if all groups are good

Queries With Aggregates

- In the SELECT line only a group property can be listed, so, the following is OK, as each of the items listed is a group property

```
SELECT SUM(Amt), MIN(Amt)
FROM Invoice
WHERE C <> 3000
GROUP BY Idate
HAVING AVG(Amt) > 100;
```

- We could list Idate too, as it is a group property too

```
SELECT Idate, SUM(Amt), MIN(Amt)
FROM Invoice
WHERE C <> 3000
GROUP BY Idate
HAVING AVG(Amt) > 100;
```

Skip to page 143

In Microsoft Access

QueriesWithAggregatesTwoSelected	
Expr1000	Expr1001
220	20
200	200
330	30

QueriesWithAggregatesThreeSelected		
ldate	Expr1001	Expr1002
	220	20
2009-02-01	200	200
2009-02-02	330	30

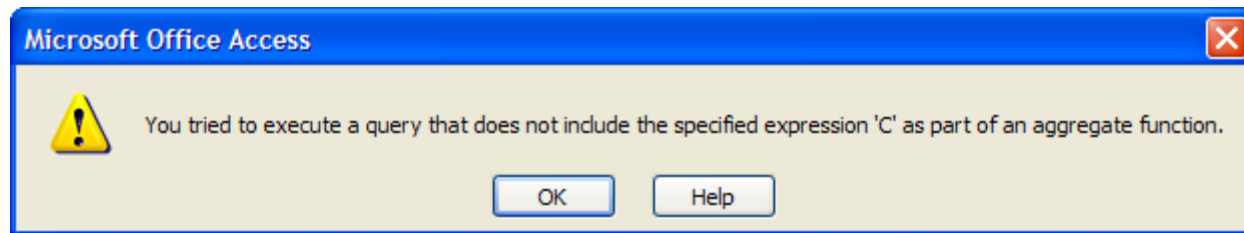
Queries With Aggregates

- But, the following is not OK, as C is not a group property, because on a specific Idate different C's can place an order

```
SELECT C  
FROM Invoice  
WHERE C <> 3000  
GROUP BY Idate  
HAVING AVG(Amt) > 100;
```

In Microsoft Access

- Got it right!



Queries With Aggregates

- One can aggregate on more than one attribute, so that the following query (shown schematically) is possible

```
SELECT Amt, Idate, MIN(C)  
FROM Invoice  
WHERE ...  
GROUP BY Amt, Idate  
HAVING ...;
```

- This will put in a single group all orders for some specific Amt placed on some specific Idate

In Microsoft Access

QueryWithAgregatesOnTwoFields		
Amt	ldate	Expr1002
20		4000
20	2009-02-03	1000
30	2009-02-02	2000
150	2009-02-02	2000
160	2009-02-03	1000
200		2000
200	2009-02-01	1000
300	2009-02-03	3000

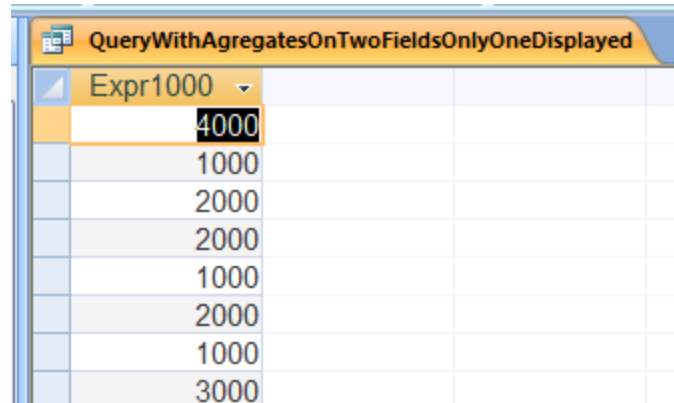
Queries With Aggregates

- The following is permitted also

```
SELECT MIN(C)  
FROM Invoice  
WHERE ...  
GROUP BY Amt, Idate  
HAVING ...;
```

- MIN(C) is a group property since every group has a single value of MIN(C)

In Microsoft Access



The screenshot shows a Microsoft Access query window titled "QueryWithAgregatesOnTwoFieldsOnlyOneDisplayed". The query results are displayed in a table with a single column labeled "Expr1000". The first row shows the aggregate result "4000", which is highlighted with a black background. The subsequent rows show individual data values: 1000, 2000, 2000, 1000, 2000, 1000, and 3000.

Expr1000
4000
1000
2000
2000
1000
2000
1000
3000

Additional/Advanced Operations in SQL

Subqueries

- In a SELECT statement, the WHERE clause can refer to a result of another query, thought of as an “inner loop,” referred to as a subquery
- Consider two relations R(A,B) and S(C,D)
- ```
SELECT A
FROM R
WHERE B > (SELECT MIN(C)
 FROM S)
```
- This will pick up all values of column A of R if the corresponding B is larger than the smallest element in the C column of S
- Generally, a result of a subquery is either one element (perhaps with duplicates) as in the above example or more than one element
- Subqueries are used very frequently, so we look at details
- We start with one element subquery results

# Subqueries

- Find a list of all I for orders that are bigger than the smallest order placed on the same date.

```
SELECT I
FROM Invoice AS Invoice1
WHERE Amt >
(SELECT MIN(Amt)
FROM Invoice
WHERE Idate = Invoice1.Idate);
```

- For each tuple of Invoice1 the value of Amt is compared to the result of the execution of the subquery.
  - The subquery is executed (logically) for each tuple of Invoice
  - This looks very much like an inner loop, executed logically once each time the outer loop “makes a step forward”
- Note that we needed to rename Invoice to be Invoice1 so that we can refer to it appropriately in the subquery.
- In the subquery unqualified Idate refers to the nearest encompassing Invoice

# Subqueries

| SubqueryOrdersBiggerThanSmallestOnThatDate |     |  |
|--------------------------------------------|-----|--|
|                                            |     |  |
|                                            | 502 |  |
|                                            | 504 |  |
|                                            | 505 |  |
|                                            | 506 |  |

# ***Subqueries***

- In addition to the  $>$  operator, we could also use other standard comparison operators between two tuple values, such as  $\geq$ ,  $<$ , etc.,
- For such comparison operators, we need to be sure that the subquery is syntactically (i.e., by its syntax) guaranteed to return only one value
- Subqueries do not add any expressive power but one needs to be careful in tracking duplicates
  - We will not do it here
- Benefits of subqueries
  - Some people find them more readable
  - Perhaps easier for the system to implement efficiently (but usually not)  
Perhaps by realizing that the inner loop is independent of the outer loop and can be executed only once

# Subqueries

- Find a list of all I for orders that are bigger than the smallest order placed on the same date
- The following will give the same result, but more clumsily than using subqueries. It might however be faster in some systems.

1. `SELECT Idate, MIN(Amt) AS MinAmt  
INTO InvoiceTemp01  
FROM Invoice  
GROUP BY Idate;`
2. `SELECT Invoice.I  
FROM Invoice, InvoiceTemp01  
WHERE Invoice.Idate = InvoiceTemp01.Idate AND Amt >  
MinAmt;`

# Subqueries

| QueryCreateInvoiceTemp01 |        | InvoiceTemp01 |
|--------------------------|--------|---------------|
| Idate                    | MinAmt |               |
|                          | 20     |               |
| 2009-02-01               | 200    |               |
| 2009-02-02               | 30     |               |
| 2009-02-03               | 20     |               |

| QueryUsingInvoiceTemp01 |     |
|-------------------------|-----|
|                         |     |
|                         | 505 |
|                         | 506 |
|                         | 502 |
|                         | 504 |



## ***Subqueries Returning a Set of Values***

- In general, a subquery could return a set of values, that is relations with more than one row in general
- In this case, we use operators that can compare a single value with a set of values.
- The two keywords are **ANY** and **ALL**
- Let  $v$  be a value,  $r$  a set of values, and  $op$  a comparison operator

Then

- “ $v \text{ op ANY } r$ ” is true if and only if  $v \text{ op } x$  is true for at least one  $x$  in  $r$
- “ $v \text{ op ALL } r$ ” is true if and only if  $v \text{ op } x$  is true for each  $x$  in  $r$

