Database Systems

Session 6 – Additional Topic

SQL DML & DDL Examples

Dr. Jean-Claude Franchitti

New York University
Computer Science Department
Courant Institute of Mathematical Sciences

Presentation material partially based on textbook slides
by Ramez Elmasri and Shamkant Navathe
Slides copyright © 2016 and on slides produced by Zvi Kedem copyright © 2014
Session Agenda

- SQL Data Manipulation Language Examples
- SQL Data Definition Language Examples
1. Session Overview
2. Data Manipulation Language for Relational DBs
3. Data Definition Language for Relational DBs
4. Summary and Conclusion
Multisets
The need to understand what happens with duplicates
SELECT statement
Operations on sets (removing and not removing duplicates)
Queries on one or more tables
Division
Three-valued logic: true, unknown and false
Treatment of NULLs
Treatment of NULLs with duplicates
Aggregates
Subqueries
Joins
Ranges and templates
Agenda (2/2)

- Inserting tuples
- Deleting tuples
- Updating (modifying tuples)
- Stored Procedure
- Recursive queries
- Embedded SQL
- Dynamic 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 2008

Many commercial implementations “close” to one of the standards
» With some parts missing and some parts added

Very powerful

At its core “full” 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

- But, after running most of the queries on Microsoft Access, which allowed easy production of “snapshots”
- Some of them were 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 1\(^{st}\) 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
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)
  - 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)
More About Multisets

- In a relational algebra, the basic object was a **set**
  - Order of elements cannot be specified
  - The multiplicity (how many times an element appearing in the set appears in it) **cannot** be specified; i.e., cannot say that “a” appears 3 times

- In SQL the basic element is a **multiset**
  - Order of elements cannot be specified
  - The multiplicity (how many times an element appearing in the set appears in it) **can** be specified; i.e., can say that “a” appears 3 times
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

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>B</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
The following two tables are not equal, because:
» There is a row that appears with different multiplicities in the two tables

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>
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

This was a subtle, but important, point
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*:

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

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
- **MINUS**, *duplicates are removed*:

SELECT * FROM R
MINUS
SELECT * FROM S;

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>
Set Operations (Not All Of Them Always Implemented)

- **INTERSECT**, *duplicates are removed*:

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

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Set Operations (Not All Of Them Always Implemented)

- **UNION ALL**, *duplicates are not removed*:

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

- 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 ALL**, *duplicates are not removed*:

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

<table>
<thead>
<tr>
<th>R A</th>
<th>S A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

  **Result**
  ```
<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
  ```

  - An element appears with the cardinality that is $\max(0, \text{cardinality in } R - \text{cardinality in } S)$
- **INTERSECT ALL**, *duplicates are not removed*:

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

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

**Result**

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

- An element appears with the cardinality that is min(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

- Sample database: *PlantCustomerInvoice.mdb* in the extras directory
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
  - The attribute “Date,” cannot be used as it is a reserved keyword
The Tables of Our Database
## Our Instance

### Plant

<table>
<thead>
<tr>
<th>P</th>
<th>Pname</th>
<th>Pcity</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>301</td>
<td>Alpha</td>
<td>Boston</td>
<td>$45,000.00</td>
</tr>
<tr>
<td>902</td>
<td>Beta</td>
<td>Boston</td>
<td>$56,000.00</td>
</tr>
<tr>
<td>903</td>
<td>Beta</td>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>904</td>
<td>Gamma</td>
<td>Chicago</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>905</td>
<td>Delta</td>
<td>Denver</td>
<td>$48,000.00</td>
</tr>
<tr>
<td>906</td>
<td>Epsilon</td>
<td>Miami</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>907</td>
<td>Beta</td>
<td>Miami</td>
<td>$65,000.00</td>
</tr>
<tr>
<td>908</td>
<td>Beta</td>
<td>Boston</td>
<td>$51,000.00</td>
</tr>
</tbody>
</table>

### Customer

<table>
<thead>
<tr>
<th>P</th>
<th>Cname</th>
<th>Ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Doe</td>
<td>Boston</td>
</tr>
<tr>
<td>2000</td>
<td>Yao</td>
<td>Boston</td>
</tr>
<tr>
<td>3000</td>
<td>Doe</td>
<td>Chicago</td>
</tr>
<tr>
<td>4000</td>
<td>Doe</td>
<td>Seattle</td>
</tr>
<tr>
<td>5000</td>
<td>Brown</td>
<td>Denver</td>
</tr>
<tr>
<td>6000</td>
<td>Smith</td>
<td>Seattle</td>
</tr>
<tr>
<td>7000</td>
<td>Yao</td>
<td>Chicago</td>
</tr>
<tr>
<td>8000</td>
<td>Smith</td>
<td>Denver</td>
</tr>
<tr>
<td>9000</td>
<td>Smith</td>
<td>Boston</td>
</tr>
</tbody>
</table>

### Invoice

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
<th>Ac</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2009-02-02</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>502</td>
<td>300</td>
<td>2009-02-03</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>2009-02-01</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>2009-02-03</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2009-02-02</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2009-02-02</td>
<td>4000</td>
<td></td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td></td>
<td></td>
<td>2000</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>2009-02-03</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>2009-02-03</td>
<td>4000</td>
<td></td>
</tr>
</tbody>
</table>
- Find Cname for all customers who are located in Boston:

```
SELECT Cname
FROM Customer
WHERE Ccity = 'Boston';
```
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.
Queries On A Single Table

- Find Pname for all plants that are located in Boston:
  
  SELECT Pname
  FROM Plant
  WHERE Pcity = 'Boston';

- Note that duplicates were not removed
Find every C who is supplied from a plant in the same city they 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
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;
```

- In some implementations AS cannot be used for renaming of tables, and only space can be used (see next)
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

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

This example shows how the space 
character is used as a renaming operator 
(does not work in Access)
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 need to be compared:

- $51,000.00
- NULL

This comparison “cannot be done”
We next introduce a new important type of query, which could have been done using relational algebra (as everything so far)

This is probably the most complex query we will discuss, so we deferred it until here

It is very important, but due to its complexity, frequently not covered in textbooks

Its building blocks (and concept behind them) are important too

So we will go over it very carefully
We first compute two tables

- $\text{CnameInCcity}(\text{Ccity}, \text{Cname})$
  This table lists all the “valid” tuples of $\text{Ccity}, \text{Cname}$; it is convenient for us to list the city first

- $\text{CnameInChicago}(\text{Cname})$
  This table lists the names of the customers located in Chicago.

We then compute 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)
- `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.
SELECT Customer.Cname INTO CnameInChicago
FROM Customer
WHERE Ccity='Chicago';
- We have reproduced them, so they are larger and we can see them clearly

<table>
<thead>
<tr>
<th>CnameInCity</th>
<th>City</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Boston</td>
<td>Yao</td>
<td></td>
</tr>
<tr>
<td>Boston</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td>Yao</td>
<td></td>
</tr>
<tr>
<td>Seattle</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Seattle</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>Denver</td>
<td>Smith</td>
<td></td>
</tr>
<tr>
<td>Denver</td>
<td>Brown</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CnameInCity</th>
<th>City</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td>Yao</td>
<td></td>
</tr>
</tbody>
</table>
In the following examples, duplicates were removed 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

We will see this in Access snapshots
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

- Stating a more natural example, which has exactly the same issues
- The following does not introduce a new database, but is simply meant to show that the problem is not artificial

<table>
<thead>
<tr>
<th>Has</th>
<th>Person</th>
<th>Tool</th>
<th>Needed</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Marsha</td>
<td>Fork</td>
<td></td>
<td>Fork</td>
</tr>
<tr>
<td></td>
<td>Marsha</td>
<td>Knife</td>
<td></td>
<td>Knife</td>
</tr>
<tr>
<td></td>
<td>Marsha</td>
<td>Spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vijay</td>
<td>Fork</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vijay</td>
<td>Knife</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dong</td>
<td>Fork</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dong</td>
<td>Spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chris</td>
<td>Spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chris</td>
<td>Cup</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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;
```

<table>
<thead>
<tr>
<th>AnswerSome</th>
<th>Ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>Seattle</td>
<td></td>
</tr>
</tbody>
</table>
In Microsoft Access

<table>
<thead>
<tr>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
</tr>
<tr>
<td>Chicago</td>
</tr>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>Chicago</td>
</tr>
<tr>
<td>Boston</td>
</tr>
</tbody>
</table>
- 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
1. TempA = (all cities)
2. TempB = (all cities, all customers); for every city all the customers in the database, *not only* 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

<table>
<thead>
<tr>
<th>TempA</th>
<th>Ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boston</td>
</tr>
<tr>
<td></td>
<td>Chicago</td>
</tr>
<tr>
<td></td>
<td>Seattle</td>
</tr>
<tr>
<td></td>
<td>Denver</td>
</tr>
</tbody>
</table>
In Microsoft Access

- Note duplicates: nothing surprising about this, as duplicates are not removed.
Asking About All

- \[ \text{SELECT } \text{Cc} \text{ity, Cname INTO tempB} \]
  \[ \text{FROM TempA, CnameInChicago;} \]

- Set of all pairs of the form (Cc\text{ity,Cname}); in fact a Cartesian product of \textit{all} cities with \textit{all} desired Cnames (\textit{not only} cities that \textit{have all} desired Cnames)

<table>
<thead>
<tr>
<th>tempB</th>
<th>Ccity</th>
<th>Cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Boston</td>
<td>Yao</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td>Yao</td>
<td></td>
</tr>
<tr>
<td>Seattle</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Seattle</td>
<td>Yao</td>
<td></td>
</tr>
<tr>
<td>Denver</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Denver</td>
<td>Yao</td>
<td></td>
</tr>
</tbody>
</table>
In Microsoft Access

