SQL-99

Contains slides made by

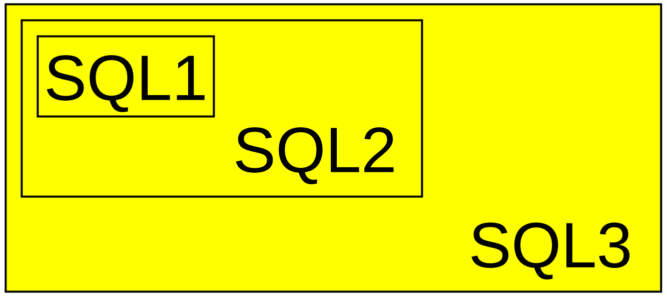
Naci Akkøk, Pål Halvorsen, Arthur M. Keller, Vera Goebel

Overview

- SQL-99
  - user-defined types (UDTs)
  - methods for UDTs
  - declarations
  - references
  - operations

SQL Development

- SQL-86 → SQL-89 (SQL1)
- SQL-92 (SQL2):
  - executable DDL
  - outer join
  - cascaded update and delete
  - temporary table
  - set operations: union, intersection, difference
  - domain definitions in schemes
  - new built-in data types
  - dynamic SQL via PREPARE and EXECUTE statements
  - transaction consistency levels
  - deferred constraint locking
  - scrolled cursors
  - SQL diagnostics
- SQL-99 (SQL3): SQL-92 + extensions



NOTE 1:
SQL-99 contains the functions from SQL-92

NOTE 2:
we are focusing on some of these extensions today

SQL-99:
User-Defined Types

- Relations are still the core abstraction.
  Classes from ODL are “translated” into
  User-Defined Types (UDTs).

- SQL allows *UDT*s that play a dual role:
  1. They can be the types of relations (tables), i.e.,
     the type of their tuple (sometimes called a *row type*).
  2. They can be the type of an attribute in a relation.

SQL-99: Defining UDTs

- UDTs are analogous to ODL class declarations, but
  - key declarations are not part of the UDT – it is part of the table declaration
  - relationships are not properties – it must be represented by own tables

- A simple form of UDTs consists of
  - keyword `CREATE TYPE`
  - name
  - keyword `AS`
  - a parenthesized, comma-separated list of attribute-type pair
  - a comma-separated list of methods including argument and return type

- Syntax: `CREATE TYPE` *T* `AS(` < list of attribute-type pairs>
  `)`
  < list of methods>`;`

SQL-99:
Bar–Beer–Sell (BBS) Example: Defining UDTs

```
CREATE TYPE BarType AS
(
    name CHAR(20),
    addr CHAR(20)
);

CREATE TYPE BeerType AS
(
    name CHAR(20),
    manf CHAR(20)
);
```

NOTE 1:
keyword `CREATE TYPE`

NOTE 2:
a name of the UDT

NOTE 3:
keyword `AS`

NOTE 4:
parenthesized, comma-separated list of attribute-type pair

NOTE 5:
additionally we may have methods (will be added later)

SQL-99:
Creating Tables – I

- UDTs do not declare relations, but we might declare one (or more) relations whose tuples are the type of an UDT.
- A simple form of relations defined from a UDT consists of
  - keyword `CREATE TABLE`
  - name
  - keyword `OF`
  - name of UDT
- Syntax: `CREATE TABLE` *S* `OF` *T*
- A relation must declare a key as keys are not part of the UDT
- Syntax:

```
CREATE TABLE S OF T (
      PRTIMARY KEY(<list of key attributes>)
);
```

SQL-99:
BBS Example: Creating Tables

```
CREATE TYPE BarType AS
(           name CHAR(20),
            addr CHAR(20)
);
CREATE TYPE BeerType AS
(           name CHAR(20),
            manf CHAR(20)
);
CREATE TABLE Bars OF BarType
(     PRIMARY KEY (name)
);
CREATE TABLE Beers OF BeerType
(     PRIMARY KEY (name)
);
```

NOTE 1:
keyword `OF` and name of UDTs are used in place of element lists in `CREATE TABLE` statements