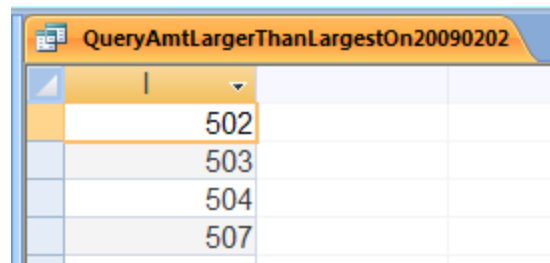
## ***Subqueries With ALL and ANY***

- Find every I for which Amt is larger than the largest Amt on February 2, 2009

```
SELECT I
FROM Invoice
WHERE Amt > ALL
(SELECT Amt
FROM Invoice
WHERE Idate = #2009-02-02#);
```

- Note, loosely speaking:  $> \text{ALL } X$  means that for every  $x$  in  $X$ ,  $> x$  holds

# ***Subqueries With ALL and ANY***



| QueryAmtLargerThanLargestOn20090202 |     |  |  |
|-------------------------------------|-----|--|--|
|                                     | 502 |  |  |
|                                     | 503 |  |  |
|                                     | 504 |  |  |
|                                     | 507 |  |  |

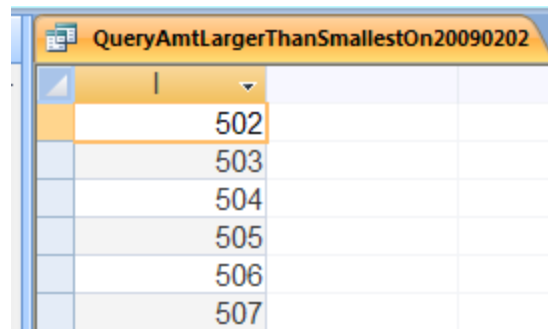
## *Subqueries With ALL and ANY*

- Find every I for which Amt is larger than the smallest Amt on February 2, 2009

```
SELECT I
FROM Invoice
WHERE Amt > ANY
(SELECT Amt
FROM Invoice
WHERE Idate = #2009-02-02#);
```

- Note, loosely speaking:  $> ANY X$  means that for at least one  $x$  in  $X$ ,  $> x$  holds

# ***Subqueries With ALL and ANY***



|     |
|-----|
| 502 |
| 503 |
| 504 |
| 505 |
| 506 |
| 507 |

## **= ALL and = ANY**

- What does = ANY mean?
  - Equal to at least one element in the result of the subquery
  - It is possible to write “IN” instead of “= ANY”
  - But better check what happens with NULLs (we do not do it here)
- What does <> ALL mean?
  - Different from every element in the subquery
  - It is possible to write “NOT IN” instead of “<> ALL”
  - But better check what happens with NULLs (we do not do it here)
- What does <> ANY mean?
  - Not equal to at least one element in the result of the subquery
  - But better check what happens with NULLs (we do not do it here)
- What does = ALL mean?
  - Equal to every element in the result of the subquery (so if the subquery has two distinct elements in the output this will be false)
  - But better check what happens with NULLs (we do not do it here)
- Note that in some systems write != instead of <>

## *Subqueries With ALL and ANY*

- Assume we have R(A,B,C) and S(A,B,C,D)
- Some systems permit comparison of tuples, such as

```
SELECT A
FROM R
WHERE (B,C) = ANY
(SELECT B, C
FROM S);
```

But some do not; then **EXISTS**, which we will see next, can be used

# *Testing for Emptiness*

- It is possible to test whether the result of a subquery is an empty relation by means of the operator **EXISTS**
- “**EXISTS R**” is true if and only if R is not empty
  - So read this: “there exists a tuple in R”
- “**NOT EXISTS R**” is true if and only if R is empty
  - So read this: “there does not exist a tuple in R”
- ***These are very important***, as they are frequently used to implement difference (MINUS or EXCEPT) and intersection (INTERSECT)
- First, a little practice, then how to do the set operations



# *Testing for Emptiness*

- Find all Cnames who do not have an entry in Invoice

```
SELECT Cname
FROM Customer
WHERE NOT EXISTS
(SELECT *
FROM Invoice
WHERE Customer.C = Invoice.C);
```

# ***Testing for Non-Emptiness***

- Find all Cnames who have an entry in Invoice

```
SELECT Cname
FROM Customer
WHERE EXISTS
(SELECT *
FROM Invoice
WHERE Customer.C = Invoice.C);
```

Skip to page 172

# ***Implementing Intersection And Difference If They Are Not Directly Available***

- We use example database ***SetOperationsInSql.mdb***
- In general, use EXISTS and NOT EXISTS
- If the tables have only one column, you may see advice to use IN and NOT IN: ***don't do that***: problems with NULLs

# Set Intersection (*INTERSECT*)

## Use *EXISTS*

```
SELECT DISTINCT *
FROM R
WHERE EXISTS
(SELECT *
FROM S
WHERE
R.First = S.First AND R.Second = S.Second);
```

| First | Second |
|-------|--------|
| a     | b      |
| a     | c      |
| b     | a      |
| b     | a      |
| b     | a      |
| b     | a      |
| b     | a      |
| b     | c      |
| b     |        |
|       | c      |
|       | c      |

| First | Second |
|-------|--------|
| b     | a      |
| b     | a      |
| b     | c      |
| c     | a      |
| c     | b      |
|       | c      |

| First | Second |
|-------|--------|
| b     | a      |
| b     | c      |

- Note that a tuple containing nulls, (NULL,c), is not in the result, and it should not be there

# ***Set Intersection (INTERSECT)***

## ***Can Also Be Done Using Cartesian Product***

```
SELECT DISTINCT *
FROM R, S
WHERE
R.First = S.First AND R.Second = S.Second)
```

# Set Difference (MINUS/EXCEPT) Use NOT EXISTS

```
SELECT DISTINCT *
FROM R
WHERE NOT EXISTS
(SELECT *
FROM S
WHERE
R.First = S.First AND R.Second = S.Second);
```

| Intersect | R      | S |
|-----------|--------|---|
| First     | Second |   |
| a         | b      |   |
| a         | c      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | c      |   |
| b         |        |   |
|           | c      |   |
|           | c      |   |

| Intersect | R      | S |
|-----------|--------|---|
| First     | Second |   |
| b         | a      |   |
| b         | a      |   |
| b         | c      |   |
| c         | a      |   |
| c         | b      |   |
|           | c      |   |

| Minus | R      | S |
|-------|--------|---|
| First | Second |   |
|       | c      |   |
| a     | b      |   |
| a     | c      |   |
| b     |        |   |

- Note that tuples containing nulls, (b,NULL) and (NULL,c), are in the result, and they should be there

# Accounting For NULLs (Perhaps Semantically Incorrectly)

```
SELECT DISTINCT *
FROM R
WHERE EXISTS (SELECT *
FROM S
WHERE (R.First = S.First AND R.Second = S.Second) OR
(R.First IS NULL AND S.First IS NULL AND R.Second =
S.Second) OR (R.First = S.First AND R.Second IS NULL AND
S.Second IS NULL) OR (R.First IS NULL AND S.First IS NULL
AND R.Second IS NULL AND S.Second IS NULL));
```

| Intersect | R      | S |
|-----------|--------|---|
| First     | Second |   |
| a         | b      |   |
| a         | c      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | c      |   |
| b         |        |   |
|           | c      |   |
|           | c      |   |

| Intersect | R      | S |
|-----------|--------|---|
| First     | Second |   |
| b         | a      |   |
| b         | a      |   |
| b         | c      |   |
| c         | a      |   |
| c         | b      |   |
|           | c      |   |

| R     | S      | IntersectAssuming! |
|-------|--------|--------------------|
| First | Second |                    |
|       | c      |                    |
| b     | a      |                    |
| b     | c      |                    |

# Accounting For NULLs (Perhaps Semantically Incorrectly)

```

SELECT DISTINCT *
FROM R
WHERE NOT EXISTS (SELECT *
FROM S
WHERE (R.First = S.First AND R.Second = S.Second) OR
(R.First IS NULL AND S.First IS NULL AND R.Second =
S.Second) OR (R.First = S.First AND R.Second IS NULL AND
S.Second IS NULL) OR (R.First IS NULL AND S.First IS NULL
AND R.Second IS NULL AND S.Second IS NULL));