<table>
<thead>
<tr>
<th>City</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>Doe</td>
</tr>
<tr>
<td>Boston</td>
<td>Yao</td>
</tr>
<tr>
<td>Boston</td>
<td>Doe</td>
</tr>
<tr>
<td>Boston</td>
<td>Yao</td>
</tr>
<tr>
<td>Chicago</td>
<td>Doe</td>
</tr>
<tr>
<td>Chicago</td>
<td>Yao</td>
</tr>
<tr>
<td>Seattle</td>
<td>Doe</td>
</tr>
<tr>
<td>Seattle</td>
<td>Yao</td>
</tr>
<tr>
<td>Denver</td>
<td>Doe</td>
</tr>
<tr>
<td>Denver</td>
<td>Yao</td>
</tr>
<tr>
<td>Seattle</td>
<td>Doe</td>
</tr>
<tr>
<td>Seattle</td>
<td>Yao</td>
</tr>
<tr>
<td>Chicago</td>
<td>Doe</td>
</tr>
<tr>
<td>Chicago</td>
<td>Yao</td>
</tr>
<tr>
<td>Denver</td>
<td>Doe</td>
</tr>
<tr>
<td>Denver</td>
<td>Yao</td>
</tr>
<tr>
<td>Boston</td>
<td>Doe</td>
</tr>
<tr>
<td>Boston</td>
<td>Yao</td>
</tr>
</tbody>
</table>
**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.
- **SELECT * INTO tempC**
  FROM tempB
  WHERE NOT EXISTS
    (SELECT *
     FROM CnameInCcity
     WHERE tempB.Ccity = CnameInCcity.Ccity
     AND tempB.Cname = CnameInCcity.Cname);

<table>
<thead>
<tr>
<th>tempC</th>
<th>Ccity</th>
<th>Cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td>Yao</td>
<td></td>
</tr>
<tr>
<td>Denver</td>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Denver</td>
<td>Yao</td>
<td></td>
</tr>
</tbody>
</table>

- 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
In Microsoft Access

<table>
<thead>
<tr>
<th>City</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
<td>Yao</td>
</tr>
<tr>
<td>Denver</td>
<td>Doe</td>
</tr>
<tr>
<td>Denver</td>
<td>Yao</td>
</tr>
<tr>
<td>Seattle</td>
<td>Yao</td>
</tr>
<tr>
<td>Denver</td>
<td>Doe</td>
</tr>
<tr>
<td>Denver</td>
<td>Yao</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>tempD</th>
<th>Ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Seattle</td>
</tr>
<tr>
<td></td>
<td>Denver</td>
</tr>
<tr>
<td></td>
<td>Denver</td>
</tr>
</tbody>
</table>
In Microsoft Access

<table>
<thead>
<tr>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seattle</td>
</tr>
<tr>
<td>Denver</td>
</tr>
<tr>
<td>Denver</td>
</tr>
<tr>
<td>Seattle</td>
</tr>
<tr>
<td>Denver</td>
</tr>
<tr>
<td>Denver</td>
</tr>
</tbody>
</table>
- SELECT * INTO AnswerAll
  FROM (SELECT *
  FROM tempA)
  MINUS
  (SELECT *
  FROM tempD);

- Set of all “good” cities, that is cities that are not “bad”

<table>
<thead>
<tr>
<th>AnswerAll</th>
<th>Ccity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td></td>
</tr>
</tbody>
</table>
- 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”
In Microsoft Access

<table>
<thead>
<tr>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>Boston</td>
</tr>
<tr>
<td>Chicago</td>
</tr>
<tr>
<td>Chicago</td>
</tr>
<tr>
<td>Boston</td>
</tr>
</tbody>
</table>
Meanings of `NULL`
- Unknown value
- Unavailable or withheld value
- Not applicable attribute

Each individual `NULL` value considered to be different from every other `NULL` value

SQL uses a three-valued logic:
- `TRUE`, `FALSE`, and `UNKNOWN`
### Table 5.1 Logical Connectives in Three-Valued Logic

<table>
<thead>
<tr>
<th></th>
<th>AND</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) TRUE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td>FALSE</td>
<td></td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td>FALSE</td>
<td>UNKNOWN</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>OR</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b) TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
<td>TRUE</td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td>FALSE</td>
<td>UNKNOWN</td>
<td></td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>TRUE</td>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>NOT</th>
<th>TRUE</th>
<th>FALSE</th>
<th>UNKNOWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>(c) TRUE</td>
<td>FALSE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FALSE</td>
<td>TRUE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>UNKNOWN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SQL allows queries that check whether an attribute value is NULL

- IS or IS NOT NULL

Query 18. Retrieve the names of all employees who do not have supervisors.

Q18: SELECT Fname, Lname FROM EMPLOYEE WHERE Super_ssn IS NULL;
We now move to look at some aspects of SQL, which are 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, we will touch on the most important aspects
- There are two variants
  - For SQL DML
  - For SQL DDL
- But the core is common
- We start with a SELECT statement
  ```sql
  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
We abbreviate:
- T for TRUE
- F for FALSE
- U for UNKNOWN

Standard 2-valued logic

<table>
<thead>
<tr>
<th>NOT</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AND</th>
<th>F</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

New 3-valued logic

<table>
<thead>
<tr>
<th>NOT</th>
<th>F</th>
<th>U</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td>F</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OR</th>
<th>F</th>
<th>U</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AND</th>
<th>F</th>
<th>U</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>F</td>
<td>U</td>
<td>U</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>U</td>
<td>T</td>
</tr>
</tbody>
</table>

U is “between” F and T, “metathink” as being “maybe T or maybe F”
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
  ```sql
  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 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
We will use a simple Microsoft Access database: *Nulls.mdb* in Extras

It has only one table
- Any comparison in which one side is NULL is UNKNOWN

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NULL</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

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

- We get:
Any comparison in which one side is NULL is UNKNOWN

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NULL</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

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

We get:

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
In Microsoft Access
Any comparison in which one side is NULL is UNKNOWN

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>NULL</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

SELECT A
FROM R
WHERE B = NULL;

We get: A
In Microsoft Access
- **Any comparison in which one side is NULL is UNKNOWN**

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>7</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>NULL</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
<td></td>
<td>9</td>
</tr>
</tbody>
</table>

- **SELECT**
  FROM R
  WHERE B <> NULL;

- **We get:**
  which is an empty table
But note what Access did, *which is wrong*: 

![Image of Microsoft Access query result](image_url)
### Oracle did it right

<table>
<thead>
<tr>
<th><strong>Script</strong></th>
<th><strong>Trace</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>drop table R;</td>
<td>Table dropped.</td>
</tr>
</tbody>
</table>
| create table R (  
  A number,  
  B number,  
  C number  
); | Table created. |
| insert into R values(1,6,8); | 1 row created. |
| insert into R values(2,7,9); | 1 row created. |
| insert into R values(3,null,8); | 1 row created. |
| insert into R values(4,null,9); | 1 row created. |
| select * from R; | A | B | C |
| select A | 1 | 6 | 8 |
| from R | 2 | 7 | 9 |
| where B <> null; | 3 | 8 | 9 |
| | 4 | 9 | |
| | no rows selected |
• Any comparison in which one side is NULL is UNKNOWN

• SELECT A
  FROM R
  WHERE B = B;

• We get:

• Because, going row by row:
  » 6 = 6 is TRUE
  » 7 = 7 is TRUE
  » NULL = NULL is UNKNOWN
  » NULL = NULL is UNKNOWN
In Microsoft Access
A new keyword made of three words: IS NOT NULL

SELECT A FROM R WHERE B IS NOT NULL;

We get:

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NULL</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

A
1
2
In Microsoft Access
A new keyword made of two words: IS NULL

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>NULL</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>NULL</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

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

```
3
4
```
In Microsoft Access
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:

- \( 5 + \text{NULL} = \text{NULL} \)
- \( 0 \times \text{NULL} = \text{NULL} \)
- \( \text{NULL} / 0 = \text{NULL} \)
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 soon

* 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

![Database Table](image.png)
- **SELECT B, C FROM R WHERE A < 6;**
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)
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)
Aggregate Functions in SQL

- Used to summarize information from multiple tuples into a single-tuple summary

**Grouping**
- Create subgroups of tuples before summarizing

**Built-in aggregate functions**
- `COUNT`, `SUM`, `MAX`, `MIN`, and `AVG`

**Functions can be used in the** `SELECT` **clause or in a** `HAVING` **clause**
NULL values discarded when aggregate functions are applied to a particular column

**Query 20.** Find the sum of the salaries of all employees of the ‘Research’ department, as well as the maximum salary, the minimum salary, and the average salary in this department.

```
Q20:   SELECT    SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
       FROM      (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
       WHERE     Dname=‘Research’;
```

**Queries 21 and 22.** Retrieve the total number of employees in the company (Q21) and the number of employees in the ‘Research’ department (Q22).

```
Q21:   SELECT    COUNT (*)
       FROM      EMPLOYEE;

Q22:   SELECT    COUNT (*)
       FROM      EMPLOYEE, DEPARTMENT
       WHERE     DNO=DNUMBER AND DNAME=‘Research’;
```
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

- Microsoft Access does not support DISTINCT
Find the average Amt in Invoice, taking into account only orders from February 2, 2009

```sql
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
In Microsoft Access
Queries With Aggregates

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

```sql
SELECT AVG(Amt)
FROM Invoice
WHERE Idate = #2008-02-02#
;```
In Microsoft Access
Find the number of different values of Amt in Invoice, taking into account only orders from February 2, 2009

```sql
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
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
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
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#;
```
In Microsoft Access
Queries With Aggregates

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

```sql
SELECT COUNT(*)
FROM Invoice
WHERE Idate = #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 laws of mathematics, for instance

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

  and not undefined or NULL
Assume you own all the plants
How much money was 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
Queries With Aggregates

- Assume you own all the plants
- How much money was made (or actually invoiced) on February 2, 2008?
- Let’s use a nice title for the column (just to practice)
  
  ```sql
  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
In some applications it may 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:
- (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
To make a smaller table, we only put the day (one digit) instead of the full date, which the database actually has.

So, instead of 2009-02-02 we just write 2.

