

Unit 5
SQL: Data Manipulation Language
For Relational Databases

SQL

- ◆ We study key features of ANSI 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)
- ◆ Many standards
 - SQL 86
 - ...
 - SQL 2016
- ◆ Many commercial implementations “close” to one of the standards
 - With some parts missing and some parts added
- ◆ Very powerful
- ◆ At its core relational algebra 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 somewhat surprising
 - Construction of ***simple and complex queries***, to show the full power of SQL
 - More than you can get from any manual
- ◆ We will not focus on
 - Any specific system
 - What you can get from a manual
 - The interaction through other languages with an SQL-based DBMS
- ◆ But, most queries have been run on different DBMSs, which allowed easy production of “snapshots”
- ◆ Some run on Oracle too

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 1st row
 - We ***can count the number of times a particular row appears*** in the table (if we want to)
 - 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

Key Differences Between Relational Algebra And SQL

- ◆ SQL contains all the power of relational algebra and more
- ◆ Many redundant operators (relational algebra had only one: intersection, which can be computed using difference)
- ◆ SQL provides statistical operators, such as AVG (average), which is very important and useful in practice
 - 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
 - But perhaps unavoidably so
 - We need to, and will, understand them
- ◆ “Pretty printing” of output: sorting and similar
 - Not difficult, but useful, we will not focus on this
- ◆ Operations for
 - Inserting
 - Deleting
 - Changing/updating (sometimes not easily reducible to deleting and inserting)

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 **are not** equal, because:
 - There is a row that appears with different multiplicities in the two tables
 - Row (2,20) appears twice in R but only once 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!

Relational Algebra vs. SQL

- ◆ We ***did not*** say that sets contain each element only once
- ◆ We said that we cannot specify (and do not care) how many times an element appears in a set
- ◆ It only matters whether it appears (at least once) or not (at all)
- ◆ ***Therefore, all that we have learned about relational algebra operations immediately applies to corresponding operations in SQL, which does care about duplicates***
- ◆ That's why it was important not to say that an element in a set appears exactly once when we studied relational algebra
- ◆ This was a subtle, but an important, point

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)***
- ◆ We proceed to progressively more and more complicated examples, starting with what we know from relational algebra
- ◆ ***A SELECT statement is also called a join: tables R1, R2, ... are “joined” when condition F holds***

Set Operations (Not All Of Them Always Implemented)

- ◆ UNION, *duplicates are removed*:

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

R	A	S	A	Result	A
	1		1		1
	1		2		2
	2		2		3
	3		4		4
	2		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 is the sum of its cardinalities in R and S

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)

- ◆ 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, *duplicates are removed*:

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

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

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 show to learn SQL
 - Therefore, a little artificial, but this does not matter

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)**

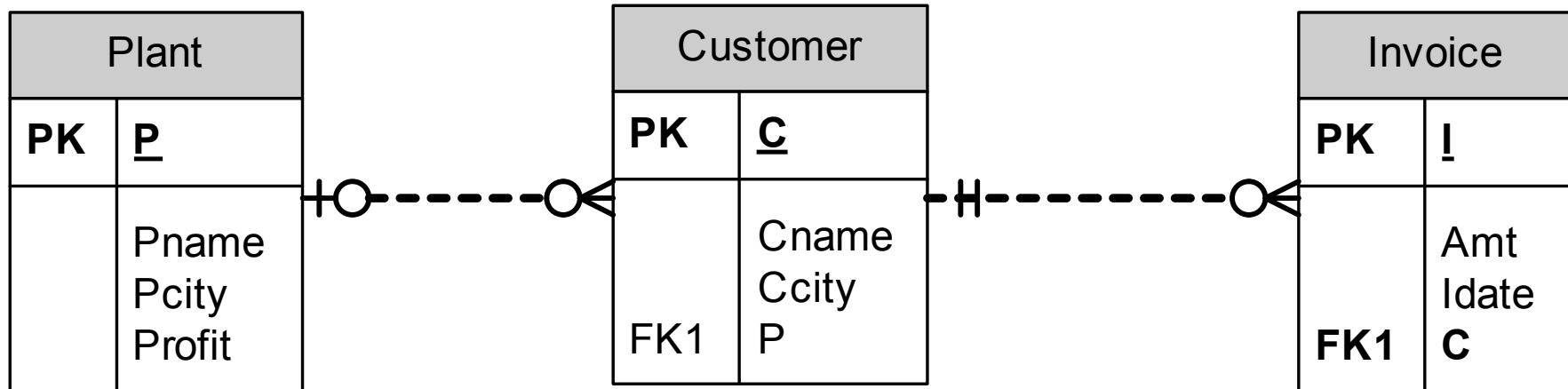
- This table describes the customers, identified by C. Each customer has a Cname and lives in a Ccity. Note that each customer is assigned to a specific P, where the orders for the customers are fulfilled. This P is a foreign key referencing Plant