```

| Intersect | R      | S |
|-----------|--------|---|
| First     | Second |   |
| a         | b      |   |
| a         | c      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | a      |   |
| b         | c      |   |
| b         |        |   |
|           | c      |   |
|           | c      |   |

| Intersect | R      | S |
|-----------|--------|---|
| First     | Second |   |
| b         | a      |   |
| b         | a      |   |
| b         | c      |   |
| c         | a      |   |
| c         | b      |   |
|           | c      |   |

| R     | S      | MinusAssumingNu |
|-------|--------|-----------------|
| First | Second |                 |
| a     | b      |                 |
| a     | c      |                 |
| b     |        |                 |



# Set Intersection For Tables With One Column

```
SELECT DISTINCT *
FROM P
WHERE A IN (SELECT A
FROM Q);
```

| P | Q |
|---|---|
| A |   |
| a |   |
| b |   |
| b |   |
| c |   |
| a |   |

| P | Q |
|---|---|
| A |   |
| b |   |
| c |   |
| c |   |
| c |   |

| P | Q |
|---|---|
| A |   |
| b |   |
| c |   |

# Set Difference For Tables With One Column

```
SELECT DISTINCT *
FROM P
WHERE A NOT IN (SELECT A
FROM Q);
```

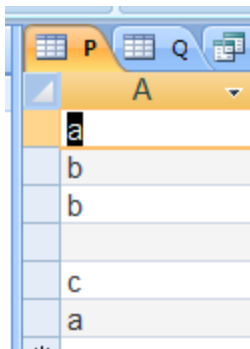


Table P: A single column 'A' with values a, b, b, c, a.

| A |
|---|
| a |
| b |
| b |
| c |
| a |

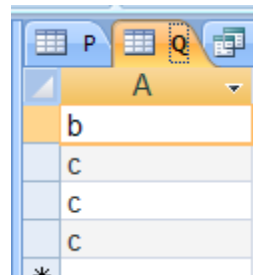
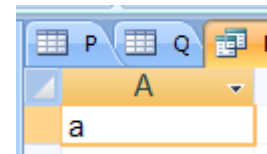


Table Q: A single column 'A' with values b, c, c, c.

| A |
|---|
| b |
| c |
| c |
| c |



Result of the query: A single column 'A' with value a.

| A |
|---|
| a |

- Note (NULL) is not in the result, so our query is not quite correct (as I have warned you earlier)

## *Using More Than One Column Name*

- Assume we have R(A,B,C) and S(A,B,C,D)
- Some systems do not allow the following (more than one item = ANY)

```
SELECT A
FROM R
WHERE (B,C) = ANY
(SELECT B, C
FROM S);
```

we can use

```
SELECT A
FROM R
WHERE EXISTS
(SELECT *
FROM S
WHERE R.B = S.B AND R.C = S.C);
```

# Joins

- SQL has a variety of “modified” Cartesian Products, called **joins**
- They are also very popular
- The interesting ones are **outer joins**, interesting when there are no matches where the condition is equality
  - Left outer join
  - Right outer join
  - Full outer join
- We will use new tables to describe them, see **OuterJoins.mdb** in Extras

| R | A | B |
|---|---|---|
|   | a | 1 |
|   | b | 2 |
|   | c | 3 |

| S | C | D |
|---|---|---|
|   | 1 | e |
|   | 2 | f |
|   | 2 | g |
|   | 4 | h |

# LEFT OUTER JOIN

- `SELECT *`  
`FROM R LEFT OUTER JOIN S`  
`ON R.B = S.C;`
- Includes all rows from the first table, matched or not, plus matching “pieces” from the second table, where applicable.
- For the rows of the first table that have no matches in the second table, NULLs are added for the columns of the second table

| R | A | B |
|---|---|---|
|   | a | 1 |
|   | b | 2 |
|   | c | 3 |

| S | C | D |
|---|---|---|
|   | 1 | e |
|   | 2 | f |
|   | 2 | g |
|   | 4 | h |

|  | A | B | C | D |
|--|---|---|---|---|
|  | a | 1 | 1 | e |
|  | b | 2 | 2 | f |
|  | b | 2 | 2 | g |
|  | c | 3 |   |   |

# ***In Microsoft Access***

| R S QueryLeftJoin QueryRightJoin |   |   |   |
|----------------------------------|---|---|---|
| A                                | B | C | D |
| a                                | 1 | 1 | e |
| b                                | 2 | 2 | g |
| b                                | 2 | 2 | f |
| c                                | 3 |   |   |

# RIGHT OUTER JOIN

- **SELECT \***  
**FROM R RIGHT OUTER JOIN S**  
**ON R.B = S.C;**
- Includes all rows from the second table, matched or not, plus matching “pieces” from the first table, where applicable.
- For the rows of the second table that have no matches in the first table, NULLs are added for the columns of the first table

| R | A | B |
|---|---|---|
|   | a | 1 |
|   | b | 2 |
|   | c | 3 |

| S | C | D |
|---|---|---|
|   | 1 | e |
|   | 2 | f |
|   | 2 | g |
|   | 4 | h |

|  | A | B | C | D |
|--|---|---|---|---|
|  | a | 1 | 1 | e |
|  | b | 2 | 2 | f |
|  | b | 2 | 2 | g |
|  |   |   | 4 | h |

# ***In Microsoft Access***

| QueryLeftJoin |  |   |  | QueryRightJoin |  |                |  |
|---------------|--|---|--|----------------|--|----------------|--|
| R             |  | S |  | QueryLeftJoin  |  | QueryRightJoin |  |
| A             |  | B |  | C              |  | D              |  |
| a             |  | 1 |  | 1              |  | e              |  |
| b             |  | 2 |  | 2              |  | f              |  |
| b             |  | 2 |  | 2              |  | g              |  |
|               |  |   |  | 4              |  | h              |  |



# ***FULL OUTER JOIN***

- **SELECT \***  
**FROM R FULL OUTER JOIN S**  
**ON R.B = S.C;**

| R | A | B |
|---|---|---|
|   | a | 1 |
|   | b | 2 |
|   | c | 3 |

| S | C | D |
|---|---|---|
|   | 1 | e |
|   | 2 | f |
|   | 2 | g |
|   | 4 | h |

|  | A | B | C | D |
|--|---|---|---|---|
|  | a | 1 | 1 | e |
|  | b | 2 | 2 | f |
|  | b | 2 | 2 | g |
|  | c | 3 |   |   |
|  |   |   | 4 | h |

- Cannot be done in Microsoft Access directly
- Can use Left Outer and Right Outer and then do a union

# INNER JOIN

- `SELECT *`  
`FROM R INNER JOIN S`  
`ON R.B = S.C;`
- This is the same as  
`SELECT *`  
`FROM R, S`  
`WHERE R.B = S.C;`

| R | A | B |
|---|---|---|
|   | a | 1 |
|   | b | 2 |
|   | c | 3 |

| S | C | D |
|---|---|---|
|   | 1 | e |
|   | 2 | f |
|   | 2 | g |
|   | 4 | h |

|  | A | B | C | D |
|--|---|---|---|---|
|  | a | 1 | 1 | e |
|  | b | 2 | 2 | f |
|  | b | 2 | 2 | g |

- Some systems use just `JOIN` instead of `INNER JOIN`

# ***In Microsoft Access***

| QueryLeftJoin |   | QueryRightJoin |   | QueryInnerJoin |  |
|---------------|---|----------------|---|----------------|--|
|               | A | B              | C | D              |  |
|               | a | 1              | 1 | e              |  |
|               | b | 2              | 2 | g              |  |
|               | b | 2              | 2 | f              |  |

# *Ranges and Templates*

- It is possible to specify ranges, or templates:
- Find all P and Pcity for plants in cities starting with letters B through D  

```
SELECT P, Pcity
FROM Plant
WHERE ((City BETWEEN 'B' AND 'E') AND (Pcity <> 'E'));
```

  - Note that we want all city values in the range B through DZZZZZ.....; thus the value E is too big, as BETWEEN includes the “end values.”

# ***In Microsoft Access***

| QueryPAndPcityForPcityBetweenBAndDZZZZZ |         |  |
|-----------------------------------------|---------|--|
| P                                       | Pcity   |  |
| 901                                     | Boston  |  |
| 902                                     | Boston  |  |
| 903                                     | Chicago |  |
| 904                                     | Chicago |  |
| 905                                     | Denver  |  |
| 908                                     | Boston  |  |

# ***Ranges and Templates***

- Find Pnames for cities containing the letter X in the second position:

```
SELECT Pname
FROM Plant
WHERE (City LIKE '_X%');
```

- % stands for 0 or more characters; \_ stands for exactly one character.

