IV) EXTENDED RELATIONAL DATABASES

Extensions of Relational Databases: A Panorama

- N1NF Models
  - VERSO (INRIA), NF2 (Darmstand)
- Logic
  - Datalog
- Extended Relational
  - Sabrina (INRIA), Postgres, Exodus (Wisconsin), Genesis (Texas), Starbust (IBM Almaden)
- Logic + NF2
  - LDL (MCC), RDL, Col (INRIA), Nail! (Stanford)
- NF2 + identity
- Semantic Models
  - IFO, FQL
- Object Oriented
  - OPL
  - Smalltalk
  - C++
- ADT PL
  - Ada, Clu
Introduction

- Reconciliation of Data Representation and associated Operations: the notion of Abstract Data Types (ADT)
  - Types are defined by their operations (i.e., use) rather than their internal presentation in a PL (i.e., representation)
- Enable user defined data structures and operations
  - Complex Value/Object Models + procedures/functions
- Distinguish between the specification of an ADT and its implementation
  - Internal representation of ADTs and code of operations are defined independently from its specification (easy modification)
  - Data are accessible only thought the associated operations (security)
- Some Early Prototypes:
  - Univ. of Berkley 1986-1990: Ingress-ADTs, Postgres (M. Stonebraker & al)
- Current standardization efforts: SQL3 ADTs

ADTs: Syntactic Aspects

- An ADT is defined syntactically by its name and the signature of its operations (for creation, access, etc.)
  - Encapsulation of data representation (abstraction)
- Example: A Bag ADT of integers
  - init: bag
  - add: int x bag ➔ bag
  - remove: int x bag ➔ bag
  - isempty: bag ➔ bool
- The semantics of an ADT is specified independently from its representation i.e., from an external view point
ADTs: Semantics Aspects

- **Axiomatic** (or algebraic) approach: an ADT is defined by axioms (i.e., equations) using the various operations
  - `isempty (init) = true`
  - `isempty (add(n,b)) = false`
  - `remove (n,init) = init`, etc.

- **Model** (or logic) based approach: an ADT is defined using well-known mathematical structures such as sets, sequences, trees, maps, etc.
  - `bag := N \rightarrow N^+` (e.g., `{7, 5, 7, 3} \rightarrow [7 \rightarrow 2, 5 \rightarrow 1, 3 \rightarrow 1]`)
  - `init := []`
  - `add (n, b) := if n not in dom(b) then b \cup [n \rightarrow 1]` else `let c = b(n) in b + [n \rightarrow c + 1]`

The DBMS Postgres Data Model

- Two kinds of **types**: Basic ones (e.g., int, float, bool, char, date) and Constructed (inherit structure and behavior from other types)
  - ADTs (to build new basic types) are defined by specifying type name, the length of the internal representation in bytes; procedures for converting a value from an external to an internal representation and vise-versa; and a default value
  - Constructed types are created by constructing a record of basic types or other constructed types

- Three kinds of **functions**: normal (written in C), operators and Postquel functions
  - Operators on ADTs are defined by specifying the number and type of operands, the return type, the precedence & associativity of the operator and the procedure that implements it
  - Functions package together any collection of commands of the Postquel query language and are considered as constructed types
define type Rectangular is (Internallength=16, Inputprocedure=chartorect, Outputprocedure = recttochar, Default="")
define type Coordinate is (Internallength=8, Inputprocedure=chartocoord, Outputprocedure = coordtochar, Default="")
define operator iswithin (Rectangular, Coordinate) returns bool is (Proc= iswithin, precedence = 3, associativity= "left", Sort= iswithinSort, Hashes, Restrict = iswithinSelect, Join= iswithinSelect, Negator= iswithinNE)
create Artifact (Title:char(50), Material:char(50), Created:date, Located:char(20))
create Painting (Style:char(10), Theme:char(20), Similar: Postquel, Placed: Coordinate) inherits Artifact
create Museum (Denom:char(20), Street:int, Num:int, City:char(10), Plan: Rectangular)
define function psimilar retrieve (Painting.title) where Painting.theme = $1.theme

Consider a core relational query language:
- QUEL (e.g., Postgres), SQL (e.g., Sabrina)

Extend the relational QL according to the following principles:
- If f: t -> t’ is the signature of a function f, and t’ is an atomic type then each atomic expression of a query that is compatible with t’ can be substituted by an expression f(x) where x of type t.
- If f t -> t’ and g t’ -> t” are the signature of the functions f and g, then g(f) is an expression of the language with signature t -> t”.
- If f: t -> t’ is the signature of a function f, and t’ is a set of tuples then each relational expression of a query that is compatible with t’ can be substituted by an expression f(x) where x of type t.
Extended Relational Query Examples

