From complex values to objects

A database perspective

Guillaume Raschia — Université de Nantes

Last update: December 12, 2012

The eNF^2 data model

eNF^2 = Extended NF^2 Model

- Extend NF^2 model by introducing
  - various **type constructors** and
  - allowing their **free combination**
- Type constructors:
  - `set` $\{.\}$: create a set type of nested type
  - `tuple` $\langle . \rangle$: tuple type of nested type
  - `list` $(.)$: list type of nested type
  - `bag` $\{\!|.|\!\}$: bag—multi-set—type of nested type
  - `array` $[.]_n$: array type of nested type
  - `map` $[./.]$: key/value dictionary type of nested types
- First two are already available in RM and NF^2

The eNF^2 data model (cont'd)

The Evolution of Data Models

b.t.w. of sort comparison

- Relational Model $\tau := \langle A_1 : \text{dom}, \ldots, A_k : \text{dom} \rangle$
- NF^2 $\tau := \text{dom} \mid \langle A_1 : \tau, \ldots, A_k : \tau \rangle \mid \{\tau\}$
- eNF^2

$$\tau := \text{dom} \mid \langle A_1 : \tau, \ldots, A_k : \tau \rangle \mid \{\tau\} \mid (\tau) \mid [\tau]_n \mid \{\!|\tau|\!\} \mid [\tau/\tau]$$

Flavors by **restrictions**, such like nested relations for NF^2

Type constructors

- $\langle . \rangle$ $\{ . \}$ $(.)$ $[.]_n$ $\{\!| . |\!\}$ $[. / .]$ a.k.a. **Parametrizable Data Types**
- Construction based on '.', the **input data type**
- Define **own operations** for access and modification
- Similar to pre-defined parametrizable data types of programming languages
  - Generics in Java `java.util`
  - Templates in C++ STL
  - Type inference in OCaml

Comparison of type constructors

Type	Dupl.	Bounded	Order	Access by	Composite
Set $\{.\}$	✘	✘	✘	Iterator	✘
Bag $\{\!\|.\|\!\}$	✔	✘	✘	Iterator	✘
Map $[./.]$	✔	✘	✘	Key	✘
List $(.)$	✔	✘	✔	Position/Iter.	✘
Array $[.]_n$	✔	✔	✔	Index	✘
Tuple $\langle . \rangle$	✔	✔	✔	Name	✔

- All but tuple type constructors are **collection data types**
- Tuple type constructor is a **composite data type**

SQL ARRAY type constructor

- Introduced within SQL:1999

```
CREATE TABLE Contacts(
    Name          VARCHAR(40),
    PhoneNumbers  VARCHAR(20) ARRAY[4],
    Addresses     AddressType ARRAY[3] );
```

SQL ARRAY type constructor (cont'd)

- Array type constructor for record insertion
- Access to elements by **explicit position** [k]

```
INSERT INTO Contacts
VALUES( 'Doe',
        ARRAY['1234','5678'],
        ARRAY[ROW( '50 Otages', 'Nantes', '44000' )] );

UPDATE Contacts
  SET PhoneNumbers[3]='91011'
  WHERE Name='Doe';
```

SQL ARRAY type constructor (cont'd)

- Alternative access to element by **unnesting of collection**

```
SELECT Name, Tel.*
FROM Contacts,
     UNNEST( Contacts.PhoneNumbers ) WITH ORDINALITY
     AS Tel(Phone, Position)
WHERE Name='Doe';
```

- Further operations:
  - size `CARDINALITY()`
  - concatenation `||`

Object structure

- Complex value—state—conforms to object structure
- Type constructors are building blocks: tuple, set, list, array, bag, dictionary
- eNF^2 as the reference model
- Implementation within SQL3

Yet another popular restriction

- **Class** of sort τ following

$$\begin{aligned} \tau &:= \langle A_1 : \varrho, \dots, A_k : \varrho \rangle \\ \varrho &:= \mathsf{dom} \mid \langle A_1 : \varrho, \dots, A_k : \varrho \rangle \mid \{\varrho\} \mid (\varrho) \mid [\varrho]_n \mid \{\!|\varrho|\!\} \mid [\varrho/\varrho] \end{aligned}$$