## ***Presenting the Result***

- It is possible to manipulate the resulting answer to a query. We present the general features by means of examples.
- For each P list the profit in thousands, order by profits in decreasing order and for the same profit value, order by increasing P:

```
SELECT Profit/1000 AS Thousands, P
FROM Plant
ORDER BY Profit DESC, P ASC;
```

# ***In Microsoft Access***

| QueryProfitDescendingPAscending |     |
|---------------------------------|-----|
| Thousands                       | P   |
| 65                              | 907 |
| 56                              | 902 |
| 51                              | 904 |
| 51                              | 906 |
| 51                              | 908 |
| 48                              | 905 |
| 45                              | 901 |
|                                 | 903 |

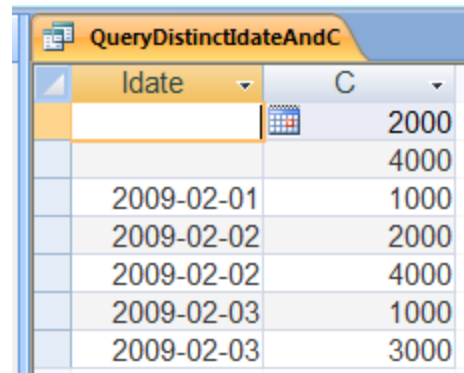


## ***Presenting the Result***

- Create the relation with attributes Idate, C while removing duplicate rows.

```
SELECT DISTINCT Idate, C
FROM Invoice;
```

# ***In Microsoft Access***



The screenshot shows a Microsoft Access query result grid. The title bar of the grid is 'QueryDistinctIdateAndC'. The grid has two columns: 'Idate' and 'C'. The 'Idate' column contains dates, and the 'C' column contains numerical values. The data is as follows:

| Idate      | C    |
|------------|------|
|            | 2000 |
|            | 4000 |
| 2009-02-01 | 1000 |
| 2009-02-02 | 2000 |
| 2009-02-02 | 4000 |
| 2009-02-03 | 1000 |
| 2009-02-03 | 3000 |

# Testing For Duplicates

- It is possible to test if a subquery returns any duplicate tuples, with NULLs ignored
- Find all Cnames that all of whose orders are for different amounts (including, of course those who have placed no orders)

```
SELECT Cname
FROM Customer
WHERE UNIQUE
(SELECT Amt
FROM Invoice
WHERE Customer.C = C);
```

- **UNIQUE** is true if there are no duplicates in the answer, but there could be several tuples, as long as all are different
- If the subquery returns an empty table, UNIQUE is true
- Recall, that we assumed that our original relations had no duplicates; that's why the answer is correct

# Testing For Duplicates

- It is possible to test if a subquery returns any duplicate tuples, with NULLs being ignored
- Find all Cnames that have at least two orders for the same amount

```
SELECT Cname
FROM Customer
WHERE NOT UNIQUE
(SELECT Amt
FROM Invoice
WHERE Customer.C = C);
```

- **NOT UNIQUE** is true if there are duplicates in the answer
- Recall, that we assumed that our original relations had no duplicates; that's why the answer is correct

***Modifying The Database***  
***INSERT, DELETE, UPDATE***

# ***Modifying the Database***

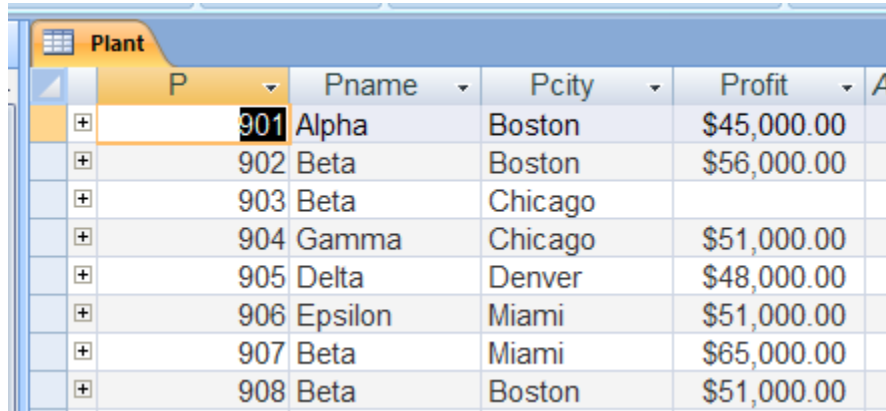
- Until now, no operations were done that modified the database
- We were operating in the realm of algebra, that is, expressions were computed from inputs.
- For a real system, we need the ability to modify the relations
- The three main constructs for modifying the relations are:
  - Insert
  - Delete
  - Update
- This in general is theoretically, especially update, quite tricky; so be careful
- Duplicates are not removed

## *Insertion of a Tuple*

- `INSERT INTO Plant (P, Pname, Pcity, Profit)`  
`VALUES ('909','Gamma',Null,52000);`
- If it is clear which values go where (values listed in the same order as the columns), the names of the columns may be omitted

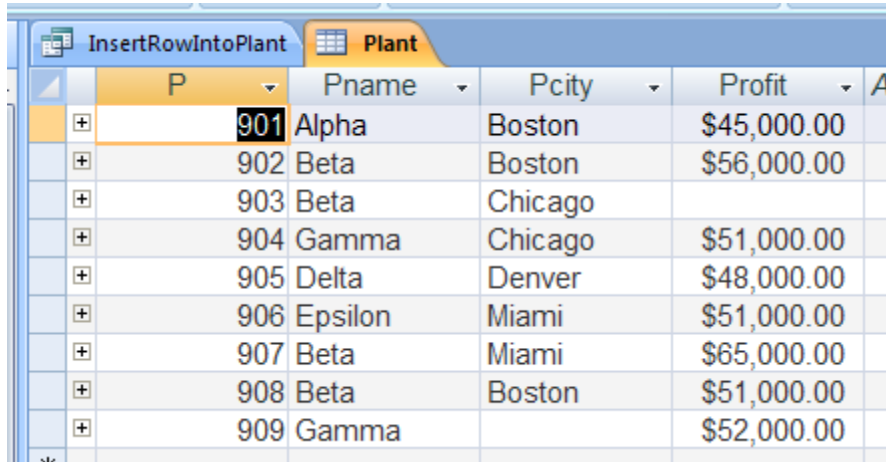
```
INSERT INTO Plant
VALUES ('909','Gamma',Null,52000);
```

# ***In Microsoft Access***



A screenshot of the Microsoft Access interface showing a table named 'Plant'. The table has five columns: 'P' (a dropdown menu), 'Pname', 'Pcity', 'Profit', and 'A'. The data is as follows:

| P   | Pname   | Pcity   | Profit      | A |
|-----|---------|---------|-------------|---|
| 901 | Alpha   | Boston  | \$45,000.00 |   |
| 902 | Beta    | Boston  | \$56,000.00 |   |
| 903 | Beta    | Chicago |             |   |
| 904 | Gamma   | Chicago | \$51,000.00 |   |
| 905 | Delta   | Denver  | \$48,000.00 |   |
| 906 | Epsilon | Miami   | \$51,000.00 |   |
| 907 | Beta    | Miami   | \$65,000.00 |   |
| 908 | Beta    | Boston  | \$51,000.00 |   |



A screenshot of the Microsoft Access interface showing the 'Plant' table with 9 rows. The 'InsertRowIntoPlant' ribbon tab is active. The data is as follows:

| P   | Pname   | Pcity   | Profit      | A |
|-----|---------|---------|-------------|---|
| 901 | Alpha   | Boston  | \$45,000.00 |   |
| 902 | Beta    | Boston  | \$56,000.00 |   |
| 903 | Beta    | Chicago |             |   |
| 904 | Gamma   | Chicago | \$51,000.00 |   |
| 905 | Delta   | Denver  | \$48,000.00 |   |
| 906 | Epsilon | Miami   | \$51,000.00 |   |
| 907 | Beta    | Miami   | \$65,000.00 |   |
| 908 | Beta    | Boston  | \$51,000.00 |   |
| 909 | Gamma   |         | \$52,000.00 |   |