No problem, as everything in the table is in the range 2009-02-01 to 2009-02-03.
Queries With Aggregates

<table>
<thead>
<tr>
<th>Invoice</th>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>502</td>
<td>300</td>
<td>3</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
<td></td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Invoice</th>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>502</td>
<td>300</td>
<td>3</td>
<td>3000</td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
<td></td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
<td></td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
<td></td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
<td></td>
</tr>
</tbody>
</table>
Queries With Aggregates

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>502</td>
<td>300</td>
<td>3</td>
<td>3000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>

- After WHERE C <> 3000

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>
# Queries With Aggregates

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>

- After GROUP BY Idate

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>
Queries With Aggregates

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>

- 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
  - \( \text{Idate} \) \( \text{AVG(Amt)} \)
    - 2 \hspace{1cm} 110
    - 1 \hspace{1cm} 200
    - 3 \hspace{1cm} 90
    - NULL \hspace{1cm} 110

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

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>504</td>
<td>160</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
<td>3</td>
<td>1000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>I</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>
### Queries With Aggregates

<table>
<thead>
<tr>
<th>Id</th>
<th>Amt</th>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>501</td>
<td>30</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
<td>2</td>
<td>4000</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
<td>1</td>
<td>1000</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
<td>NULL</td>
<td>2000</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
<td>NULL</td>
<td>4000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Idate</th>
<th>SUM(Amt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>330</td>
</tr>
<tr>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>NULL</td>
<td>220</td>
</tr>
</tbody>
</table>
In Microsoft Access

<table>
<thead>
<tr>
<th>Date</th>
<th>Expr1001</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009-02-01</td>
<td>220</td>
</tr>
<tr>
<td>2009-02-02</td>
<td>330</td>
</tr>
</tbody>
</table>
Grouping: The GROUP BY and HAVING Clauses

- **Partition** relation into subsets of tuples
  - Based on **grouping attribute(s)**
  - Apply function to each such group independently

- **GROUP BY** clause
  - Specifies grouping attributes

- If NULLs exist in grouping attribute
  - Separate group created for all tuples with a NULL value in grouping attribute
- **HAVING clause**
  - Provides a condition on the summary information

---

**Query 28.** For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

Q28:  
```sql
SELECT Dnumber, COUNT (*) 
FROM DEPARTMENT, EMPLOYEE 
WHERE Dnumber=Dno AND Salary>40000 AND 
( SELECT Dno 
  FROM EMPLOYEE 
  GROUP BY Dno 
  HAVING COUNT (*) > 5 )
```
SELECT <attribute and function list>
FROM <table list>
[ WHERE <condition> ]
[ GROUP BY <grouping attribute(s)> ]
[ HAVING <group condition> ]
[ ORDER BY <attribute list> ];
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

  ```sql
  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

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

[Images of two tables, one labeled "QueriesWithAggregatesTwoSelected" and the other labeled "QueriesWithAggregatesThreeSelected", showing data with columns labeled "Expr1000", "Expr1001", and "Expr1002". The tables contain data entries such as 220, 20, 200, 200, 330, 30, 220, 20, 200, 200, 330, 30.]
But, the following is not OK, as C is not a group property, because on a specific ldate different C’s can place an order

```
SELECT C
FROM Invoice
WHERE C <> 3000
GROUP BY ldate
HAVING AVG(Amt) > 100;
```
In Microsoft Access

- Got it right!
One can aggregate on more than one attribute, so 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
The following is permitted also

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

<table>
<thead>
<tr>
<th>Expr1000</th>
<th>4000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>3000</td>
</tr>
</tbody>
</table>
- Nested queries
  - Complete select-from-where blocks within WHERE clause of another query
- Outer query
- Comparison operator \( \text{IN} \)
  - Compares value \( v \) with a set (or multiset) of values \( V \)
  - Evaluates to \( \text{TRUE} \) if \( v \) is one of the elements in \( V \)
Q4A: SELECT DISTINCT Pnumber
FROM PROJECT
WHERE Pnumber IN
  ( SELECT Pnumber
    FROM PROJECT, DEPARTMENT, EMPLOYEE
    WHERE Dnum=Dnumber AND
    Mgr_ssn=Ssn AND Lname='Smith' )

  OR
  Pnumber IN
  ( SELECT Pno
    FROM WORKS_ON, EMPLOYEE
    WHERE Essn=Ssn AND Lname='Smith' );
Nested Queries (cont’d.)

- Use tuples of values in comparisons
  - Place them within parentheses

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE (Pno, Hours) IN ( SELECT Pno, Hours
                          FROM WORKS_ON
                          WHERE Essn='123456789' );
```
Use other comparison operators to compare a single value $v$

- $= \text{ANY (or = SOME)}$ operator
  - Returns TRUE if the value $v$ is equal to some value in the set $V$ and is hence equivalent to IN

- Other operators that can be combined with ANY (or SOME): $>$, $\geq$, $<$, $\leq$, and $\neq$

```
SELECT Lname, Fname
FROM EMPLOYEE
WHERE Salary > ALL
(SELECT Salary
FROM EMPLOYEE
WHERE Dno=5);
```
Avoid potential errors and ambiguities

- Create tuple variables (aliases) for all tables referenced in SQL query

---

**Query 16.** Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```sql
Q16: SELECT E.Fname, E.Lname
     FROM EMPLOYEE AS E
     WHERE E.Ssn IN ( SELECT Essn
                       FROM DEPENDENT AS D
                       WHERE E.Fname=DDEPENDENT_name
                          AND E.Sex=D.Sex );
```
Correlated Nested Queries

- **Correlated** nested query
  - Evaluated once for each tuple in the outer query
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(A,B).
- `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.
- 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.

  ```sql
  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

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>502</td>
</tr>
<tr>
<td>504</td>
</tr>
<tr>
<td>505</td>
</tr>
<tr>
<td>506</td>
</tr>
</tbody>
</table>
Subqueries

- In addition to the > operator, we could also use other standard comparison operators between two tuple values, such as >=, <>, 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;`

2. `SELECT Invoice.I FROM Invoice, InvoiceTemp01 WHERE Invoice.Idate = InvoiceTemp01.Idate AND Amt > MinAmt;`
Subqueries
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 $\text{op}$ a comparison operator.

Then

- “$v \text{ op } \text{ANY } r$” is true if and only if $v \text{ op } x$ is true for at least one $x$ in $r$.
- “$v \text{ op } \text{ALL } r$” is true if and only if $v \text{ op } x$ is true for each $x$ in $r$.
Find every I for which Amt is larger than the largest Amt on February 2, 2009

```sql
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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>502</td>
</tr>
<tr>
<td></td>
<td>503</td>
</tr>
<tr>
<td></td>
<td>504</td>
</tr>
<tr>
<td></td>
<td>507</td>
</tr>
</tbody>
</table>
Find every I for which Amt is larger than the smallest Amt on February 2, 2009

```sql
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
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 “= ANY”
- 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
The EXISTS and UNIQUE Functions in SQL

- **EXISTS function**
  - Check whether the result of a correlated nested query is empty or not

- **EXISTS and NOT EXISTS**
  - Typically used in conjunction with a correlated nested query

- **SQL function UNIQUE (Q)**
  - Returns **TRUE** if there are no duplicate tuples in the result of query Q
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);
```
Find all Cnames who have an entry in Invoice

```
SELECT Cname
FROM Customer
WHERE EXISTS
(SELECT *
FROM Invoice
WHERE Customer.C = Invoice.C);
```
Implementing Intersection & Difference If They Are Not Directly Available

- See *SetOperationsInSql.mdb* in extras

- 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 it: problems with NULLs
SELECT DISTINCT *  
FROM R  
WHERE EXISTS  
(SELECT *  
FROM S  
WHERE  
R.First = S.First AND R.Second = S.Second);

- Note that a tuple containing nulls, (NULL,c), is not in the result, and it should not be there
SELECT DISTINCT *
FROM R
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);
```

- Note that tuples containing nulls, (b,NULL) and (NULL,c), are in the result, and they should be there
SELECT DISTINCT * 
FROM R 
WHERE EXISTS (SELECT * 
FROM S 
SELECT DISTINCT *
FROM R
WHERE NOT EXISTS (SELECT *
FROM S
SELECT DISTINCT * 
FROM P 
WHERE A IN (SELECT A 
FROM Q);
Set Difference For Tables With One Column

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

- Note (NULL) is not in the result, so our query is not quite correct (as per previous warning)
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);
```
We want to compute the set of Cnames that have at least all the Cnames that Chicago has.

<table>
<thead>
<tr>
<th>CnameInCcity</th>
<th>Ccity</th>
<th>Cname</th>
<th>CnameInChicago</th>
<th>Cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boston</td>
<td>Doe</td>
<td></td>
<td></td>
<td>Doe</td>
</tr>
<tr>
<td>Boston</td>
<td>Yao</td>
<td></td>
<td></td>
<td>Yao</td>
</tr>
<tr>
<td>Boston</td>
<td>Smith</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chicago</td>
<td>Doe</td>
<td></td>
<td></td>
<td>Doe</td>
</tr>
<tr>
<td>Chicago</td>
<td>Yao</td>
<td></td>
<td></td>
<td>Yao</td>
</tr>
<tr>
<td>Seattle</td>
<td>Doe</td>
<td></td>
<td></td>
<td>Doe</td>
</tr>
<tr>
<td>Seattle</td>
<td>Smith</td>
<td></td>
<td></td>
<td>Smith</td>
</tr>
<tr>
<td>Denver</td>
<td>Smith</td>
<td></td>
<td></td>
<td>Smith</td>
</tr>
<tr>
<td>Denver</td>
<td>Brown</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
- List all cities, the set of whose profits, contains all the profits that are in Chicago.