- Objects are **instances** of classes, a.k.a. class members
- An object o satisfies sort τ of its class c
- Essentially `struct` in C Programming Language

User-Defined Type in SQL3

UDT's occur at two levels:

- Columns of relations
- Tuples of relations

```
CREATE TYPE AddressType AS ( Street CHAR(50),
                             City   CHAR(50),
                             Zip    CHAR(5) );

CREATE TYPE BarType     AS ( Name   CHAR(20),
                             Addr   AddressType );

CREATE TABLE Bars OF BarType ( PRIMARY KEY (Name) );
```

Encapsulated object vs. row

- `Bars` is unary: tuples are objects with 2 components
- Grant **access privilege** to components
- **Type constructor**

```
INSERT INTO Bars
VALUES BarType( 'Le Flesselles',
                AddressType( '50 Otages',
                             'Nantes',
                             '44000' ) );
```

Encapsulated object vs. row (cont'd)

- **Observer** $A()$ and **Mutator** $A(v)$ for each attribute A
- Calls to implicit *getters* and *setters*, **redefinition** allowed

```
UPDATE Bars
SET Bars.Addr.Street('Allée Flesselles')
WHERE Bars.Name = 'Le Flesselles';

SELECT B.Name, B.Addr FROM Bars B;
```

Excerpt of the result set:
BarType('Le Flesselles',
AddressType ('Allée Flesselles', 'Nantes', '44000'))

Object behavior

Method := signature + body

Operation that apply to objects of a type

- $f(x)$ is invoked by sending a message to object o: $o.f(3)$
- Method
  - returns single value (may be a collection)
  - is typically written in general-purpose PL
  - could have unexpactable **side-effect**
- Implementation within ODL and SQL3

Disclaimer

Insight into object behavior is out of the scope of this series of slides

Corollary: main focus is the **structural part**

A word on eNF^2 in Oracle

- Supports **majority** of standard features as part of its object-relational extension—since 8i
  - **Multi-set** type constructor as `NESTED TABLE` type
  - **Array** type constructor as `VARRAY` type
  - **Object** (and **Tuple**) type constructor as `OBJECT` type
- Uses different syntax than ANSI/ISO SQL standard...

Alternative languages

Definition of object structures

- DDL part of SQL3 — OR-Databases
- DDL part of [your favorite or-dbms] — OR-Databases
- Entity/Relationship (E/R) Model — Relational Databases
- Object Description Language (ODL) — OO-Databases
- Unified Modeling Language (UML) — OO-PL
- ...

ODMG ODL

Example

```
class Bar {
  attribute string                              name;
  attribute struct addr {string street,
                         string city,
                         int    zip}    address;
  attribute enum lic {full, beer, none} license;
  attribute set< string >               drinks;
}
```

- Primitive types: int, real, char, string, bool, and *enumeration*
- Composite type: *structure*
- Collection types: set, array, bag, list, and dictionary

Class hierarchy

- Reuse of class definition
- A subclass is a refinement of its superclass

Definition (Class hierarchy)

A class hierarchy $(\mathcal{C}, \sigma, \prec)$ has 3 components:

1. a set $\mathcal{C}$ of class names
2. types τ's associated with these classes: $\sigma(c) = \tau$
3. specification of the is-a relationship $\prec$ between classes

Subtyping relationship

A subtype **inherits** value—and behavior—of a predefined type

Definition (Subtyping)

Let $(\mathcal{C}, \sigma, \prec)$ be a class hierarchy; subtyping relationship is the smallest partial order $\leq$ over types σ satisfying:

1. dom $\leq$ dom
2. if $\tau_i \leq \tau_i'$, $1 \leq i \leq n$, then $\langle A_1 : \tau_1, \ldots, A_n : \tau_n, \ldots, A_{n+k} : \tau_{n+k} \rangle \leq \langle A_1 : \tau_1', \ldots, A_n : \tau_n' \rangle$
3. if $\tau_1 \leq \tau_2$, then $\{\tau_1\} \leq \{\tau_2\}$, $(\tau_1) \leq (\tau_2)$, $\{\!|\tau_1|\!\} \leq \{\!|\tau_2|\!\}$ and $[\tau_1]_n \leq [\tau_2]_n$
4. if $\tau_1 \leq \tau_3$ and $\tau_2 \leq \tau_4$, then $[\tau_1/\tau_2] \leq [\tau_3/\tau_4]$
5. for each τ, $\tau \leq \text{ANY}$ (i.e. ANY is the top of the hierarchy)