## ***Insertion of a Tuple***

- If values of some columns are not specified, the default values (if specified in SQL DDL, as we will see later; or perhaps NULL) will be automatically added
- **INSERT INTO** Plant (P, Pname, Pcity)  
**VALUES** ('910','Gamma',Null);

# ***In Microsoft Access***

| InsertRowIntoPlant |     |         |         |             |   |
|--------------------|-----|---------|---------|-------------|---|
| Plant              |     |         |         |             |   |
|                    | P   | Pname   | Pcity   | Profit      | A |
| +                  | 901 | Alpha   | Boston  | \$45,000.00 |   |
| +                  | 902 | Beta    | Boston  | \$56,000.00 |   |
| +                  | 903 | Beta    | Chicago |             |   |
| +                  | 904 | Gamma   | Chicago | \$51,000.00 |   |
| +                  | 905 | Delta   | Denver  | \$48,000.00 |   |
| +                  | 906 | Epsilon | Miami   | \$51,000.00 |   |
| +                  | 907 | Beta    | Miami   | \$65,000.00 |   |
| +                  | 908 | Beta    | Boston  | \$51,000.00 |   |
| +                  | 909 | Gamma   |         | \$52,000.00 |   |

| InsertTupleIntoPlantWithDefaultValue |     |         |         |             |   |
|--------------------------------------|-----|---------|---------|-------------|---|
| Plant                                |     |         |         |             |   |
|                                      | P   | Pname   | Pcity   | Profit      | A |
| +                                    | 901 | Alpha   | Boston  | \$45,000.00 |   |
| +                                    | 902 | Beta    | Boston  | \$56,000.00 |   |
| +                                    | 903 | Beta    | Chicago |             |   |
| +                                    | 904 | Gamma   | Chicago | \$51,000.00 |   |
| +                                    | 905 | Delta   | Denver  | \$48,000.00 |   |
| +                                    | 906 | Epsilon | Miami   | \$51,000.00 |   |
| +                                    | 907 | Beta    | Miami   | \$65,000.00 |   |
| +                                    | 908 | Beta    | Boston  | \$51,000.00 |   |
| +                                    | 909 | Gamma   |         | \$52,000.00 |   |
| +                                    | 910 | Gamma   |         |             |   |

## ***Insertion From A Table***

- Assume we have a tableCandidate(C,Cname,Ccity,Good) listing potential customers
  - First, for each potential customer, the value of Good is Null
  - Later it becomes either Yes or No
- We can insert part of this “differential table” into customers:

```
INSERT INTO Customer (C, Cname, Ccity, P)
SELECT C, Cname, Ccity, NULL
FROM Candidate
WHERE Good = 'YES';
```
- In general, we can insert any result of a query, as long as compatible, into a table

# *In Microsoft Access*

| Customer Candidate |       |       |         |     |
|--------------------|-------|-------|---------|-----|
| C                  | Cname | Ccity | P       | A   |
| +                  | 1000  | Doe   | Boston  | 901 |
| +                  | 2000  | Yao   | Boston  | 902 |
| +                  | 3000  | Doe   | Chicago | 903 |
| +                  | 4000  | Doe   | Seattle |     |
| +                  | 5000  | Brown | Denver  | 903 |
| +                  | 6000  | Smith | Seattle | 907 |
| +                  | 7000  | Yao   | Chicago | 904 |
| +                  | 8000  | Smith | Denver  | 904 |
| +                  | 9000  | Smith | Boston  | 903 |

| Customer Candidate |       |         |      |  |
|--------------------|-------|---------|------|--|
| C                  | Cname | Ccity   | Good |  |
| 9001               | Qin   | Boston  | YES  |  |
| 9002               | Doe   | Chicago | NO   |  |
| 9003               | Rao   | Chicago |      |  |

| Candidate InsertGoodCandidatesIntoCustomer Customer |       |       |         |     |  |
|-----------------------------------------------------|-------|-------|---------|-----|--|
| C                                                   | Cname | Ccity | P       | Ac  |  |
| +                                                   | 1000  | Doe   | Boston  | 901 |  |
| +                                                   | 2000  | Yao   | Boston  | 902 |  |
| +                                                   | 3000  | Doe   | Chicago | 903 |  |
| +                                                   | 4000  | Doe   | Seattle |     |  |
| +                                                   | 5000  | Brown | Denver  | 903 |  |
| +                                                   | 6000  | Smith | Seattle | 907 |  |
| +                                                   | 7000  | Yao   | Chicago | 904 |  |
| +                                                   | 8000  | Smith | Denver  | 904 |  |
| +                                                   | 9000  | Smith | Boston  | 903 |  |
| +                                                   | 9001  | Qin   | Boston  |     |  |

# *Deletion*

- `DELETE`  
`FROM Candidate`  
`WHERE Good = 'Yes';`
- This removes rows satisfying the specified condition
  - In our example, once some candidates were promoted to customers, they are removed from Candidate

# ***In Microsoft Access***

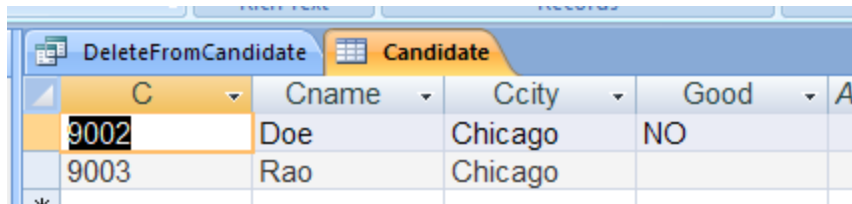
| Customer |       | Candidate |      |  |
|----------|-------|-----------|------|--|
| C        | Cname | Ccity     | Good |  |
| 9001     | Qin   | Boston    | YES  |  |
| 9002     | Doe   | Chicago   | NO   |  |
| 9003     | Rao   | Chicago   |      |  |

| DeleteFromCandidate |       | Candidate |      |   |
|---------------------|-------|-----------|------|---|
| C                   | Cname | Ccity     | Good | A |
| 9002                | Doe   | Chicago   | NO   |   |
| 9003                | Rao   | Chicago   |      |   |

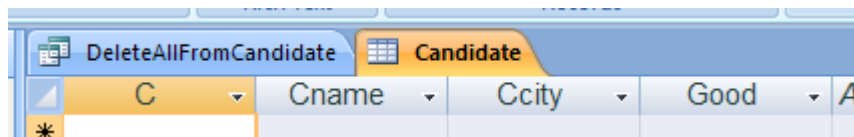
# ***Deletion***

- **DELETE**  
**FROM** Candidate;
- This removes all the rows of a table, leaving an empty table; but the table remains
- Every row satisfied the empty condition, which is equivalent to: “WHERE TRUE”

# ***In Microsoft Access***



| C    | Cname | Ccity   | Good | A |
|------|-------|---------|------|---|
| 9002 | Doe   | Chicago | NO   |   |
| 9003 | Rao   | Chicago |      |   |



| C | Cname | Ccity | Good | A |
|---|-------|-------|------|---|
| * |       |       |      |   |



## ***Another Way to Compute Difference***

- Standard SQL operations, such as EXCEPT do not work in all implementations.
- To compute  $R(A,B) - S(A,B)$ , and to keep the result in  $R(A,B)$ , one can do:

```
DELETE FROM R
WHERE EXISTS
 (SELECT *
 FROM S
 WHERE R.A = S.A AND R.B = S.B);
```

- But duplicates are not removed
  - Of course no copy of a tuple that appears in both R and S remains in R
  - But if a tuple appears several times in R and does not appear in S, all these copies remain in R

# *Update*

- **UPDATE** Invoice  
**SET** Amt = Amt + 1  
**WHERE** Amt < 200;
- Every tuple that satisfied the WHERE condition is changed in the specified manner (which could in general be quite complex)