```sql
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
In Microsoft Access
### Why Division Was So Difficult

- Let us return to a previous example

<table>
<thead>
<tr>
<th>Has</th>
<th>Person</th>
<th>Tool</th>
<th>Needed</th>
<th>Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Marsha</td>
<td>Fork</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Marsha</td>
<td>Knife</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Marsha</td>
<td>Spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vijay</td>
<td>Fork</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Vijay</td>
<td>Knife</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dong</td>
<td>Fork</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Dong</td>
<td>Spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chris</td>
<td>Spoon</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Chris</td>
<td>Cup</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Asking About Some And About All

- List all Persons, whose set of Tools contains \textbf{at least one} Tool that is (also) in Needed
- This will be easy
- List all Persons, whose set of Tools contains \textbf{at least all} the Tools that are (also) in Needed
- This will be harder
- This is technically called a \textit{division}: Has divided by Needed
- List all Persons, whose set of Tools contains *at least one* Tool that is (also) in Needed
- The result can be expressed using a logical formula with an existential quantifier:

\[ \{ p \mid \exists t [ t \in N \land (p, t) \in H ] \} \]
- The standard SELECT ... FROM ... WHERE ...
  easily expresses the existential quantifier above
- \( p \) “is good” if it has at least one needed tool
Asking About All

- List all Persons, whose set of Tools contains \textit{at least all} the Tools that are (also) in Needed.
- The result can be expressed using a logical formula with a universal quantifier:
  \[
  \{ p \mid \forall t[t \in N \rightarrow (p,t) \in H] \}
  \]
- Using, the following four facts in predicate calculus, we can rewrite out formula, using an existential quantifier, and that is what we, in effect, did while using SQL:
  \[
  \neg\neg\alpha \equiv \alpha
  \]
  \[
  \alpha \rightarrow \beta \equiv \neg\alpha \lor \beta
  \]
  \[
  \neg(\alpha \lor \beta) \equiv \neg\alpha \land \neg\beta
  \]
  \[
  \forall x[A(x)] \equiv \neg\exists x[\neg A(x)]
  \]
  \[
  \{ p \mid \neg\exists t[t \in N \land (p,t) \notin H] \}
  \]
Key Ideas

- Division is really an application of a universal quantifier
- Comparison of pure relational algebra to SQL DML
Joined Tables in SQL and Outer Joins

- **Joined table**
  - Permits users to specify a table resulting from a join operation in the FROM clause of a query

- The FROM clause in Q1A
  - Contains a single joined table

Q1A:  
```
SELECT Fname, Lname, Address  
FROM (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)  
WHERE Dname='Research';
```
Specify different types of join

- NATURAL JOIN
- Various types of OUTER JOIN

NATURAL JOIN on two relations R and S

- No join condition specified
- Implicit EQUIJOIN condition for each pair of attributes with same name from R and S
- **Inner join**
  - Default type of join in a joined table
  - Tuple is included in the result only if a matching tuple exists in the other relation

- **LEFT OUTER JOIN**
  - Every tuple in left table must appear in result
  - If no matching tuple
    - Padded with NULL values for attributes of right table
- **RIGHT OUTER JOIN**
  - Every tuple in right table must appear in result
  - If no matching tuple
    - Padded with NULL values for the attributes of left table

- **FULL OUTER JOIN**
- Can nest join specifications
SQL has a variety of “modified” Cartesian Products, called joins

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
- 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.

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>f</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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.

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>f</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>4</td>
<td>h</td>
</tr>
</tbody>
</table>
### In Microsoft Access

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>f</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>h</td>
</tr>
</tbody>
</table>
- SELECT *
  FROM R FULL OUTER JOIN S
  ON R.B = S.C;

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>g</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>h</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>f</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Digression: Execution Plan Matters

- Consider a database consisting of 3 relations
  - Lives(Person, City) about people in the US, about 300,000,000 tuples
  - Oscar(Person) about people in the US who have won the Oscar, about 1,000 tuples
  - Nobel(Person) about people in the US who have won the Nobel, about 100 tuples

- How would you answer the question, trying to do it most efficiently “by hand”?
- Produce the relation Good_Match(Person1, Person2) where the two Persons live in the same city and the first won the Oscar prize and the second won the Nobel prize

- How would you do it using SQL?
Digression: Execution Plan Matters

- **SELECT** Oscar.Person Person1, Nobel.Person Person2
  **FROM** Oscar, Lives Lives1, Nobel, Lives Lives2
  **WHERE** Oscar.Person = Lives1.Person
  **AND** Nobel.Person = Lives2.Person
  **AND** Lives1.City = Lives2.City;

very inefficient

- Using various joins (which, we did not cover) or intermediate tables, we can specify easily the “right order,” in effect producing
  - Oscar_PC(Person,City), listing people with Oscars and their cities
  - Nobel_PC(Person,City), listing people with Nobels and their cities

- Then producing the result from these two small relations
- This is much more efficient
- But the cleanest way is to use “big cartesian product”
It is possible to specify ranges, or templates:

Find all P and Pcity for plants in cities starting with letters B through D

```sql
SELECT P, Pcity
FROM Plant
WHERE ((City BETWEEN 'B' AND 'I') 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.”
### Query to Find Cities Between Specific Codes

<table>
<thead>
<tr>
<th>P</th>
<th>Pcity</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>Boston</td>
</tr>
<tr>
<td>902</td>
<td>Boston</td>
</tr>
<tr>
<td>903</td>
<td>Chicago</td>
</tr>
<tr>
<td>904</td>
<td>Chicago</td>
</tr>
<tr>
<td>905</td>
<td>Denver</td>
</tr>
<tr>
<td>908</td>
<td>Boston</td>
</tr>
</tbody>
</table>
Find pnames for cities containing the letter X in the second position:

```sql
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:

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

![Query Profit Descending P Ascending Table]

<table>
<thead>
<tr>
<th>Thousands</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>65</td>
<td>907</td>
</tr>
<tr>
<td>56</td>
<td>902</td>
</tr>
<tr>
<td>51</td>
<td>904</td>
</tr>
<tr>
<td>51</td>
<td>906</td>
</tr>
<tr>
<td>51</td>
<td>908</td>
</tr>
<tr>
<td>48</td>
<td>905</td>
</tr>
<tr>
<td>45</td>
<td>901</td>
</tr>
<tr>
<td></td>
<td>903</td>
</tr>
</tbody>
</table>
Create the relation with attributes Idate, C while removing duplicate rows.

\[
\text{SELECT DISTINCT Idate, C FROM Invoice;}
\]
In Microsoft Access

<table>
<thead>
<tr>
<th>Idate</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>4000</td>
</tr>
<tr>
<td>2009-02-01</td>
<td>1000</td>
</tr>
<tr>
<td>2009-02-02</td>
<td>2000</td>
</tr>
<tr>
<td>2009-02-02</td>
<td>4000</td>
</tr>
<tr>
<td>2009-02-03</td>
<td>1000</td>
</tr>
<tr>
<td>2009-02-03</td>
<td>3000</td>
</tr>
</tbody>
</table>
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, or course those who have placed no orders)
  
  ```sql
  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
  
  ```sql
  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
- 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
- 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

<table>
<thead>
<tr>
<th>P</th>
<th>Pname</th>
<th>Pcity</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>Alpha</td>
<td>Boston</td>
<td>$45,000.00</td>
</tr>
<tr>
<td>902</td>
<td>Beta</td>
<td>Boston</td>
<td>$56,000.00</td>
</tr>
<tr>
<td>903</td>
<td>Beta</td>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>904</td>
<td>Gamma</td>
<td>Chicago</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>905</td>
<td>Delta</td>
<td>Denver</td>
<td>$48,000.00</td>
</tr>
<tr>
<td>906</td>
<td>Epsilon</td>
<td>Miami</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>907</td>
<td>Beta</td>
<td>Miami</td>
<td>$65,000.00</td>
</tr>
<tr>
<td>908</td>
<td>Beta</td>
<td>Boston</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>909</td>
<td>Gamma</td>
<td></td>
<td>$52,000.00</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>P</th>
<th>Pname</th>
<th>Pcity</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>Alpha</td>
<td>Boston</td>
<td>$45,000.00</td>
</tr>
<tr>
<td>902</td>
<td>Beta</td>
<td>Boston</td>
<td>$56,000.00</td>
</tr>
<tr>
<td>903</td>
<td>Beta</td>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>904</td>
<td>Gamma</td>
<td>Chicago</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>905</td>
<td>Delta</td>
<td>Denver</td>
<td>$48,000.00</td>
</tr>
<tr>
<td>906</td>
<td>Epsilon</td>
<td>Miami</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>907</td>
<td>Beta</td>
<td>Miami</td>
<td>$65,000.00</td>
</tr>
<tr>
<td>908</td>
<td>Beta</td>
<td>Boston</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>909</td>
<td>Gamma</td>
<td></td>
<td>$52,000.00</td>
</tr>
</tbody>
</table>

