The SQL Language

Elisa Bertino
CS Department and CERIAS
Purdue University

SQL is a language for data definition and manipulation, initially developed at the IBM research laboratory in California. It became a standard in 1986 (SQL-1986). The last developed standard is SQL:1999 (known also as SQL3), which includes object-relational features. The previous standard is SQL-92 (known also as SQL2).

The SQL Language

It is the language commonly adopted by commercial DBMS, including:
- SQL/DS and DB2 (IBM)
- Oracle
- SQL Server (Microsoft)
- Informix
- Sybase
- CA-Ingres (Computer Associates)

SQL – Data types

SQL:1999 has the following categories of data types:
- Pre-defined types
  - Structured types
  - User-defined types
- In this part of the course, we will see the pre-defined types
  - The structured and user-defined types will be discussed when talking about the object-relational data model
- The pre-defined types are divided in five categories
  - Numeric types
  - Binary types
  - String types
  - Temporal types
  - Boolean types
**SQL - Numeric data types**

- **Exact numeric types**
  - Represent integer values or decimal values with a fixed number of digits for the fractional part (ex. 75, -6.2)

- **Approximated numeric types**
  - Are numeric values represented according to the floating point representation (ex. 1256E-4)

**SQL – Numeric Data Types**

Exact numeric types:

- **INTEGER** represents integer values
  - The precision (total number of digits) is expressed in number of bits or digits, depending on the specific implementation of SQL

- **SMALLINT** represents integer values
  - The values of this type are used for optimization in that they require less storage space
  - The only constraint (for systems supporting both the INTEGER and SMALLINT data types) is that the precision of this data type be not greater than the precision of the INTEGER data type

**SQL - Numeric Data Types**

- **NUMERIC** represents decimal values (stored according to a non-floating-point representation)
  - It is characterized by a precision (total number of digits) and a scale (number of digits reserved for the fractional part)
  - The specification of this data type has the form
    - `NUMERIC [(precision, [scale])]`
  - For example, `NUMERIC (4,1)` corresponds to values in the range [-999.9, 999.9]
  - Both precision and scale can be omitted; the default is 1 for the precision and 0 for the scale

**SQL - Numeric Data Types**

- **DECIMAL** also represents decimal values
  - It is similar to the NUMERIC type
  - The specification of this data type has the form
    - `DECIMAL [(precision, [scale])]`
  - The difference between NUMERIC and DECIMAL (for systems supporting both) is that the first data type must be implemented exactly with the required precision; by contrast the second data types can be implemented with a higher precision
SQL - Numeric Data Types

- REAL represents floating point numbers
  - The precision depends on the specific implementation of SQL
- DOUBLE PRECISION represents floating numbers with double precision
  - The precision depends on the specific implementation of SQL
- FLOAT allows one to specify the required precision

SQL - Binary Data Types

- BIT represents bitstrings of predefined maximum length
  - The specification has the format BIT ([length])
  - The default is 1
- BIT VARYING represents bitstrings of predefined maximum length
  - The specification has the format BIT VARYING ([length])

The difference is that for the BIT a fixed storage allocation technique is used, whereas for BIT VARYING different strategies can be adopted.

SQL – String Data Types

- CHARACTER represents character strings of maximum predefined length
  - It is often shortened as CHAR
  - Its specification has the format CHAR ([length]) where length is the maximum length for strings
  - The default is 1
- CHARACTER VARYING represents variable length strings of maximum predefined length
  - It is often shortened as VARCHAR
  - Its specification has the format VARCHAR ([length])

The difference between these two types is that for the type CHAR a space of size equal to ‘length’ is always allocated (thus increasing storage overhead when there are shorter strings), whereas different implementation strategies are adopted for the VARCHAR type.
SQL - String Data Types

- **CHARACTER LARGE OBJECT (CLOB)**
  - It allows one to store character strings of very large dimensions; it is useful for storing text data and therefore for supporting MM
  - It is possible to associate with the type of data CHAR, a reference CHARACTER SET and the related COLLATION (ordering of characters in the set)
  - For each of the string data types, there exists the version NATIONAL

SQL – Temporal Data Types

- **DATE** represents dates expressed as year (4 digits), month (2 digits between 1 and 12), day (2 digits between 1 and 31 and subject to further restrictions depending on the month)
  - Values of this type have the format `DATE 'yyyy-mm-dd'`

- **TIME** represents times expressed as hour (2 digits), minute (2 digits) and second (2 digits)
  - Values of this type have the format `TIME 'hh:mm:ss[.nnnnnn]'

- **TIMESTAMP** represents the “concatenation” of the previous two data types
  - It represents timestamps that consist of: year, month, day, hour, minute, second and microsecond
  - The values of this type have the format `TIMESTAMP 'yyyy-mm-dd hh:mm:ss[.nnnnnn]]'

- **INTERVAL** represents a temporal interval; temporal intervals are specified with reference to temporal qualifiers:
  - YEAR, MONTH, DAY, HOUR, MINUTE, SECOND
  - Examples: INTERVAL '3' YEAR
  - Example: INTERVAL '5' MINUTE

SQL - Boolean Data Types

- **BOOLEAN** represents Boolean values
  - Such values are TRUE, FALSE, UNKNOWN
  - The UNKNOWN value has been introduced for the management of null values
  - Later we will see how the Boolean operators AND, OR, NOT are to be extended to deal with the UNKNOWN value
### SQL – Data Types

**Example from DB2**

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMALLINT</td>
<td>Integers 16 bits</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Integers 32 bits</td>
</tr>
<tr>
<td>DECIMAL (p,s)</td>
<td>Decimal numbers with precision p and scale s</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>Floating numbers 64 bits FLOAT and DOUBLE</td>
</tr>
<tr>
<td>CHAR(n)</td>
<td>Character strings of length n, with n ≤ 254, and default = 1</td>
</tr>
<tr>
<td>VARCHAR(n)</td>
<td>Character strings of variable length, with maximal equal to n, where n ≤ 4000</td>
</tr>
<tr>
<td>DATE</td>
<td>year, month, day</td>
</tr>
<tr>
<td>TIME</td>
<td>hour, minute, second</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>year, month, day, hour, minute, second</td>
</tr>
</tbody>
</table>

### SQL - DDL: Table creation

- Tables are created through the command `CREATE TABLE`
- Syntax:
  ```sql
  CREATE TABLE TableName (spec_col1 [, ..., spec_col_n])
  ```
  where:
  - `TableName` is the name of the relation being created
  - `spec_col_i` (i=1, ..., n) is a column specification clause; it has the following format:
    ```sql
    ColumnName Domaini [DEFAULT DefaultValue]
    ```

### SQL - Domains

- It is possible to define domains and use them in table definitions
- A domain is a different name for a data type and in addition may include other information (namely, default values and constraints on the values)
- Syntax:
  ```sql
  CREATE DOMAIN DomainName AS TypeDescription
  [DEFAULT DefaultValue];
  ```
**SQL - Domains**

- Example
  ```sql
  CREATE DOMAIN JobDomain AS CHAR(10) DEFAULT 'employee';
  ```
- CREATE TABLE Employees
  ```sql
  (Emp# Decimal(4),
  Name Char(20),
  Job JobDomain,
  HiringD Date,
  Salary Decimal(7,2),
  Bonus Decimal(7,2) DEFAULT 0,
  Dept# Decimal(2));
  ```

**SQL – Integrity Constraints**

- A constraint is a rule that specifies conditions on the values taken by the tuples in a table.
- We can associate a constraint with: a table, an attribute, a domain.
- SQL supports a variety of constraint types:
  - Keys (UNIQUE and PRIMARY KEY)
  - Mandatory values (NOT NULL)
  - Foreign keys (FOREIGN KEY)
  - CHECK constraints (on attributes or tuple) and assertions (constraints CHECK on several tuples or tables)

- It is in addition possible to specify whether a given constraint has to be verified as soon as an operation is executed which could violate the constraint (NON DEFERRABLE) or at the end of the transaction (DEFERRABLE)
- The constraints cannot contain conditions the evaluation of which may give different results depending on when the constraints are evaluated (for example reference to the system time)

**SQL - Keys**

- Keys are specified through the keywords UNIQUE or PRIMARY KEY
  - UNIQUE assures that there are no two tuples (in a given relation) that have the same not null values for the attributes for which the UNIQUE constraint is specified (note: the UNIQUE attributes may contain null values)
  - PRIMARY KEY imposes that for each tuple the attributes that are part of the primary key have not null values and that no two tuples exist with the same values for all these attributes
- A table may several attributes (or sets of attributes) that UNIQUE but only one attribute (or set attributes) which is a PRIMARY KEY
If the key consists of a single attribute, we can just include the UNIQUE or PRIMARY KEY keyword in the column specification.

otherwise the special clauses

PRIMAR KEY (ColumnNameList)

or

UNIQUE(ColumnNameList)

can be used as part of the CREATE TABLE Command.

Example:

CREATE TABLE Employees

(Emp# Decimal(4) PRIMARY KEY,
Name Char(20),
Job DomainJob,
HiringD Date,
Salary Decimal(7,2),
Bonus Decimal(7,2) DEFAULT 0,
Dept# Decimal(2));

Example:

CREATE TABLE Employees

(Emp# Decimal(4) PRIMARY KEY,
Name Char(20),
Job DomainJob,
HiringD Date,
Salary Decimal(7,2),
Bonus Decimal(7,2) DEFAULT 0,
Dept# Decimal(2));

Example:

CREATE TABLE Movies

(Title Char(20),
Year Integer,
Studio Char(20),
Color Boolean,
PRIMARY KEY (Title,Year));
### SQL - NOT NULL

- To specify that an attribute cannot have null value (and thus it is mandatory that the attribute has a value different from null for all tuples) the NOT NULL keyword must be included in the column.
- Columns that are declared PRIMARY KEY cannot have null values; for these columns there is no need to specify the NOT NULL keyword.

