Object-Relational Concepts

These slides take a closer look as some of the features of SQL:1999 and SQL:2003.

- SQL:1999 (also called SQL3): A relatively new standard which embodies some ideas of the object-oriented philosophy.

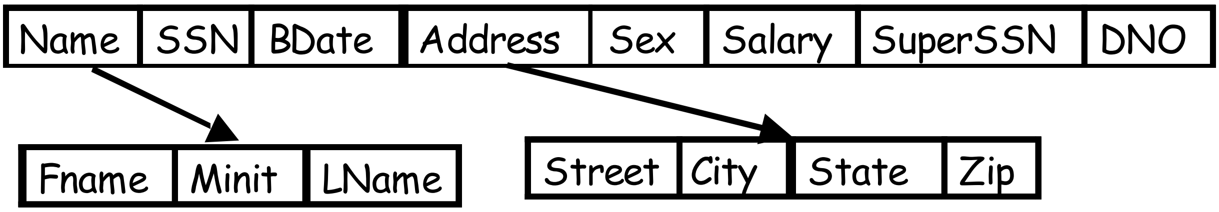
- SQL:2003 (also called SQL:200n,SQL4): The latest standard, which adds XML support and a few other features to SQL:1999..

Both standards provide nearly full backward compatibility with SQL2 (SQL-92), the “purely relational” standard.

Row types:

SQL:1999 supports the idea of a row type:

Here is how to recapture a structure such as the following:




```
CREATE ROW TYPE EmployeeType
(
Name        NameType,
SSN         Char(9)        NOT NULL,
BDate       Date,
Address     AddressType,
Sex         Char,
Salary      Decimal(10,2),
SuperSSN    Char(9);
DNO         Int            NOT NULL
);

CREATE ROW TYPE NameType,
(
LName       Varchar(15),
FName       Char,
MInit       Varchar(15)
);
```

```
CREATE ROW TYPE AddressType,
(
Street       Varchar(15),
City         Varchar(15),
State        Char(2),
Zip          Char(5)
);

CREATE TABLE Employee
     OF TYPE EmployeeType
(PRIMARY KEY SSN);
```

Example query (note use of ..):

```
SELECT   Name..LName, SSN,
FROM     Employee
WHERE    Address..State = 'NH';
```

or

```
SELECT   Employee.Name..LName,
         Employee.SSN,
FROM     Employee
WHERE    Employee.Address..State = 'NH';
```

Collection Types:

- SQL:1999 supports only the ARRAY collection type.
- SQL:2003 supports MULTISET as well, which is not a mathematical multiset, but just an ordinary set.

  The SQL declarations below are used to recapture a table with the following format:

Department				
Dname	Dnumber	MGRSSN	MGR-Startdate	DLocations
Research	5	333445555	1998-05-22	{Bellaire, Sugarland, Houston}
Administration	4	987654321	1995-01-01	Stafford
Headquarters	1	888665555	1981-06-19	Houston

```
CREATE ROW TYPE DepartmentType,
(
DName          Varchar(15),
DNumber        Int,
MgrSSN         Char(9),
MgrStartDate   Date,
DLocations     Varchar(15) Multiset
);
```

```
CREATE TABLE Department
      OF TYPE DepartmentType,
(PRIMARY KEY DNumber);
```

To find the locations of the Research department:

```
SELECT  L.DLocation
FROM    Department D, TABLE(D.DLocations) L
WHERE   D.DName = 'Research';
```

To count the locations of each department:

```
SELECT  DName, COUNT(DLocations)
FROM    Department
GROUP BY DName;
```

Comments:

- There are operations for union, intersection, list concatenation, and the like.
- Reference types are not allowed as values (see below).

Reference Types:

Object identity is recaptured via the notion of a reference type.

Example: Instead of using foreign keys, it is possible (and perhaps more natural) to use reference types:

Here is an example, using some types defined previously
(Address_Type, EmployeeType, DepartmentType):

```
CREATE ROW TYPE EmployeeType
(
Name         NameType,
SSN          Char(9)        NOT NULL,
BDate        Date
Address      AddressType,
Sex          Char,
Salary       Decimal(10,2),
Supervisor   Ref(EmployeeType),
DeptRef      Ref(DepartmentType)   NOT NULL
);

CREATE TABLE Employee
     OF TYPE EmployeeType,
(PRIMARY KEY SSN);
```

To access reference types, a C-style notation is used.

The following delivers a list of employee last names, the name of the department, and the last name of the supervisor.

```
SELECT Name..LName,
       DeptRef->Dname,
       Supervisor->Name..LName
FROM   Employee;
```

With reference types, the need for explicit keys in constructed types becomes less clear.

```
CREATE ROW TYPE ProjectType,
(
PName       Varchar(15)       NOT NULL,
PNumber     Int               NOT NULL,
PLocation   Varchar(15),
DNum        Int
);

CREATE TABLE Project
OF ProjectType,
(PRIMARY KEY Pnumber);

CREATE ROW TYPE WorksOnType,
(
EmployeeRef     Ref(EmployeeType) NOT NULL,
ProjectRef      Ref(ProjectType)  NOT NULL,
Hours           Decimal(3,1)
);

CREATE TABLE Works_On
OF WorksOnType,
(PRIMARY KEY EmployeeRef, ProjectRef);
```

Even in SQL:2003, multisets of reference types are not allowed.