NOTE 2:
primary key is defined by the keywords `PRIMARY KEY` followed by a parenthesized, comma-separated list of key attributes

NOTE 3:
other elements of a table declaration may be added similarly, e.g., foreign keys, tuple based constrains, etc., which apply to this table only, not UDT

NOTE 4:
usually we have one relation per UDT, but we may have several

SQL-99:
References – I

- If a table is created using a UDT, we may have a *reference column* serving as an *identity*
  - it can serve as a primary key
  - can be a system generated, unique value

- To refer to tuples in a table with a referece column, an attribute may have as type a reference to another type.
  - If `T` is a UDT, then `REF(T)` is the type of a reference to a `T` object.
  - Unlike OODBS, references are values that can be seen by queries.

SQL-99: References – II

- For a reference attribute to be able to refer to a relation, the relation must be *referenceable*.
- A table is made referenceable by including a clause in the *table declaration* (this not part of the UDT).
- Syntax: `REF IS` <attribute name> <generated>
- The <attribute name> will serve as the object identifier
- The <generated> is telling how the id is generated, either:

1. `SYSTEM GENERATED`, the DBMS maintains a unique value in this column for each tuple
2. `DERIVED`, the DBMS uses the primary key of the relation to produce unique values for each tuple

SQL-99:
BBS Example: References – I

```
CREATE TYPE BarType AS (
    name CHAR(20),
    addr CHAR(20),
    bestSeller REF(BeerType) SCOPE Beers
);
CREATE TYPE BeerType AS (
    name CHAR(20),
    manf CHAR(20)
);
CREATE TABLE Bars OF BarType (
    PRIMARY KEY (name)
);
CREATE TABLE Beers OF BeerType (
    REF IS beerID SYSTEM GENERATED
    PRIMARY KEY (name)
);
```

NOTE 1:
`bestSeller` is a reference to a `BeerType` object

NOTE 2:
`bestSeller` must refer to objects in the `Beers` relation whose type is `BeerType`

NOTE 3:
the relation `Beers` must be referenceable

NOTE 4:
the “ID” is system generated

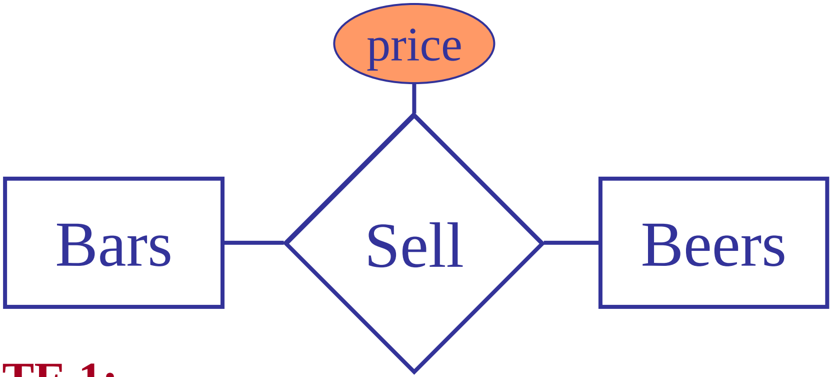
NOTE 5:
only single references is possible this way, not set

SQL-99:
BBS Example: References – II

```
CREATE TYPE BarType AS
(   name CHAR(20),
    addr CHAR(20)
);

CREATE TYPE BeerType AS
(   name CHAR(20),
    manf CHAR(20)
);

CREATE TYPE MenuType AS
(   bar    REF(BarType),
    beer   REF(BeerType)
);
```



NOTE 1:
Bars sell beers (and beers are sold at bars), but we cannot directly represent this SET relationship in type bar and beer as in ODL

NOTE 2:
we need a separate relation to represent such sets, with references to the two types (possibly with a scope)

NOTE 3:
if the relationship has properties as price, we even in ODL we must have a separate class