Covariance and contravariance

Subtyping follows from wider to narrower: **covariance** only

Definition (Covariance)

Typing rules preserve the ordering on $\leq$

About the map type constructor

$$
\begin{tikzcd}
A : \tau_1 \arrow[r, "{[a, b]}"] \arrow[dr, dashed] & B : \tau_2 \\
C : \tau_3 \arrow[u] \arrow[r, dashed] \arrow[ur, dashed] & D : \tau_4 \arrow[u]
\end{tikzcd}
$$

- **Contravariance** reverses the ordering: type safety in PL

Well-formed class hierarchy

Property

A class hierarchy $(\mathcal{C}, \sigma, \prec)$ is **well-formed** iff for each pair c_1, c_2 of classes, $c_1 \prec c_2$ implies $\sigma(c_1) \leq \sigma(c_2)$

Example

- $\sigma(\text{Person}) = \langle \text{name} \rangle$
- $\sigma(\text{Teacher}) = \langle \text{name}, \text{dpts} : \{\langle \text{id} \rangle\} \rangle$
- $\sigma(\text{Student}) = \langle \text{name}, \text{major}, \text{enrol} : \{\text{dom}\} \rangle$
- $\sigma(\text{Lecturer}) = \langle \text{name}, \text{dpts} : \{\langle \text{id}, \text{office} \rangle\}, \text{contacts} : [\text{dom}]_3 \rangle$
- $\sigma(\text{Tutor}) = \langle \text{name}, \text{dpts} : \{\langle \text{id} \rangle\}, \text{labs} : \{\!|\langle \text{day}, \text{room} \rangle|\!\} \rangle$

$\{$Student, Teacher$\} \prec$ Person, and $\{$Lecturer, Tutor$\} \prec$ Teacher

Subtyping within SQL

`UNDER` clause with `NOT FINAL` statement in the base type

```
CREATE TYPE PersonType AS (
    Name        VARCHAR(20)   NOT NULL,
    DateOfBirth DATE,
    Gender      CHAR)
NOT FINAL;

CREATE TYPE StudentType UNDER PersonType AS (
    StudentID   VARCHAR(10),
    Major       VARCHAR(20)
);

CREATE TABLE Student OF StudentType;
```

Multiple Inheritance

- **More than one superclass** vs. single inheritance
- The is-a relationship forms a directed acyclic graph (DAG)
- A subclass inherits state and behavior from **all** its superclasses
- Potential for ambiguity, e.g., fields with the same name
- The "*diamond problem*": $D \prec \{B, C\} \prec A$
- SQL does not support multiple inheritance of UDT's

Inheritance within ODL

```
class Person {
    attribute string    name;
    attribute character gender; }

class Teacher extends Person {...}
class Student extends Person {...}

class TeachingFellow extends Teacher, Student {
    attribute string    degree; }
```

- How many names and genders for a single TF ?!

Membership in a class hierarchy

Definition (Object assignment)

A function π mapping each name in $\mathcal{C}$ to a finite set of objets

- Proper extension of c: $\pi(c)$
- Set of database objects: $O = \{\pi(c) \mid c \in \mathcal{C}\}$
- **Extension** of c: $\pi^*(c) = \bigcup\{\pi(x) \mid x \in \mathcal{C} \ \wedge \ x \prec c\}$
- $\pi^*(c_1) \subseteq \pi^*(c_2)$ whenever $c_1 \prec c_2$

Definition (Substitutability principle)

Value of type S can be sustituted to value of its supertype T

Alternative memberships

Properties

- **Complete** assignment: $\pi^*(c) \neq \emptyset \rightarrow \pi(c) = \emptyset$
- **Disjoint** assignment: $\pi(c_1) \cap \pi(c_2) \neq \emptyset \rightarrow (c_1 \prec c_2 \vee c_2 \prec c_1)$

