Unit 6
SQL: Data Manipulation Language
For Relational Databases 2
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 I own all the plants

How much money I made (or actually invoiced) on February 2, 2009?

Let’s use a nice title for the column (just to practice)

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

Logically, it makes sense that we get 330
In Microsoft Access
Queries With Aggregates

- Assume I own all the plants
- How much money did I make (or actually invoice) 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 would make sense to get 0
- However, we get NULL
In Microsoft Access
Queries With Aggregates

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

- It is possible to have a quite sophisticated query, which shows the importance of this construct:

  (Completely) ignoring all orders placed by $C = 3000$, list for each $Idate$ the sum of all orders placed, if the average order placed was larger than 100

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

- The order of execution is:
  1. FROM
  2. WHERE
  3. GROUP
  4. HAVING
  5. SELECT

- We will trace this example to see how this works
Queries With Aggregates

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

<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.
# 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:

<table>
<thead>
<tr>
<th>Idate</th>
<th>AVG(Amt)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>110</td>
</tr>
<tr>
<td>1</td>
<td>200</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td>NULL</td>
<td>110</td>
</tr>
</tbody>
</table>

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></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 at the group rather than the 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
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 since it was a parameter to GROUP BY

```sql
SELECT Idate, SUM(Amt), MIN(Amt)
FROM Invoice
WHERE C <> 3000
GROUP BY Idate
HAVING AVG(Amt) > 100;
```
In Microsoft Access
But, the following is not OK, as C is not a group property, because on a specific Idate different C’s can place an order

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

Got it right!
Queries With Aggregates

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

```sql
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

<table>
<thead>
<tr>
<th>Amt</th>
<th>Idate</th>
<th>Expr1002</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>2009-02-03</td>
<td>4000</td>
</tr>
<tr>
<td>20</td>
<td>2009-02-03</td>
<td>1000</td>
</tr>
<tr>
<td>30</td>
<td>2009-02-02</td>
<td>2000</td>
</tr>
<tr>
<td>150</td>
<td>2009-02-02</td>
<td>2000</td>
</tr>
<tr>
<td>160</td>
<td>2009-02-03</td>
<td>1000</td>
</tr>
<tr>
<td>200</td>
<td>2009-02-03</td>
<td>2000</td>
</tr>
<tr>
<td>200</td>
<td>2009-02-01</td>
<td>1000</td>
</tr>
<tr>
<td>300</td>
<td>2009-02-03</td>
<td>3000</td>
</tr>
</tbody>
</table>
Queries With Aggregates

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>3000</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>
Subqueries

In a SELECT statement, the WHERE clause can refer to a result of another query, thought of as an “inner loop,” referred to as a subquery.

Consider two relations R(A,B) and S(C,D)

```
SELECT A
FROM R
WHERE B > (SELECT MIN(C)
            FROM S)
```

This will pick up all values of column A of R if the corresponding B is larger than the smallest element in the C column of S.

Generally, a result of a subquery is either one element (perhaps with duplicates) as in the above example or more than one element.

Subqueries are used very frequently, so we look at details.

We start with one element subquery results.
Subqueries

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

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

For each tuple of Invoice1 the value of Amt is compared to the result of the execution of the subquery.

- The subquery is executed (logically) for each tuple of Invoice
- This looks very much like an inner loop, executed logically once each time the outer loop “makes a step forward”

Note that we needed to rename Invoice to be Invoice1 so that we can refer to it appropriately in the subquery.

In the subquery unqualified Idate refers to the nearest encompassing Invoice
Subqueries
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, and 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
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

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

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

• Note, loosely speaking: > ALL X means that for every x in X, > x holds
Subqueries With ALL and ANY
Subqueries With ALL and ANY

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

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

• Note, loosely speaking: > ANY X means that for at least one x in X, > x holds
Subqueries With ALL and ANY
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

  ```sql
  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 \texttt{EXISTS}
- “\texttt{EXISTS R}” is true if and only if \( R \) is not empty
  - So read this: “there exists a tuple in \( R \)”
- “\texttt{NOT EXISTS R}” is true if and only if \( R \) is empty
  - So read this: “there does not exist a tuple in \( R \)”

- \textit{These are very important}, as they are frequently used to implement difference (\texttt{MINUS} or \texttt{EXCEPT}) and intersection (\texttt{INTERSECT})

- First, a little practice, then we’ll see how to do the set operations
Testing for Emptiness