### InsertTupleIntoPlantWithDefaultValue

<table>
<thead>
<tr>
<th>P</th>
<th>Pname</th>
<th>Pcity</th>
<th>Profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>Alpha</td>
<td>Boston</td>
<td>$45,000.00</td>
</tr>
<tr>
<td>902</td>
<td>Beta</td>
<td>Boston</td>
<td>$56,000.00</td>
</tr>
<tr>
<td>903</td>
<td>Beta</td>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>904</td>
<td>Gamma</td>
<td>Chicago</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>905</td>
<td>Delta</td>
<td>Denver</td>
<td>$48,000.00</td>
</tr>
<tr>
<td>906</td>
<td>Epsilon</td>
<td>Miami</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>907</td>
<td>Beta</td>
<td>Miami</td>
<td>$65,000.00</td>
</tr>
<tr>
<td>908</td>
<td>Beta</td>
<td>Boston</td>
<td>$51,000.00</td>
</tr>
<tr>
<td>909</td>
<td>Gamma</td>
<td></td>
<td>$52,000.00</td>
</tr>
<tr>
<td>910</td>
<td>Gamma</td>
<td></td>
<td></td>
</tr>
</tbody>
</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:

```sql
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

<table>
<thead>
<tr>
<th>C</th>
<th>Cname</th>
<th>Ccity</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Doe</td>
<td>Boston</td>
<td>901</td>
</tr>
<tr>
<td>2000</td>
<td>Yao</td>
<td>Boston</td>
<td>902</td>
</tr>
<tr>
<td>3000</td>
<td>Doe</td>
<td>Chicago</td>
<td>903</td>
</tr>
<tr>
<td>4000</td>
<td>Doe</td>
<td>Seattle</td>
<td>907</td>
</tr>
<tr>
<td>5000</td>
<td>Brown</td>
<td>Denver</td>
<td>904</td>
</tr>
<tr>
<td>6000</td>
<td>Smith</td>
<td>Seattle</td>
<td>907</td>
</tr>
<tr>
<td>7000</td>
<td>Yao</td>
<td>Chicago</td>
<td>904</td>
</tr>
<tr>
<td>8000</td>
<td>Smith</td>
<td>Denver</td>
<td>904</td>
</tr>
<tr>
<td>9000</td>
<td>Smith</td>
<td>Boston</td>
<td>903</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>Cname</th>
<th>Ccity</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>9001</td>
<td>Qin</td>
<td>Boston</td>
<td>YES</td>
</tr>
<tr>
<td>9002</td>
<td>Doe</td>
<td>Chicago</td>
<td>NO</td>
</tr>
<tr>
<td>9003</td>
<td>Rao</td>
<td>Chicago</td>
<td></td>
</tr>
</tbody>
</table>
- **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

<table>
<thead>
<tr>
<th>C</th>
<th>Cname</th>
<th>Ccity</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>9001</td>
<td>Qin</td>
<td>Boston</td>
<td>YES</td>
</tr>
<tr>
<td>9002</td>
<td>Doe</td>
<td>Chicago</td>
<td>NO</td>
</tr>
<tr>
<td>9003</td>
<td>Rao</td>
<td>Chicago</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>Cname</th>
<th>Ccity</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>9002</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9003</td>
<td>Rao</td>
<td>Chicago</td>
<td></td>
</tr>
</tbody>
</table>
- **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

<table>
<thead>
<tr>
<th>C</th>
<th>Cname</th>
<th>Ccity</th>
<th>Good</th>
</tr>
</thead>
<tbody>
<tr>
<td>9002</td>
<td>Doe</td>
<td>Chicago</td>
<td>NO</td>
</tr>
<tr>
<td>9003</td>
<td>Rao</td>
<td>Chicago</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
</tr>
</thead>
</table>
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:

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

- But duplicates 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 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

<table>
<thead>
<tr>
<th>Invoice</th>
<th>AddToAmtInInvoice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Amt</td>
</tr>
<tr>
<td>501</td>
<td>30</td>
</tr>
<tr>
<td>502</td>
<td>300</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
</tr>
<tr>
<td>504</td>
<td>160</td>
</tr>
<tr>
<td>505</td>
<td>150</td>
</tr>
<tr>
<td>506</td>
<td>150</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
</tr>
<tr>
<td>508</td>
<td>20</td>
</tr>
<tr>
<td>509</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Invoice</th>
<th>AddToAmtInInvoice</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Amt</td>
</tr>
<tr>
<td>501</td>
<td>31</td>
</tr>
<tr>
<td>502</td>
<td>300</td>
</tr>
<tr>
<td>503</td>
<td>200</td>
</tr>
<tr>
<td>504</td>
<td>161</td>
</tr>
<tr>
<td>505</td>
<td>151</td>
</tr>
<tr>
<td>506</td>
<td>151</td>
</tr>
<tr>
<td>507</td>
<td>200</td>
</tr>
<tr>
<td>508</td>
<td>21</td>
</tr>
<tr>
<td>509</td>
<td>21</td>
</tr>
</tbody>
</table>
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.
We briefly go over some additional extension, to overcome restrictions of SQL.
It is generally possible to “wrap” SQL statements (or similar to SQL statement) using programming language constructs, such as:

- IF
- ELSE
- ELSEIF
- WHILE
- FOR
- REPEAT

So, one can write a procedure that reads some values from some table and based on these values decides to do something to another table.

Pretty straightforward but syntax rather nonstandard with different systems using different syntax.

It is also possible to invoke procedures written in a standard programming language.
Ancestor/Descendant Query

- As we have seen, given Birth(Parent, Child) it is not possible, using standard SQL to compute Answer(Ancestor, Descendant)
- This is an example of a recursive query
- It can be evaluated using an appropriate procedure “outside” standard SQL, or using a special extension: WITH RECURSIVE
WITH RECURSIVE Answer(Ancestor,Descendant) AS
(
SELECT Parent AS Ancestor, Child AS Descendant
FROM Birth
UNION
SELECT Birth.Parent, Ancestor.Descendant
FROM Birth, Answer
WHERE Birth.Child = Answer.Ancestor
)
SELECT *
FROM Answer

We populate Answer with Birth, and then as long as possible add longer and longer “chains” one generation at a time, until all of the Answer is obtained
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 an request to the relational database to update your account
» . . .
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.
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):

  ```sql
  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

  ```sql
  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
  
  ```sql
  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.
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

```sql
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;
```
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
Summary

- Complex SQL
  - Nested queries, joined tables, outer joins, aggregate functions, grouping

- Views
  - Virtual or derived tables
1. Session Overview
2. Data Manipulation Language for Relational DBs
3. Data Definition Language for Relational DBs
4. Summary and Conclusion
Agenda (1/2)

- CREATE for defining tables
  - Specifying domains
  - PRIMARY KEY
  - UNIQUE
  - FOREIGN KEY
  - NOT NULL
  - CHECK
  - DEFAULT

- Unknowns

- Maintenance of referential integrity

- Constraint checking
  - NOT DEFERRABLE
  - DEFERRABLE

- ASSERTION
Trigger “on” INSERT, UPDATE, DELETE, “firing” BEFORE, AFTER, INSTEAD

Views

Updating views with SQL UPDATE

Updating views with INSTEAD TRIGGERs

ALTER, DROP, REPLACE

Privileges:
  » Select
  » Insert
  » Update
  » Delete
  » References
Schema Change Statements in SQL

- **Schema evolution commands**
  - Can be done while the database is operational
  - Does not require recompilation of the database schema
The DROP Command

- DROP command
  - Used to drop named schema elements, such as tables, domains, or constraint

- Drop behavior options:
  - CASCADE and RESTRICT

- Example:
  - DROP SCHEMA COMPANY CASCADE;
**The ALTER Command**

- **Alter table actions** include:
  - Adding or dropping a column (attribute)
  - Changing a column definition
  - Adding or dropping table constraints

- **Example:**
  
  ```sql
  ALTER TABLE COMPANY.EMPLOYEE ADD COLUMN Job VARCHAR(12);
  ```

- **To drop a column**
  
  Choose either **CASCADE** or **RESTRICT**
The ALTER Command (cont’d.)

- Change constraints specified on a table
  - Add or drop a named constraint

```sql
ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFKEY CASCADE;
```
• This is the database we will define
• We do not pay attention to domains of attributes as there is not much interesting in this
We will only some of the basic capabilities for defining a relational database

The standard is very extensive and provides for a rich repertoire of useful capabilities

We can only touch on some of them

But enough for defining reasonable complexity databases
Basic Definition

- CREATE TABLE Plant ( 
  P CHAR(10),
  Pname CHAR VARYING(10),
  Pcity CHAR VARYING(10),
  Profit NUMBER
);

- This is a minimal definition
  » Name of the table
  » Names of the columns
  » Domains of the columns
CREATE TABLE Customer ( 
C CHAR(10),
Cname CHAR VARYING(10),
Ccity CHAR VARYING(10),
P CHAR(10)
);

This is a minimal definition
» Name of the table
» Names of the columns
» Domains of the columns
Basic Definition

- CREATE TABLE Invoice ( 
  I CHAR(10),
  Amt NUMBER,
  Idate DATE,
  C CHAR(10)
);

- This is a minimal definition
  » Name of the table
  » Names of the columns
  » Domains of the columns
Permitted Data Types (Data Domains)

- SQL standard specifies permitted data types, which can be roughly grouped into several families:
  - Integers (small or long)
  - Real numbers (standard or double length and with various precisions)
  - Character strings (fixed or variable length)
  - Bit strings (fixed or variable length)
  - Dates and times (various specifications with various time “granularity”)

- Systems have different implementations and modifications of the standard
- In some of the slides, new concepts will be introduced
- The SQL specifications will be in color and bold to draw attention to them
- CREATE TABLE Plant ( 
  P CHAR(10) NOT NULL, 
  Pname CHAR VARYING(10), 
  Pcity CHAR VARYING(10), 
  Profit NUMBER, 
  CONSTRAINT C_20 PRIMARY KEY (P), 
  CONSTRAINT C_30 UNIQUE (Pcity, Profit), 
  CONSTRAINT C_40 CHECK ( Pcity <> Pname ), 
  CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) ) 
); 

- This is a minimal definition 
  » Name of the table 
  » Names of the columns 
  » Domains of the columns
CREATE TABLE Plant ( 
P CHAR(10) NOT NULL,
Pname CHAR VARYING(10),
Pcity CHAR VARYING(10),
Profit NUMBER,
CONSTRAINT C_20 PRIMARY KEY (P),
CONSTRAINT C_30 UNIQUE (Pcity, Profit),
CONSTRAINT C_40 CHECK ( Pcity <> Pname ),
CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) )
);

Specifies that the values in these columns (could be more than one such column) must not be NULL
CREATE TABLE Plant ( 
P CHAR(10) NOT NULL, 
Pname CHAR VARYING(10), 
Pcity CHAR VARYING(10), 
Profit NUMBER, 
CONSTRAINT C_20 PRIMARY KEY (P), 
CONSTRAINT C_30 UNIQUE (Pcity, Profit), 
CONSTRAINT C_40 CHECK ( Pcity <> Pname ), 
CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) )
);