Each class may have direct subclasses with:

- complete vs. partial assignment
- disjoint vs. overlapping assignment

Extension in ODL

- **Extent** declaration: *named* set of objects of the same type
  - Class ∼ Schema of a relation
  - Extent ∼ Instance of a relation
- Optional **Key** declaration: unicity constraint

```
class Course ( extent Courses
               keys   id, (dept, title), (classroom, time) )
{...};

SELECT c.id, c.title FROM Courses c
WHERE c.dept='Computer Science';
```

- Object Query Language (OQL): SQL-like for pure object db's
- Alias for extent (c) is mandatory: typical class member

"Subtabling" within SQL

No native extension for types in SQL: create table for each UDT

Table inheritance!

```
CREATE TABLE Person OF PersonType;
CREATE TABLE Student OF StudentType UNDER Person;
```

- A `Person` row matches at most one `Student` row
- A `Student` row matches exactly one `Person` row
- Inherited columns are inserted only into `Person` table
- Delete `Student` row deletes matching `Person` row

"Subtabling" within SQL (cont'd)

- Default: retrieve the extension π^*(Person) with all subtable rows

```
SELECT P.Name FROM Person P;
```

- `ONLY` clause: retrieve the proper extension π(Person)

```
SELECT P.Name FROM ONLY (Person) P;
```

Open issues

Multiple-table inheritance ? Propagation of referential integrity constraints ? Index ? . . .

Basics of relational mapping

- Classes are all distinct tables
- Keys must be defined
- The three ways to cope with class hierarchy:
  1. E/R-style: one partial table by subclass with key+specific fields
  2. OO-style: one full table by subclass
  3. Null-style: all subclasses embedded within one single base table

Example

Person(<u>name</u>, gender)
Teacher(<u>name</u>, dpt)
Student(<u>name</u>, major)

Person(<u>name</u>, gender)
Teacher(<u>name</u>, gender, dpt)
Student(<u>name</u>, gender, major)

Person(<u>name</u>, gender, dpt, major)

Exercises 1/2

1. Definitions

eNF^2, Class, Class member, UDT, Observer, Mutator, Method, ODL, Class hierarchy, Subtyping, Covariance, Diamond problem, Substitutability principle, Extension, OQL

2. True or False?

i) eNF^2 dominates NF^2.

ii) Map is a composite type constructor.

iii) SQL implements eNF^2.

iv) Subtyping rules are covariant only.

v) $\pi(c) = \emptyset$ except for the leaves, is a full complete assignment.

vi) SQL Subtabling implements relational mapping in E/R-style.

Exercises 2/2

3. Misc

1. Exhibit a class hierarchy that is not well formed.
2. Give in ODL the `Person` class hierarchy of the example slide.

4. Problem

How many relations are required, using the OO-style mapping, if there is a 3-level hierarchy with out-degree 4, and that hierarchy is: (a) disjoint and complete at each level, (b) disjoint and partial at each level, and (c) overlapping and partial.

Object identity

- Persistent objects are given an **Object IDentifier** (OID)
- Used to manage *inter-object references*
- OID's are
  - **unique** among the set of objects stored in the DB
  - **immutable** even on update of the object value
  - **permanent** all along the object lifecycle
- OID's are not based on physical representation/storage of object (i.e., $\neq$ ROWID or TID, $\neq$ @object)

Ultimate object representation

Definition (Object)

An object is a pair (o, ϑ), with o being the OID and ϑ is the value

- Object identity is given by the OID
- Object value is not required to be unique

Values by example

- In the *class*-oriented restriction of eNF^2, values ϑ are
  - tuple-based complex values:
    $(o_1, \langle \text{title} : \text{'cs123'}, \text{desc} : \text{'...'} \rangle)$
    $(o_2, \langle \text{title} : \text{'cs987'}, \text{desc} : \text{'...'} \rangle)$
    $(o_3, \langle \text{name} : \text{'Doe'}, \text{major} : \text{'cs'}, \text{year} : \text{'junior'}, \text{enrol} : \{o_1, o_2\} \rangle)$
  - OID to achieve aliasing: (o_4, o_3)
  - *nil* for nullable reference: $(o_5, \textit{nil})$