SQL-99: References – III

- References may be given a *scope*, i.e., the name of the relation to whose tuples are referred to
- Syntax: `S REF(T) SCOPE R --` (an attribute `S` of type `REF(T)` referring to a tuple in relation `R`)
- If no scope is given, the reference can go to tuples of any relation of type `T`
- Example

```
CREATE TYPE MenuType AS (
     bar   REF(BarType) SCOPE Bars,
     beer  REF(BeerType) SCOPE Beers,
     price FLOAT
  );
```

NOTE:
`Bars` and `Beers` are relations defined using the `BarType` and `BeerType`, respectively

SQL-99: Methods – I

- UDTs can have associated methods. They work on objects whose type is the UDT (applied on tuples)

- Similar to *Persistent Stored Modules* (PSM), which are general purpose functions allowed to be stored together with the schema and used in SQL (described in Chapter 8), but methods are
  - declared in the UDT using a `METHOD` clause
  - defined separately in a `CREATE METHOD` statement

SQL-99: Methods – II

- There is a special tuple variable `SELF` that refers to that object to which the method is applied, i.e.,
  can use `SELF.a` to access the object attribute `a`
- In the method declaration, arguments need a mode, like `IN`, `OUT`, or `INOUT`, but the mode does not appear in the definition.
- Many methods will take no arguments (relying on `SELF`)
- All methods must return a value of some type
- A method is applied using "dot", e.g., `t.updatePrice(...)`

SQL-99:
Methods: Declaration

- A declaration of a method for a UDT consists of
  - keyword `METHOD`
  - name of the method
  - keyword `RETURNS`
  - the return type

- Declaration syntax:

  `METHOD` <name> `RETURNS` <return type>;

SQL-99: Methods: Definitions

- A definition of a method for a UDT consists of
  - keywords `CREATE METHOD`
  - name of the method including arguments and their type
  - keyword `RETURNS` and the return type
  - keyword `FOR` and the name of the UDT in which the method is declared
  - body of the method (as PSM functions)
- Definition syntax (body):

```
CREATE METHOD <name> RETURNS <return type> FOR <name of UDT>
BEGIN
        <method body>
END
```

SQL-99 Example Methods: Declarations and Definitions – I

- Example:

```
CREATE TYPE MenuType AS
(  bar   REF(BarType) SCOPE Bars,
   beer  REF(BeerType) SCOPE Beers,
   price FLOAT
)
METHOD updatePrice
(  IN p  float
)
RETURNS BOOLEAN;

CREATE METHOD updatePrice
(  p  float
)
RETURNS BOOLEAN FOR MenuType
BEGIN
       <body>
END;
```

NOTE 1:
Declaration in UDT

NOTE 2:
Definition separately, outside the UDT

NOTE 3:
parameters, mode only in declaration

NOTE 4:
the body is written in the same language as the PSM functions, e.g., SQL/PSM used in the book

NOTE 5:
can use built-in `SELF`

NOTE 6:
p necessary, as it is used to change the value of the price attribute, e.g., `p` is added to `SELF.price`

SQL-99:
New Operations

- All appropriate SQL operations applying to tables defined using UDTs are allowed, but there are also some new features:
  - using references
  - accessing UDT attributes
  - creating UDT objects
  - order relationships

SQL-99 – New Operations: Following References – I

- If *x* is a value of type `REF(T)`, then `x` refers to some tuple *t* of type `T`
- The attributes of tuple *t* can be obtained by using the `->` operator
  - essentially as in C
  - if *x* is a reference to tuple *t* and *a* is an attribute in *t*, then `x->a` is the value if attribute *a* in *t*
- Example: Find the beers served at “Joe’s”

```
    SELECT beer->name
    FROM Sells
    WHERE bar->name = 'Joe''s';
```

NOTE 1:
`Sells` is a table with `menuType` as type

NOTE 2:
the attributes of a tuple is accessed using the `->` operator

NOTE 3:
single-quoted strings

SQL-99 – New Operations: Following References – II

- The tuple `t` can be obtained by using the `DEREF` operator if `x` is a reference
- Example:
  Find the bars (all attributes) serving “Bud”

```
SELECT DEREF(bar)
FROM Sells
WHERE beer->name = 'Bud';
```