Find all Cnames who do not have an entry in Invoice

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

- In general, use EXISTS and NOT EXISTS

- If the tables have only one column, you may see advice to use IN and NOT IN: don’t do that: problems with NULLs
**Set Intersection (INTERSECT)**

Use **EXISTS**

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

Note that a tuple containing nulls, (NULL,c), is not in the result, and it should not be there.
Set Intersection (INTERSECT) Can Also Be Done Using Cartesian Product

```
SELECT DISTINCT *
FROM R
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.
Accounting For NULLs
(Perhaps Semantically Incorrectly)

SELECT DISTINCT *
FROM R
WHERE EXISTS (SELECT *
FROM S
Accounting For NULLs
(Perhaps Semantically Incorrectly)

SELECT DISTINCT *
FROM R
WHERE NOT EXISTS (SELECT *
FROM S
**Set Intersection For Tables With One Column**

```
SELECT DISTINCT *  
FROM P  
WHERE A IN (SELECT A  
FROM Q);
```
Set Difference For Tables With One Column

```
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 I have warned you earlier)
Using More Than One Column Name

Assume we have $R(A,B,C)$ and $S(A,B,C,D)$

Some systems do not allow the following (more than one item = ANY)

\[
\text{SELECT A} \\
\text{FROM R} \\
\text{WHERE (B,C) = ANY} \\
(\text{SELECT B, C} \\
\text{FROM S});
\]

we can use

\[
\text{SELECT A} \\
\text{FROM R} \\
\text{WHERE EXISTS} \\
(\text{SELECT *} \\
\text{FROM S} \\
\text{WHERE R.B = S.B AND R.C = S.C});
\]
We want to compute the set of cities that have at least all the names that Chicago has.

<table>
<thead>
<tr>
<th>CnameInCcity</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>CnameInChicago</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doe</td>
<td></td>
</tr>
<tr>
<td>Yao</td>
<td></td>
</tr>
</tbody>
</table>
Computing Division Concisely

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

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

This is really the same as before
• I leave it to you to figure the details out, if you like
In Microsoft Access
SQL has a variety of “modified” Cartesian Products, called joins.

They are also very popular.

The most 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.
**LEFT OUTER JOIN**

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

- Includes all rows from the first table, matched or not, plus matching “pieces” from the second table, where applicable.

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

---

<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>
### 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>g</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>2</td>
<td>f</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.

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

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

A | B | C | D
---|---|---|---
a | 1 | 1 | e |
b | 2 | 2 | f |
b | 2 | 2 | g |
4 | h |
In Microsoft Access
**FULL OUTER JOIN**

```sql
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></td>
<td>1</td>
<td>e</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>g</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>h</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>4</td>
<td>h</td>
</tr>
</tbody>
</table>

Cannot be done in Microsoft Access directly
Can use Left Outer and Right Outer
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 'E') AND (Pcity <> 'E'));
```

• Note that we want all city values in the range B through DZZZZZ....; thus the value E is too big, as BETWEEN is inclusive.
In Microsoft Access

<table>
<thead>
<tr>
<th>P</th>
<th>Pcity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Boston</td>
</tr>
<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:

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

• % stands for 0 or more characters; _ stands for exactly one character (unfortunately SQL uses these and not the more commonly-used regular expression syntax).
Presenting the Result

It is possible to manipulate the resulting answer to a query. We present the general features by means of examples.

For each P list the profit in thousands, order by profits in decreasing order and for the same profit value, order by increasing P:

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

<table>
<thead>
<tr>
<th>Thousands</th>
<th>P</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</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>
Presenting the Result

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

```
SELECT DISTINCT Idate, C
FROM Invoice;
```
In Microsoft Access
Testing For Duplicates

It is possible to test whether a subquery returns any duplicate tuples, with NULLs ignored.

Find all Cnames that all of whose orders are for different amounts (including, of course those who have placed no orders).

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

UNIQUE is true if there are no duplicates in the answer, but there could be several tuples, as long as all are different.

If the subquery returns an empty table, UNIQUE is true.

