DATABASDESIGN FÖR INGENJÖRER - 1DL124

Sommar 2005

En introduktionskurs i databassystem

http://user.it.uu.se/~udbl/dbt-sommar05/
alt. http://www.it.uu.se/edu/course/homepage/dbdesign/st05/

Kjell Orsborn
Uppsala Database Laboratory
Department of Information Technology, Uppsala University,
Uppsala, Sweden

Introduction to AMOS II and AMOSQL

Kjell Orsborn

Department of Information Technology
Uppsala University, Uppsala, Sweden

Iris/OpenODB/Odapter/AMOS II Object-Relational DBMS

IRIS

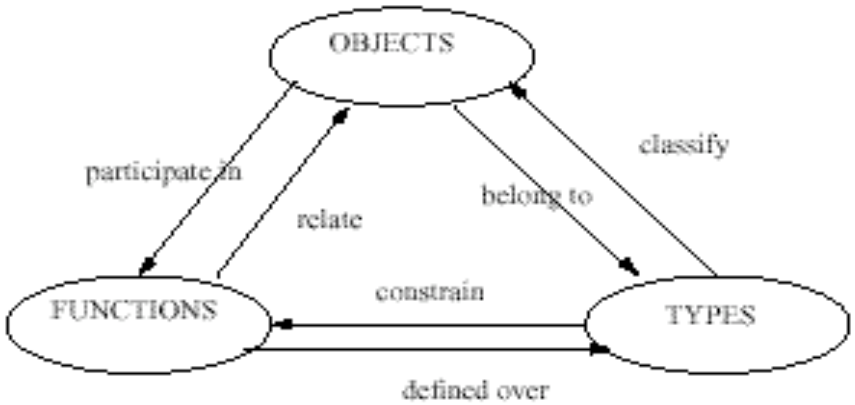
- 1st Object-Relational DBMS: Iris research prototype developed in Database Technology Department of HP Laboratories
- Iris' query language OSQL is a *functional* query language
- OpenODB/Odapter is the HP product based on Iris

AMOS II

- AMOS II developed at UDBL but has its roots in Iris
- AMOS II runs on PCs under Windows NT/2000 and Solaris
- AMOS II uses query language *AMOSQL*
- AMOS II system is a fast *main-memory* DBMS
- AMOS II has single user or optional client-server configuration
- The object part of SQL99 is close to AMOSQL
- Mediator facilities: AMOS II is also a *multi-database (mediator)* system for integrationg data from other databases

AMOS II / Iris Data Model

- Basic elements in the AMOS II data model



AMOS II Data Model

Objects:

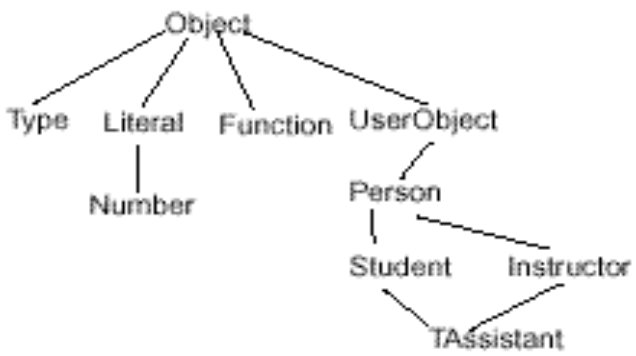
- Atomic entities (no attributes)
- Belong to one or more *types* where one type is *the most specific type*
- Regard database as *set of objects*
- Built-in atomic types, *literals*:
  - String, Integer, Real, Boolean
- *Collection* types:
  - Bag, Vector
- *Surrogate* types:
  - objects have unique *object identifiers (OIDs)*
  - explicit creation and deletion
  - DBMS manages OIDs

  AMOSQL example:
  - `create person instances :tore;`

AMOS II Data Model

Types:

- *Classification* of objects
  - groups of OIDs belong to different types
- *Multiple inheritance* supported
- Organized in a type/subtype Directed Acyclic Graph
  - defines that OIDs of one type is a subset of OIDs of other types
- Types and functions are objects too
  - of types “type” and “function”
- Part of the AMOS II type hierarchy:



AMOS II Data Model

Types continued…:

- Every object is an instance of *at least one type*
- A *type set* is associated with each OID
- Each OID has one *most specific type*
- Each surrogate type has an *extent* which is the set of objects having that type in its type set.
- System understands *subtype/supertype* relationships
- Objects of *user-defined types* are instances of type `Type` and subtypes of `UserObject`
- User defined objects always contains class `UserObject` in its type set
- Object types may change dynamically (*roles*)

AMOS II Data Model

Functions:

- Define *semantics* of objects:
  - properties of objects
  - relationships among objects
  - views on objects
  - stored procedures for objects
- Functions are instances of type `Function`
- More than one argument allowed
- Bag valued results allowed, e.g. `Parents`
- Multiple valued results allowed
- Sets of multiple tuple valued results most general

AMOS II Data Model

- A function has two parts:
- 1) *signature*:
  - name and types or arguments and results
  - examples:

```
name(person p) -> charstring n
name(department d) -> charstring n
dept(employee e) -> department d
plus(number x, number y) ->number r
children(person m, person f) -> bag of person c
marriages(person p) -> bag of <Person s, Integer year>
```

- 2) *implementation*:
  - specifies how to compute outputs from valid inputs
  - non-procedural specifications, except for stored procedures
- A function also contains an *extent*, i.e. a set of mappings from argument(s) to result(s)
  - for example:

```
name(:tore) = 'Tore'
name(:d1) = 'Toys'
dept(:tore) = :d1
plus(1,2) = 3 or (1+2 = 3) Indefinite extent!
children(:tore,:ulla) = {:karl,:oskar}
marriages(:tore) = {<:eva, 1971>,<:ulla,1981>}
```

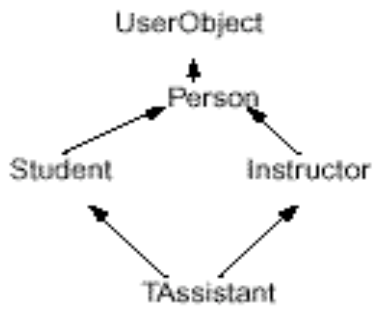
AMOS II Data Model

AMOSQL has four kinds of functions:

- 1) *stored functions* (c.f. relational tables, object attributes)
  - values stored explicitly in database
- 2) *derived functions* (c.f. relational views, object methods)
  - defined in terms of queries and other functions using AMOSQL
  - compiled and optimized by Amos when defined for later use
- 3) *database procedures* (c.f. stored procedures, object methods)
  - for procedural computations over the database
- 4) *foreign functions* (c.f. object methods)
  - escape to programming language (Java, C, or Lisp) e.g. for foreign database access
- Functions can also be overloaded:
  - *overloaded functions* have several different definition depending on the types of their arguments and results.

AMOSQL language - schema definition and manipulation

- Creating types:
  - `create type Person;`
  - `create type Student under Person;`
  - `create type Instructor under Person;`
  - `create type TAssistant under Student, Instructor;`



AMOSQL language - schema manipulation

- Delete a type:
  - `delete type Person;`
  - referential integrity maintained
  - types `Person, Student, Instructor` and `TAssistent` also deleted
- Create functions:
  - `create function name (Person p) -> Charstring nm as stored;`
  - `create function name (Course) -> Charstring as stored;`
  - `create function teaches(Instructor) -> bag of Course as stored;`
  - `create function enrolled(Student) -> bag of Course as stored;`
  - `create function instructors(Course c) -> Instructor i as`
    `select i where teaches(i) = c;`
  - The `instructors` function is the inverse of `teaches`

AMOSQL language - schema manipulation

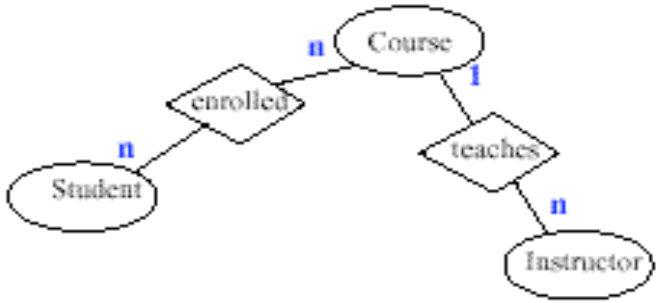
- Delete functions:
  - delete function `teaches`;
  - referential integrity maintained.
    - e.g. function `instructors` also deleted
- Defining type and attributes:
  - ```
    create type Person properties
    (name Charstring,
    birthyear Integer,
    hobby Charstring);
    ```
  - `name, birthyear, hobby` are defined together with type `Person`
- Above equivalent to:
  - ```
    create type Person;
    create function name(Person) -> Charstring as stored;
    create function birthyear(Person) -> Integer as stored;
    create function hobby(Person) -> Charstring as stored;
    ```

AMOSQL language - schema manipulation

- Example of inherited properties:
  - `create type Person properties`
    `(name Charstring key,`
    `age Integer,`
    `spouse Person);`
  - `create type Employee under Person properties`
    `(dept Department);`
  - `Employee` will have functions (attributes) `name, age, spouse, dept`
- Can easily extend with new functions:
  - `create function phone(Person) -> Charstring as stored;`