#### Example:
```
CREATE TABLE Employee
(Emp# Decimal(4) PRIMARY KEY,
SSN Char(9) UNIQUE,
Name Char(20) NOT NULL,
Job DomainJob,
HiringD Date,
Salary Decimal(7,2) NOT NULL,
Bonus Decimal(7,2) DEFAULT 0,
Dept# Decimal(2) NOT NULL);
```

### SQL - Foreign Keys

- Foreign keys are specified through the FOREIGN KEY clause which also part of the CREATE TABLE command.
- Syntax:
  ```
  FOREIGN KEY (ColumnNameList)
  REFERENCES ReferencedTable name
  [MATCH {FULL | PARTIAL | SIMPLE}]
  [ON DELETE { NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT}]
  [ON UPDATE { NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT}]
  [, FOREIGN KEY.....]);
  ```
- ColumnNameList is a list of attributes that corresponds to one key (UNIQUE or primary) of the reference table.
- It is not necessary that the attribute names be the same, but the domains must be compatible.
- It is possible (mandatory in case of ambiguities) to explicitly indicate which are the referenced columns.
- If the foreign key consists of a single attribute, the foreign key specification can be included in the column definition by using the clause REFERENCES ReferencedTable name.
The type of match is meaningful in the case of foreign keys consisting of more than one attribute and in presence of null values.

- MATCH SIMPLE: the referential integrity constraint is satisfied if for each tuple of the referencing table the following condition is verified:
  - At least one of the columns of the foreign key is NULL, or
  - None of such columns is NULL and a tuple exists in the referenced table the key of which is equal to the value(s) of such columns.

This is the default.

MATCH FULL: the referential integrity constraint is satisfied if for each tuple of the referencing table the following condition is verified:

- All columns of the foreign key are NULL, or
- None of such columns is NULL and a tuple exists in the referenced table the key of which is equal to the value(s) of such columns.

MATCH PARTIAL: the referential integrity constraint is satisfied if for each tuple of the referencing table the values of the non null columns of the foreign key correspond to the values of the key of a tuple in the referenced table.

Example:
- Referenced table includes tuples having the following keys:
  - 10, 'mary'
  - 20, 'joseph'
- In the case of MATCH SIMPLE:
  - The following values for foreign keys are correct:
    - 10, 'mary'
    - NULL, 'joseph'
    - 10, NULL
    - NULL, 'luke'
    - 30, NULL
  - Wrong values:
    - 10, 'joseph'
    - 10, 'luke'

Example (con't):
- In the case of MATCH FULL:
  - The following values for foreign keys are correct:
    - 10, 'mary'
    - NULL, NULL
  - The following are wrong values:
    - 10, NULL
    - NULL, 'joseph'
    - NULL, 'luke'
    - 30, NULL
    - 10, 'joseph'
    - 10, 'luke'
**SQL - Foreign Keys**

**Match types**

- Example (con’t)
  - In the case of MATCH PARTIAL
    - The following values for foreign keys are correct
      
      - 10, ‘mary’
      - NULL, NULL
      - 10, ‘joseph’
  
    - The following values are incorrect
      
      - NULL, ‘luke’
      - 30, NULL
      - 10, ‘joseph’
      - 10, ‘luke’

**Options concerning the actions to be executed when a referenced tuple is deleted (ON DELETE):**

- NO ACTION: the deletion of a tuple in the referenced table is executed only if no tuple exists in the referencing relation that has as foreign key the key of the tuple to be deleted
- RESTRICT: the semantics is the same as the NO ACTION, with the difference that this option is applied right away, whereas NO ACTION is applied after the SQL statement actions have finished, including application of any other referential action and any trigger executions.
  
  - In other words, the RESTRICT option does not allow temporary constraint violations, whereas the NO ACTION does

- CASCADE: the deletion of a tuple from the referenced table implies the deletion of all the tuples in the referencing table that have as foreign key the same key of the tuple to be deleted

- SET NULL: the deletion of a tuple from the referenced table implies that in all tuples of the referencing table the foreign key, the values of which are the same of the key of the tuple to be deleted, are set to NULL (if admitted)

- SET DEFAULT: the deletion of a tuple from the referenced table implies that in all tuples of the referencing table the foreign key, the values of which are the same of the key of the tuple to be deleted, are set to the default value

**Options concerning the actions to be executed when a referenced tuple is deleted (ON UPDATE):**

- They have the same meaning of the deletion options
- difference: the CASCADE option has the effect of assigning to the foreign keys the new value of the key of the referenced tuple

- default: NO ACTION both for deletion and update
In case of several references, the order according to which the various options are considered is as follows:
- RESTRICT
- CASCADE, SET NULL, SET DEFAULT
- NO ACTION

In the case of insertion or update to the referencing table it is not possible to specify any option and it is always applied the NO ACTION option.

Example

```sql
CREATE TABLE Professors
(P# Char(7) NOT NULL,
 Pname Varchar(20) NOT NULL,
 Address Varchar(15) NOT NULL,
 PRIMARY KEY (P#));

CREATE TABLE Students
(S# Char(6) NOT NULL,
 Sname Varchar(20) NOT NULL,
 Address Varchar(15),
 Birthdate Date NOT NULL,
 PRIMARY KEY (S#));

CREATE TABLE Advisor
(P# Char(7) NOT NULL,
 S# Char(6) NOT NULL,
 PRIMARY KEY (S#),
 FOREIGN KEY (P#) REFERENCES Professors
 ON DELETE RESTRICT
 ON UPDATE CASCADE,
 FOREIGN KEY (S#) REFERENCES Students
 ON DELETE CASCADE,
 ON UPDATE CASCADE);
```

Example (con't)

```sql
CREATE TABLE Employees
(Emp# Decimal(4) PRIMARY KEY,
 Name Char(20),
 Job DomainJob,
 HiringD Date,
 Salary Decimal(7,2),
 Bonus Decimal(7,2) DEFAULT 0,
 Dept# Decimal(2) REFERENCES Departments);

CREATE TABLE Departments
(Dept# Decimal(2) PRIMARY KEY,
 DeptName Char(20),
 Office# Decimal(4),
 Division Char(2),
 Manager Decimal(4) REFERENCES Employees);
```
**SQL – CHECK constraints on attributes**

- The column specification is extended with the CHECK clause; such clause specifies a condition, that is a predicate or a Boolean combination of predicates (component WHERE of an SQL query)
- Example
  
  ```
  Job Char(10) CHECK (Job IN ('manager', 'engineer', 'technician', 'secretary'))
  ```
  
  such condition may include subqueries that may refer other tables; however the constraint is checked only upon modifications to the values of the column with which the constraint is associated

**SQL – CHECK constraints on tuples**

- The specification of a table is extended with the CHECK clause; such clause specifies a condition, that is a predicate or a Boolean combination of predicates (component WHERE of an SQL query)
- Example
  
  ```
  CHECK (Salary > Bonus)
  ```
  
  such condition may include subqueries that may refer other tables; however the constraint is checked only upon modifications to the values of the columns referenced in the constraint

**SQL – Assertions**

- Assertions are elements of the schema; they are used to specify constraints that involve several tables
- Syntax:
  
  ```
  CREATE ASSERTION Name
  CHECK (Condition)
  ```
  
  The condition is a predicate or a Boolean combination of predicates (component WHERE of an SQL query)
SQL – Assertions

- Example
  
  ```sql
  CREATE ASSERTION ASingleManagerPerDepartment 
  CHECK (NOT EXISTS (SELECT * FROM Employees 
      WHERE Job = 'manager' 
      GROUP BY Dept# 
      HAVING COUNT(*) > 1))
  ```

- Not all DBMS support all those constraints; in particular assertions are not generally supported.

- Most DBMS only support constraints the evaluation of which can be executed by accessing a single tuple.

- Motivation: evaluation efficiency.

SQL – Constraints

- It is possible to assign a name to a constraint by using the CONSTRAINT clause; such clause contains a name and the CHECK clause.

- Examples:
  
  - `Emp# Char(6) CONSTRAINT KeyEmp PRIMARY KEY`
  
  - `CONSTRAINT SalaryOk CHECK (Salary > Bonus)`

- The specification of names for constraints is useful in order to reference the constraints (for example for deleting them).

SQL - Constraints

- Example

  ```sql
  CREATE TABLE Exams 
  (S# Char(6) REFERENCES Students 
    ON DELETE CASCADE, 
  C# Char(6) REFERENCES Courses, 
  Mark Integer NOT NULL, 
  PRIMARY KEY (S#,C#), 
  CONSTRAINT Mark-constr CHECK Mark > 18 AND Mark < 30); 
  ```

- Constraint evaluation:

  1. A constraint is violated by a tuple if the condition evaluated on the tuple is False.
  2. A constraint the evaluation of which is True or Unknown (because of null values) is not violated.
  3. When a set of tuples are inserted or modified, the violation of a constraint for a single tuple in the set results in the abort of the execution of the entire statement, that is, no tuple in the set is inserted or modified; the transaction however continues its execution.
## SQL - DDL
### Deletions and modifications

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DROP TABLE R</strong></td>
<td>Drop relation R</td>
<td>DROP TABLE Employees;</td>
</tr>
<tr>
<td><strong>RENAME R&lt;sub&gt;a&lt;/sub&gt; TO R&lt;sub&gt;b&lt;/sub&gt;</strong></td>
<td>Changes the name of a relation from R&lt;sub&gt;a&lt;/sub&gt; into R&lt;sub&gt;b&lt;/sub&gt;</td>
<td>RENAME Employees TO Emp;</td>
</tr>
</tbody>
</table>

### Deletions and modifications

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALTER TABLE R ADD COLUMN spec_col</strong></td>
<td>It adds a new column to a relation</td>
<td>ALTER TABLE Employees ADD COLUMN (Proj# Decimal(3));</td>
</tr>
<tr>
<td><strong>ALTER TABLE R ALTER COLUMN spec_col</strong></td>
<td>It modifies a column of a relation</td>
<td>ALTER TABLE Employees ALTER COLUMN (Proj# Decimal(4));</td>
</tr>
<tr>
<td><strong>ALTER TABLE R DROP COLUMN nome_col</strong></td>
<td>It drops a column from a relation</td>
<td>ALTER TABLE Employees DROP COLUMN Bonus;</td>
</tr>
</tbody>
</table>

### Deletions and modifications

<table>
<thead>
<tr>
<th>Action</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ALTER DOMAIN D SET DEFAULT default_value</strong></td>
<td>It modifies the default value of a domain</td>
<td></td>
</tr>
<tr>
<td>**DROP DOMAIN D (RESTRICT</td>
<td>CASCADE)**</td>
<td>It removes a domain</td>
</tr>
<tr>
<td></td>
<td><strong>RESTRICT:</strong> the domain is removed only if there are no relations using it</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>CASCADE:</strong> in each relation using the domain, the name of the domain is replaced by its definition (notice that such change does not impact the data stored in the relation)</td>
<td></td>
</tr>
</tbody>
</table>
SQL - DDL