# ***In Microsoft Access***

| Invoice   AddToAmtInInvoice |     |            |      |    |  |
|-----------------------------|-----|------------|------|----|--|
| I                           | Amt | Idate      | C    | Ac |  |
| 501                         | 30  | 2009-02-02 | 2000 |    |  |
| 502                         | 300 | 2009-02-03 | 3000 |    |  |
| 503                         | 200 | 2009-02-01 | 1000 |    |  |
| 504                         | 160 | 2009-02-03 | 1000 |    |  |
| 505                         | 150 | 2009-02-02 | 2000 |    |  |
| 506                         | 150 | 2009-02-02 | 4000 |    |  |
| 507                         | 200 |            | 2000 |    |  |
| 508                         | 20  | 2009-02-03 | 1000 |    |  |
| 509                         | 20  |            | 4000 |    |  |

| Invoice   AddToAmtInInvoice |     |            |      |    |  |
|-----------------------------|-----|------------|------|----|--|
| I                           | Amt | Idate      | C    | Ac |  |
| 501                         | 31  | 2009-02-02 | 2000 |    |  |
| 502                         | 300 | 2009-02-03 | 3000 |    |  |
| 503                         | 200 | 2009-02-01 | 1000 |    |  |
| 504                         | 161 | 2009-02-03 | 1000 |    |  |
| 505                         | 151 | 2009-02-02 | 2000 |    |  |
| 506                         | 151 | 2009-02-02 | 4000 |    |  |
| 507                         | 200 |            | 2000 |    |  |
| 508                         | 21  | 2009-02-03 | 1000 |    |  |
| 509                         | 21  |            | 4000 |    |  |

# ***Update***

- But this gets quite “strange,” and incorrect if the same tuple could be updated in different ways if it satisfies a different condition, the system will reject this
- Example
  - A student can have only one major (we will see how to specify this later) and we tell the database to change each student major to X, if the student took a course in department X
  - If students can take courses in several departments, the above cannot work

# ***Recursion***

# ***Recursion***

- We have argued previously that given a relation Birth(Parent,Child) it is not possible to create the associated Lineage(Anccestor,Descendant) using relational algebra (and what we know so far)
- SQL has an extension that allows to do that in a clean way
- Previously strange hacks were needed
- We will look on how to do it next using the cleaner way

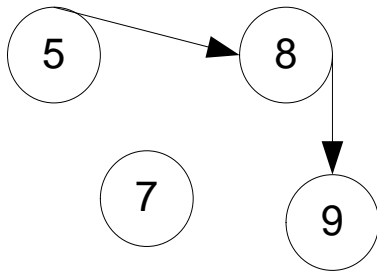
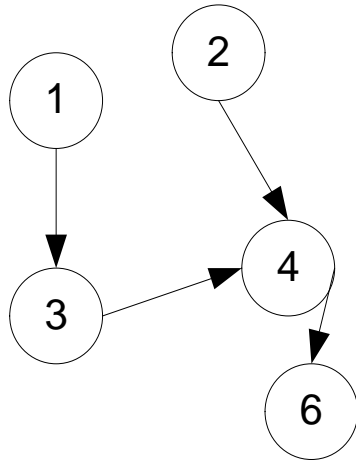
# Recursion in SQL



- `WITH Lineage(ancestor, descendant) AS`  
    (`SELECT Parent, Child`  
      `FROM Birth`  
      `UNION ALL`  
      `SELECT Parent, Descendant`  
      `FROM Lineage`  
      `WHERE Child = Ancestor`  
    )  
  
■ Lineage is initialized as Birth (paths of length 1)  
  
■ In every SELECT it is another generation is “spanned”  
and paths 1 arc longer are added to Lineage, more  
specifically, we go “one generation higher”  
  
■ Next we revisit an old example

# Using Recursion on a DAG

- Given **Arc** we would like to compute **Route**



| Arc | Tail | Head |
|-----|------|------|
| 1   | 3    |      |
| 2   | 4    |      |
| 3   | 4    |      |
| 4   | 6    |      |
| 5   | 8    |      |
| 8   | 9    |      |

| Route | Tail | Head |
|-------|------|------|
| 1     | 3    |      |
| 1     | 4    |      |
| 1     | 6    |      |
| 2     | 4    |      |
| 2     | 6    |      |
| 3     | 4    |      |
| 3     | 6    |      |
| 4     | 6    |      |
| 5     | 8    |      |
| 5     | 9    |      |
| 8     | 9    |      |

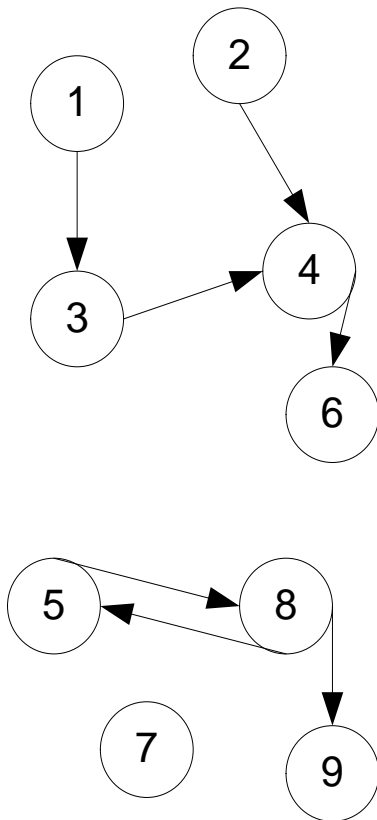


# A Query And Its Execution

|                                | TAIL  | HEAD  |
|--------------------------------|-------|-------|
|                                | ----- | ----- |
| CREATE TABLE arc (             |       |       |
| tail NUMBER,                   |       |       |
| head NUMBER,                   |       |       |
| PRIMARY KEY(tail, head)        |       |       |
| );                             | 1     | 3     |
| INSERT INTO arc VALUES (1, 3); | 2     | 4     |
| INSERT INTO arc VALUES (2, 4); | 3     | 4     |
| INSERT INTO arc VALUES (3, 4); | 4     | 6     |
| INSERT INTO arc VALUES (4, 6); | 5     | 8     |
| INSERT INTO arc VALUES (5, 8); | 8     | 9     |
| INSERT INTO arc VALUES (8, 9); | 1     | 4     |
| WITH route(tail, head) AS      | 2     | 6     |
| (    SELECT a.tail, a.head     | 3     | 6     |
| FROM arc a                     | 5     | 9     |
| UNION ALL                      | 1     | 6     |
| SELECT a.tail, r.head          |       |       |
| FROM route r                   |       |       |
| JOIN arc a                     |       |       |
| ON a.head = r.tail             |       |       |
| );                             |       |       |
| SELECT * FROM route;           |       |       |

# Using Recursion On A Directed Graph

- Given **Arc** we would like to compute **Route**



| Arc | Tail | Head |
|-----|------|------|
| 1   | 3    |      |
| 2   | 4    |      |
| 3   | 4    |      |
| 4   | 6    |      |
| 5   | 8    |      |
| 8   | 5    |      |
| 8   | 9    |      |

| Route | Tail | Head |
|-------|------|------|
| 1     | 3    |      |
| 1     | 4    |      |
| 1     | 6    |      |
| 2     | 4    |      |
| 2     | 6    |      |
| 3     | 4    |      |
| 3     | 6    |      |
| 4     | 6    |      |
| 5     | 8    |      |
| 5     | 9    |      |
| 8     | 5    |      |
| 8     | 9    |      |

# A Query And Its Execution

```
CREATE TABLE arc(
 tail NUMBER,
 head NUMBER,
 PRIMARY KEY(tail, head)
);
INSERT INTO arc VALUES (1,3);
INSERT INTO arc VALUES (2,4);
INSERT INTO arc VALUES (3,4);
INSERT INTO arc VALUES (4,6);
INSERT INTO arc VALUES (5,8);
INSERT INTO arc VALUES (8,5);
INSERT INTO arc VALUES (8,9);
WITH route(tail, head) AS
(
 SELECT a.tail, a.head
 FROM arc a
 UNION ALL
 SELECT a.tail, r.head
 FROM route r
 JOIN arc a
 ON a.head = r.tail
)
SELECT * FROM route;
```

ERROR:  
ORA-32044: cycle  
detected while  
executing recursive  
WITH query

Execution failed because  
the graph was not acyclic  
(it had a cycle)

It is possible to write a  
more complex query that  
could handle more  
general graphs.

