From complex values to objects

A database perspective

Guillaume Raschia — Nantes Université

Last update: October 4, 2022

eNF²

Contents

The eNF² Data Model

eNF² = Extended NF² Model

- Extend NF² 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` $(. \rightarrow .)$: key/value dictionary type of nested types
- First two are already available in RM and NF²

The eNF^2 Data Model (cont'd)

The Evolution of Data Models

b.t.w. of sort comparison

- Relational Model $\tau := \langle A_1 \!:\! \mathrm{dom}, \ldots, A_k \!:\! \mathrm{dom} \rangle$
- NF^2 $\tau := \mathrm{dom} \mid \langle A_1 \!:\! \tau, \ldots, A_k \!:\! \tau \rangle \mid \{\tau\}$
- eNF^2 $\tau := \mathrm{dom} \mid \langle A_1 \!:\! \tau, \ldots, A_k \!:\! \tau \rangle \mid \{\tau\} \mid (\tau) \mid [\tau]_n \mid \{\!|\tau|\!\} \mid (\tau \to \tau)$

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

Type Constructors

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

Comparison of Type Constructors

Type	Dupl.	Bounded	Order	Access by
Set $\{ . \}$	✘	✘	✘	Iterator
Bag $\{\!| . |\!\}$	✔	✘	✘	Iterator
Map $(. \to .)$	✔	✘	✘	Key
List $(.)$	✔	✘	✔	Index/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**

Type Constructors in SQL

- `MULTISET`
- `ROW`
- `ARRAY`

SQL ARRAY Type Constructor

- Introduced within SQL-99

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


SQL ARRAY Type Constructor (cont'd)

- Array type constructor with record insertion
- Access to elements by **index** 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 elements 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 ||

Classes

(Yet Another) Popular Restriction of eNF^2

Class

The outermost type constructor is a tuple

- A complex value conforms to sort τ of an object structure: it is an **instance** of its **class**
- Type constructors are building blocks: tuple, set, list, array, bag, dictionary
- eNF^2 is the reference model
- Implementation in SQL3 b.t.w. of **User-Defined Types**

User-Defined Types 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 an unary relation: tuples are objects (with 2 components)
- Grant **access privilege** to the 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'))

A Word About eNF^2 in Oracle

- Supports a **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...

Object Behavior

Method := **signature** + **body**

Operation that applies to objects of a given 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 unexpectable **side-effect**
- Implementation in SQL3

Disclaimer

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

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

Alternative Languages

Practical definition of object structures

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

Example in ODMG ODL

```
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 *enum*
- Composite type: *structure*
- Collection types: set, array, bag, list, and dictionary

Hierarchy & Subtyping

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 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?!

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) )
{...};
```

```
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 $\pi^*(\text{Person})$ with all subtable rows

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

- `ONLY` clause: retrieve the proper extension $\pi(\text{Person})$

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

Open issues

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

Basics of Relational Mapping of Class Hierarchy

- 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)

ID’s & Relationships

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, nil)

Composition Graph

Structural representation of an object as a labeled directed graph

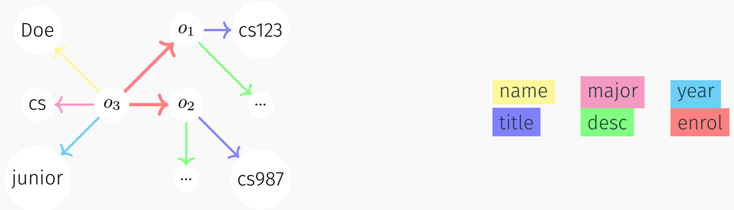
$$\text{struct}(o) := G(V, E)$$

where

- Vertices $V \subset O \cup \mathrm{dom}$ 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



Extend to a—cyclic—graph: *teacher* → *dpt* → *employees*

Statement

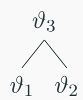
Object db is mainly **a huge persistent relational graph**

Object Expansion

Definition (Expansion)

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

Example of expand(o_3)



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

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 Sells 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

Translate into Relationships in ODL

- 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 the direction: `contains` or `isIncluded` or both

ODL example

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


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:

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

Epilogue

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

Impedance Mismatch Revisited

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

```sql
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`

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