Deletions and modifications

- ALTER TABLE R DROP CONSTRAINT C
- ALTER DOMAIN D DROP CONSTRAINT C
  - It removes the constraint C from relation R or from domain D
- ALTER TABLE R ADD CONSTRAINT C ...
- ALTER DOMAIN D ADD CONSTRAINT C ...
  - It adds the constraint C to relation R or to domain D
- DROP ASSERTION A
  - It removes assertion A

SQL - Queries

The basic format of an SQL query is:

```
SELECT R1.C1, R2.C2, ...., Rn.Cn
FROM  R1, R2, ...., Rk
WHERE F;
```

- R1, R2, ...., Rk is a list of distinct relation names
- Ri1.C1, Ri2.C2, ...., Rin.Cn is a list of column names
- The notation R.C denotes the column of name C of relation R
- If the FROM clause lists only one relation or there is only one relation with a column with name C, then we can use C instead of R.C
- F is a predicate similar to the predicates seen for the relational operation σ

Examples from the employee database

- Q1: select all employees having a salary greater than 2000
  
  \[\sigma_{\text{Salary}>2000}(\text{Employees})\]
  
  SELECT * FROM Employees
  WHERE Salary >2000;

  - The * symbol in the projection clause of the query denotes that all the columns must be retrieved
SQL - Queries

Results of Q1:

<table>
<thead>
<tr>
<th>Emp#</th>
<th>Name</th>
<th>Job</th>
<th>HireD</th>
<th>Salary</th>
<th>Bonus</th>
<th>Dept#</th>
</tr>
</thead>
<tbody>
<tr>
<td>7566</td>
<td>Pink</td>
<td>manager</td>
<td>02/04/81</td>
<td>2975.00</td>
<td>?</td>
<td>20</td>
</tr>
<tr>
<td>7698</td>
<td>Black</td>
<td>manager</td>
<td>01/05/81</td>
<td>2850.00</td>
<td>?</td>
<td>30</td>
</tr>
<tr>
<td>7782</td>
<td>Neri</td>
<td>engineer</td>
<td>01/06/81</td>
<td>2450.00</td>
<td>200.00</td>
<td>10</td>
</tr>
<tr>
<td>7839</td>
<td>Dare</td>
<td>engineer</td>
<td>17/11/81</td>
<td>2600.00</td>
<td>300.00</td>
<td>10</td>
</tr>
<tr>
<td>7977</td>
<td>Green</td>
<td>manager</td>
<td>10/12/80</td>
<td>3000.00</td>
<td>?</td>
<td>10</td>
</tr>
</tbody>
</table>

Examples from the employee database

- Q2: retrieve the name and the department number of the employees that are engineers and have a salary greater than 2000

\[ \pi_{Name, Dept\#}(\sigma_{Salary>2000 \land Job='Engineer'}(Employees)) \]

```
SELECT Name, Dept# FROM Employees
WHERE Salary>2000 AND Job = 'engineer';
```

- Results of Q2:

<table>
<thead>
<tr>
<th>Name</th>
<th>Dept#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neri</td>
<td>10</td>
</tr>
<tr>
<td>Dare</td>
<td>10</td>
</tr>
</tbody>
</table>

- Q3: retrieve the number of the employees that work at department 30 and are engineers or technician

\[ \pi_{Emp\#}(\sigma_{Dept\#=30 \land (Job='engineer' \lor Job='technician')}(Employees)) \]

```
SELECT Emp\# FROM Employees
WHERE Dept#=30 AND (Job = 'engineer' OR Job = 'technician');
```

- Results of Q3:

<table>
<thead>
<tr>
<th>Emp#</th>
</tr>
</thead>
<tbody>
<tr>
<td>7499</td>
</tr>
<tr>
<td>7521</td>
</tr>
<tr>
<td>7844</td>
</tr>
<tr>
<td>7900</td>
</tr>
</tbody>
</table>

Conditions on value ranges

- The operator BETWEEN determines the tuples that for a given attribute have a value in a given range

format:

\[ C \text{ BETWEEN } v_1 \text{ AND } v_2 \]

- Negated form

\[ C \text{ NOT BETWEEN } v_1 \text{ AND } v_2 \]
SQL - Queries

Conditions on value ranges

- **example**

  ```sql
  SELECT Name, Salary FROM Employees WHERE Salary BETWEEN 1100 AND 1400;
  ```

- **results**

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adams</td>
<td>1100.00</td>
</tr>
<tr>
<td>Mills</td>
<td>1300.00</td>
</tr>
</tbody>
</table>

SQL - Queries

Search of a value in a set

- **example**

  ```sql
  SELECT * FROM Departiments WHERE Dept# IN (10,30);
  ```

- **results**

<table>
<thead>
<tr>
<th>Dept#</th>
<th>DeptName</th>
<th>Office</th>
<th>Division</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Building Construction</td>
<td>1100</td>
<td>D1</td>
<td>7977</td>
</tr>
<tr>
<td>30</td>
<td>Road Maintenance</td>
<td>5100</td>
<td>D2</td>
<td>7698</td>
</tr>
</tbody>
</table>

SQL - Queries

Comparisons among character strings

- **example**

  ```sql
  SELECT * FROM Departiments WHERE DeptName LIKE 'B%';
  ```

- **results**

<table>
<thead>
<tr>
<th>DeptName</th>
<th>Office</th>
<th>Division</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Construction</td>
<td>1100</td>
<td>D1</td>
<td>7977</td>
</tr>
</tbody>
</table>

- **example**

  ```sql
  SELECT * FROM Departiments WHERE DeptName LIKE '_%';
  ```

- **results**

<table>
<thead>
<tr>
<th>DeptName</th>
<th>Office</th>
<th>Division</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Road Maintenance</td>
<td>5100</td>
<td>D2</td>
<td>7698</td>
</tr>
</tbody>
</table>

SQL - Queries

Search of a value in a set

- **example**

  ```sql
  SELECT * FROM Departiments WHERE Dept# IN (SELECT Dept# FROM Departiments WHERE Dept# = 10);
  ```

- **results**

<table>
<thead>
<tr>
<th>DeptName</th>
<th>Office</th>
<th>Division</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Construction</td>
<td>1100</td>
<td>D1</td>
<td>7977</td>
</tr>
</tbody>
</table>

SQL - Queries

Comparisons among character strings

- **example**

  ```sql
  SELECT * FROM Departiments WHERE DeptName LIKE '%a%';
  ```

- **results**

<table>
<thead>
<tr>
<th>DeptName</th>
<th>Office</th>
<th>Division</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Road Maintenance</td>
<td>5100</td>
<td>D2</td>
<td>7698</td>
</tr>
</tbody>
</table>
Comparisons among character strings

- Example: determine all employees that have 'R' as third character in their names
  
  ```sql
  SELECT Name FROM Employees
  WHERE Name LIKE '__R%';
  ```

  Results:
  - Name
  - Martin
  - Neri
  - Dare
  - Turni
  - Ford

The operator SIMILAR TO allows one to specify more complex matching conditions but it is very inefficient.

- For this reason, it is more frequently used in the specification of CHECK constraints than in queries.
- A comparison predicate expressed with the SIMILAR TO has the following format:

  ```sql
  C [NOT] SIMILAR TO pattern
  ```

  where `pattern` is a character string that may contain, in addition to characters `%` and `_`, various operators.

Operators:

- `*`, repetition from 0 to n
- `+`, repetition from 1 to n
- `[CharacterSet]`, a character from the set `CharacterSet`
- `[Character1-Character2]`, a character in the range
- `[^SetorRange]`, negation
- `[:ALPHA:], [:UPPER:], [:LOWER:], [:DIGIT:], [:ALNUM:]`, alphabetic characters, uppercase, lowercase, numeric, alphanumeric
- `|`, or
- `||`, concatenation

Example: `[A-C][D-G][H-J][K-M][N-P][Q-S][T-V][W-Z][0-9][A-Z][a-z]`

Comparison among temporal intervals

- The predicate OVERLAPS returns the Boolean value true if two temporal intervals have a non-empty intersection.
- This predicate applies to pairs of temporal intervals expressed as (beginning,end) or to pairs expressed as (beginning,temporal-length).

Examples:

- (DATE '1994-01-01', INTERVAL '05' MONTH) OVERLAPS (DATE '1993-07-01', INTERVAL '08' MONTH)
Ordering query results
- In the examples seen so far, the ordering of the tuples results of a query is determined by the system (essentially depends from the strategy used to execute the query)

- It possible to specify a different ordering by adding to the query the ORDER BY clause

- example: retrieve salary, job and name of the employees of department #30, and order the result according to the values of attributes salary

```
SELECT Salary, Job, Name
FROM Employees
WHERE Dept# = 30
ORDER BY Salary;
```

```
results
Salary Job Name
800,00 technician Andrews
800,00 technician White
800,00 secretary Martin
1500,00 technician Turni
1950,00 engineer Gianni
2850,00 manager Black
```
Duplicate elimination

- Suppose that we wish to retrieve the list of all jobs appearing in the Employees relation.
- `SELECT Job FROM Employees;`

.results

- engineer
- technician
- manager
- secretary
- engineer
- technician
- manager
- secretary
- engineer
- manager

Duplicate elimination

- It is possible to require the duplicate elimination by using the `DISTINCT` clause.
- `SELECT DISTINCT Job FROM Employees;`

/results

- engineer
- technician
- manager
- secretary

Join

- The join operation is an important operation in that allows one to correlate data stored by different relations.
- In SQL the join can be expressed as a Cartesian product to which one or more join predicates are applied.
- A join predicates expresses a relationship that must be verified by the tuples result of the query.
- SQL also provides an explicit join operation.

/example: retrieve the name of the department where employee Red works
- `SELECT DeptName
FROM Employees, Departments
WHERE Name = 'Red' AND
Employees.Dept# = Departments.Dept#;`
SQL - Queries

Join

- SQL-1999 also provides explicit join operators
- Such operators produce relations and therefore they can be used in the FROM clause of queries
- The simplest of such operators is the CROSS JOIN corresponding to the Cartesian product
  - ex. Employees CROSS JOIN Departments
- The theta-join is expressed by the JOIN ON operator
  - ex. Employees JOIN Departments ON Dept# > Emp#

Example of the various syntactical expressions

- SELECT DeptName
  FROM Employees JOIN Departments
  ON Employees.Dept# = Departments.Dept#
  WHERE Name = 'Red';
- SELECT DeptName
  FROM Employees NATURAL JOIN Departments
  WHERE Name = 'Red';
- SELECT DeptName
  FROM Employees JOIN Departments USING (Dept#)
  WHERE Name = 'Red';

Notice that the SQL NATURAL JOIN does not fully correspond to the natural join as defined in the relational algebra: duplicate attributes are not eliminated and the schema of the resulting relation is the same as the Cartesian product.