Some constraint on the tables
» Constraint name, here C_20, is not required, but it is a very good idea to give unique names to a constraint, so it can be later DROPPed or ALTERed by referring to it by its name
» Constraint name should reflect something about the constraint, to save space I used short names
CREATE TABLE Plant ( 
P CHAR(10) NOT NULL, 
Pname CHAR VARYING(10), 
Pcity CHAR VARYING(10), 
Profit NUMBER, 
CONSTRAINT C_20 PRIMARY KEY (P), 
CONSTRAINT C_30 UNIQUE (Pcity, Profit), 
CONSTRAINT C_40 CHECK ( Pcity <> Pname ), 
CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) )
);

The column P is the primary key (only one possible)
» This requires that it must not be NULL (this is not necessary to state in some systems, as the primary key condition automatically forces it by SQL standard)

Primary key could be several columns, e.g., PRIMARY KEY(Pcity, Profit); but not in our example
CREATE TABLE Plant (  
P CHAR(10) NOT NULL,  
Pname CHAR VARYING(10),  
Pcity CHAR VARYING(10),  
Profit NUMBER,  
CONSTRAINT C_20 PRIMARY KEY (P),  
CONSTRAINT C_30 UNIQUE (Pcity, Profit),  
CONSTRAINT C_40 CHECK ( Pcity <> Pname ),  
CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) )  
);

The “subtuple” PCITY,PNAME is a candidate key
  » There is no requirement, in general, about any of its column being not NULL
  » To reiterate: all the columns of the primary key must not be NULL
### Check (and Unknown)

- CREATE TABLE Plant (  
P CHAR(10) NOT NULL,  
Pname CHAR VARYING(10),  
Pcity CHAR VARYING(10),  
Profit NUMBER,  
CONSTRAINT C_20 PRIMARY KEY (P),  
CONSTRAINT C_30 UNIQUE (Pcity, Profit),  
CONSTRAINT C_40 CHECK ( Pcity <> Pname ),  
CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) )
);

- Every tuple must satisfy this condition
- The condition is **satisfied**, when it is either  
  » **TRUE**, or  
  » **UNKNOWN** (so if Pcity is Null, this condition is satisfied)
- **Recall in SQL DML: UNKNOWN implies “not satisfied”**
CREATE TABLE Plant (  
P CHAR(10) NOT NULL,  
Pname CHAR VARYING(10),  
Pcity CHAR VARYING(10),  
Profit NUMBER,  
CONSTRAINT C_20 PRIMARY KEY (P),  
CONSTRAINT C_30 UNIQUE (Pcity, Profit),  
CONSTRAINT C_40 CHECK ( Pcity <> Pname ),  
CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) )
);

This is: (Pcity = 'Chicago') → (Profit > 1000)  
By standard rules of Boolean operators (propositional calculus)
Check (and Unknown)

- CREATE TABLE Plant ( 
  P CHAR(10) NOT NULL, 
  Pname CHAR VARYING(10), 
  Pcity CHAR VARYING(10), 
  Profit NUMBER, 
  CONSTRAINT C_20 PRIMARY KEY (P), 
  CONSTRAINT C_30 UNIQUE (Pcity, Profit), 
  CONSTRAINT C_40 CHECK ( Pcity <> Pname ), 
  CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) ) 
); 

- Returning to semantics of UNKNOWN and OR, this constraint has to evaluate to TRUE or UNKNOWN to be satisfied, so we need ( Pcity is not Chicago or is NULL) or ( Profit is greater than 1000 or is NULL)

- So for Chicago the profit is greater than 1000 or is NULL
CREATE TABLE Customer (  
C CHAR(10) NOT NULL,
Cname CHAR VARYING(10) DEFAULT (NULL),
Ccity CHAR VARYING(10),
P CHAR(10) DEFAULT ('Main'),
CONSTRAINT C_60 PRIMARY KEY (C),
CONSTRAINT C_70 FOREIGN KEY (P) REFERENCES Plant ON DELETE SET NULL );

It is possible to specify defaults
  » E.g., when a tuple is inserted and only C and Ccity are specified, the system knows to specify NULL for Cname and Main for P
CREATE TABLE Customer (  
C CHAR(10) NOT NULL,  
Cname CHAR VARYING(10) DEFAULT (NULL),  
Ccity CHAR VARYING(10),  
P CHAR(10) DEFAULT ('Main'),  
CONSTRAINT C_60 PRIMARY KEY (C),  
CONSTRAINT C_70 FOREIGN KEY (P) REFERENCES Plant ON DELETE SET NULL);

- P in Customer has to reference the primary key of Plant
- This means that one of two conditions is satisfied
  - P has a non NULL value and this value of P appears in Plant
  - P is NULL
    Of course, if P were specified as NOT NULL, this could not be the case
On Delete Set Null

- CREATE TABLE Customer (  
  C CHAR(10) NOT NULL,  
  Cname CHAR VARYING(10) DEFAULT (NULL),  
  Ccity CHAR VARYING(10),  
  P CHAR(10) DEFAULT ('Main'),  
  CONSTRAINT C_60 PRIMARY KEY (C),  
  CONSTRAINT C_70 FOREIGN KEY (P)  
  REFERENCES Plant ON DELETE SET NULL  
);  
- P in Customer has to reference the primary key of Plant  
- But note, that P in Customer is not required to be NOT NULL  
- We have a specification that if P listed in some tuple of Customer is deleted from Plant (that is the tuple with this value of primary key is deleted), then that value of P in Plant is automatically replaced by NULL
Not Null

- CREATE TABLE Invoice (  
  I CHAR(10) NOT NULL,  
  Amt NUMBER,  
  Idate DATE,  
  C CHAR(10) NOT NULL,  
  CONSTRAINT C_80 PRIMARY KEY (I),  
  CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES Customer ON DELETE CASCADE  
);

- NOT NULL can be specified for columns not in the primary key
On Delete Cascade

- CREATE TABLE Invoice (  
  I CHAR(10) NOT NULL,  
  Amt NUMBER,  
  Idate DATE,  
  C CHAR(10) NOT NULL,  
  CONSTRAINT C_80 PRIMARY KEY (I),  
  CONSTRAINT C_90 FOREIGN KEY (C)  
  REFERENCES Customer ON DELETE CASCADE  
);  

- We have a specification that if C listed in some tuple of Invoice is deleted from Customer (that is the tuple with this value of primary key is deleted), all the tuples with this value of C in Invoice must be deleted.
In order to maintain referential integrity constraints, the system will reject any operation that will violate it.

» There are subtle interactions if NULLs are present; we will not discuss them here

CREATE TABLE Invoice (  
  I CHAR(10) NOT NULL,  
  Amt NUMBER,  
  Idate DATE,  
  C CHAR(10) NOT NULL,  
  CONSTRAINT C_80 PRIMARY KEY (I),  
  CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES Customer ON . . .  
);
This constraint “will act” when:

- An **INSERT** or an **UPDATE** on Invoice is attempted that would produce there a value of \( C \) that does not exist in Customer.
- A **DELETE** or an **UPDATE** on Customer is attempted that will leave tuples in Invoice in which the value of \( C \) does not appear in any tuple of Customer.

The default is **NO ACTION**, that is the above will not be permitted.

We will briefly discuss other options in case of **UPDATE**s of Customer and skip what happens in other cases:

- **CASCADE**: the new value of the primary key is copied to the foreign key
- **SET NULL**: the new value of the foreign key is **NULL**
- **SET DEFAULT**: the new value of the foreign key is a specified default value (which of course has to appear in Customer)
It is generally a good idea to start with a basic definition and augment it with constraints later.

We see how this is done.
Basic Definition

- CREATE TABLE Plant (P CHAR(10) NOT NULL, Pname CHAR VARYING(10), Pcity CHAR VARYING(10), Profit NUMBER);
CREATE TABLE Customer (  
C CHAR(10) NOT NULL,  
Cname CHAR VARYING(10) DEFAULT (NULL),  
Ccity CHAR VARYING(10),  
P CHAR(10) DEFAULT ('Main')  
);
CREATE TABLE Invoice ( 
  I CHAR(10) NOT NULL, 
  Amt NUMBER, 
  Idate DATE, 
  C CHAR(10) NOT NULL 
);
ALTER TABLE Plant ADD CONSTRAINT C_20 PRIMARY KEY (P);

ALTER TABLE Customer ADD CONSTRAINT C_60 PRIMARY KEY (C);

ALTER TABLE Invoice ADD CONSTRAINT C_80 PRIMARY KEY (I);

ALTER TABLE Customer ADD CONSTRAINT C_70 FOREIGN KEY (P) REFERENCES Plant ON DELETE SET NULL;

ALTER TABLE Invoice ADD CONSTRAINT C_90 FOREIGN KEY (C) REFERENCES Customer ON DELETE CASCADE;
Altering The Definition To Add Constraints

- ALTER TABLE Plant ADD CONSTRAINT C_30 UNIQUE (Pcity, Profit);
- ALTER TABLE Plant ADD CONSTRAINT C_40 CHECK ( Pcity <> Pname );
- ALTER TABLE Plant ADD CONSTRAINT C_50 CHECK ( (Pcity <> 'Chicago') OR (Profit > 1000) );
Referencing Unique

- Foreign key can also refer to UNIQUE and not only to PRIMARY KEY
- So we could also add to our database such a constraint, for which we look at an example
- CREATE TABLE Test (TestID CHAR(10) NOT NULL, TestPname CHAR VARYING(10), TestPcity CHAR VARYING(10), TestProfit NUMBER);
- ALTER TABLE Test ADD CONSTRAINT C_99 FOREIGN KEY (TestPcity, TestProfit) REFERENCES Plant(Pcity, Profit);
Sometimes It Is Necessary To Define Tables First & Then Add Constraints

