Object-Oriented DBMS’s

- ODMG = Object Data Management Group: an OO standard for databases.
- ODL = Object Description Language: design in the OO style.
- OQL = Object Query Language: queries an OO database with an ODL schema, in a manner similar to SQL.

ODMG Compliant Databases

- Jasmine (Computer Associates)
- TITANIUM (MicroDB Systems)
- ObjectStore (EXcelon)
- Objectivity (Objectivity)
- POET Object Server (POET Soft.)
- Versant ODBMS (Versant Corp.)

OO databases account for less than 2% of database market.

ODL Overview

- Class declarations (interfaces).
- Interface includes:
  1. Name for the interface.
  2. Key declaration(s), which are optional.
  3. Extent declaration = name for the set of currently existing objects of a class.
  4. Element declarations. An element is an attribute, a relationship, or a method.

ODL Class Declarations

```java
interface <name> {
  elements = attributes, relationships, methods
}
```

- Element Declarations
  ```java
  attribute <type> <name>;
  relationship <rangetype> <name>;
  ```

- Relationships involve objects; attributes involve non-object values, e.g., integers.
Method Example

float gpa(in name) raises(noGrades)
- float = return type.
- in: indicates the argument (a student name) is read-only.
- Other options: out, inout.
- noGrades is an exception that can be raised by method gpa.

ODL Relationships

- Only binary relations supported.
- Multiway relationships require a "connecting" class, as discussed for E/R model.
- Relationships come in inverse pairs.
- Example: Sells between beers and bars is represented by a relationship in bars, giving the beers sold, and a relationship in beers giving the bars that sell it.

More ODL Relationships

- Many-many relationships have a set type (called a collection type) in each direction.
- Many-one relationships have a set type for the one, and a simple class name for the many.
- One-one relations have classes for both.

Beers-Bars-Drinkers Example

interface Beers {
  attribute string name;
  attribute string manf;
  relationship Set<Bars> servedAt inverse Bars::serves;
  relationship Set<Drinkers> fans inverse Drinkers::likes;
}
- An element from another class is indicated by <class>::
- Form a set type with Set<type>.

Bars ODL

interface Bars {
  attribute string name;
  attribute Struct Addr (string street, string city, int zip) address;
  attribute Enum Lic (full, beer, none) licenseType;
  relationship Set<Drinkers> customers inverse Drinkers::frequents;
  relationship Set<Beers> serves inverse Beers::servedAt;
}
- Structured types have names and bracketed lists of field-type pairs.
- Enumerated types have names and bracketed lists of values.

Drinkers ODL

interface Drinkers {
  attribute string name;
  attribute Struct Bars::Addr address;
  relationship Set<Beers> likes inverse Beers::fans;
  relationship Set<Bars> frequents inverse Bars::customers;
}
- Note reuse of Addr type.
ODL Type System

- Basic types: int, real/float, string, enumerated types, and classes.
- Type constructors: Struct for structures and four collection types: Set, Bag, List, and Array.
- Limitation on nesting:
  - Relationships: class – collection
  - Attributes: basic – struct – collection

Many-One Relationships

- Don't use a collection type for relationship in the "many" class.
- Drinkers have favorite beers:
  ```java
  interface Drinkers {
    ...
    relationship Beers favoriteBeer
    inverse Beers::realFans;
    ...
  }
  ```
- Also add to Beers:
  ```java
  relationship Set<Drinkers> realFans
  inverse Drinkers::favoriteBeer;
  ```

Example: Multiway Relationship

- Consider a 3-way relationship bars-beers-prices. We have to create a connecting class BBP.
  ```java
  interface Prices {
    attribute real price;
    relationship Set<BBP> toBBP
    inverse BBP::thePrice;
  }
  interface BBP {
    relationship Bars theBar inverse ...
    relationship Beers theBeer inverse ...
    relationship Prices thePrice
    inverse Prices::toBBP;
  }
  ```

Example (contd.)

- Inverses for theBar, theBeer must be added to Bars, Beers.
- Better in this special case: make no Prices class; make price an attribute of BBP.
- Notice that keys are optional.
  - BBP has no key, yet is not "weak." Object identity suffices to distinguish different BBP objects.

Roles in ODL

- Names of relationships handle roles. E.g. Spouses and Drinking Buddies
  ```java
  interface Drinkers {
    attribute string name;
    attribute Struct Bars::Addr address;
    relationship Set<Beers> likes
    inverse Beers::fans;
    relationship Set<Bars> frequents
    inverse Bars::customers;
    relationship Drinkers husband
    inverse wife;
    relationship Drinkers wife
    inverse husband;
    relationship Set<Drinkers> buddies
    inverse buddies;
  }
  ```
- Notice that Drinkers:: is optional when the inverse is a relationship of the same class.