The result of a join operation is a relation; it is thus possible to require that such relation be ordered also according to the values of columns from different tables and to eliminate duplicate tuples.
Join

- example: for each employee retrieve the name, the salary, the job and the name of the department where the employee works. The resulting tuples must be ordered according to the name of the department (ascending order) and to the salary (descending order).

```
SELECT DeptName, Name, Job, Salary
FROM Employees, Departments
WHERE Employees.Dept# = Departments.Dept#
ORDER BY DeptName, Salary DESC;
```
### SQL - Queries

#### Outer join - examples

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>S</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>2</td>
<td></td>
<td>3</td>
<td>v</td>
</tr>
</tbody>
</table>

- SELECT A, B, C FROM R NATURAL LEFT OUTER JOIN S;
- SELECT A, B, C FROM R NATURAL RIGHT OUTER JOIN S;
- SELECT A, B, C FROM R NATURAL FULL OUTER JOIN S;

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>x</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>3</td>
<td>v</td>
</tr>
</tbody>
</table>

#### Union join

- There is an additional join operator: UNION JOIN
- The result of R UNION JOIN S is a table that includes each column and each tuple of table R and table S; the tuples are completed with NULL values for each column that is not present in the original tuple
- example: R UNION JOIN S

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

#### Expressions and functions

- the predicates used in queries can involve, in addition to column names, also expressions
- expressions are formulated by applying operators to the values that tuples have for the columns
- examples of expressions and functions are the ones defined for numbers, strings, dates and times
- expressions and functions may appear in the projection clause, in the WHERE clause, and in the assignment expressions of the UPDATE statement
SQL - Queries

Expressions and functions

- An expression used in the projection clause results in a virtual column, that is a column not present in the queried table.
- The virtual columns are not physically stored in the table; they are materialized as a result of queries.
- It is possible to assign a name to a virtual column with the clause `AS ColumnName`.

SQL - Queries

Arithmetic expressions and functions

- Simple expressions can be formulated by applying the arithmetic operators (+, -, *, /) to the values that columns have in tuples.

Example: retrieve the name, the salary, the bonus, and the sum of the salary and the bonus for all the engineers.

```
SELECT Name, Salary, Bonus, Salary + Bonus
FROM Employees
WHERE Job = 'engineer';
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Bonus</th>
<th>TotalSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>1600.00</td>
<td>500.00</td>
<td>2100.00</td>
</tr>
<tr>
<td>Neri</td>
<td>2450.00</td>
<td>200.00</td>
<td>2650.00</td>
</tr>
<tr>
<td>Dare</td>
<td>2000.00</td>
<td>300.00</td>
<td>2300.00</td>
</tr>
<tr>
<td>Adams</td>
<td>1100.00</td>
<td>500.00</td>
<td>1600.00</td>
</tr>
<tr>
<td>Giani</td>
<td>1950.00</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Mill</td>
<td>1300.00</td>
<td>150.00</td>
<td>1450.00</td>
</tr>
</tbody>
</table>

In this example, a name is assigned to the virtual column `TotalSalary`.

Example: retrieve the name, the salary, the bonus, and the sum of the salary and the bonus for all the engineers such that the sum of salary and bonus is greater than 2000.

```
SELECT Name, Salary, Bonus, Salary + Bonus AS TotalSalary
FROM Employees
WHERE Job = 'engineer' AND Salary + Bonus > 2000;
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
<th>Bonus</th>
<th>TotalSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red</td>
<td>1600.00</td>
<td>500.00</td>
<td>2100.00</td>
</tr>
<tr>
<td>Neri</td>
<td>2450.00</td>
<td>200.00</td>
<td>2650.00</td>
</tr>
<tr>
<td>Dare</td>
<td>2000.00</td>
<td>300.00</td>
<td>2300.00</td>
</tr>
<tr>
<td>Adams</td>
<td>1100.00</td>
<td>500.00</td>
<td>1600.00</td>
</tr>
<tr>
<td>Giani</td>
<td>1950.00</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Mill</td>
<td>1300.00</td>
<td>150.00</td>
<td>1450.00</td>
</tr>
</tbody>
</table>

In this example, a name is assigned to the virtual column `TotalSalary`.

...
Arithmetic expressions and functions

- **functions:**
  - ABS(n) computes the absolute value of the numeric value n
  - MOD(n, b) computes the integer remained of the division of n by b
- several DBMS provide many other functions: square root, integer part, trigonometric functions
- the functions can be used in the projection clause and in the WHERE clause
  
  ex. WHERE MOD(A, 5) > 3 where A is a column name

Expressions and functions for strings

- Concateation operator denoted by `||`
  
  SELECT LastName || ' ' || FirstName || ' ' || Address
  FROM People;
  returns a unique string that include last name, first name, and address separated by a white space
- **functions:**
  - SUBSTRING (str FROM m [ FOR n]) given a string str, it extracts from it the substring starting at the character of position m for a length n (if n is specified) or until the last character of str
  - TRIM ([{LEADING|TRAILING|BOTH} [str1] FROM] str2)
    it removes from str2 the characters in str1, from the initial/subsequent positions or both (default: BOTH)
  - (UPPER | LOWER) (str)
    they transform the string str in upper case or lower case, respectively

Expressions and functions for strings

- **functions:**
  - POSITION (str1, IN str2)
    it returns the position of the first character in which str1 appears as substring in str2
  - {BIT|CHAR|OCTET}_LENGTH(str)
    it returns the length of string str, expressed as number of bits, characters, bytes, respectively
  - There are also functions for conversion between character sets
Expressions and functions for dates and temporal values

- It is possible to apply the arithmetic + and – to temporal intervals and between temporal values and temporal intervals.
- It is also possible to apply * and / between a temporal interval and a number; function ABS can be applied to temporal intervals.
- Functions:
  - Zero-argument functions: CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, LOCALTIMESTAMP
  - EXTRACT (field FROM expr)
    - It extracts from expr the field corresponding to the temporal qualifier
      ex. EXTRACT (DAY FROM '1969-10-08')

Example: the head of the HR department would like to have a meeting with all new employees of department 10 after 90 days from the hiring date. For each employee for whom the meeting has not taken place, the secretary of the head has to determine the name, the hiring date and the date of the meeting.

```
SELECT Name, HiringDate, HiringDate + 90 DAYS
FROM Employees
WHERE HiringDate + 90 DAYS > CURRENT_DATE
AND Dept# = 10;
```

CAST expressions

- It is possible to convert a value to another type through the CAST operator
  - CAST (ScalarExpression) AS Target
- Examples of possible conversions:
  - between numeric values (exact or approximated) and strings (fixed or variable length)
  - between bit strings and character strings
  - between temporal values and character strings
  - between exact numeric values and temporal intervals

CAST expressions

- It is also possible to perform cast operations on the various temporal values:
  - from DATE to TIMESTAMP – the TIME field is assigned the value 00:00:00
  - from TIME to TIMESTAMP – the DATE field is assigned the value CURRENT_DATE
  - from TIMESTAMP to DATE/TIME – a project operation is executed on the field of interest
  - between intervals of different granularities, when it is possible to perform the conversion
    ex. CAST(INTERVAL '3' YEAR TO INTERVAL MONTH) the result is INTERVAL '36' MONTH
An aggregate function allows one to retrieve aggregate information from sets of tuples. The aggregate functions are based on two main concepts:
- Partitioning of the set of tuples in a relation based on the values of one or more columns of the relation. The columns to be used for the partitioning are specified in an additional query clause, the GROUP BY clause.
- Computation of the aggregate function on each partition (also called group) obtained by the partitioning. An aggregate function has as input a column and it is applied to the set of values of this column, extracted from the tuples belonging to the same partition.

Common aggregate functions are:
- MAX, MIN, SUM, AVG, COUNT
Some systems also include STDEV and VARIANCE.
All aggregate functions, except COUNT, must be applied to sets of simple values and cannot be applied to sets of tuples; therefore they have as argument a column name and optionally the DISTINCT qualifier.

The COUNT function can have three different types of arguments:
- The special character '*' - in this case, the function returns the number of tuples in a given group.
  example: COUNT(*)
- A column name – in this case, the function returns the number of not null values for the column.
  example: COUNT(Salary)
- A column name with the DISTINCT qualifier – in this case, the function returns the number of not null distinct values for the column.
  example: COUNT(DISTINCT Salary)

For the MIN and MAX functions, the use of the DISTINCT qualifier does not change the results with respect to the case when the qualifier is not used.

For the SUM and AVG functions, if the argument is:
- A column name – the function is applied to all the not null values of the column.
  example: SUM(Salary)
- A column name with the DISTINCT qualifier – the function is applied to all not null distinct values of the column.
  example: SUM(DISTINCT Salary)
It is possible to apply the aggregate functions without performing the partitioning. In such case the functions are applied to a single group which includes all tuples of the relation.

Example: determine the highest salary of employees

```sql
SELECT MAX(Salary) FROM Employees;
```

Results

<table>
<thead>
<tr>
<th>MAX(Salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000.00</td>
</tr>
</tbody>
</table>

Example: group the employees according to the department number and determine the maximum salary for each group

```sql
SELECT Dept#, MAX(Salary) FROM Employees GROUP BY Dept#;
```

Results

<table>
<thead>
<tr>
<th>Dept#</th>
<th>MAX(Salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3000.00</td>
</tr>
<tr>
<td>20</td>
<td>2975.00</td>
</tr>
<tr>
<td>30</td>
<td>2850.00</td>
</tr>
</tbody>
</table>

In a query containing a GROUP BY clause, each tuple of the result relation represents a group of tuples from the relation on which the query is executed. In the example there are three groups: one for each value of Dept#

The MAX function is applied for each such group; it is applied to the set of values of Salary for the tuples in each such group.

Several columns can be used to partition the tuples.

The aggregate functions can also been used when queries contain joins.