Composition graph

Structural representation of an object as a labeled directed graph

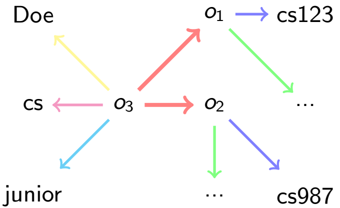
$$\text{struct}(o) := G(V, E)$$

where

- Vertices $V \subset O \cup \text{dom}$ are OID's and atomic values
- Edges $E \subseteq V \times \mathcal{A} \times V$ are labeled with symbols from $\mathcal{A}$, the set of field names
- Draw an edge (o_i, x) whenever $x \in \{o_j, a\}$ occurs in the value of o_i, a being an atomic value in dom

Composition graph (cont'd)

Example for object o_3



name major year
title desc enrol

Extend to a—cyclic—graph: *teacher* $\rightarrow$ *dpt* $\rightarrow$ *employees*

Statement

Object db is essentially **a huge persistent relational graph**

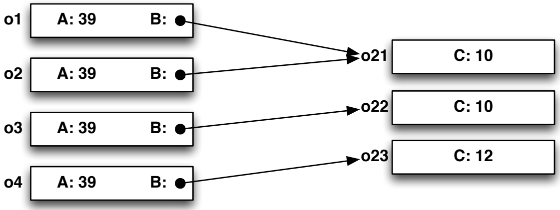
Equality

- **Identity** ($==$) is checked by means of OID's comparison
- Composition graph allows to compare two objects for equality
  - **Shallow equality** ($=$): graphs must be identical, including OID's
  - **Deep equality** ($=_*$): isomorphic graphs with different OID's but atomic values are equal

Properties

$$
\begin{aligned}
o_i == o_j &\longrightarrow o_i = o_j \\
o_i = o_j &\longrightarrow o_i =_* o_j
\end{aligned}
$$

Equality (cont'd)



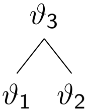
True	False
$o_1.B == o_2.B$	$o_1 == o_2$
$o_1 = o_2$	$o_1 = o_3$
$o_1 =_* o_3$	$o_1 =_* o_4$
$o_{21} = o_{22}$	$o_{21} == o_{22}$

Object expansion

Definition (Expansion)

Expansion of an object o, denoted $\text{expand}(o)$, is the—possibly infinite—tree obtained by replacing each object by its value recursively

Example of $\text{expand}(o_3)$



- Infinite expansion: cycle in the composition graph
- Deep equality can be checked from expansion traversal

Object persistence

- In OO-PL, objects are **transient**
- Persistence is orthogonal to object types—classes
- Many policies to come up with **persistent** objects:
  - Object creation and explicit declaration
  - Homogeneous collection by **extension**
  - **Reachability**: declare persistent objects by name, and the system makes persistent all reachable objects at any level of the composition graph

Reachability vs. extension

Names

1. Extension: all class members are designated by the name of a single collection
2. Reachability: objects are linked to the name of their **root of persistence**

On delete

1. Extension: must detect an orphan (?) and retrieve the object from its collection name
2. Reachability: garbage collecting when an object has a null in-degree

Types revisited

The family of *types* τ over the set $\mathcal{C}$ of class names is as follows:

$$\tau := \langle A_1 : \varrho, \ldots, A_k : \varrho \rangle$$
$$\varrho := \mathsf{dom} \mid c \mid \langle A_1 : \varrho, \ldots, A_k : \varrho \rangle \mid \{\varrho\} \mid (\varrho) \mid [\varrho]_n \mid \{\!|\varrho|\!\} \mid [\varrho/\varrho]$$

- dom may be refined into primitive types of the language
- c is any class name in $\mathcal{C}$

Types revisited (cont')