◆ **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

Note: could not use attribute “Date,” as it is a reserved keyword

The Tables of Our Database



Our Instance

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

Plant 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

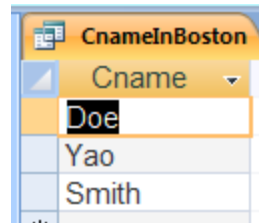
Plant Customer Invoice				
	I	Amt	Idate	C
	501	30	2009-02-02	2000
	502	300	2009-02-03	3000
	503	200	2009-02-01	1000
	504	160	2009-02-03	1000
	505	150	2009-02-02	2000
	506	150	2009-02-02	4000
	507	200		2000
	508	20	2009-02-03	1000
	509	20		4000

Important note for later: a customer can be in several cities.

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';
```

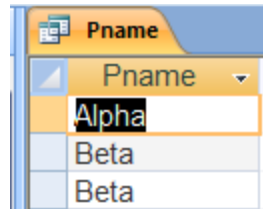
- The asterisk, *, stands for the sequence of all the columns, in this case, C, Cname, Ccity, P

CustomerInBoston				
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' entry is highlighted with a black background and white text.

Pname
Alpha
Beta
Beta

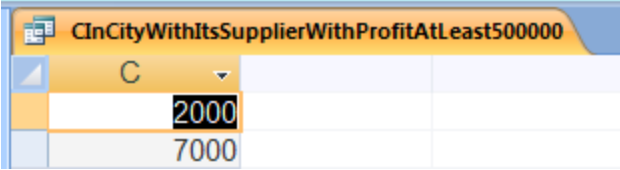
- ◆ Note that duplicates were not removed

Queries on a Single Table (Continued)

- ◆ Find every C who is supplied from a plant in the same city that it is 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.Pcity = Second.Pcity AND First.Profit > Second.Profit;
```

PlantBiggerWithPlantSmaller	
Bigger	Smaller
908	901
902	901
907	906
902	908

- ◆ ***In some implementations AS cannot be used for renaming of tables, and only space can be used***

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:
 - \$50,000.00
 - NULL
- ◆ This comparison “cannot be done”

Division

- ◆ We next introduce a new important type of query, which could have been done using relational algebra (as everything we have done so far)
- ◆ This is probably the most complex query we will discuss, so we deferred it until now
- ◆ *It is very important, but due to its complexity frequently not covered in textbooks*
- ◆ Its building blocks (and concepts behind them) are important too
- ◆ So I will go over it very carefully

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)

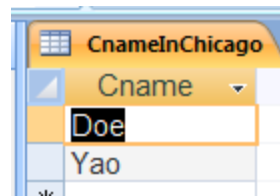
CnameInCcity

- ◆ **SELECT** Ccity, Cname **INTO** CnameInCcity
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

- ◆ **SELECT** Customer.Cname **INTO** CnameInChicago
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 Vs. Asking About All

- ◆ In the following examples, I removed duplicates to save space
- ◆ In SQL duplicates will not be removed, but it will not change the meaning of the result: still the right answers will be obtained

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

Another Example

- ◆ I will state a more natural example, which has exactly the same issues
- ◆ I did not want to introduce a new database, so this is just to show that the problem is not artificial

Has	Person	Tool
	Marsha	Fork
	Marsha	Knife
	Marsha	Spoon
	Vijay	Fork
	Vijay	Knife
	Dong	Fork
	Dong	Spoon
	Chris	Spoon
	Chris	Cup

Needed	Tool
	Fork
	Knife

Asking About Some And About All

- ◆ List all Persons, whose set of Tools contains **at least one** Tool that is (also) in Needed
- ◆ This will be easy