Object-Relational Systems

- Object-oriented ideas enter the relational world.
  - Keep relation as the fundamental abstraction.
- Compare with object-oriented DBMS, which use the class as the fundamental abstraction and tacks on relations as one of many types.
Motivation

- Allow DBMS's to deal with specialized types – maps, signals, images, etc. – with their own specialized methods.
- Supports specialized methods even on conventional relational data.
- Supports structure more complex than flat files.

Issues

1. Basic ideas from SQL standards documents.
2. Use Oracle 8i/9i notation when similar.
3. Introduce some new concepts from Oracle.

User-Defined Types

- SQL allows UDT's that play a dual role:
  1. They can be the types of relations; i.e., the type of their tuple.
     • Sometimes called a row type.
  2. They can be the type of an attribute in a relation.

Defining UDT's – Example in Oracle Syntax

CREATE TYPE BarType AS OBJECT (
  name CHAR(20) UNIQUE,
  addr CHAR(20)
); /

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

CREATE TYPE MenuType AS OBJECT (
  bar REF BarType,
  beer REF BeerType,
  price FLOAT
); /

Creating Tables

- Type declarations do not create tables.
- They are used in place of element lists in CREATE TABLE statements.

CREATE TABLE Bars OF BarType;
CREATE TABLE Beers OF BeerType;
CREATE TABLE Sells OF MenuType;

Example (contd.)

CREATE TYPE MenuType AS OBJECT (
  bar REF BarType,
  beer REF BeerType,
  price FLOAT
); /

- In Oracle, type definitions must be followed by a slash (/) in order to get them to compile.
- The SQL standard is similar, but OBJECT is not used after AS.
Values of User-Defined Types – Oracle Approach

- Each UDT has a type constructor of the same name.
- Values of that type are the values of its fields wrapped in the constructor.

```sql
SELECT * FROM Bars;
```

produces values such as

```sql
BarType('Spoon', 'Wells')
```

- Note: Oracle doesn't show the type constructor for some queries.

Accessing Fields of an Object – Oracle Approach

- The dot operator works as expected.
- Thus, if we want the bar name and address without the constructor:

```sql
SELECT bb.name, bb.addr
FROM Bars bb;
```

- The alias bb is not always necessary but it's better to use always use it.
- SQL standard: Same idea, but the attribute is treated as a generator method, e.g., `bb.name()`.

Inserting Values – Oracle Approach

- Use INSERT in Oracle, but wrap the inserted object in its type-constructor.

```sql
INSERT INTO Bars VALUES(
    BarType('Spoon', 'Wells')
);
```

- SQL standard involves generator and mutator methods; see text.

Types for Columns

- A UDT can also be the type of a column.
- Example: create an address type for use with bars and drinkers.

```sql
CREATE TYPE AddrType AS OBJECT (  
    street CHAR(30),  
    city CHAR(20),  
    zip INT  
);
```

Types for Columns (contd.)

- We can then create a table of drinkers that includes their name, address, and favorite beer.
- The beer is included as a beer object, which unnormalizes the relation but is legal.

```sql
CREATE TABLE Drinker (  
    name CHAR(30),  
    addr AddrType,  
    favBeer BeerType  
);
```

Need to Use Aliases

- If you access an attribute whose type is an object type, you must use an alias for the relation. E.g.,

```sql
SELECT favBeer.name
FROM Drinker;
```

will not work in Oracle; neither will:

```sql
SELECT Drinker.favBeer.name
FROM Drinker;
```

You have to say:

```sql
SELECT dd.favBeer.name
FROM Drinker dd;
```
XML

- eXtensible Markup Language
  - Semantic tags
  - Models text documents, semistructured data.

Schema for XML

- Document Type Definitions (DTD)
  - Grammar describing the legal nesting of tags.
- XML Schema
  - Standard in progress.

DTD Example

```
<!DOCTYPE Bars [  
  <!ELEMENT BARS (BAR*)>  
  <!ELEMENT BAR (NAME, BEER+)>  
  <!ELEMENT NAME (#PCDATA)>  
  <!ELEMENT BEER (NAME, PRICE)>  
  <!ELEMENT PRICE (#PCDATA)>  
]>```

DTD Explained

- `<ELEMENTS element-name (components)>`
  - Components are nested within the element in the specified order.
  - Multiplicity:
    - `*` means zero or more.
    - `+` means one or more.
    - `?` means zero or one.
  - `#PCDATA` – parsed character data.

Other XML Features

- Attribute lists
- References
- Standalone XML

XML Document Example

```
<?XML VERSION = "1.0" STANDALONE="no"?>
<!DOCTYPE Bars SYSTEM "bar.dtd">
<BARS>
  <BAR>
    <NAME>Spoon</NAME>
    <BEER>
      <NAME>Bud</NAME>
      <PRICE>3.5</PRICE>
    </BEER>
    <BEER>
      <NAME>Guinness</NAME>
      <PRICE>5.5</PRICE>
    </BEER>
  </BAR>
</BARS>```