# ***Triggers***

# Triggers

- These are actions that can be taken **before/after/instead** **INSERT**, **UPDATE**, or **DELETE**
- Triggers are both complex and powerful, we just touch briefly on them here
- We will discuss:
  - **AFTER** (next)
  - **INSTEAD** (later)
- Assume that after a new Customer is inserted into the database, if Cname is Xiu, the system will “automatically” **CREATE** a new plant in the city Xiu lives, with “properties related to Xiu,” which we will understand by looking at the example
- Let us look at (Prof Kedem tested this in Oracle)
  - The exact trigger in Oracle
  - A partial trace of the execution in Oracle

## *Defining A Trigger*

- **CREATE TRIGGER** Trigger01  
**AFTER INSERT ON** Customer  
**REFERENCING NEW AS** newcustomer  
**FOR EACH ROW**  
**WHEN** (newcustomer.Cname = 'Xiu')  
**BEGIN**  
**INSERT INTO** Plant **VALUES**(:newcustomer.C,  
'Xiu\_Plant', :newcustomer.Ccity, NULL);  
**END** Trigger01;  
  
.  
RUN;
- This was the exact Oracle syntax
- **NEW** refers to added rows
- If rows were deleted (not in our example!), we could refer to them as **OLD**

# Our Database

- Customer and Plant before Insert

| C     | CNAME | CCITY   | P     |
|-------|-------|---------|-------|
| ----- | ----- | -----   | ----- |
| 1000  | Doe   | Boston  | 901   |
| 2000  | Yao   | Boston  | 902   |
| 3000  | Doe   | Chicago | 903   |
| 4000  | Doe   | Seattle |       |
| 5000  | Brown | Denver  | 903   |
| 6000  | Smith | Seattle | 907   |
| 7000  | Yao   | Chicago | 904   |
| 8000  | Smith | Denver  | 904   |
| 9000  | Smith | Boston  | 903   |

| P     | PNAME   | PCITY   | PROFIT |
|-------|---------|---------|--------|
| ----- | -----   | -----   | -----  |
| 901   | Alpha   | Boston  | 45000  |
| 902   | Beta    | Boston  | 56000  |
| 903   | Beta    | Chicago |        |
| 904   | Gamma   | Chicago | 51000  |
| 905   | Delta   | Denver  | 48000  |
| 906   | Epsilon | Miami   | 51000  |
| 907   | Beta    | Miami   | 65000  |
| 908   | Beta    | Boston  | 51000  |

# *Insertion*

- `INSERT INTO Customer  
VALUES(1001,'Xiu','Boston',null);`
  
- Note that the INSERT statement could have inserted many tuples into Customer, for instance, if a whole table was inserted into Customer
  - We had an example of such “candidate customers” being inserted into Customer, once Good became Yes



# Our Database

- Customer and Plant after Insert

| C     | CNAME | CCITY   | P     |
|-------|-------|---------|-------|
| ----- | ----- | -----   | ----- |
| 1000  | Doe   | Boston  | 901   |
| 2000  | Yao   | Boston  | 902   |
| 3000  | Doe   | Chicago | 903   |
| 4000  | Doe   | Seattle |       |
| 5000  | Brown | Denver  | 903   |
| 6000  | Smith | Seattle | 907   |
| 7000  | Yao   | Chicago | 904   |
| 8000  | Smith | Denver  | 904   |
| 9000  | Smith | Boston  | 903   |
| 1001  | Xiu   | Boston  |       |

| P     | PNAME     | PCITY   | PROFIT |
|-------|-----------|---------|--------|
| ----- | -----     | -----   | -----  |
| 901   | Alpha     | Boston  | 45000  |
| 902   | Beta      | Boston  | 56000  |
| 903   | Beta      | Chicago |        |
| 904   | Gamma     | Chicago | 51000  |
| 905   | Delta     | Denver  | 48000  |
| 906   | Epsilon   | Miami   | 51000  |
| 907   | Beta      | Miami   | 65000  |
| 908   | Beta      | Boston  | 51000  |
| 1001  | Xiu_Plant | Boston  |        |

# ***Triggers in MySQL***

<https://dev.mysql.com/doc/refman/5.7/en/trigger-syntax.html>

# PL/SQL

- This is a procedural language extension, and we will look just at an example

```
■ SET SERVEROUTPUT ON
 DECLARE
 VID EMPLOYEE.ID%TYPE;
 BEGIN
 SELECT ID INTO VID
 FROM EMPLOYEE
 WHERE NAME = 'Yiling';
 IF SQL%FOUND THEN
 DBMS_OUTPUT.PUT_LINE('Employee with name
"Yiling" has ID ' || VID);
 END IF;
 END;
/
```

- This will print

Employee with name "Yiling" has ID 0000000003

# ***Programs Interacting With SQL***

# ***DBMS vs. Programming Language***

- DBMS (at our level)
  - knows what tables are
  - does not know what files are
- A Programming Language
  - does not know what tables are
  - knows what files are
- An API or an interface of some sort is provided so
  - that a program works on local variables or files
  - appropriate calls are made to tell the DBMS how the queries/changes to the local variables are to be translated into SQL (or SQL-like) statements that the DBMS understands

# ***SQL Embedded In A Host Language***

- Scenario
  - You go to an ATM to withdraw some money
  - You swipe your card, something (a program, not a relational database) reads it
  - You punch in your PIN, a program reads it
  - The program talks to a relational database to see if things match, assume that they do
  - You ask for a balance, a program reads what you punched and formulates a query to a relational database and understands the answer and shows you on the screen
  - You want to withdraw money, a program formulates a request to the relational database to update your account
  - . . .
- You need to have an interface between a programming language and SQL, such as
- Embedded SQL
- Java Database Connectivity (JDBC) API

# ***SQL Embedded in A Host Language***

- Sometimes, we need to interact with the database from programs written in another host language
- The advantage of this is that we are able to use the structure of the database, its layers, indices, etc
- The disadvantage is, the host language does not understand the concepts of relations, tuples, etc
- We use a version of SQL, called Embedded SQL, for such interactions
- We concentrate on static embedded SQL
- Look at mysql portion of:  
[https://www.python-course.eu/sql\\_python.php](https://www.python-course.eu/sql_python.php)
- Skip the rest of these notes.

# ***SQL Commands As Procedure Calls***

- SQL commands in host languages, could at a gross level be considered procedure calls
- ANSI standard specified Embedded SQL for some programming languages only
- There are two main types of operations:
  - Those working on a tuple
  - Those working on a relation



# ***Common Variables***

- Variables in the host language that are used to communicate with the SQL module must be declared as such
- Assuming we want to act on the relation plants, we would write in our host program something similar to:
  - EXEC SQL BEGIN DECLARE SECTION;  
VAR  
Plant: INTEGER;  
Plantname: ...;  
Plantcity: ...;  
Plantprofit: ...;  
EXEC SQL END DECLARE SECTION;

## ***A Fragment of a Host Program***

- We could write the following program fragment in our host program (note ":" before variable name):

```
EXEC SQL SELECT P
FROM Plant
INTO :Plant
WHERE Profit = :Plantprofit;
```

after Plantprofit is set to a correct value in the host program

- We could also write

```
EXEC SQL INSERT INTO Plant
VALUES(:Plant, :Plantname,
:Plantcity, :Plantprofit);
```

after Plant, Plantname, Plantcity, Plantprofit are set to correct values in the host program

# ***Treatment of NULLS***

- Sometimes the value inserted or retrieved will be NULL
- However host language does not know how the database is coding NULLs.
- It is possible to use special indicator variables to indicate that the value is actually NULL
  - EXEC SQL SELECT profit  
INTO :Plantprofit INDICATOR :Ind  
WHERE C = 75;
- Here if host language variable Ind is negative, it means that Plantprofit does not contain an actual value, but NULL was returned by the SQL system

# SQL Codes

- As part of the declaration section, a variable, generally referred to as **SQLCODE**, must be declared
- It is set by SQL to indicate whether the operation was successful, and if not what kind of problems may have occurred

# ***Handling Sets Of Tuples (Relations)***

- To handle a relation in a host language, we need a looping mechanism that would allow us to go through it a tuple at a time
  - We have seen before how to handle a tuple at a time.
- The mechanism for handling relations is referred to as **CURSOR**