NOTE 1:
`Bar` is reference to a tuple in table `Bars`

NOTE 2:
`DEREF(bar)` gets the referenced tuples

```
SELECT bar
From Sells
Where beer->name = ‘Bud’;
```

NOTE 3:
`SELECT bar`, without `DEREF`, would return only system-generated values serving as the IDs of the tuples – not the information in the tuples themselves

SQL-99 – New Operations: Accessing UDT Attributes

- A tuple defined by a UDT is analogous to an object – not a list of components corresponding to the attributes of a UDT
- Example:
  the relation `bars` is defined using the UDT `barType`
  - this UDT has two attributes, i.e., `name` and `addr`,
  - a tuple `t` in `bars` has only one component, i.e., the object itself
- Every UDT has implicitly defined *observer methods* for each attribute.
  - `x()` is the name of the observer method for an attribute `x`
  - returns the value of attribute `x` in the UDT
  - is applied as all other methods on this UDT, i.e., using “dot”
  - if `t` is of UDT type `T` and `x` is an attribute of `T`, then `t.x()` is the value of `x` in `t`

SQL-99 – New Operations: Creating Data Elements

- *Generator methods* create objects of UDT type `T`:
  - same name as the UDT itself, i.e., `T()`
  - takes no arguments
  - invoked without being applied to objects
  - returns an object of type `T` with no values in the various components

SQL-99 – New Operations: Updating Data Elements

- *Mutator methods* update attributes in objects of UDT type `T`:
  - for each attribute `x` in `T`, there is a mutator method `x(v)`
  - when applied to an object `T`, `x(v)` changes the value of `x` to `v`
- Note: both *mutator* (`x(v)`) and *observer* (`x()`) methods have the same name, but only a *mutator* method has a parameter

SQL-99 – New Operations Example: Creating and Updating Data Elements

- Example:

  PSM procedure inserting new bars into the `Bars` relation

```
CREATE PROCEDURE insertBar (
      IN n CHAR(20),
      IN a CHAR(20)
)
DECLARE newBar BarType;
BEGIN
      SET newBar = BarType();
      newBar.name(n);
      newBar.addr(a);
      INSERT INTO Bars VALUES(newBar);
END;
```

NOTE 1: the UDT `BarType` has two attributes, i.e., `name` and `addr`, which are parameters

NOTE 2: declaration of a variable of type `BarType`

NOTE 3: `newBar` is assigned a value of an empty `BarType` object using the `BarType()` generator method

NOTE 4: we apply mutator methods for the attributes in `BarType` UDT, i.e, `name(n)` and `addr(a)`, on the `newBar` object using “dot” notation

NOTE 5: we insert the object newBar of type BarType into the table Bars. NB! Simpler ways may exist to insert objects

SQL-99 – New Operations: Comparing Objects – I

- There are no operations to compare two objects whose type is some UDT *by default*, i.e, we cannot
  - eliminate duplicates
  - use `WHERE` clauses
  - use `ORDER BY` clauses
- SQL-99 allows to specify comparison or ordering using `CREATE ORDERING` statements for UDTs

SQL-99 – New Operations: Comparing Objects – II

- Equality for an UDT named T:

  ```
  CREATE ORDERING FOR T EQUALS ONLY BY STATE
  ```

  (equal if all corresponding components have the same value)

- Apply all comparison operators for an UDT named T:

  ```
  CREATE ORDERING FOR T ORDERING FULL BY RELATIVE
  WITH F
  ```

  (all comparison operators - <, <=, >, >=, =, and <> - may be applied on two objects using an integer function F which *must be implemented* separately)

  Example: $<$: $F(x_1, x_2) < 0$ if $x_1 < x_2$
  $>$: $F(x_1, x_2) > 0$ if $x_1 > x_2$
  $=$: $F(x_1, x_2) = 0$ if $x_1 = x_2$
  etc.