Example: group the employees by department name and job; for each group we want to determine: the sum of salaries, the number of employees in each group, the average of salaries

```sql
SELECT DeptName, Job, SUM(Salary), COUNT(*), AVG(Salary) FROM Departments, Employees WHERE Departments.Dept#=Employees.Dept# GROUP BY DeptName, Job;
```
SQL - Aggregate Functions

### Results

<table>
<thead>
<tr>
<th>DeptName</th>
<th>Job</th>
<th>SUM(Salary)</th>
<th>COUNT(*)</th>
<th>AVG(Salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Construction</td>
<td>engineer</td>
<td>5750,00</td>
<td>3</td>
<td>1916,66</td>
</tr>
<tr>
<td>Building Construction</td>
<td>manager</td>
<td>3000,00</td>
<td>1</td>
<td>3000,00</td>
</tr>
<tr>
<td>Research</td>
<td>engineer</td>
<td>2700,00</td>
<td>2</td>
<td>1350,00</td>
</tr>
<tr>
<td>Research</td>
<td>manager</td>
<td>2975,00</td>
<td>1</td>
<td>2975,00</td>
</tr>
<tr>
<td>Research</td>
<td>secretary</td>
<td>1800,00</td>
<td>2</td>
<td>900,00</td>
</tr>
<tr>
<td>Road Maintenance</td>
<td>engineer</td>
<td>1950,00</td>
<td>1</td>
<td>1950,00</td>
</tr>
<tr>
<td>Road Maintenance</td>
<td>manager</td>
<td>2850,00</td>
<td>1</td>
<td>2850,00</td>
</tr>
<tr>
<td>Road Maintenance</td>
<td>secretary</td>
<td>800,00</td>
<td>1</td>
<td>800,00</td>
</tr>
<tr>
<td>Road Maintenance</td>
<td>technician</td>
<td>3100,00</td>
<td>3</td>
<td>1033,33</td>
</tr>
</tbody>
</table>

**Important restriction:** The projection clause of a query including a GROUP BY clause can only include:

- One or more columns from the set of columns that appear in the GROUP BY clause
- Aggregate functions

- The aggregate functions can also appear in arithmetic expressions
  - Ex. `SUM(Salary) + SUM(Bonus)`

### HAVING Clause

- It is possible to specify search conditions against groups of tuples
- Example: consider the groups in the previous query and suppose that we wish to retrieve only the groups that include at least two employees

```sql
SELECT DeptName, Job, SUM(Salary), COUNT(*), AVG(Salary)
FROM Departments, Employees
WHERE Departments.Dept# = Employees.Dept#
GROUP BY DeptName, Job
HAVING COUNT(*) > 2;
```

### Results

<table>
<thead>
<tr>
<th>DeptName</th>
<th>Job</th>
<th>SUM(Salary)</th>
<th>COUNT(*)</th>
<th>AVG(Salary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Building Construction</td>
<td>engineer</td>
<td>5750,00</td>
<td>3</td>
<td>1916,66</td>
</tr>
<tr>
<td>Research</td>
<td>engineer</td>
<td>2700,00</td>
<td>2</td>
<td>1350,00</td>
</tr>
<tr>
<td>Research</td>
<td>secretary</td>
<td>1800,00</td>
<td>2</td>
<td>900,00</td>
</tr>
<tr>
<td>Road Maintenance</td>
<td>technician</td>
<td>3100,00</td>
<td>3</td>
<td>1033,33</td>
</tr>
</tbody>
</table>

- The HAVING clause can be a Boolean combination of predicates; such predicates however may only involve aggregate functions
### SQL - Aggregate Functions

- An "execution" model
  1. The search condition specified in the WHERE clause of the query is applied to all the tuples of the relation input to the query. The evaluation is performed on each single tuple.
  2. The tuples obtained by step 1 are partitioned according to the GROUP BY clause.
  3. The search condition specified in the HAVING clause is applied to each group of tuples obtained at the previous step.
  4. The groups obtained by step 3 are the ones verifying the query. For such groups the aggregate functions specified in the projection clause are computed; the results of such functions are the results of the query.

### SQL – Null values

- SQL uses a three-valued logics to evaluate the truth value of a search condition (where clause).
  - True (T), False (F), Unknown (?)
- The evaluation for a simple predicate evaluated on an attribute having null value returns ? (Unknown)
- The truth value of a Boolean combination of predicates return is determined according to the tables.
- A tuple for which the predicate truth value is ? is not returned by the query.
- If the evaluation of a constraint predicate is ? The constraint is not violated.

#### SQL – AND and OR

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

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

#### SQL – NOT

<table>
<thead>
<tr>
<th>NOT</th>
<th>T</th>
<th>F</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td></td>
</tr>
</tbody>
</table>
### SQL - Null values

- SELECT * FROM R WHERE A=a AND B=b;
  
  The truth value for each of the tuples of R is as follows:
  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
  
  no tuple verify the query

- SELECT * FROM R WHERE C=c1;
  
  The truth value for each of the tuples of R is as follows:
  
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
  
  the tuple that verifies the query is t2

- SELECT * FROM R WHERE B IS NULL;
  
  returns tuples t1 and t3

- SELECT * FROM R WHERE B IS NULL AND C IS NULL;
  
  returns tuple t3

- SELECT * FROM R WHERE B IS NULL OR C IS NULL;
  
  returns tuples t1 and t3

### SQL - Null values

- the IS NOT NULL predicate applied to a given attribute of a tuple returns True if the tuple has a non null value for the attribute

- example:
  
  - SELECT * FROM R WHERE B IS NOT NULL;
    
    returns tuple t2

- Returns that the following queries:
  
  - SELECT * FROM R WHERE B = B;
  
  - SELECT * FROM R WHERE B = 'b' OR B <> 'b';
  
  - SELECT * FROM R WHERE B IS NOT NULL;
    
    are all equivalent

### SQL - Null values

- In the expressions if one of the argument NULL, then the entire expression id NULL

- example: Salary + Bonus

- In the aggregate functions null values are ignored

- example: SUM(Bonus)

- This has as the consequence that
  
  SUM(Col1 + Col2) can give a different result from
  
  SUM(Col1) + SUM(Col2)
SQL - Null values

- If e1 and e2 are NULL, e1 = e2 is not True; it is Unknown.
- Thus two expressions with null value are not considered to be equal but are not distinct, that is, they are considered duplicates:
  - SELECT DISTINCT: there is at most a NULL in the result
  - GROUP BY: there is at most a group with NULL
- There is a predicate IS DISTINCT FROM that has the same as <> except that for the null value, that is, if e1 and e2 are NULL:
  - e1 <> e2 returns UNKNOWN, but
  - e1 IS DISTINCT FROM e2 returns FALSE

SQL - Subqueries

- One of the reasons making SQL a powerful language is the possibility of specifying complex queries in terms of simpler queries, through the subquery mechanism.
- The WHERE clause of a query (called external query) can contain another query (called subquery).
- The subquery is used to determine one or more values to be used as comparison values in predicates in the external query.

Example: determine all employees that have the same job of the employee Gianni:

```
SELECT Name, Job FROM Employees
WHERE Job = (SELECT Job FROM Employees
WHERE Name = 'Gianni');
```

- The subquery returns as value 'engineer'
- The external query then determines all employees that are engineers
- The result is (Red, Neri, Dare, Adams, Gianni, Mill)

Example: determine all employees having a salary higher than the average of the salaries of all employees:

```
SELECT Name, Salary FROM Employees
WHERE Salary > (SELECT AVG(Salary)
FROM Employees);
```
SQL - Subqueries

- **results**
  - Name | Salary
  - Pink  | 2975.00
  - Black | 2850.00
  - Neri  | 2450.00
  - Dare  | 2000.00
  - Gianni| 1950.00
  - Verdi | 3000.00

- a subquery may include another subquery, join predicates, and all predicates we have seen so far
- the subqueries can also be used in the data manipulation statements (Insert, Delete, Update)

SQL - Subqueries

- in the examples seen so far, subqueries return a single value
- such subqueries are referred to as **scalar** subqueries
- if a scalar subquery returns several tuples, an error is returned at run-time
- if we want to use a subquery returning several values (referred to as **table** subquery) we need to specify how the returned values have to be used in the **WHERE** clause

SQL - Subqueries

- the **ANY** and **ALL** quantifiers are used for such purpose
- such quantifiers are inserted between the comparison operator and the subquery (there is also the **SOME** quantifier, synonym of **ANY**)
- example: determine the salary, job, name and department number of all employees having a salary greater than the salary of **at least one** employee from the department #30

SELECT Salary, Job, Name, Dept#
FROM Employees WHERE Salary > ANY (SELECT Salary
FROM Employees
WHERE Dept#=30);

SQL - Subqueries

- If the quantifier **ALL** is used, the query returns the employees having a salary greater than all the values returned by the subquery
**SQL - Subqueries**

- Example: determine the salary, job, name and department number of all employees having a salary greater than the salary of all employees from the department #30

```
SELECT Salary, Job, Name, Dept#
FROM Employees WHERE Salary > ALL (SELECT Salary FROM Employees WHERE Dept#=30);
```

- Results

<table>
<thead>
<tr>
<th>Salary</th>
<th>Job</th>
<th>Name</th>
<th>Dept#</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000.00</td>
<td>manager</td>
<td>Green</td>
<td>10</td>
</tr>
<tr>
<td>2975.00</td>
<td>manager</td>
<td>Pink</td>
<td>20</td>
</tr>
</tbody>
</table>

**SQL - Subqueries**

- The following shorthands are defined for ANY and ALL
  - IN equivalent to = ANY
  - NOT IN equivalent to ≠ ALL

- Example: determine the name and job of the employees from the department #10 that have the same job as some employee from the department #30

```
SELECT Name, Job FROM Employees WHERE Dept#=10 AND Job IN (SELECT Job FROM Employees WHERE Dept#=30);
```

- Results

<table>
<thead>
<tr>
<th>Job</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>manager</td>
<td>Green</td>
</tr>
<tr>
<td>technician</td>
<td>Andrews</td>
</tr>
<tr>
<td>technician</td>
<td>White</td>
</tr>
<tr>
<td>technician</td>
<td>Turni</td>
</tr>
<tr>
<td>secretary</td>
<td>Martin</td>
</tr>
</tbody>
</table>

**SQL - Subqueries**

- Example (2): determine the name and job of the employees from the department #10 having a job not present in department #30

```
SELECT Name, Job FROM Employees WHERE Dept#=10 AND Job NOT IN (SELECT Job FROM Employees WHERE Dept#=30);
```

- Results

<table>
<thead>
<tr>
<th>Job</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>manager</td>
<td>Green</td>
</tr>
<tr>
<td>engineer</td>
<td>Mill</td>
</tr>
</tbody>
</table>

**SQL - Subqueries**

- The WHERE clause of a query may include any combination of simple predicates and predicates with subqueries