Example: Suppose it is desired to collect the set of dependents for each employee as an attribute of the dependent relationship. Sadly, the following does not work.

```
CREATE ROW TYPE DependentType
(
EmployeeRef      Ref(EmployeeType)   NOT NULL,
DependentName NameType;              NOT NULL,
Sex                 Char,
BDate               Date,
Relationship       Varchar(8)
);

CREATE TABLE Dependent
OF DependentType,
(PRIMARY KEY EmployeeRef, DependentName);

CREATE ROW TYPE EmployeeType
(
Name          NameType,
... <other declarations here, same as before>
DeptRef       Ref(DepartmentType)   NOT NULL,
Dependents  Set(Ref(Dependent))
);

CREATE TABLE Employee
      OF TYPE EmployeeType,
(PRIMARY KEY SSN);
```

One could do the following:

```
CREATE ROW TYPE EmployeeType
(
Name          NameType,
... <other declarations here, same as before>
DeptRef  Ref(DepartmentType)   NOT NULL,
Dependents  DependentType  Multiset
);

CREATE TABLE Employee
      OF TYPE EmployeeType,
(PRIMARY KEY SSN);
```

However, now the Employee relation contains actual sets of tuples, rather than references to tuples which presumably live in the Dependent relation. This leads to two options.

1. Do away with the Dependent relation entirely.
   - This leads to navigation problems similar to those encountered in the legacy hierarchical model.
   - To process all dependents, one must traverse the employee relation and then examine the Dependents attribute of each tuple.

2. Keep both the Dependent relation and the set of dependents in the Employee relation.
   - This leads to an update and consistency nightmare, since there are now two copies of each dependent tuple.

Explicit identity:

In object-oriented programming languages, it is usually the case that object identity is hidden. In object-oriented database situations, this need not be the case.

Here is an example in which an explicit primary key and object identifier called ID is generated by the system:

```
CREATE ROW TYPE EmployeeType
(
ID          Ref(EmployeeType)   NOT NULL,
Name        NameType,
SSN         Char(9);            NOT NULL,
BDate       Date;
Address     AddressType,
Sex         Char,
Salary      Decimal(10,2),
Supervisor  Ref(EmployeeType),
DeptRef     Ref(DepartmentType)   NOT NULL
);

CREATE TABLE Employee
     OF TYPE EmployeeType
VALUES FOR ID ARE SYSTEM GENERATED;
(PRIMARY KEY ID);
```

Subtypes and Inheritance:

Example: Define a special type of Employee called Manager. A tuple of manager type has all of the fields of a tuple of EmployeeType, plus the field DeptSupervised.

```
CREATE ROW TYPE EmployeeType
(
ID              Ref(EmployeeType)   NOT NULL,
...
...
DeptRef         Ref(DepartmentType)   NOT NULL
);

CREATE ROW TYPE ManagerType
UNDER EmployeeType
(
DeptSupervised     DepartmentType;
);

CREATE TABLE Employee
      OF TYPE EmployeeType
VALUES FOR ID ARE SYSTEM GENERATED;
(PRIMARY KEY ID);

CREATE TABLE Manager
      OF TYPE ManagerType
       UNDER Employee;
```

Behavior of subtypes and inheritance:

Insertion:

- Insertion into the Manager table automatically inserts into the Employee table.

- Insertion into the Employee table has no effect on the Manager table.

Deletion:

- Deletion from the Manager table automatically deletes the corresponding tuple from the Employee table as well!!!

- Deletion from the Employee table also deletes any corresponding tuples from the Manager table.

Update:

- Any update of an attribute other than DeptSupervised affects both tables.

- An update to DeptSupervised affects only the Manager table.

Consequences:

- How does one promote Lou to be a manager?
- How does one remove Lou as a manager, while leaving him as an employee?

Answers:

It is necessary to delete the “Lou” tuple from the old relation(s), and then insert a new tuple.

The utility of this construct is thus not very clear.

User-Defined Types:

- Row types are not encapsulated. Any operators may manipulate them.
- SQL:1999 also supports encapsulated types, with associated functions (methods).
- Values for attributes may not be altered, or even read, except by using the methods.

Example: A name type with a function which returns the whole name as one string:

```
CREATE TYPE NameADT
(
LName    Varchar(15),
FName    Varchar(15),
MInit    Char,
NameLFM FnLFM,
NameFML FnFML,
FUNCTION NameLFM(:n NameADT)
             RETURNS Varchar(35);
   :s VarChar(31);
   BEGIN
      :s := STRCAT(:n.FName, ‘ ‘);
      :s := STRCAT(:s, :n.MInit);
      :s := STRCAT(:s, ‘. ‘);
      :s := STRCAT(:s, :n.LName);
      RETURN(:s);
   END;
);
```

The type also includes certain built-in functions:

- A *constructor* function which generates a new, null object of this type.

- One *observer* function for each attribute, which allows one to examine the value of that attribute. These typically have the A.B format, for compatibility with other SQL data types.

- One *mutator* function for each attribute, which allows one to change the value of that attribute.

Privileges may be granted to these functions, so that, for example, some users may be able to look at the values of attributes without changing them.

The privilege scheme follows the grant/revoke format.

- External functions (written in some other programming language) are also possible.

Other SQL:1999 features:

- Recursive queries (*e.g.,* Ancestor);

- Triggers (one action forces the execution of another)

- New data types:
- Boolean
- CLOB (Character large object)
- BLOB (Binary Large Object)

- User-defined subtypes
- Example: Weight as a subtype of Int
- Problem: A very ugly and strict typecasting system.

Other SQL:2003 features:

- SQL/XML

- New data types:
    - Bigint
    - Multiset
    - XML
-
- Table functions

- CREATE TABLE LIKE

- Merge

- Sequence generators