SQL-99: UDTs (revisited) – Type of a Column

- A UDT can also be the type of a column.
- Example:
  Let’s create an address type to use in bars (replacing the string)

```
CREATE TYPE AddrType AS (
    street CHAR(30),
    city CHAR(20),
    zip INTEGER
);

CREATE TYPE BarType AS (
    name CHAR(20),
    addr AddrType
);
```

NOTE 1:
the `addr` attribute of the UDT `BarType` has changed to an own UDT – composite types

- Problem: how can we sort all bars alphabetically?
- Need a way to compare the objects

SQL-99 – New Operations: Comparing Objects (revisited) – lexicographical ordering

- First, the UDT `AddrType`:

```
CREATE ORDERING FOR AddrType
ORDER FULL BY RELATIVE WITH AddrComp;

CREATE FUNCTION AddrComp (
     IN x1 AddrType,
     IN x2 AddrType
) RETURNS INTEGER
IF           x1.city() < x2.city()       THEN RETURN(-1)
ELSEIF       x1.city() > x2.city()       THEN RETURN(1)
ELSEIF       x1.street() < x2.street()   THEN RETURN(-1)
ELSEIF       x1.street() > x2.street()   THEN RETURN(1)
ELSE RETURN(0)
END IF;
```

NOTE 1: all comparison operators may be applied

NOTE 2: comparison is performed in function `AddrComp`

NOTE 3: we first compare `city`, if equal we look at `street`

NOTE 5: if `x1.a < x2.a` **return -1**

NOTE 6: if all `x1.a = x2.a` **return 0**

NOTE 7: has to use observer methods to get value

NOTE 4: if `x1.a > x2.a` **return 1**

SQL-99 – New Operations:
Comparing Objects (revisited) – lexicographical ordering

- Second, the UDT `BarType`:

```
CREATE ORDERING FOR BarType
ORDER FULL BY RELATIVE WITH BarComp;
```

NOTE 1:
all comparison operators may be applied

```
CREATE FUNCTION BarComp (
    IN x1 BarType,
    IN x2 BarType
) RETURNS INTEGER
IF        x1.name() < x2.name()  THEN RETURN(-1)
ELSEIF    x1.name() > x2.name()  THEN RETURN(1)
ELSEIF    x1.addr() < x2.addr()  THEN RETURN(-1)
ELSEIF    x1.addr() > x2.addr()  THEN RETURN(1)
ELSE RETURN(0)
END IF;
```

NOTE 2:
we first compare `name`, if equal we look at `addr`

NOTE 3:
as the `addr` itself is a UDT, it will again use the its own comparison function `AddrComp`

SQL-99 BBS Example: Using Methods – I

- Example:

  add method for retrieving price including tip

```
CREATE TYPE MenuType AS (
     bar   REF(BarType) SCOPE Bars,
     beer  REF(BeerType) SCOPE Beers,
     price FLOAT
)
METHOD priceTip (IN p float)
RETURNS FLOAT;

CREATE METHOD priceTip (p float)
RETURNS FLOAT FOR MenuType
BEGIN
     RETURN (1 + p) *    SELF.price;
END;
CREATE TABLE Sells OF MenuType;
```

NOTE 1:
tip is given in percent

NOTE 2:
the value returned is the price, found by using `SELF`, increased by p percent (`FLOAT`)

NOTE 3:
create table `sells` from UDT `MenuType`

SQL-99:
BBS Example: Using Methods – II

- Example:

  find beers and price with and without tip on “Joe’s” bar

```
             2   4
SELECT s.beer->name(),s.price(),s.priceTip(0.15)
FROM Sells s1          4         5
           2   4    3
WHERE s.bar->name() = ‘Joe’’s’
```

NOTE 1:
Renaming allowed

NOTE 2:
since `beer` and `bar` are references we have to use the `->` operator

NOTE 3:
`bar` is a reference to an object whose type is a UDT. However, the value returned by the `name()` observer method is a text string. Thus, NO comparison operators have to be defined – use only traditional text comparison

NOTE 4:
since `Sells` objects have a UDT type and `beer` and `bar` are references to objects whose types are UDTs, we must use observer methods to retrieve the attribute values

NOTE 5:
methods are applied using “dot” notation