- Example: determine the employees with the same job as employee Gianni or with a salary greater or equal than the salary of employee Ford; the results must be ordered according to job and salary

```
SELECT Name, Job, Salary FROM Employees WHERE Job = (SELECT Job FROM Employees WHERE Name = 'Gianni') OR Salary ≥ (SELECT Salary FROM Employees WHERE Name = 'Ford') ORDER BY Job, Salary;
```
SQL - Subqueries

- A subquery may include another subquery
- Example: list the names and jobs of the employees of the department #10 having the same job of some employee from the Research department

```
SELECT Name, Job FROM Employees
WHERE Dept# = 10 AND Job IN
    (SELECT Job FROM Employees
     WHERE Dept# = (SELECT Dept# FROM Departments
                    WHERE DeptName = 'Research'));
```

SQL – Correlated subqueries

- In the examples seen, each subquery is executed just once and the returned value (or set of values) is used in the WHERE clause of the external query
- It is possible to specify subqueries that are repeatedly executed for each candidate tuple considered in the evaluation of the external query
- Example: determine the employees having a salary greater than the average of the salaries of their own departments

```
SELECT Name FROM Employees
WHERE Salary > (average of the salaries of the employees from the department of the candidate employee);
```

We also need a subquery to compute the average of the salaries of the department for each candidate tuple; such subquery would have the form:

```
(SELECT AVG(Salary) FROM Employees
 WHERE Dept# = (SELECT Dept# FROM Departments
                  WHERE DeptName = 'Research'));
```

The external query can then compare the salary of the employee being considered with the value returned by the subquery
Such type of queries is called correlated, because each execution of the subquery is correlated to the values of one or more attributes of the candidate tuples in the external query. In order to reference the columns of the tuples, candidates in the external query, variables (referred to as relation aliases) can be used in the two queries are on the same relation; a relation alias is defined in the external query and referenced in the subquery.

Two are the main concepts underlying correlation:

a) a correlated subquery references one or more attributes of tuples from the external query
b) if a subquery select tuples from the same relation referred by the external query, a relation alias must be defined for such relation in the external query; the subquery must use the alias to reference the attribute values in the candidate tuples from the external query.

The use of the alias is not mandatory if the external query and the subquery refer to different relations, even though it is useful for clarity.

example: the following query expresses in SQL the query we discussed; in addition the results must be ordered:

```
SELECT Dept#, Name, Salary FROM Employees X
WHERE Salary > (SELECT AVG(Salary)
    FROM Employees
    WHERE X.Dept# = Dept#)
ORDER BY Dept#;
```

example: determine name and salary of employees that are managers of their departments

```
SELECT Name, Salary FROM Employees
WHERE Emp# = (SELECT Manager FROM Departments
    WHERE Employees.Dept# = Dept#)
```

is equivalent to

```
SELECT Name, Salary FROM Employee X
WHERE Emp# = (SELECT Manager FROM Departments
    WHERE X.Dept# = Dept#)
```
SQL - Correlated subqueries

- the alias can also be used in queries that do not include subqueries
- they are useful when we want to reference to two different tuples of the same relation
- example: determine name and salary of the employees that earn more than the manager of their department

```sql
SELECT X.Name, X.Salary
FROM Employees X, Employees Y, Departments
WHERE X.Dept# = Departments.Dept# AND
       Departments.Manager = Y.Emp# AND
       X.Salary > Y.Salary;
```

SQL - Correlated subqueries

**ALL and ANY**

- example: determine the departments such that all employees of such departments earn more than 1000

```sql
SELECT * FROM Department X
WHERE 1000 < ALL (SELECT Salary FROM Employees
WHERE Dept#=X.Dept#);
```

- example: determine the departments that have at least an employee earning more than 1000

```sql
SELECT * FROM Departments X
WHERE 1000 < ANY (SELECT Salary FROM Employees
WHERE Dept#=X.Dept#);
```

Note: such query can simply be expressed with a join

SQL - Correlated subqueries

**EXISTS and NOT EXISTS**

- the predicate EXISTS(sq) returns the Boolean value True if the subquery returns at least a tuple; returns the Boolean value False otherwise
- the predicate NOT EXISTS(sq) returns the Boolean value True if the subquery does not return any tuple; it returns the Boolean value False otherwise
- note: the evaluation of predicates with these two operators never returns the Unknown value

```sql
SELECT Name FROM Employees X
WHERE EXISTS (SELECT * FROM Employees
WHERE X.Emp# = Reports_to);
```

SQL - Correlated subqueries

**EXISTS and NOT EXISTS**

- example: suppose that the Employees relation had an additional column, Reports_to, that records an emp# for each tuple of Employees. Given an employee E, such number denotes the employee to whom E reports
- We want to determine the name of all employees having at least an employee reporting to them

```sql
SELECT Name FROM Employees X
WHERE EXISTS (SELECT * FROM Employees
WHERE X.Emp# = Reports_to);
```
We want to determine the name of all employees that do not have any employee reporting to them.

```
SELECT Name FROM Employees X
WHERE NOT EXISTS (SELECT * FROM Employees WHERE X.Emp# = Reports_to);
```

The correlated subqueries and the NOT EXISTS operator allows one to express the division operation.

Example: consider the relations Projects(Proj#, Pname, Budget) and Participants(Dept#, Proj#).

Determine the names of the departments participating to all projects with a budget greater than 50,000.

```
SELECT DeptName FROM Departments X
WHERE NOT EXISTS (SELECT * FROM Projects Y
WHERE Budget > 50,000 AND
NOT EXISTS (SELECT * FROM Participants
WHERE X.Dept# = Dept# AND
Y.Proj# = Proj#));
```

The specification of the division in SQL is based on the notion of counterexample.

A department X verifies the query if we cannot find a project with a budget greater than 50,000 to which X does not participate.

```
SELECT DeptName FROM Departments X
WHERE NOT EXISTS (SELECT * FROM Projects Y
WHERE Budget > 50,000 AND
NOT EXISTS (SELECT * FROM Participants
WHERE X.Dept# = Dept# AND
Y.Proj# = Proj#));
```

The UNIQUE(sq) predicate returns the Boolean value True if the subquery returns tuples that are all distinct.

It returns the Boolean value False if the subquery results contain duplicates.

Note: the evaluation of predicates with this operator never returns Unknown value.
**SQL - Correlated subqueries**

- **UNIQUE**
  - example: determine the departments that do not have two (or more) employees with the same job

  ```sql
  SELECT Dept# FROM Employees X
  WHERE UNIQUE (SELECT Job
  FROM Employees
  WHERE Dept# = X.Dept#);
  ```

**SQL – Set operations**

- **Union**
  - a query or subquery may consist of several queries combined UNION operator
  - the UNION operator returns all the distinct tuples returned by at least one of the queries to which it is applied
  - example: consider the following relations, having the same schema as the Employees relation

    ```sql
    Part_Time_Employees
    In_Sabbatical_Employees
    ```

  ```sql
  SELECT Name, Emp# FROM Employees
  WHERE Salary IN
  (SELECT Salary FROM Part_Time_Employees
  WHERE Job = 'technician'
  UNION
  SELECT Salary FROM In_Sabbatical_Employees
  WHERE Job = 'engineer');
  ```

**SQL - Set operations**

- **Union**
  - Determine all employees that have the same salary of part time technicians or engineers that are in sabbatical

    ```sql
    SELECT Name, Emp# FROM Employees
    WHERE Salary IN
    (SELECT Salary FROM Part_Time_Employees
    WHERE Job = 'technician'
    UNION
    SELECT Salary FROM In_Sabbatical_Employees
    WHERE Job = 'engineer');
    ```

**SQL - Set operations**

- **Union**
  - The UNION imposes some restrictions on the queries on which it operates
  - The queries must return the same number of columns, and the corresponding columns must have the same domain (it is not required that the length be the same) or compatible domains
  - If nothing is specified about the correspondence of columns, the correspondence is based on the column position in the projection clauses, independently from their names
Union

- If the keyword CORRESPONDING is used after the UNION, then the match among columns is not based on the position but on the column names.
- It is also possible to explicitly specify the columns to be matched by their name by the CORRESPONDING BY(ColumnNameList) clause.
- If the ORDER BY clause is used, such clause must be the only one in the query and has to be specified at the end of the query and not at the end of each SELECT clause.

Example:

```sql
SELECT Name, Emp# FROM Part_Time_Employees
UNION
SELECT Name, Emp# FROM In_Sabbatical_Employees
ORDER BY 2;
```

Union

- When one specifies the columns on which to perform the ordering, the relative positions of columns have to be used and not the names, in that names could be different in different tables.
- Example:

```sql
SELECT Name, Emp# FROM Part_Time_Employees
UNION
SELECT Name, Emp# FROM In_Sabbatical_Employees
ORDER BY 2;
```

Intersect and Except

- The INTERSECT and EXCEPT (or MINUS) operators execute the intersection and difference.
- The conditions for the application of these operators are the same of the union.
- Example:

```sql
SELECT Name FROM Part_Time_Employees
INTERSECT
SELECT Name FROM In_Sabbatical_Employees;
```
**SQL - Set operations**

**Intersect and Except**
- determine the names of the part time employees that are not in sabbatical
  
  ```sql
  SELECT Name FROM Part_Time_Employees EXCEPT
  SELECT Name FROM In_Sabbatical_Employees;
  ```

- If the available SQL language does not support the INTERSECT and EXCEPT operations, such operations can be executed through EXISTS and NOT EXISTS

**Example**
- determine the names of the part time employees that are in sabbatical
  
  ```sql
  SELECT Name FROM Part_Time_Employees X
  WHERE EXISTS (SELECT * FROM In_Sabbatical_Employees
  WHERE Emp#=X.Emp#);
  ```

- determine the names of the part time employees that are not in sabbatical
  
  ```sql
  SELECT Name FROM Part_Time_Employees X
  WHERE NOT EXISTS (SELECT *
  FROM In_Sabbatical_Employees
  WHERE Emp#=X.Emp#);
  ```

**SQL - DML Insertion**

- The INSERT statement has the following format
  
  ```sql
  INSERT INTO R [(C1,C2,...,Cn)]
  {VALUES (e1,e2,...,en) | sq};
  ```

  where:
  - R is the name of the relation on which the insert is executed
  - C1,C2,...,Cn is the list of columns of the new tuple(s) to which values are assigned
  - All columns not explicitly listed receive the null value or the default value (if specified in the create command for R)
  - If such list is not specified, then a list including all columns of R is assumed

- `e1,e2,...,en` is the list of values to be assigned to the new tuple

- the values are assigned based on a positional correspondence; value ei (i=1,...,n) is assigned to column Ci