- ◆ List all Persons, whose set of Tools contains **at least all** the Tools that are (also) in Needed
- ◆ This will be harder

Asking About Some

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

```
SELECT Ccity INTO AnswerSome  
FROM CnameInCcity, CnameInChicago  
WHERE CnameInCcity.Cname = CnameInChicago.Cname;
```

AnswerSome	Ccity
	Boston
	Chicago
	Seattle

Asking About All

- ◆ We will proceed in stages, producing temporary tables, to understand how to do it
- ◆ It is possible to do it using one query, which we will see later
- ◆ We will start with the roadmap of what we will actually do
- ◆ We will produce some intermediate tables

Roadmap

1. TempA = (all cities)
2. TempB = (all cities, all desired customers); for every city all the desired customers, **not only** the desired customers *actually* 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 cities in TempC = 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

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

Asking About All

- ◆ `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 desired Cname; this is a “bad” Ccity with a proof why it is bad

tempC	Ccity	Cname
	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

Asking About All

- ◆ `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

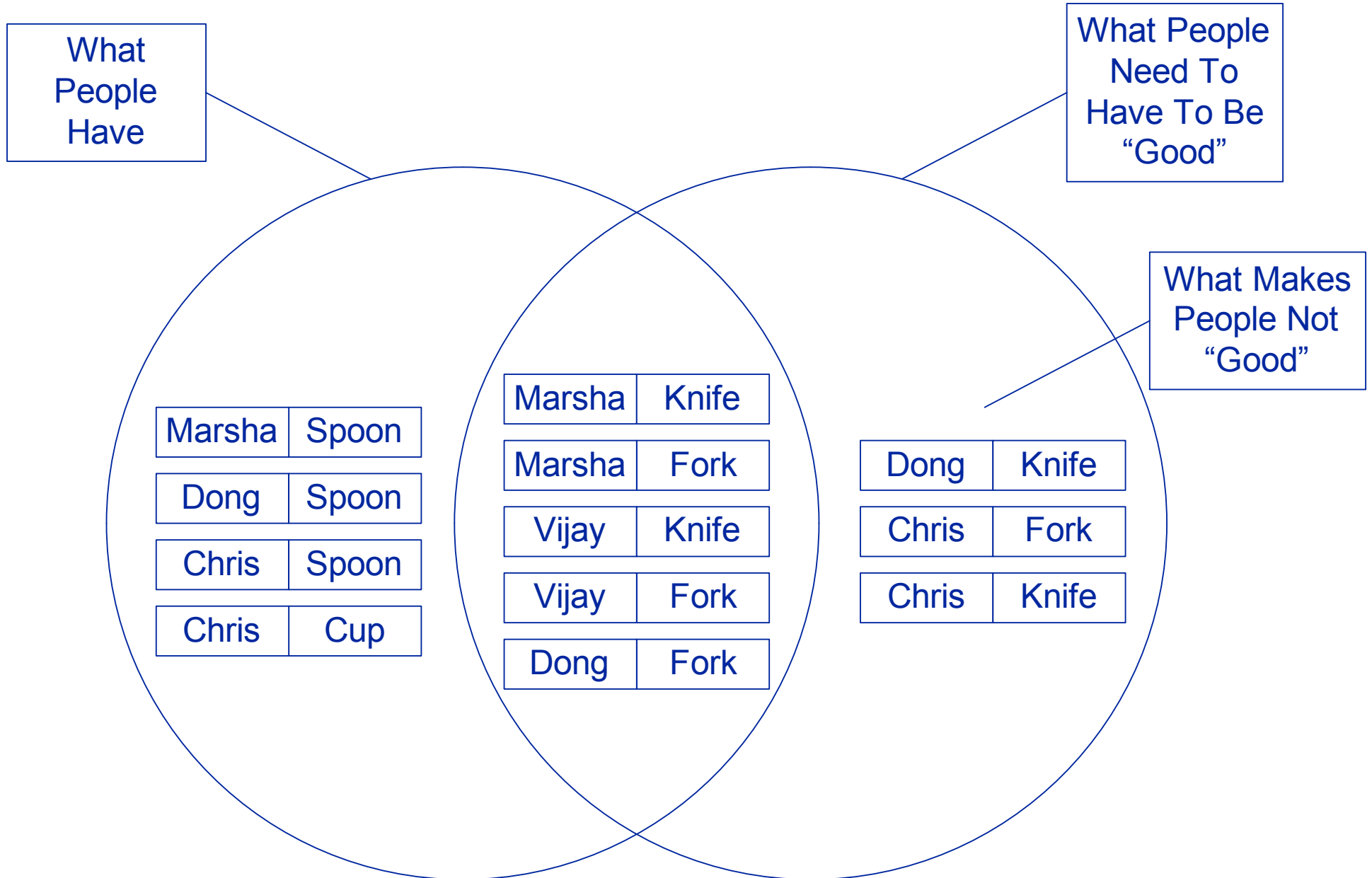
Why Division Was So Difficult

- ◆ We review our other example

Has	Person	Tool
	Marsha	Fork
	Marsha	Knife
	Marsha	Spoon
	Vijay	Fork
	Vijay	Knife
	Dong	Fork
	Dong	Spoon
	Chris	Spoon
	Chris	Cup

Needed	Tool
	Fork
	Knife

Venn Diagram To Explain Division Query



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 SQL DML***
 - ***For SQL DDL***
- ◆ 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?
- ◆ 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, 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 FALSE, then it is not passed to SELECT
 - If the condition on the tuple is UNKNOWN, then it is not passed to SELECT
- ◆ ***In this context, of SQL DML queries, UNKNOWN behaves exactly the same as FALSE***
- ◆ So why introduce it? Because it will behave differently in the context of SQL DDL, as we will see later