- If you define a foreign key constraint, it cannot refer to a table that has not yet been designed
- Consider the following Visio diagram

```
<table>
<thead>
<tr>
<th>Husband</th>
<th>Wife</th>
</tr>
</thead>
<tbody>
<tr>
<td>PK</td>
<td>PK</td>
</tr>
<tr>
<td>Hname</td>
<td>Wname</td>
</tr>
<tr>
<td>FK1</td>
<td>FK1</td>
</tr>
<tr>
<td>Wname</td>
<td>Hname</td>
</tr>
</tbody>
</table>
```

- You have “circular” dependencies
  - You cannot fully define Husband before Wife
  - You cannot fully define Wife before Husband
- Therefore
  1. Produce basic definitions for Husband and Wife
  2. Alter them by adding constraints later
UNIQUE and PRIMARY KEY

- Back to our old example
- CREATE TABLE City
  Country NOT NULL,
  State,
  Name NOT NULL,
  Longitude NOT NULL,
  Latitude NOT NULL
);

- A city can be identified in one of two ways
  » By its geographic location: Longitude and Latitude
  » By its official “hierarchy of names”: Country, State, Name

- It may be the case that some countries are not divided into states (or equivalent units)
  » For them it is natural to allow State to be NULL, as opposed to faking something
- The following is OK
- CREATE TABLE City
  Country NOT NULL,
  State,
  Name NOT NULL,
  Longitude NOT NULL,
  Latitude NOT NULL,
  UNIQUE (Country, State, Name),
  PRIMARY KEY (Longitude, Latitude) );
The following *is not OK*

```sql
CREATE TABLE City
Country NOT NULL,
State,
Name NOT NULL,
Longitude NOT NULL,
Latitude NOT NULL,
PRIMARY KEY (Country, State, Name),
UNIQUE (Longitude, Latitude) );
```

- Because State could be NULL, not permitted in primary key
- We will see why primary keys should not contain NULLs (there are other reasons for this too)
Small database

CREATE TABLE City_Population
Country NOT NULL,
State,
Name NOT NULL,
Longitude NOT NULL,
Latitude NOT NULL,
Population,
PRIMARY KEY (Country, State, Name),
UNIQUE (Longitude, Latitude) );

CREATE TABLE City_Size
Country NOT NULL,
State,
Name NOT NULL,
Longitude NOT NULL,
Latitude NOT NULL,
Size,
PRIMARY KEY (Country, State, Name),
UNIQUE (Longitude, Latitude) );
We want to combine information about cities from both tables

SELECT *
FROM City_Population, City_Size
WHERE (City_Population.Country = City_Size.Country
AND City_Population.State = City_Size.State
AND City_Population.Name = City_Size.Name) ;

We will not get anything for cities in countries that are not divided into states!

Because the result of comparison of say (Monaco, NULL, Monaco-Ville) = (Monaco, NULL, Monaco-Ville) is UNKNOWN

Therefore, we cannot have (Country,State,Name) as PRIMARY KEY
The following can be done if we want to use UNIQUE set of attributes for joining in our example

```
SELECT *
FROM City_Population, City_Size
WHERE City_Population.Country = City_Size.Country
AND City_Population.Name = City_Size.Name
AND ( City_Population.State = City_Size.State
    OR (City_Population.State IS NULL
        AND City_Size.State IS NULL ) )
```
When Are Constraints Checked?

- Essentially, each row of the TABLE has to satisfy the constraint.
- Constraints are checked as tables are modified (immediately or deferred until later, generally until the end of a transaction).
- The actual checking is done either after each statement or at the end of a transaction.
  - It is done at the end, to allow changes that cannot be done in a single statement.
  - For example if Total = Checking + Savings and money is moved from Checking to Savings this constraint could be violated in the middle of the move, but must be satisfied before and after the move.
- So as part of specification of a constraint one can specify:
  - NOT DEFERRABLE (this is the default), or
  - DEFERRABLE.
Specifying Constraints as Assertions and Actions as Triggers

- **CREATE ASSERTION**
  - Specify additional types of constraints outside scope of built-in relational model constraints

- **CREATE TRIGGER**
  - Specify automatic actions that database system will perform when certain events and conditions occur
CREATE ASSERTION

- Specify a query that selects any tuples that violate the desired condition
- Use only in cases where it is not possible to use CHECK on attributes and domains

```
CREATE ASSERTION SALARY_CONSTRAINT
CHECK ( NOT EXISTS ( SELECT * 
    FROM EMPLOYEE E, EMPLOYEE M,
    DEPARTMENT D
    WHERE E.Salary > M.Salary
    AND E.Dno = D.Dnumber
    AND D.Mgr_ssn = M.Ssn ) );
```
Introduction to Triggers in SQL

- **CREATE TRIGGER** statement
  - Used to monitor the database
- Typical trigger has three components:
  - Event(s)
  - Condition
  - Action
Assertions

- Assertion is like a CHECK constraint, **but** it is not attached to a TABLE definition; it is “free floating”

- **CREATE ASSERTION** Assertion01
  
  ```sql
  CHECK
  ( (SELECT COUNT (*) FROM Plant) + (SELECT COUNT (*) FROM Customer) < 1000 );
  ```

- Assertions are more natural than previously described constraints, especially when referring to several tables

- However, they are frequently not implemented, e.g., Oracle

- It is very difficult to implement them both correctly and efficiently
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 in this unit:
  - 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 (this was tested in Oracle)
  - The exact trigger in Oracle
  - A partial trace of the execution in Oracle
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**
### Customer and Plant before Insert

<table>
<thead>
<tr>
<th>C</th>
<th>CNAME</th>
<th>CCITY</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Doe</td>
<td>Boston</td>
<td>901</td>
</tr>
<tr>
<td>2000</td>
<td>Yao</td>
<td>Boston</td>
<td>902</td>
</tr>
<tr>
<td>3000</td>
<td>Doe</td>
<td>Chicago</td>
<td>903</td>
</tr>
<tr>
<td>4000</td>
<td>Doe</td>
<td>Seattle</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>Brown</td>
<td>Denver</td>
<td>903</td>
</tr>
<tr>
<td>6000</td>
<td>Smith</td>
<td>Seattle</td>
<td>907</td>
</tr>
<tr>
<td>7000</td>
<td>Yao</td>
<td>Chicago</td>
<td>904</td>
</tr>
<tr>
<td>8000</td>
<td>Smith</td>
<td>Denver</td>
<td>904</td>
</tr>
<tr>
<td>9000</td>
<td>Smith</td>
<td>Boston</td>
<td>903</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>PNAME</th>
<th>PCITY</th>
<th>PROFIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>Alpha</td>
<td>Boston</td>
<td>45000</td>
</tr>
<tr>
<td>902</td>
<td>Beta</td>
<td>Boston</td>
<td>56000</td>
</tr>
<tr>
<td>903</td>
<td>Beta</td>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>904</td>
<td>Gamma</td>
<td>Chicago</td>
<td>51000</td>
</tr>
<tr>
<td>905</td>
<td>Delta</td>
<td>Denver</td>
<td>48000</td>
</tr>
<tr>
<td>906</td>
<td>Epsilon</td>
<td>Miami</td>
<td>51000</td>
</tr>
<tr>
<td>907</td>
<td>Beta</td>
<td>Miami</td>
<td>65000</td>
</tr>
<tr>
<td>908</td>
<td>Beta</td>
<td>Boston</td>
<td>51000</td>
</tr>
</tbody>
</table>
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.
### Customer and Plant after Insert

<table>
<thead>
<tr>
<th>C</th>
<th>CNAME</th>
<th>CCITY</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Doe</td>
<td>Boston</td>
<td>901</td>
</tr>
<tr>
<td>2000</td>
<td>Yao</td>
<td>Boston</td>
<td>902</td>
</tr>
<tr>
<td>3000</td>
<td>Doe</td>
<td>Chicago</td>
<td>903</td>
</tr>
<tr>
<td>4000</td>
<td>Doe</td>
<td>Seattle</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>Brown</td>
<td>Denver</td>
<td>903</td>
</tr>
<tr>
<td>6000</td>
<td>Smith</td>
<td>Seattle</td>
<td>907</td>
</tr>
<tr>
<td>7000</td>
<td>Yao</td>
<td>Chicago</td>
<td>904</td>
</tr>
<tr>
<td>8000</td>
<td>Smith</td>
<td>Denver</td>
<td>904</td>
</tr>
<tr>
<td>9000</td>
<td>Smith</td>
<td>Boston</td>
<td>903</td>
</tr>
<tr>
<td>1001</td>
<td>Xiu</td>
<td>Boston</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>P</th>
<th>PNAME</th>
<th>PCITY</th>
<th>PROFIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>901</td>
<td>Alpha</td>
<td>Boston</td>
<td>45000</td>
</tr>
<tr>
<td>902</td>
<td>Beta</td>
<td>Boston</td>
<td>56000</td>
</tr>
<tr>
<td>903</td>
<td>Beta</td>
<td>Chicago</td>
<td></td>
</tr>
<tr>
<td>904</td>
<td>Gamma</td>
<td>Chicago</td>
<td>51000</td>
</tr>
<tr>
<td>905</td>
<td>Delta</td>
<td>Denver</td>
<td>48000</td>
</tr>
<tr>
<td>906</td>
<td>Epsilon</td>
<td>Miami</td>
<td>51000</td>
</tr>
<tr>
<td>907</td>
<td>Beta</td>
<td>Miami</td>
<td>65000</td>
</tr>
<tr>
<td>908</td>
<td>Beta</td>
<td>Boston</td>
<td>51000</td>
</tr>
<tr>
<td>1001</td>
<td>Xiu_Plant</td>
<td>Boston</td>
<td></td>
</tr>
</tbody>
</table>
Concept of a view in SQL

- Single table derived from other tables
- Considered to be a virtual table
### CREATE VIEW command

- Give table name, list of attribute names, and a query to specify the contents of the view

**Example:**

V1:

```
CREATE VIEW WORKS_ON1
AS SELECT Fname, Lname, Pname, Hours
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE Ssn=Essn AND Pno=Pnumber;
```

V2:

```
CREATE VIEW DEPT_INFO(Dept_name, No_of_emps, Total_sal)
AS SELECT Dname, COUNT (*), SUM (Salary)
FROM DEPARTMENT, EMPLOYEE
WHERE Dnumber=Dno
GROUP BY Dname;
```
- Specify SQL queries on a view
- View always up-to-date
  - Responsibility of the DBMS and not the user
- `DROP VIEW` command
  - Dispose of a view
- Complex problem of efficiently implementing a view for querying

- **Query modification** approach
  - Modify view query into a query on underlying base tables
  - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute
View materialization approach
- Physically create a temporary view table when the view is first queried
- Keep that table on the assumption that other queries on the view will follow
- Requires efficient strategy for automatically updating the view table when the base tables are updated
- **Incremental update strategies**
  - DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table
View Update and Inline Views

- Update on a view defined on a single table without any aggregate functions
  - Can be mapped to an update on underlying base table
- View involving joins
  - Often not possible for DBMS to determine which of the updates is intended
- **Clause** `WITH CHECK OPTION`
  - Must be added at the end of the view definition if a view is to be updated

- **In-line view**
  - Defined in the `FROM` clause of an SQL query
We now proceed to the definition of the user level, that is to the definition of views. Generally speaking, a view consists of “continuously current” table that is derived by means of a `SELECT` statement from other tables.