- sq is a subquery (mutually exclusive with respect the VALUES clause)

- the tuples returned by the subquery are inserted in relation R

- the projection clause of sq must contains columns (or more in general expressions) compatible with the columns of R to which values are assigned

- therefore the domain of column Ci (i=1,...,n) must be compatible with the domain of the i-th column (or expression) in the projection clause of sq
Example of insertion with explicit values
- A new department has to be inserted; the number is 40, the name is Applied Research; the division is D2; the manager is Blacchi (Emp# 7698); the office is 6100

```
INSERT INTO Departments
VALUES (40, 'Applied Research', 6100, 'D2', 7698);
```

Example of insertion of tuples the values of which are obtained from a subquery
- Suppose that we would like to create a relation Promotions, that includes some of the columns of the Employees relation: Name, Salary, Bonus
- We would like to insert in such relation all engineers having a bonus greater than the 25% of their salary; such information must be extracted from the Employees relation
  
  ```
  INSERT INTO Promotions (Name, Salary, Bonus)
  SELECT Name, Salary, Bonus
  FROM Employees WHERE Bonus > 0.25*Salary AND Job = 'engineer';
  ```
- The engineers that are inserted are Red and Adams

The statement for the deletions has the following format

```
DELETE FROM R [alias] [WHERE F];
```

where:
- R is the name of the relation on which the deletion has to be executed
- The name of the relation may have an alias associated if it is necessary to refer to the tuples of this relation in some subquery in the WHERE clause F
- F is the qualification clause and specifies the tuples to be deleted
- If no qualification clause is specified all tuples are deleted

The UPDATE statement has the following format

```
UPDATE R [alias]
SET C1={e1 | NULL}, ...., Cn={en | NULL}
[WHERE F];
```

where:
- R is the name of the relation being modified
- The name of the relation may have an alias associated if it is necessary to refer to the tuples of this relation in some subquery in the WHERE clause F
SQL - DML Update

- \( C_i = \{e_i | \text{NULL}\} \) \((i=1,\ldots,n)\) is assignment expression specifying that column \( C_i \) must be assigned the value of expression \( e_i \).
- Such expression can be a constant, or an arithmetic or string expression, often function of the current values of the tuples to be modified, or a subquery.
- Alternatively it is possible to specify that the column be assigned the null value.
- \( F \) is the qualification clause selecting the tuples to be modified.
- If no qualification clause is specified, all tuples are modified.

Example: the salary of all technicians must be increased of 100

```
UPDATE Employees
SET Salary = Salary + 100
WHERE Job = 'technician';
```

Example: the employee Gianni must be promoted to manager and his salary increased of 10%

```
UPDATE Employees
SET Job = 'manager',
Salary = 1.10*Salary
WHERE Name = 'Gianni';
```

In an Update statement, subqueries can be used for:

(a) selecting the tuples to be modified
(b) determining the new values to be assigned to tuples

Example (a): consider a Bonus relation and assume to increase of 5% the salary of all employees in such relation

```
UPDATE Employees
SET Salary = 1.05* Salary
WHERE Emp# IN
(SELECT Emp# FROM Bonus);
```

Example (b): assign to all technicians a salary equal to 110% of the average of the salaries of all the technicians

```
UPDATE Employees
SET Salary = (SELECT 1.1*AVG(Salary)
FROM Employees
WHERE Job = 'technician')
WHERE Job = 'technician';
```

The updates in SQL are executed according to a set-oriented semantics: the WHERE clause and the value of the expressions in the SET clause are evaluated only once, then the updates are executed on all the tuples “simultaneously”.

SQL - DML Null values

- For assigning null values in the Update statement, the NULL keyword is used (to not to be confused with the 'NULL' string)

```
UPDATE R
SET B=NULL WHERE C=c2;
```

result:

<table>
<thead>
<tr>
<th>R</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>?</td>
<td>c1</td>
<td>t1</td>
</tr>
<tr>
<td>a1</td>
<td>?</td>
<td>c2</td>
<td>t2</td>
</tr>
<tr>
<td>a2</td>
<td>?</td>
<td>?</td>
<td>t3</td>
</tr>
</tbody>
</table>

SQL – DML example

- Suppose to create a Project relation with the following command

```
CREATE TABLE Projects (Proj# Decimal(3) NOT NULL, 
                      ProjName Varchar(5), 
                      Budget Decimal(7,2));
```

- Suppose to insert some tuples into this relation

```
INSERT INTO Projects VALUES (101,'Alpha', 96000);
INSERT INTO Projects VALUES (102,'Beta', 82000);
INSERT INTO Projects VALUES (103,'Gamma', 15000);
```

- Because each employee is assigned to a project, we would like to extend the Employees relation with a new column that records the number of the project for each employee

```
ALTER TABLE Employees
ADD COLUMN (Proj# Decimal(3));
```

- Suppose to assign all technicians (of every department) and all employees of department #20 to project 101

```
UPDATE Employees SET Proj#=101
WHERE Dept#>20 OR Job = 'technician';
```

- Suppose to assign all employees not assigned to any project to project 102

```
UPDATE Employees SET Proj#=102
WHERE Proj# IS NULL;
```

SQL – DML example

<table>
<thead>
<tr>
<th>Emp#</th>
<th>Name</th>
<th>Job</th>
<th>HiringDate</th>
<th>Salary</th>
<th>Bonus</th>
<th>Dept#</th>
<th>Proj#</th>
</tr>
</thead>
<tbody>
<tr>
<td>7369</td>
<td>Red</td>
<td>engineer</td>
<td>17/12/80</td>
<td>1600.00</td>
<td>500.00</td>
<td>20</td>
<td>101</td>
</tr>
<tr>
<td>7499</td>
<td>Andrews</td>
<td>technician</td>
<td>20/02/81</td>
<td>800.00</td>
<td></td>
<td>30</td>
<td>101</td>
</tr>
<tr>
<td>7521</td>
<td>White</td>
<td>technician</td>
<td>20/02/81</td>
<td>800.00</td>
<td>100.00</td>
<td>30</td>
<td>101</td>
</tr>
<tr>
<td>7556</td>
<td>Pink</td>
<td>manager</td>
<td>22/04/81</td>
<td>2075.00</td>
<td></td>
<td>20</td>
<td>101</td>
</tr>
<tr>
<td>7554</td>
<td>Martin</td>
<td>secretary</td>
<td>28/06/81</td>
<td>800.00</td>
<td></td>
<td>30</td>
<td>102</td>
</tr>
<tr>
<td>7688</td>
<td>Black</td>
<td>manager</td>
<td>01/05/81</td>
<td>2600.00</td>
<td></td>
<td>30</td>
<td>102</td>
</tr>
<tr>
<td>7782</td>
<td>Black</td>
<td>engineer</td>
<td>01/06/81</td>
<td>2600.00</td>
<td></td>
<td>30</td>
<td>102</td>
</tr>
<tr>
<td>7788</td>
<td>Scott</td>
<td>secretary</td>
<td>06/11/81</td>
<td>800.00</td>
<td></td>
<td>20</td>
<td>101</td>
</tr>
<tr>
<td>7893</td>
<td>Dave</td>
<td>engineer</td>
<td>17/11/81</td>
<td>2000.00</td>
<td></td>
<td>30</td>
<td>102</td>
</tr>
<tr>
<td>7894</td>
<td>Tarti</td>
<td>technician</td>
<td>08/06/81</td>
<td>1500.00</td>
<td></td>
<td>30</td>
<td>101</td>
</tr>
<tr>
<td>7979</td>
<td>Adams</td>
<td>engineer</td>
<td>28/05/81</td>
<td>1100.00</td>
<td></td>
<td>30</td>
<td>101</td>
</tr>
<tr>
<td>7960</td>
<td>Garrett</td>
<td>engineer</td>
<td>03/12/81</td>
<td>1000.00</td>
<td></td>
<td>30</td>
<td>102</td>
</tr>
<tr>
<td>7962</td>
<td>Ford</td>
<td>secretary</td>
<td>03/12/81</td>
<td>1000.00</td>
<td></td>
<td>20</td>
<td>101</td>
</tr>
<tr>
<td>7934</td>
<td>Mill</td>
<td>engineer</td>
<td>23/01/82</td>
<td>1300.00</td>
<td></td>
<td>10</td>
<td>102</td>
</tr>
<tr>
<td>7977</td>
<td>Green</td>
<td>manager</td>
<td>10/12/80</td>
<td>3000.00</td>
<td></td>
<td>10</td>
<td>102</td>
</tr>
</tbody>
</table>
in SQL it is possible to define alternative views of the same data

A view is a virtual relation (like a window) through which it is possible to access the data stored in the real relations (called base relations)

A view does not physically store tuples, but it can be used almost like a base relation

A view is defined by a query on one or more base relations and/or other views

A view is materialized by executing the query that defines it

The view mechanism is useful for
- simplifying data accesses
- providing logical independence
- assuring data confidentiality

The command for creating views has the following format

CREATE VIEW V [(ColumnNameList)] AS Q
[WITH [LOCAL] CASCADED] CHECK OPTION;

where:

- V is the name of the view being created; such name must be unique with respect to all names of relations and view defined by the user who is defining V
- Q is the definition query of the view
  - a view has the same number of columns (base or virtual) as the number of columns in the projection clause of Q
  - ColumnNameList is a list of column names to be assigned to the view columns; such list is not mandatory, except when the query contains in the projection clause aggregate functions or expressions

example: create a view returning a subset of the set of tuples of the Employees relation; the view must include the columns Emp#, Name and Job for the employees of department #10

CREATE VIEW Emp10 AS
SELECT Emp#, Name, Job
FROM Employees WHERE Dept# = 10;

The names of the columns of view Emp10 are: Emp#, Name, Job
SQL - Views

- On a view it is possible to execute (with some restrictions) both queries and changes.
- Example: select all tuples of view Emp10
  ```sql
  SELECT * FROM Emp10;
  ```
- Results:
<table>
<thead>
<tr>
<th>Emp#</th>
<th>Name</th>
<th>Job</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
<td>Neri</td>
<td>engineer</td>
</tr>
<tr>
<td>7839</td>
<td>Dare</td>
<td>engineer</td>
</tr>
<tr>
<td>7934</td>
<td>Mills</td>
<td>engineer</td>
</tr>
<tr>
<td>7977</td>
<td>Green</td>
<td>manager</td>
</tr>
</tbody>
</table>