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

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

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

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

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

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

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

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:
 - $5 + \text{NULL} = \text{NULL}$
 - $0 * \text{NULL} = \text{NULL}$
 - $\text{NULL} / 0 = \text{NULL}$

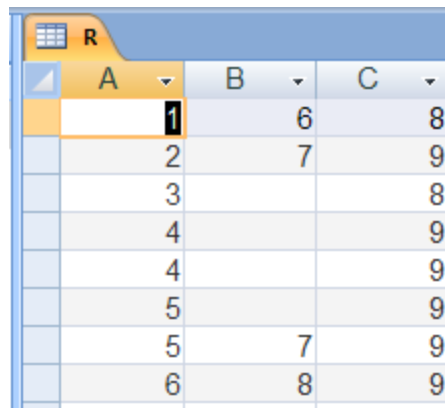
NULLs

- ◆ 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 use one table



A screenshot of a database table named 'R'. The table has three columns: A, B, and C. The data is as follows:

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

- ◆ 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

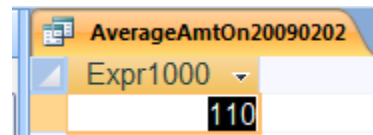
- ◆ It is sometimes important to specify whether duplicates should or should not be removed before the appropriate aggregate operator is applied
- ◆ Modifiers to aggregate operators
 - ***ALL*** (default, do not remove duplicates)
 - ***DISTINCT*** (remove duplicates)
 - ***COUNT*** can also have * specified,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

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



A screenshot of a database query result. The title bar of the window is 'AverageAmtOn20090202'. The table has one column labeled 'Expr1000' and one row with the value '110'.

Expr1000
110

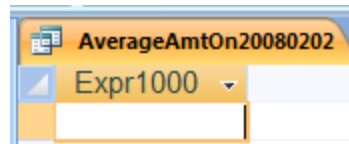
Queries With Aggregates

- ◆ 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#;`
- ◆ 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#;
```



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#;
```

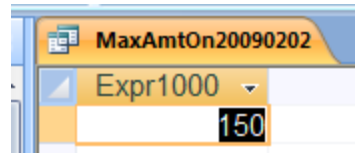
- Here we had to remove duplicates, to get the right answer

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



The screenshot shows a Microsoft Access query result window. The title bar of the window is labeled 'MaxAmtOn20090202'. The window displays a single row of data. The first column is labeled 'Expr1000' and contains the value '150'.