- Find the museums with a total surface greater that 100
  
  \[
  \text{retrieve } (\text{m.Denom}) \text{ using Museum } [\text{m}] \text{ where } \text{surface } (\text{m.Plan}) > 100
  \]

- Find the paintings of the San Diego Museum exhibited within the area specified by the rectangular “50,100:100,70”
  
  \[
  \text{retrieve } (\text{p.title}) \text{ using Painting } [\text{p}], \text{ Museum } [\text{m}] \text{ where } \text{p.Located} = \text{“San Diego Museum” and p.located} = \text{m.Denom and iswithin (intersect (m.Plan, “50,100:100,70”), p.Placed)}
  \]

Extended Relational: Pending Issues

- How ADTs are defined using a RDBMS?
  - Need for DBMS DDL & DML extension (e.g., SQL3)
    - Internal operations are written in extended SQL
    - External operations are written in the PL (C in Postgres, Lisp in Sabrina) and only their signature appears in the definition of the ADT

- How ADTs can be interpreted by the Query Language?
  - Need for conversions of the ADT declaration (structure + operations) into the internal DBMS representation:
    - The function code is implemented using the internal representation of the ADT in the DBMS
    - The function code is implemented using a specific PL understood by the DBMS

- How queries can be optimized?
  - Need to index the attributes of ATDs (user defined index?)
  - Autogenous vs. Exogenous extensions
Autogenous vs. Exogenous Extensions

V) DATABASE PROGRAMMING LANGUAGES
Preliminary Typing Issues

◆ Why Typing is useful?
  • Fundamental Programming discipline for better specification of code
◆ Type Checking: Verification of correct functions on correct objects
  • Vassilis - 5 (error !!)
◆ Type Inference: Decide the correct type of an expression
  • Age(Vassilis) - 5 -> int
◆ Strong vs. Weak Typing
  • Each variable or function of the PL has a type (C or C++ vs Lisp)
◆ Static vs. Dynamic Typing
  • Each variable or function type is known during compilation (C vs. C++)
➲ Compromise: expressiveness (flexibility) vs. security (error detection)

Two forms of Persistence

1. The same structure should be used for persistent or transient values
2. The type of a value is retained whether it is persistent or not and that no software can ever violate the constraints appropriate to its type
3. The same names should be preserved
The PPL Approach

- How programming languages can be enhanced with data persistence?
  - Integration of a PL with DBMS functionality (Pascal-R, Adaplex, etc.)
  - Extension of a PL with DBMS functionality (PS-Algol, Galileo, etc.)
- Coupling DBMS functionality with a PL for data intensive applications:
  - Data structuring in form of collections
  - Filtering primitives for collection data (bulk types)
- Main characteristics
  - Persistence orthogonal to types, behavioral transparency, etc.
  - Abstract Data Types
- But query optimization is quite difficult to achieve
  - Programs are much more complicated to optimize than relational queries

Pascal-R [J. Schmidt 1980]

- Integration of the Pascal type system with the relational model
  - tuple ↔ record type (no variant)
  - relation ↔ new type relation
  - database ↔ new type database
- Introduction of new operations for set values (relations)
  - The in operator tests tuple membership in a relation
  - The each operator address the result of a relation selection operation
  - The for each operator scans the tuples of a relation
- Limits:
  - The record type allows to define only tuples with atomic attributes
  - Incomplete type/persistence orthogonality: only relation type is persistent
Pascal-R: Example

```pascal
Artist = relation Name of record
  Name: string;
  Style: string;
  Live-Time: string;
  Nationality: string;
end;

Painting = relation Name of record
  Name: string;
  Painter: string;
  Material: string;
  Date: string;
  Museum: string;
end;

Fine-Art = database
  artists: Artist;
  paintings: Painting;
end;

do
  for each p in Fine-Art. Paintings: p. Painter = a. name
do writeln (p. name);
```

PS-Algol [M. Atkinson & al 1981]

- The type system used by PS-algol is the same with S-algol
  - basic types: integer, real, boolean, string and pointer
  - constructed types: vector (dynamic table) and structure (tuple)
  - relation types: a new type table (of pointers to vector or structure instances)
- The procedures used by PS-algol are manipulated as normal values
  - Possible definition of ADTs and object encapsulation
  - The scan procedure allow iteration over the elements of a collection (vector) by applying boolean functions (with side-effects)
  - The is and isn’t primitives allow to verify the type of values
- Limits:
  - Unlike Pascal, PS-Algol pointers are untyped (strings): no static typing