For example, we could write

```sql
CREATE VIEW GoodPlant
AS SELECT *
FROM Plant
WHERE Profit > .0;
```

We could now execute a query against the view

```sql
SELECT P
FROM GoodPlant
WHERE City = 'Chicago';
```

This will give all P for Chicago where Profit is positive.
Views Versus Snapshots

- View is not a snapshot, which is static
- View can be thought of as a procedure call
- Therefore we should think of the following procedure for computing the answer to the last query:
  - The system computes the value of the table GoodPlant
  - The system executes the query against the table GoodPlant
- In actuality, the system may compute the answer differently, however, the result will be equivalent to the canonical procedure described above
In general, almost any query definition could be used to define a view, so we could have:

```
CREATE VIEW Customer_In_The_City
AS SELECT Cname
FROM Plant, Customer
WHERE Pcity = Ccity
AND Plant.C = Customer.C;
```

Views can also be defined **WITH CHECK OPTION**, which we will discuss later.
Updating Views

- Views, in principle, can be updated just like the base tables
- However, all updates to views must be reflected in a correct update to the base table.
- Let us start with the view
  
  CREATE VIEW GoodPlant
  AS SELECT *
  FROM Plant
  WHERE Profit > 0.0;
- Then, it is clear what should be inserted into the table Plant if the following is issued:
  
  INSERT INTO GoodPlant
  VALUES (675, 'Major', 'Philadelphia', .25);
Consider now the view

```
CREATE VIEW SomePlant
AS SELECT P, Pname, City
FROM Plant;
```

Then, if the value of Profit can be NULL or has a defined default value, it is clear what should be inserted into the table Plant if the following is issued:

```
INSERT INTO SomePlant
VALUES (675, 'Major', 'Philadelphia');
```
Consider the view

```
CREATE VIEW Plant_In_Chicago
AS SELECT * 
FROM Plant
WHERE City = 'Chicago';
```

According to SQL the following update is valid

```
INSERT INTO Plant_In_Chicago
VALUES (897,'Minor','Philadelphia',.1);
```

It is reflected properly in the base table Plant, however, it does not show in the view, of course
Instead, if we define the view

```
CREATE VIEW Plant_In_Chicago
AS SELECT *
FROM Plant
WHERE City = 'Chicago'
WITH CHECK OPTION;
```

Then the update

```
INSERT INTO Plant_In_Chicago
VALUES (897,'Minor','Philadelphia',.1);
```

will be rejected
Some Views Cannot Be Updated

- Consider the view

  ```sql
  CREATE VIEW Profit_On_Date
  AS SELECT Profit, Date
  FROM Plant, Invoice, Customer
  WHERE Plant.P = Customer.P
  AND Invoice.C = Customer.C;
  ```

- There is no meaning to the update

  ```sql
  INSERT INTO Profit_On_Date
  VALUES (0.9, 2009-02-01);
  ```

  Why?

  Because there is no well-defined way for reflecting this update in the base tables
  Several tables would need to be modified in a non-deterministic fashion
Consider the view

CREATE VIEW Avg_Amt
AS SELECT AVG(Amt)
FROM Invoice
WHERE Idate = '2009-02-01';

It is not permitted to issue:

INSERT INTO Avg_Amt
VALUES (75);

as there is no way of changing the base tables in a well-defined way.
Consider the view

```
CREATE VIEW Cities_With_Plant
AS SELECT Pcity
FROM Plant;
```

It is not permitted to issue

```
INSERT INTO Cities_With_Plant
VALUES ('Palm Beach');
```

as P cannot have a NULL value, as it was the primary key.
The following are the major conditions (there are others) that must be true for an updatable view:

- Is drawn from one TABLE
  No joins, unions, differences, intersections

- If the underlying TABLE is a view, it must be updateable

- The SELECTed columns are column references (each column at most once and without DISTINCT) and not values or aggregates

- No GROUP BY
Some Views That Should Be Updateable

- It may make sense to update views that the SQL standard does not allow to update and it is now sometimes permissible; that is, in some implementations
- If we have two tables
  - \( R(\text{SSN}, \text{Salary}) \)
  - \( S(\text{SSN}, \text{Address}) \)

- Consider the view

  CREATE VIEW RS  
  AS SELECT R.SSN AS SSN, Salary, Address  
  FROM R, S  
  WHERE R.SSN = S.SSN ;

- And it is perfectly clear what to do if a new employee is inserted into RS: i.e., how to reflect this in R and in S
Updating Views

- SQL prohibits this
- But Oracle actually will execute correctly
- But Oracle will do very strange things too when you attempt to update views in strange ways
- The standard mechanism for updating views when it makes sense uses **INSTEAD** triggers
Using A Trigger To Update A View

- **CREATE TABLE**  \( r \) (  
a CHAR (10) NOT NULL,  
b CHAR (10) NOT NULL,  
  PRIMARY KEY (a)  
);  

- **CREATE TABLE**  \( s \) (  
a CHAR (10) NOT NULL,  
c CHAR (10) NOT NULL,  
  PRIMARY KEY (a)  
);  

- **CREATE VIEW**  \( t \) AS  
  SELECT r.a AS a, r.b AS b, s.c AS c  
  FROM r, s  
  WHERE r.a = s.a;
CREATE TRIGGER trigger02
INSTEAD OF UPDATE ON t
REFERENCING NEW AS new
BEGIN UPDATE s
  SET c = :new.c
WHERE a = :old.a;
END trigger02;
.
RUN;

UPDATE t
SET c = 'q'
WHERE a = '2';
Using A Trigger To Update A View

- Tables R, S, and view T before update on the view

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>n</td>
</tr>
</tbody>
</table>
Using A Trigger To Update A View

- Tables R, S, and view T after update on the view using trigger02

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>q</td>
</tr>
<tr>
<td>3</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>q</td>
</tr>
</tbody>
</table>
Triggers will allow you to do very strange things
CREATE TRIGGER trigger03
INSTEAD OF UPDATE ON t
REFERENCING NEW AS new
BEGIN UPDATE new
SET b = :new.c
WHERE a = :old.a;
END trigger03;
.
RUN

UPDATE t
SET c = 'q'
WHERE a = '2';
Using A Trigger To Update (?) A View

- Tables R, S, and view T before update on the view

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>f</td>
<td>n</td>
</tr>
</tbody>
</table>
Using A Trigger To Update (?) A View

- Tables R, S, and view T after update on the view using trigger03

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td>2</td>
<td>q</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>3</td>
<td>o</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>e</td>
<td>m</td>
</tr>
<tr>
<td>2</td>
<td>q</td>
<td>n</td>
</tr>
</tbody>
</table>
In general, if an object is CREATED, in can subsequently be
- ALTERed (some features are changed)
- DROPped (removed)

Sometimes it can be
- REPLACEd (by a new object)

This is why it is generally a good idea to name constraints, assertions, triggers, etc, while creating them
Privileges can be granted to user or PUBLIC for:

- Operations
- References on
- Base tables
- Views

These are technically part of *Data Control Language* or *DCL*.
Types of Privileges

- Select
- Insert
- Update
- Delete
- References
A typical instruction is:

```sql
GRANT SELECT, INSERT
ON Customer
TO Li, Brown;
```

Privileges can be restricted to columns:

```sql
GRANT SELECT
ON Customer.City
TO Li, Brown;
```

It is possible to grant all privileges by:

```sql
GRANT ALL
ON Customer
TO Li, Brown;
```
It is possible to allow the users to pass the privileges to other users by issuing:

- `GRANT SELECT, INSERT ON Customer TO Li, Brown WITH GRANT OPTION;`

Then Li can issue

- `GRANT SELECT ON Customer.City TO JONES;`
It is possible to allow a user to use columns in a table as foreign keys referring to primary keys in a table to which the user has no privileges:

- **GRANT ALL**
  ON Invoice
  TO Li;

- **GRANT REFERENCES** (C)
  ON Customer
  TO Li;

This privilege must be explicitly granted because Li may be able to check if a particular C appears in Customer

- To check if C = 1 appears in Customer, Li attempts to INSERT an Invoice from C = 1
- If C = 1 does not appear in Customer, the database will complain about violation of FOREIGN KEY constraint
- If C = 1 appears in Customer, the database will not complain about violation of FOREIGN KEY constraint
- This is how Li can check this and that’s why it is explicitly permitted
It is possible to grant privileges on views. Of course, the privilege must be meaningful. That is a privilege to update can be given only on a view that can be updated, etc.
• Privileges can be revoked

• There are various ways to specify what happens with privileges granted by somebody from whom a privilege is taken away
• CREATE ASSERTION and CREATE TRIGGER
Summary

- Data Manipulation Language for Relational DB
- Data Definition Language for Relational DB