Recall that we assumed that our original relations had no duplicates; that’s why the answer is correct.
Testing For Duplicates

- It is possible to test whether 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.
Modifying the Database

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

- INSERT INTO Plant (P, Pname, Pcity, Profit)
  VALUES ('909','Gamma',Null,52000);

- If it is clear which values go where (values listed in the same order as the columns), the names of the columns may be omitted

  INSERT INTO Plant
  VALUES ('909','Gamma',Null,52000);
In Microsoft Access
**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

<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>
**Insertion From A Table**

- Assume we have a table `Candidate(C,Cname,Ccity,Good)` listing potential customers
  - First, for each potential customer, the value of `Good` is Null
  - Later it becomes either Yes or No

- We can insert part of this “differential table” into customers:
  ```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></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>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>P</th>
<th>Ac</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Doe</td>
<td>Boston</td>
<td>901</td>
<td></td>
</tr>
<tr>
<td>2000</td>
<td>Yao</td>
<td>Boston</td>
<td>902</td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td>Doe</td>
<td>Chicago</td>
<td>903</td>
<td></td>
</tr>
<tr>
<td>4000</td>
<td>Doe</td>
<td>Seattle</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>Brown</td>
<td>Denver</td>
<td>903</td>
<td></td>
</tr>
<tr>
<td>6000</td>
<td>Smith</td>
<td>Seattle</td>
<td>907</td>
<td></td>
</tr>
<tr>
<td>7000</td>
<td>Yao</td>
<td>Chicago</td>
<td>904</td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td>Smith</td>
<td>Denver</td>
<td>904</td>
<td></td>
</tr>
<tr>
<td>9000</td>
<td>Smith</td>
<td>Boston</td>
<td>903</td>
<td></td>
</tr>
<tr>
<td>9001</td>
<td>Qin</td>
<td>Boston</td>
<td>901</td>
<td></td>
</tr>
</tbody>
</table>
Deletion

DELETE
FROM Candidate
WHERE Good = 'Yes';

This removes rows satisfying the specified condition
• In our example, once some candidates were promoted to customers, they are removed from Candidate
### In Microsoft Access

#### Customer 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>

#### DeleteFromCandidate Table

<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>
Deletion

DELETE
FROM Candidate;

This removes all the rows of a table, leaving an empty table; but the table remains

Every row satisfied the empty condition, which is equivalent to: “WHERE TRUE”
In Microsoft Access

- DeleteFromCandidate

- DeleteAllFromCandidate
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);
  ```
- Duplicates are not removed
  - Of course no copy of a tuple that appears in both $R$ and $S$ remains in $R$
  - But if a tuple appears several times in $R$ and does not appear in $S$, all these copies remain in $R$
Every tuple that satisfied the WHERE condition is changed in the specified manner (which could in general be quite complex)
In Microsoft Access

![Database Table Screenshot]

© 2014 Zvi M. Kedem
Update

But this gets quite “strange,” and incorrect if the same tuple could be updated in different ways if it satisfies a different condition; the system will reject this.

Example

A student can have only one major (we will see how to specify this later) and we tell the database to change each student major to X, if the student took a course in department X.

If students can take courses in several departments, the above cannot work.
Recursion

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

WITH Lineage(ancestor, descendant) AS
  (
    SELECT Parent, Child
    FROM Birth
    UNION ALL
    SELECT Parent, Descendant
    FROM Lineage
    WHERE Child = Ancestor
  )
Triggers

These are actions that can be taken before/after/instead INSERT, UPDATE, or DELETE

Triggers are both complex and powerful, we just touch briefly on them here

We will discuss:
  • AFTER (next)
  • INSTEAD (later)

Assume that after a new Customer is inserted into the database, if Cname is Xiu, the system will “automatically” CREATE a new plant in the city Xiu lives, with “properties related to Xiu,” which we will understand by looking at the example

Let us look at (I tested this in Oracle)
  • The exact trigger in Oracle
  • A partial trace of the execution in Oracle
Defining A Trigger

CREATE TRIGGER Trigger01
AFTER INSERT ON Customer
REFERENCING NEW AS newcustomer
FOR EACH ROW
WHEN (newcustomer.Cname = 'Xiu')
BEGIN
INSERT INTO Plant VALUES (:newcustomer.C, 'Xiu_Plant', :newcustomer.Ccity, NULL);
END Trigger01;
.
RUN;

This is the exact Oracle syntax
NEW refers to added rows
If rows were deleted (not in our example!), we could refer to them as OLD
## Our Database

### Customer and Plant before Insert

<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
## Our Database

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

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

This will print

Employee with name "Yiling" has ID 0000000003
Scenario

- You go to an ATM to withdraw some money
- You swipe your card, something (a program, not a relational database) reads it
- You punch in your PIN, a program reads it
- The program talks to a relational database to see if things match, assume that they do
- You ask for a balance, a program reads what you punched and formulates a query to a relational database and understands the answer and shows you on the screen
- You want to withdraw money, a program formulates a request to the relational database to update your account
- ...
SQL Embedded in A Host Language

- Sometimes, we need to interact with the database from programs written in another host language.
- The advantage of this is that we are able to use the structure of the database, its layers, indices, etc.
- The disadvantage is, the host language does not understand the concepts of relations, tuples, etc.
- We use a version of SQL, called Embedded SQL, for such interactions.
- We concentrate on static embedded SQL.
- We just sketch this very briefly.
SQL Commands As Procedure Calls

SQL commands in host languages could at a gross level be considered procedure calls.