# *Usage Of CURSOR*

- **DECLARE** a **CURSOR**, in a way similar to defining a query
  - As a consequence, the relation is defined, but is not computed
- **OPEN** a **CURSOR**
  - The relation is now computed, but is not accessible.
- **FETCH CURSOR** is executed in order to get a tuple
  - This is repeated, until all tuples are processed
  - The current tuple is referred to as **CURRENT**
  - Of course, some condition must be checked to make sure there are still tuples to be processed. **SQLCODE** is used for this
- **CLOSE** the **CURSOR**
  - Delete the relation

## ***Example Of Using A CURSOR***

- Increase the profit of all plants in Miami by 10%, if the profit is less than 0.1. This is what is written in the host, non-SQL, program

```
Plantcity:='Miami';
EXEC SQL DECLARE CURSOR Todo AS
SELECT *
FROM Plant
WHERE CITY = :Plantcity;
```

```
EXEC SQL OPEN CURSOR Todo;
```

```
WHILE SQLCODE = 0 DO
BEGIN
 EXEC SQL FETCH Todo
 INTO :Plant, :Plantname,
 :Plantcity, :Plantprofit;
 IF :Plantprofit < 0.1 THEN
 EXEC SQL UPDATE Plant
 SET Profit = Profit*1.1
 WHERE CURRENT OF Todo
 END;
```

```
EXEC SQL CLOSE CURSOR Todo;
```

# ***Dynamic Embedded SQL***

- Previously described embedded SQL was static
- The queries were fully specified (the relations, the columns, etc.), therefore they could be preprocessed before the program started executing
- Dynamic embedded SQL allows submission during execution of strings to SQL, which are interpreted and executed
- Useful when program execution can “take many different paths”
- Useful to allow users to submit spontaneous queries during execution of the program



# ***Dynamic Embedded SQL***

- Assume that x is a string variable in your host language
- Put in x a string that is an SQL statement
- **EXEC SQL PREPARE** y from :x ;
  - The string is parsed and compiled and the result put in y, so that the SQL statement is understood and ready to be submitted
- **EXEC SQL EXECUTE** y
  - Execute this SQL statement
- **EXEC SQL EXECUTE IMMEDIATE** :x ;
  - This combines both statements above
  - Good if the statement is executed once only, otherwise, unnecessarily parsing and compiling are repeated for each query execution

# ***Java Database Connectivity***

- There is an API to connect to Oracle DBMS from Java
- This is possible also for some other Database Management Systems
- We will not go over that but you can look at an example (which will not be discussed) in Extras

# ***Predicate Calculus vs. Relational Algebra***

# ***Very Basic Predicate Calculus***

- This is very important conceptually but very rarely covered in Database Systems classes
- We will learn why the operation of division was so difficult to implement and we used double negations (implemented sometimes as double MINUS)
- To remind

$$\{y \mid \exists x[A(x, y)]\}$$

means the set of all  $y$  for which there exists an  $x$  such that  $A(x,y)$  is true

- To remind

$$\{y \mid \forall x[A(x, y)]\}$$

means the set of all  $y$  for which for all  $x$ ,  $A(x,y)$  is true

## ***Expressing Implication Using a Negation and a Disjunction***

- This you probably know, but let's review why  $\alpha \rightarrow \beta$  is the same as (is equivalent to)  $\neg\alpha \vee \beta$
- It is easier to think about the negation of  $\alpha \rightarrow \beta$
- This intuitively means that  $\alpha$  is true but  $\beta$  is false
- Or,  $\alpha \wedge \neg\beta$  is true
- The negation of this is  $\neg(\alpha \wedge \neg\beta)$ , that is  $\neg\alpha \vee \beta$
- And we have shown that

$$\alpha \rightarrow \beta \equiv \neg\alpha \vee \beta$$

## Asking About Some

- List all Persons, who took at least one Required Course
- The result can be expressed using a logical formula with an **existential quantifier**:

$$\{p \mid \exists c[c \in r \wedge (p, c) \in t]\}$$

- We want every Person for which there is a Course in Required, such that (Person, Course) is in Took
- The standard SELECT ... FROM ... WHERE ... easily expresses the existential quantifier above

```
SELECT Took.p
FROM Required, Took
WHERE Required.c = Took.c;
```

# Asking About All

- List all Persons, who Took **at least all** the Courses that are Required
- The result can be expressed using a logical **formula** with a universal quantifier:

$$\{p \mid \forall c[c \in r \rightarrow (p, c) \in t]\}$$

- We can read this as

We want every Person such that for every Course if Course is in Required then (Person, Course) is in Took

- It is not easy to express this using the standard SELECT ... FROM ... WHERE ...



# Expressing Division Using an Existential Quantifier

- We will start with our original expression and replace it by equivalent expressions
- On the right, we provide a justification for the step

$$\{p \mid \forall c[c \in r \rightarrow (p, c) \in t]\}$$

$$\{p \mid \forall c[c \notin r \vee (p, c) \in t]\} \quad \alpha \rightarrow \beta \equiv \neg \alpha \vee \beta$$

$$\{p \mid \neg \exists c \neg [c \notin r \vee (p, c) \in t]\} \quad \forall x[A(x)] \equiv \neg \exists x[\neg A(x)]$$

$$\{p \mid \neg \exists c[c \in r \wedge (p, c) \notin t]\} \quad \neg(\alpha \vee \beta) \equiv \neg \alpha \wedge \neg \beta$$

- This what we did when we computed division earlier, using “double negation”
- Using predicate calculus is easier than using relational algebra

# ***SQL and Predicate Calculus***

- SQL (and relational algebra) implement the existential quantifier essentially directly
- SQL (and relational algebra) do not implement the universal quantifier directly so it needs to be expressed essentially as we did, using the existential quantifier
- There were proposals to actually use predicate calculus and not relational algebra to query databases
- But it was believed that programmers would not want to write logical expressions
- Omitting some minor technicalities, relational algebra is equivalent in expressive power to predicate calculus
- But predicate calculus is more natural, once you learn it
- Proof rather easy, using transformations similar to what we have just done

## ***Summary: Differences Between SQL And “Pure” Relational Algebra***

# Key Differences Between Relational Algebra And SQL

- SQL data model is a **multiset** not a set; still rows in tables (we sometimes continue calling relations)
  - Still no order among rows: no such thing as 1<sup>st</sup> row
  - We can (if we want to) count how many times a particular row appears in the table
  - We can remove/not remove duplicates as we specify (most of the time)
  - There are some operators that specifically pay attention to duplicates
  - We **must** know whether duplicates are removed (and how) for each SQL operation; luckily, easy
- Many redundant operators (relational algebra had only one: intersection)
- SQL provides statistical operators, such as AVG (average)
  - Can be performed on subsets of rows; e.g. average salary per company branch

# ***Key Differences Between Relational Algebra And SQL***

- Every domain is “enhanced” with a special element: NULL
  - Very strange semantics for handling these elements
- “Pretty printing” of output: sorting, and similar
- Operations for
  - Inserting
  - Deleting
  - Changing/updating (sometimes not easily reducible to deleting and inserting)
- “Hooks” for “more general-purpose” languages
- Triggers

# Basic Syntax Comparison

| Relational Algebra                                        | SQL                                                                                                                                          |
|-----------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| $\pi_{a, b}$                                              | SELECT a, b                                                                                                                                  |
| $\sigma_{(d > e) \wedge (f = g)}$                         | WHERE d > e AND f = g                                                                                                                        |
| $p \times q$                                              | FROM p, q                                                                                                                                    |
| $\pi_{a, b} \sigma_{(d > e) \wedge (f = g)} (p \times q)$ | SELECT a, b<br>FROM p, q<br>WHERE d > e AND f = g;<br>{must always have SELECT even if all attributes are kept, can be written as: SELECT *} |
| $\rho_{b/a}$                                              | a AS {or blank space} b                                                                                                                      |
| $p = \text{result}$                                       | INSERT INTO p<br>result {assuming p was empty}                                                                                               |
| $\pi_{a, b}(p)$ {assume a, b are the only attributes}     | SELECT *<br>FROM p;                                                                                                                          |

# ***Key Ideas***

# ***Key Ideas***

- Multisets
- Nulls
- Typical queries
  - Microsoft Access
  - Oracle
- Division
- Joins
- Aggregates
- Duplicates
- Aggregate operators
- Subqueries



# ***Key Ideas***

- Insertion
- Deletion
- Update
- Recursion
- Triggers
- PL/SQL
- Interface with other languages
- Predicate calculus vs. relational algebra