Expr1000
150

Queries With Aggregates

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

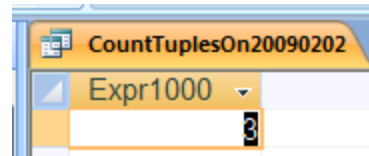
```
SELECT MIN(Amt)
FROM Invoice
WHERE Idate = #2009-02-02#;
```

- 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 Idate = #2009-02-02#;
```



CountTuplesOn20090202	
Expr1000	3

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 laws of mathematics, for instance

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

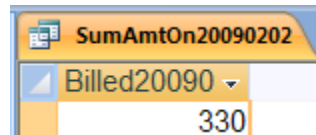
and not undefined or NULL

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

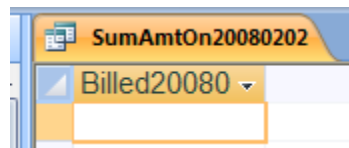


SumAmtOn20090202	
Billed20090	330

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



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 only put 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

ssldate	SUM(Amt)
2	330
1	200
NULL	220

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

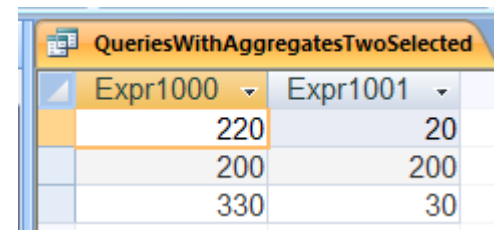
Queries With Aggregates

- ◆ 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

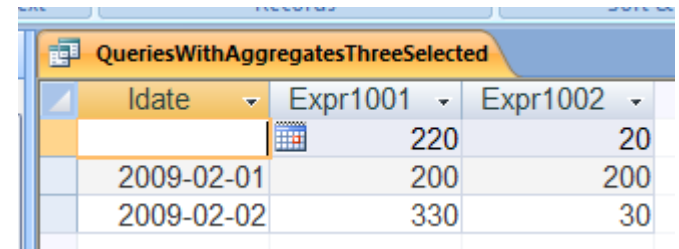
```
SELECT SUM(Amt), MIN(Amt)
FROM Invoice
WHERE C <> 3000
GROUP BY Idate
HAVING AVG(Amt) > 100;
```



Expr1000	Expr1001
220	20
200	200
330	30

- ◆ 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;
```



Idate	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;
```


Our Instance

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

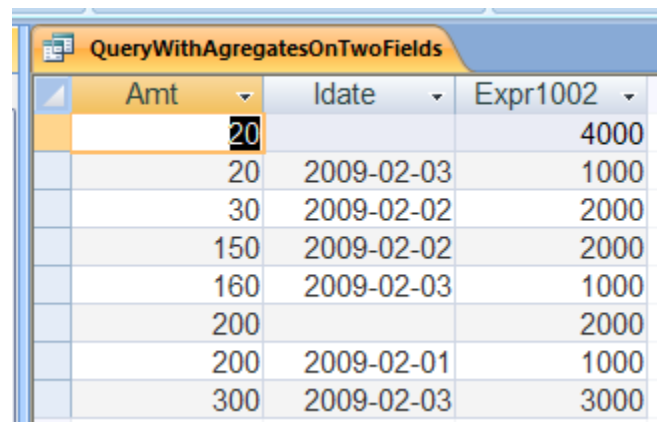
Invoice				
	I	Amt	Idate	C
	501	30	2009-02-02	2000
	502	300	2009-02-03	3000
	503	200	2009-02-01	1000
	504	160	2009-02-03	1000
	505	150	2009-02-02	2000
	506	150	2009-02-02	4000
	507	200		2000
	508	20	2009-02-03	1000
	509	20		4000

Queries 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

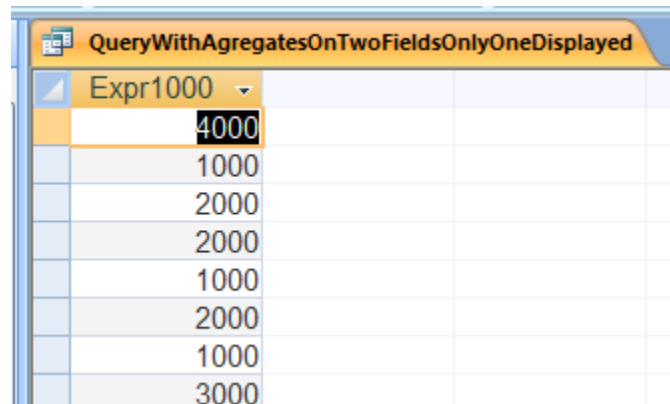


Amt	Idate	Expr1002
20		4000
20	2009-02-03	1000
30	2009-02-02	2000
150	2009-02-02	2000
160	2009-02-03	1000
200		2000
200	2009-02-01	1000
300	2009-02-03	3000

Queries With Aggregates

- ◆ The following is permitted also

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



Expr1000
4000
1000
2000
2000
1000
2000
1000
3000

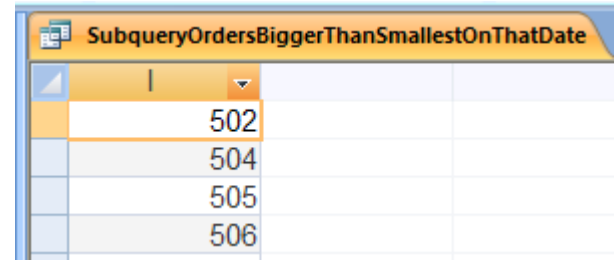
Subqueries

- ◆ 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);
```