SQL - Views

Use of join

- Example: create a view Personnel that includes the columns Name and Job from relation Employees and the name of the project in which each employee works.
  ```sql
  CREATE VIEW Personnel AS
  SELECT Name, Job, ProjName
  FROM Employees, Projects
  WHERE Employees.Proj# = Projects.Proj#;
  ```

SQL - Views

Use of expressions and functions

- It is possible to define views through queries that include expressions or functions.
- Such expressions are similar to the other view column, but their values are computed from the base table each time the view is materialized.
- Such columns are often called virtual columns; it is mandatory to specify a name in the view for each such column.
- Example: define a view reporting the annual salary of employees.

```sql
CREATE VIEW V1 (Name, Monthly_Salary, Yearly_Salary, Dept#) AS
SELECT Name, Salary, Salary*12, Dept#
FROM Employees;
```

Suppose to execute the following query:
```sql
SELECT * FROM V1 WHERE Dept#=30;
```

The following results are obtained:
<table>
<thead>
<tr>
<th>Name</th>
<th>Monthly_Salary</th>
<th>Yearly_Salary</th>
<th>Dept#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew</td>
<td>800.00</td>
<td>9600.00</td>
<td>30</td>
</tr>
<tr>
<td>White</td>
<td>800.00</td>
<td>9600.00</td>
<td>30</td>
</tr>
<tr>
<td>Martin</td>
<td>800.00</td>
<td>9600.00</td>
<td>30</td>
</tr>
<tr>
<td>Blaschi</td>
<td>1500.00</td>
<td>18000.00</td>
<td>30</td>
</tr>
<tr>
<td>Turni</td>
<td>1950.00</td>
<td>23400.00</td>
<td>30</td>
</tr>
<tr>
<td>Galini</td>
<td>1950.00</td>
<td>23400.00</td>
<td>30</td>
</tr>
</tbody>
</table>
It is also possible to define views through queries that include aggregate functions and GROUP BY clauses.

Example: define a view reporting for each department some simple statistics concerning the employee salary.

```
CREATE VIEW Dept_S (Dept#, Min_S, Avg_S, Max_S, Total) AS
SELECT Dept#, MIN(Salary), AVG(Salary), MAX(Salary), SUM(Salary)
FROM Employees GROUP BY Dept#;
```

The query:

```
SELECT Dept#, Min_S, Max_S, Total FROM Dept_S;
```

Would return the following results:

<table>
<thead>
<tr>
<th>Dept#</th>
<th>Min_S</th>
<th>Max_S</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1300.00</td>
<td>3000.00</td>
<td>8750.00</td>
</tr>
<tr>
<td>20</td>
<td>800.00</td>
<td>2975.00</td>
<td>7445.00</td>
</tr>
<tr>
<td>30</td>
<td>800.00</td>
<td>2850.00</td>
<td>8700.00</td>
</tr>
</tbody>
</table>

Once a view is created, it can be considered (almost) as a base table.

There could be however restrictions on the queries and the updates that can be executed on views.

An example of restriction on queries:

- It is not possible to use aggregate functions on the columns of views that are defined as aggregate functions.
  
  (some DBMS actually allow such queries)
- When one issues a query on a view, the system replaces in the query the view with its definition.
- Example: consider the Personnel view and the query:

  SELECT Name, ProjName FROM Personnel
  WHERE Job = 'manager';

The query obtained by the composition is the following:

```
SELECT Name, ProjName FROM Employees, Projects
WHERE Employees.Proj# = Projects.Proj#
AND Job = 'manager';
```

In the case of aggregate functions applied to columns defined as aggregate functions such composition is not possible.
Problem 1: insertion

CREATE VIEW V3 AS
    SELECT Emp#, Name, Job, HiringData, Dept#
    FROM Employees
    WHERE Job < 'manager';

- The schema of V3 is (Emp#, Name, Job, HiringDate, Dept#)
- Suppose to insert in the view the following tuple (8001, 'Smith', 'technician', '13-12-205', 20)
- such insert is transformed in un insert operation on the Employees relation

- If for such columns a default value has been specified, then such value can be assigned to the inserted tuple
- another solution is to assign the null value
- such solution introduces problems if the value of Salary and Bonus must be different from the null value

Problem 2: deletion

CREATE VIEW V4 AS
    SELECT Name, Office
    FROM Employees, Departments
    WHERE Employees.Dept# = Departments.Dip#;

- the schema of V4 is (Name, Office)
- Suppose that the following deletion is executed
  DELETE FROM V4 WHERE Name = 'Red' AND Office = 2100;
- such delete can be translated as follows
  DELETE FROM Employees WHERE Name = 'Red';
  DELETE FROM Departments WHERE Office = 2100;
Problem 2: deletion

- state of the view after the deletion

<table>
<thead>
<tr>
<th>Name</th>
<th>Office</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew</td>
<td>5100</td>
</tr>
<tr>
<td>White</td>
<td>5100</td>
</tr>
<tr>
<td>Martin</td>
<td>5100</td>
</tr>
<tr>
<td>Blacchi</td>
<td>5100</td>
</tr>
<tr>
<td>Black</td>
<td>1100</td>
</tr>
</tbody>
</table>

- The previous updates have side-effects on the view
- If the user executes a select operation on the view after the update, all tuples related to employees having the same office as employee Red have been removed from the view

Problem 3: update

Create view V5 (Emp#, TotalSalary) as
SELECT Emp#, Salary+Bonus
FROM Employees;

Suppose to execute the following update on such view
UPDATE V5 SET TotalSalary = TotalSalary * 1.2
WHERE Emp# = 7782;

Such update can be mapped onto different updates on the base table: increasing Salary, increasing Bonus, increasing both Salary and Bonus

In general one should know the inverse of the function that defines TotalSalary

The execution of an update operation on a view is propagated to the table(s) on which the view is defined

Example of restrictions:

1) It is possible to execute the DELETE operation if the view definition query verifies the following conditions:
   - The query is on a single table
   - It does not include the GROUP BY clause, the DISTINCT clause, or aggregate functions
2) It is possible to execute the UPDATE operation if the view definition query verifies the previous conditions and in addition:
   - The modified column is not defined by an expression
3) It is possible to execute the INSERT operation if the view definition query verifies the previous conditions and in addition:
   - each column for which no default value has been specified and for which there is a NOT NULL constraint is present in the view

Example: consider the following view
CREATE VIEW EmpHS AS
SELECT Emp#, Name, Salary FROM Employees
WHERE Salary > 3000;

Suppose to insert in the view the following tuple
(200, Haas, 2000)
the condition specified in the query defining the view is not verified by such tuple; therefore such tuple is inserted but it cannot be retrieved by queries on the views.

To make sure that the tuples inserted through a view (or modified through a view) be accepted only if they verify the condition in the view definition query, the CHECK OPTION is used.

The format of a view definition is the following:
CREATE VIEW ViewName [(ColumnNameList)]
AS Query
WITH [LOCAL|CASCADED] CHECK OPTION

The difference between LOCAL and CASCADED is relevant only when a view is defined in terms of another view.
**SQL - View - check option**

- example: if the previous view were defined as follows

  ```sql
  CREATE VIEW EmpHS AS
  SELECT Emp#, Name, Salary FROM Employees
  WHERE Salary > 3000
  WITH CHECK OPTION;
  ```

- The insertion on the view of the tuple
  
  (200, Haas, 2000)

  would not be executed

**SQL - View - check option**

- If a view V1 is defined in terms of another view V2, then
  - if V1 is defined WITH LOCAL CHECK OPTION, the insertions executed on V1 must verify
    1. The definition of V1
    2. The definition of V2 only if V2 is in turn defined with check option
  - if V1 is defined WITH CASCADED CHECK OPTION, the insertions executed on V1 must verify
    1. The definition of V1
    2. The definition of V2 (independently from whether V2 is defined with check option or not)

**SQL - View – schema modifications**

- format of the statement for removing views
  ```sql
  DROP VIEW V;
  ```
  where V is the name of the view to be deleted

- format of the statement for changing the name of a view
  ```sql
  RENAME V₁ to V₂;
  ```

- It is not possible, instead, to modify the definition of a view; the only way is to re-define the view

**SQL - View – schema modifications**

- What happens to a view V when a table (or view) used in the view definition query is dropped?

  - In the DROP TABLE and DROP VIEW commands we can specify the RESTRICT or CASCADE options
    - if RESTRICT is specified, the command is executed only if the table/view is not used in the definition of other views (or other elements of the schema, for example assertions)
    - if CASCADE is specified, the command has the effect of dropping also the views (and the other elements of the schema) the definition of which is based on the table/view to be deleted
The set of catalogs is a system database containing information (descriptors) concerning the various database objects: tables, views, constraints, indexes, access permissions.

The catalogs are organized as relations.

Example:
- There is a tuple for each relation or view, with the following attributes:
  - name
  - name of the creator user
  - type (T/V)
  - number of columns

- There is a tuple for each column of each relation or view, including the following attributes:
  - column name
  - name of the table or view to which the column belongs
  - ordinal position of the column among the columns of the same table or view
  - data type of the column

Note: the catalogs have different names in different DBMS.

SQL – Examples of catalogs:

SYSTABLES - includes a tuple for each table or view given a table or view, some of the information recorded in a catalog tuple are:
- name (TABNAME)
- name of creator use (DEFINER)
- type (TYPE, T=table, V=view, A=alias)
- number of columns (COLCOUNT)

SYSCOLUMNS - includes a tuple for each column of each table or view in the database given a column, some of the information recorded in a catalog tuple are:
- the column name (COLNAME)
- the name of the table or view to which the column belongs (TABNAME)
- the ordinal position of the column among the columns belonging to the same table or view (COLNO)
- the column type (TYPENAME)

Note: on the set of tables that store the catalogs views are very often defined.
It is possible to query catalogs as if they were ordinary relations

- Find all tables having a column the name of which starts S
- Find all the columns of the Employees table
- Find how many tables have been created by user Smith

In general it is not possible to modify the catalogs

- The catalogs are automatically modified by the DBMS upon execution of DDL commands

Some exceptions to the restriction on the catalog updates are represented for example by the possibility that DBA have to modify the catalogs containing data statistics (used in query optimization)

The catalogs also include information about the tables storing them

The entries related to the catalog relations are not created as effect of DDL commands

Such entries are generated as part of the installation procedure of the DBMS (they are "hard-wired" in the system)

Some aspects of the SQL:1999 data model that have not been discussed so far include:

- object-relational modeling features:
  - collection types, row types, reference types, user-defined types
  - inheritance
- triggers
- recursion in queries and views
- OLAP constructs for data analysis
- stored procedures