ANSI standard specified Embedded SQL for some programming languages only.

There are two main types of operations:
- Those working on a tuple
- Those working on a relation
**Common Variables**

- Variables in the host language that are used to communicate with the SQL module must be declared as such.

- Assuming we want to act on the relation plants, we would write in our host program something similar to:
  
  ```sql
  EXEC SQL BEGIN DECLARE SECTION;
  VAR
  Plant: INTEGER;
  Plantname: ...;
  Plantcity: ...;
  Plantprofit: ...;
  EXEC SQL END DECLARE SECTION;
  ```
A Fragment of a Host Program

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

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

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

We could also write

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

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

- Sometimes the value inserted or retrieved will be NULL
- However host language does not know how the database is coding NULLs.
- It is possible to use special indicator variables to indicate that the value is actually NULL
  - `EXEC SQL SELECT profit INTO :Plantprofit INDICATOR :Ind WHERE C = 75;`
- Here if host language variable Ind is negative, it means that Plantprofit does not contain an actual value, but NULL was returned by the SQL system
As part of the declaration section, a variable, generally referred to as **SQLCODE**, must be declared. It is set by SQL to indicate whether the operation was successful, and if not, what kind of problems may have occurred.
Handling Sets Of Tuples (Relations)

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

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

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

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

EXEC SQL OPEN CURSOR Todo;

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

EXEC SQL CLOSE CURSOR Todo;
```
Previous 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
Key Ideas

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

- Insertion
- Deletion
- Update
- Recursion
- Triggers
- PL/SQL
- Interface with other languages
Optional Material
Asking About Some

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

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

- The standard SELECT … FROM … WHERE … easily expresses the existential quantifier above
- p “is good” if it has taken at least 1
 Asking About All

- List all Persons, who Took \textit{at least all} the Courses that are Required

- The result can be expressed using a logical \textit{formula} with a universal quantifier:

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

- 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

\[
\neg \neg \alpha \equiv \alpha
\]

\[
\alpha \implies \beta \equiv \neg \alpha \lor \beta
\]

\[
\neg (\alpha \lor \beta) \equiv \neg \alpha \land \neg \beta
\]

\[
\forall x [ A(x) ] \equiv \neg \exists x [ \neg A(x) ]
\]

- Our \textit{formula} is equivalent to

\[
\{ p \mid \neg \exists c [ c \in r \land (p, c) \notin t ] \}
\]
Reiteration: Differences Between SQL And “Pure” Relational Algebra

- This is for those who want to have concise description
- This part of the Unit is optional
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\textsuperscript{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

Many redundant operators (relational algebra had only one: intersection)

SQL provides statistical operators, such as AVG (average)

- Can be performed on subsets of rows; e.g. average salary per company branch
Key Differences
Between Relational Algebra And SQL

☐ Every domain is “enhanced” with a special element: NULL
  • Very strange semantics for handling these elements

☐ “Pretty printing” of output: sorting, and similar

☐ Operations for
  • Inserting
  • Deleting
  • Changing/updating (sometimes not easily reducible to deleting and inserting)
## Basic Syntax Comparison

<table>
<thead>
<tr>
<th>Relational Algebra</th>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\pi_{a, b}$</td>
<td>SELECT $a, b$</td>
</tr>
<tr>
<td>$\sigma_{(d &gt; e)}$</td>
<td>WHERE $d &gt; e$ AND $f = g$</td>
</tr>
<tr>
<td>$p \times q$</td>
<td>FROM $p, q$</td>
</tr>
</tbody>
</table>
| $\pi_{a, b}$      | SELECT $a, b$  
|                   | FROM $p, q$  
|                   | WHERE $d > e$ AND $f = g$ |
| Renaming using some syntax | AS {or blank space} |
| $p := \text{result}$ | INSERT INTO $p$  
|                   | result {assuming $p$ was empty} |
| $\pi_{a, b}$ attributes) | SELECT *  
|                   | FROM $p$; |