I	
502	
504	
505	
506	

- ◆ 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

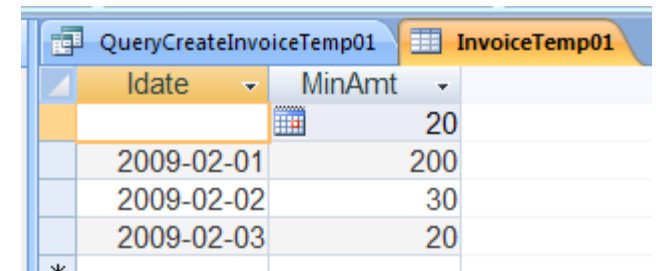
Subqueries

- ◆ 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
 - 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

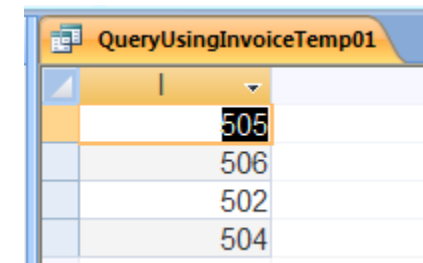
1. `SELECT Idate, MIN(Amt) AS MinAmt
INTO InvoiceTemp01
FROM Invoice
GROUP BY Idate;`



The screenshot shows a SQL query window titled 'QueryCreateInvoiceTemp01' with a result grid titled 'InvoiceTemp01'. The grid has two columns: 'Idate' and 'MinAmt'. The data rows are:

Idate	MinAmt
	20
2009-02-01	200
2009-02-02	30
2009-02-03	20

2. `SELECT Invoice.I
FROM Invoice, InvoiceTemp01
WHERE Invoice.Idate = InvoiceTemp01.Idate AND Amt > MinAmt;`



The screenshot shows a SQL query window titled 'QueryUsingInvoiceTemp01' with a result grid. The grid has one column: 'I'. The data rows are:

I
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

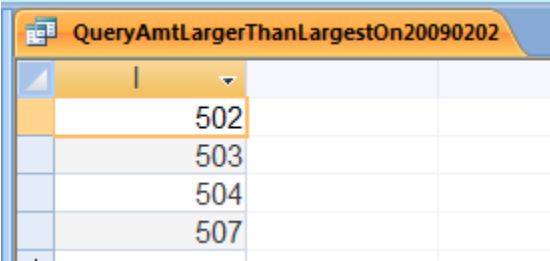
- “ $v \text{ op ANY } r$ ” is true if and only if $v \text{ op } x$ is true for at least one x in r
- “ $v \text{ op ALL } r$ ” is true if and only if $v \text{ op } x$ is true for each x in r

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



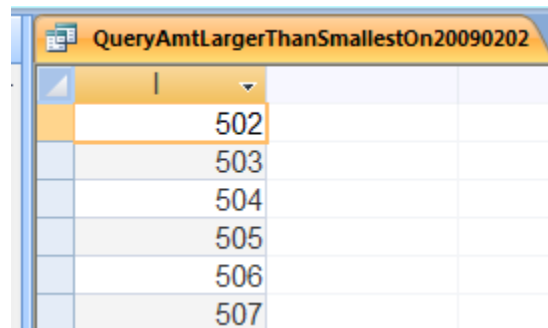
I
502
503
504
507

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**



I
502
503
504
505
506
507
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)