- Object assignment π is basically OID assignment
- $\textsc{any} \in \mathcal{C}$ is a—singular—type such that $\mathrm{dom}(\textsc{any}) = [\![O]\!]$
- Subtyping relationship is extended to:
  6. if $c_1 \prec c_2$, then $c_1 \leq c_2$
- Semantics of a class c is $\mathrm{dom}(c) = \pi^*(c) \cup \{nil\}$

SQL3 References

Principle

If τ is a type, then REF(τ) is a **type of references** to τ

- Weak translation of OID's into SQL world
- Unlike OID's, a REF is **visible** although it is gibberish

```
CREATE TYPE SellType AS (
  bar   REF(BarType)  SCOPE Bar,
  beer  REF(BeerType) SCOPE Beer,
  price FLOAT );
```

Following REF's and dereferencing

```
CREATE TABLE Sell OF SellType (
  REF IS sellID SYSTEM GENERATED,
  PRIMARY KEY (bar, beer) );

SELECT DEREF(s.beer) AS beer
FROM Sells s
WHERE s.bar->name = 'Le Flesselles';
```

- It would have required a join or nested query otherwise

Relationships

- Operate at the type system—class definition—level
- Connect entities/classes/types one with each other
- Binary relationships as **partial multi-valued functions**
- Decide for a direction: `contains` or `isIncluded` or both
- Prevent from redundancy: computed relationships
- Multiway relationships simulated by *connecting* classes

ODL example

```
class Sell {
  attribute    real price;
  relationship Bar  theBar;
  relationship Beer theBeer;
}
```

Multiplicity of relationships

- For binary relationships
  - One-one: `class/class`
  - Many-one: `set`[1]`/class`
  - Many-many: `set/set`
- 'One' means **at most one**
- Many-one variants: **aggregation** and **composition** (1..1)
- Weak class: id depends on master class id's ($\neq$ OID's)

```
class Employee (key (name, affiliated_to)) {  // weak class
  attribute    string        name;
  relationship Department    affiliated_to;   // one-one
  relationship set<Task>     assigned_to;     // many-one
  relationship list<Project> participates_in; // many-many
}
```

[1] or any collection type constructor.

Basic properties of partial multi-valued functions

- A function $f : X \to Y$ can be
  - **total**: domain of f is X
  - **injective**: for all a,b in domain of f, if $f(a) = f(b)$ then $a = b$
  - **surjective**: range (or codomain) of f is Y
  - **bijective**: both injective and surjective
- The **inverse** function $f^{-1} : Y \to X$ satisfies $f^{-1} \circ f = \mathrm{Id}$, with $\circ$ extended to sets of values

Comments

- Impact on design choices and further encoding
- Problems arise especially when mixed with inheritance

Implementation of the inverse function

- Two relationships are right inverses by means of
  - Both ways of the same link in graphical languages
  - `inverse` statement in ODL
  - Not supported in SQL3

Example in ODL

```
class Employee { ...
  relationship list<Project> participates_in
               inverse       Project::members; }
class Project { ...
  relationship set<Employee> members
               inverse       Employee::participates_in; }
```

Relational mapping

- Relationships are essentially all distinct tables
- Key fields both parts come into play
- Exceptions:
  - Supporting relationship of a weak class does not require a separate table
  - Inlining of aggregations and compositions
- Discussion about inlining each *-one relationship
- Implement relationships one way only

OQL features

- Query can include **path expressions** rather than joins:

```
SELECT s.beer.name, s.price
FROM Sell
WHERE s.bar.name='Le Flesselles';
```

- Alternative query

```
SELECT s.beer.name, s.price
FROM Bar b, b.beerSold s
WHERE b.name='Le Flesselles';
```

- Collections cannot be further extended by dot notation
- Collections can be part of the FROM clause

OQL features (cont'd)

- Result type is basically $\{\!|\langle.\rangle|\!\}$
- Complex result type can be constructed in query

```
SELECT DISTINCT struct( e.name,
                        projects:(
    SELECT p.projectId
    FROM e.participates_in AS p) )
FROM Employees AS e;
```

- Result type:

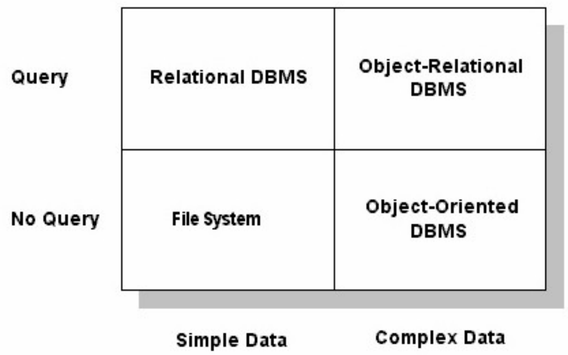
$$\{\langle \text{name}:\text{string}, \text{projects}:\{\!|\text{int}|\!\}\rangle\}$$

From Lineland to Spaceland

Object-Oriented paradigm brings to the—relational—data world

- Mashup of:
  1. Databases
  2. OO Programming Languages
  3. Conceptual/Semantic Modeling
- Practical approaches to contemporary issues
- Lack of strong mathematical foundations

The Matrix



M. Stonebraker: *Object-Relational DBMS: The Next Great Wave*, MK, 1998

15 years later, OO-DBMS in South-East quadrant is questionable

Impedance Mismatch revisited

Find a sunset picture taken within a coastal zone by a professional photographer

```
SELECT p.id
FROM slides p, area a, a.landmarks l
WHERE sunset (p.picture) AND
      p.owner.occupation = 'photographer' AND
      a.type = 'coastal' AND
      contains (p.caption, l.name) ;
```

- User-defined functions: `sunset()` `contains()`
- Path expression: `P.owner.occupation`
- Collection as table: `area.landmarks`

The First Manifesto

M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. **The Object-Oriented Database System Manifesto**. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223-240, Kyoto, Japan, December 1989

13 must-have features of OO-DBMS

- 8 from **Object-Oriented Programming Languages**
  complex objects, object identity, encapsulation, types and classes, inheritance, polymorphism, completeness, extensibility
- 5 from **Databases**
  persistence, secondary storage management, concurrency, recovery, *ad hoc* query facility

ODMG Standard

- Object Database Management Group
  - **1991** ODMG was created by R. Cattell of Sun Microsystems
  - **2000** Latest standard: ODMG 3.0
  - **2001** ODMG disbanded to focus on Java Data Object (JDO)
  - **2006** OMG Object Database Technology (ODBT) Working Group for the 4th generation of OO-DBMS standard
- Four components
  1. Object Model
  2. Object Definition Language (ODL)
  3. Object Query Language (OQL)
  4. Language Binding for C++, Java, Smalltalk

Object Query Language (OQL)

- Extension of the SQL-92 standard: object-oriented notions, like complex objects, object identity, path expressions, operation invocation etc.
- High level constructs to deal with sets of objects and primitives for structures, list, arrays etc.
- Functional language where operators can freely be composed, as long as the operands respect the type system
- OQL does not provide explicit **update** operators but rather invokes operations defined on objects for that purpose

OO-DBMS vs. OR-DBMS vs. O/R Mapping

Relation as first-class citizen?

- Yes: SQL3
  - PostgreSQL, IBM DB2, Oracle, Microsoft SQL Server, Sybase
- No: ODMG ODL+OQL
  - db4o, Versant, ObjectStore, ObjectDB, Native Queries, LINQ
- Don't care: PL coupled with (R-)DBMS Mapping Framework
  - Hibernate, JPA, JDO, CodeIgniter, Symfony, Django, EF

Exercises 1/2

1. Definitions

OID, Composition graph, Shallow equality, Deep equality, Expansion, Persistence by reachability, Partial multi-valued function

2. True or False? ✎

i) OID's are kind of primary keys.

ii) Given $=$ and $=_*$ are resp. $=_0$ and $=_\infty$, then $=_{k+1}$ refines $=_k$.

iii) One-one relationships are injective functions.

iv) Contravariance is meaningless.

v) ODL allows for multiway relationships.

Exercises 2/2

3. ODL relationships ✐

1. Give an ODL design with all the inverse relationships for the `Person` class to represent a genealogy.
2. What makes a relationships its own inverse ?

4. Problem

Prove that expand(o) is a regular tree, i.e., it has a finite number of distinct subtrees.