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);
```

Set Intersection (*INTERSECT*)

Use *EXISTS*

```
SELECT DISTINCT *  
FROM R  
WHERE 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	

Intersect	R	S
First	Second	
b	a	
b	c	

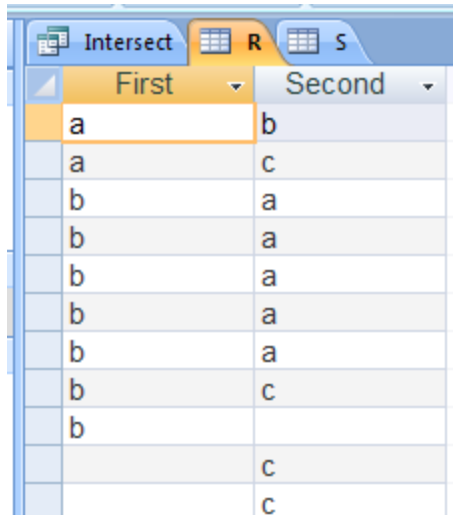
- ◆ Note that a tuple containing nulls, (NULL,c), is not in the result, and it should not be there

Set Intersection (INTERSECT) Can Also Be Done Using Cartesian Product

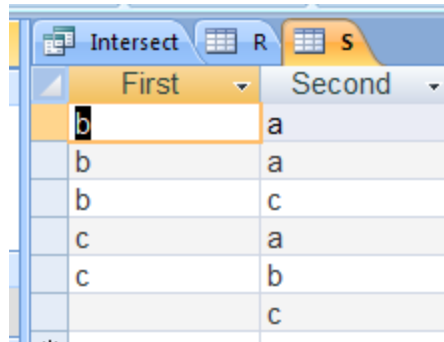
```
SELECT DISTINCT *  
FROM R, S  
WHERE  
(R.First = S.First AND R.Second = S.Second)
```

Set Difference (MINUS/EXCEPT) Use NOT EXISTS

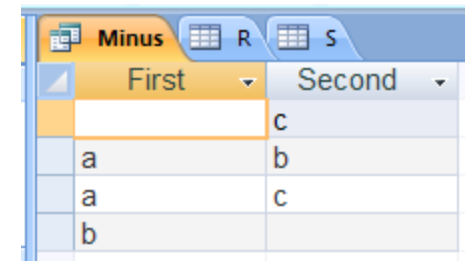
```
SELECT DISTINCT *  
FROM R  
WHERE NOT EXISTS  
(SELECT *  
FROM S  
WHERE  
R.First = S.First AND R.Second = S.Second);
```



First	Second
a	b
a	c
b	a
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
	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);
```

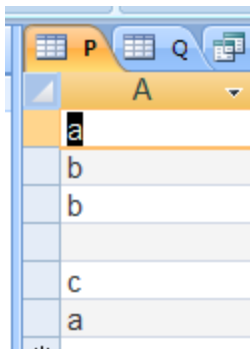


Table P: A single column table with values a, b, b, c, a.

A
a
b
b
c
a

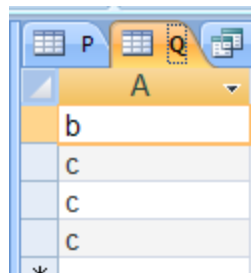
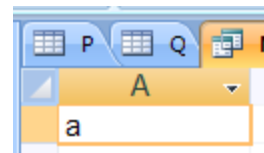


Table Q: A single column table with values b, c, c, c.

A
b
c
c
c



Result of the query: A single column table 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)

Back To Division

- ◆ We want to compute the set of Ccities that have at least all the Cnames that Chicago has

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

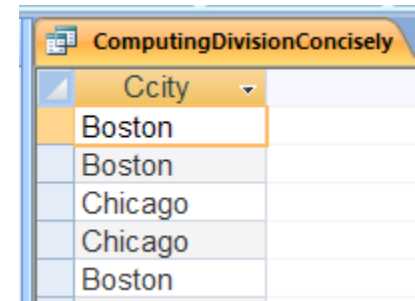
CnameInChicago	Cname
	Doe
	Yao

Computing Division Concisely

- ◆ List all Ccities, the set of whose Cnames, contains all the Cnames that are in Chicago.

```
SELECT Ccity
FROM CnameInCcity AS CnameInCcity1
WHERE NOT EXISTS
(SELECT Cname
FROM CnameInChicago
WHERE Cname NOT IN
(SELECT Cname
FROM CnameInCcity
WHERE CnameInCcity.Ccity = CnameInCcity1.Ccity));
```

- ◆ This is really the same as before
 - I leave it to you to figure this out



Ccity
Boston
Boston
Chicago
Chicago
Boston

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 two tables to describe them

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		

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 rows of second table that have no matches in first table, NULLs are added for the columns of 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

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

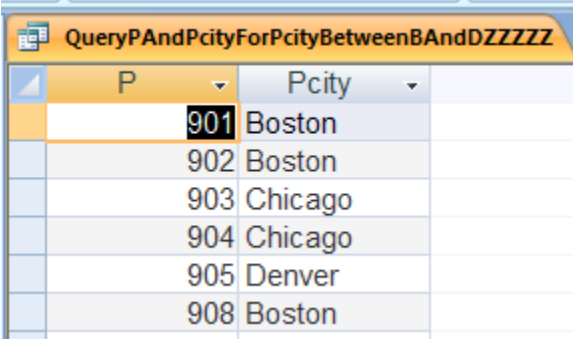
- ◆ Can not be done in some DBMSs; use Left Outer and Right Outer

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 ((Pcity 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.”



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:

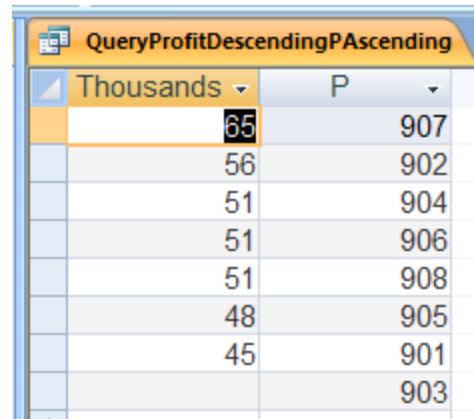
```
SELECT Pname  
FROM Plant  
WHERE (City LIKE '_X%');
```

- % 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;
```

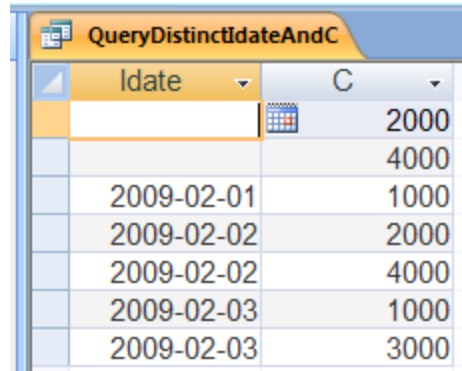


Thousands	P
65	907
56	902
51	904
51	906
51	908
48	905
45	901
	903

Presenting the Result

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

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



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 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 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

- ◆ 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);

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

InsertRowIntoPlant				
	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
+	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);

InsertRowIntoPlant				
	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
+	909	Gamma		\$52,000.00

InsertTupleIntoPlantWithDefaultValue				
	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
+	909	Gamma		\$52,000.00
+	910	Gamma		

Insertion From A Table

- ◆ Assume we have a table Candidate(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

Result

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

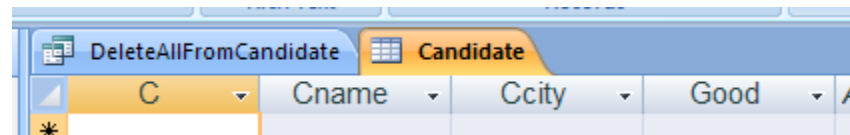
- ◆ **DELETE**
FROM Candidate
WHERE Good = 'Yes';
- ◆ This removes rows satisfying the specified condition
 - In our example, once some candidates were promoted to customers, they are removed from Candidate

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”



DeleteAllFromCandidate					Candidate				
C	Cname	Ccity	Good	A	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)

Invoice				
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				
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	

Key Ideas

- ◆ Multisets
- ◆ Nulls
- ◆ Typical queries
- ◆ Division
- ◆ Joins
- ◆ Aggregates
- ◆ Duplicates
- ◆ Aggregate operators
- ◆ Subqueries
- ◆ Insertion
- ◆ Deletion
- ◆ Update