AMOSQL language - schema manipulation

- Modeling relationships with cardinality constraints
  - `create function enrolled(Student e nonkey) -> Course c nonkey as stored;`
  - `create function teaches(Instructor i key) -> Course c nonkey as stored;`



- Modeling *properties of relationships* by multi-argument *stored* functions:
  - `create function score(Student,Course) -> Integer s as stored;`
- Modeling *properties of relationships* by multi-argument *derived* functions:
  - `create function instructors(Student s, Course c) -> Teacher t as select t where teaches(t) = c and enrolled(s) = c;`

AMOSQL language - data definition and manipulation

- *Instance creation*:
  -
```
create Person(name, birthyear) instances
:risch ('T.J.M. Risch', 1949),
:ketabchi ('M.A. Ketabchi', 1950);
```
  - equivalent formulation:
```
create Person instances :ketabchi, :risch;
set name(:risch) = 'T.J.M. Risch';
set birthyear(:risch) = 1949;
set name(:ketabchi)= 'M.A. Ketabchi';
set birthyear(:ketabchi)=1950;
```
- *Instance deletion*:
  -
```
delete :risch;
delete :ketabchi;
```

AMOSQL language - data manipulation

- *Calling* functions:
  - `name(:risch);`
    `'T.J.M. Risch'`
  - equivalent formulation:
    `select name(:risch);`
    `'T.J.M. Risch'`
- *Adding* elements to bag-valued functions:
  - `add hobbies(:risch) = 'Painting';`
    `add hobbies(:risch) = 'Fishing';`
    `add hobbies(:risch) = 'Sailing';`
    `hobbies(:risch);`
    `'Painting'`
    `'Fishing'`
    `'Sailing'`

AMOSQL language - data definition and manipulation

- *Removing* elements from set-valued functions:
  - `remove hobbies(:risch) = 'Fishing';`
    `hobbies(:risch);`
    `'Painting'`
    `'Sailing'`
- *Adding type* to object:
  - `add type Teacher to :risch;`
    `set teaches(:risch)= :math;`
- *Removing type* from object:
  - `remove type Teacher from :risch;`
    `teaches(:risch);`
    `Error: Function teaches not defined for object`
  - This will also implicitly do
    `remove teaches(:risch) = :math;`
    Good for database evolution.

AMOSQL queries

- AMOSQL power: relationally complete and more
- General format:
  - ```
    select <expressions>
    from <variable declarations>
    where <predicate>;
    ```
- Example:
  - ```
    select name(p), birthyear(p) from Person p;
    ```
- Function composition simplifies queries that traverse function graph (Daplex semantics):
  - ```
    name(parents(friends(:risch)));
    ```
- More SQLish:
  - ```
    select n
    from Charstring n, Person par, Person fr
    where n = name(par) and
       par = parents(fr) and
       fr = friends(:risch);
    ```
- Works also for bag-valued arithmetic functions:
  - ```
    sqrt(sqrt(16.0));
    2.0
    -2.0
    ```
    ```

AMOSQL examples

- Examples of functions and *ad hoc queries*

```
create function income(Person) -> Integer as stored;
create function taxes(Person) -> Integer as stored;
create function parents(Person) -> bag of Person as stored;
create function netincome(Person p) -> Integer as
          select income(p)-taxes(p);
create function sparents(Person c) -> Student as
          select parents(c); /* Parent if parent is student;
          bag of implicit for derived functions */
create function grandsparentsnetincomes(Person c) -> Integer as
          select netincome(sparents(parents(c)));
select name(c)
from Person c
where grandsparentsnetincomes(c) > 100000 and income(c) <10000;
```

AMOSQL aggregation functions

- An aggregation function is a function that coerces some value to a single unit, a *bag*, before it is called.
- "bagged" arguments are not "distributed" as for other AMOSQL functions (no Daplex semantics for aggregation functions)
  - `count(parents(friends(:risch)));`
    `5`
- Signature:
  - `create function count(bag of Object) -> Integer as foreign ...;`
- Nested queries, local bags:
  - `sum(select income(p) from Person p);`

AMOSQL quantification

- Quantifiers
- Existential and universal quantification over subqueries supported through two aggregation operators:
  - `create function notany(bag of object) -> boolean;`
  - `create function some(bag of object) -> boolean;`

  `some` tests if there exists some element in the bag

  `notany` tests if there does not exist some element in the bag
- Example:

```
– create function maxincome(Dept d) -> Integer as
  select income(p)
  from Employee p
  where dept(p) = d and
    notany(select true from Employee q where income(q) > income(p));
```

AMOSQL advanced updates

- Set-oriented updates
- Setting multiple function instances:
  - ```
    set salary(e) = s
    from Employee e, Integer s
    where s=salary(manager(e));
    ```
- Removing values from set-valued functions:
  - ```
    remove friends(:risch) = f
    from Person f
    where age(f) > age(:risch);
    ```
  - ```
    remove friends(:risch) = p from Person p
    where count(friends(p))>5;
    ```
    ```

AMOSQL stored procedures

- Database Procedures
- For example to encapsulate database updates:
  -
```
create function creperson(charstring nm, integer inc) -> person p
as
begin
  create person instances p;
  set name(p) = nm;
  set income(p) = inc;
  result p
end;
```
- Optimized iterative update:
  -
```
create function RemoveOldFriends(Person p) -> boolean as
begin
  remove friends(p) = s
  from Person s
  where age(s) > age(p);
  end;

RemoveOldFriends(:risch);
```

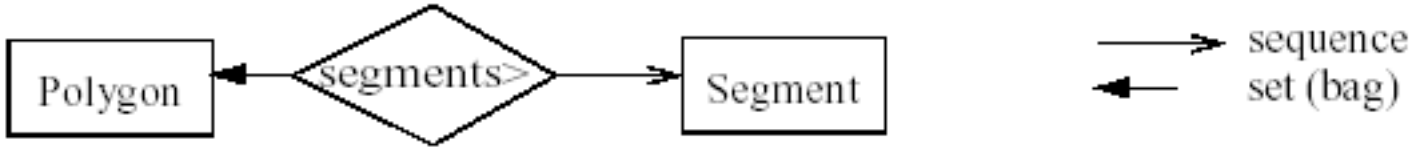
AMOSQL sequences

Vectors (ordered sequences of objects)

- The datatype vector stores ordered sequences of objects of any type
- Vector declarations can be parameterized by declaring the type `Vector of <type>` as for example:
  - ```
    create type Segment properties
    (start Vector of Real,
     stop Vector of Real);
    ```
  - ```
    create type Polygon properties
    (segments Vector of Segment);
    ```
- Vector values have system provided constructors:
  - ```
    create Segment instances :s1, :S2;
    set start(:s1)=Vector of Real(1.1, 2.3);
    set stop(:s1)=Vector of Real(2.3, 4.6);
    set start(:s2)=Vector of Real(2.8, 5.3);
    ```
  - ```
    create Polygon instances :p1;
    set segments(:p1)=Vector of Segment(:s1, :s2);
    ```

AMOSQL sequences

- Extended ER notation:



- Vector types can be used as any other type
- E.g. functions on sequences can be defined:
  - `create function square(Number r)->Number as select r * r;`
  - `create function positive(Number r)->Number as select r where r>=0;`
  - `create function length(Segment l) -> real as select positive(sqrt(square(start(l)[0] - stop(l)[0]) + square(start(l)[1] - stop(l)[1])));`
  - `create function length(Polygon p) -> real as select sum(select length(segments(p)[i]) from Integer i);`
- Vector queries:
  - `length(:s1);`
  - `length(:p1);`
  - `select s from Segment s where length(s) > 1.34;`

AMOSQL schema queries

- System data can be queried as any other database data as for example:
- Find the names of the supertypes of EMPLOYEE:
  - `name(supertypes(typenamed("EMPLOYEE")));`
    `"PERSON"`
- Find the resolvents of an overloaded function:
  - `name(resolvents(functionnamed("AGE")));`
    `"DEPARTMENT.AGE->INTEGER"`
    `"PERSON.AGE->INTEGER"`
- Find the types of the first argument of each resolvent of a function:
  - `name(resolventtype(functionnamed("AGE")));`
    `"DEPARTMENT"`
    `"PERSON"`
- Find all functions whose single argument have type PERSON
  - `attributes(typenamed('PERSON"));`
    `"NAME"`
    `"AGE"`

How to run AMOS II

- Install system on your PC by downloading it from
  - `http://www.csd.uu.se/~udbl/amos/`
- Run AMOS II with:
  - `amos2`
- User's guide in:
  - `http://www.csd.uu.se/~udbl/amos/doc/amos_users_guide.html`
- Simple AMOS II tutorial in
  - `http://www.csd.uu.se/~udbl/amos/doc/tut.pdf`

(AM)OSQL in Iris/OpenODB/AMOS II

- Summary:

- (AM)OSQL provides flexible OR DBMS capabilities
- Not hard wired object model, but dynamically extensible model
- Extended subset of object part of SQL99
- Very good support for ad hoc queries
- Good schema modification operations
- Object views
- The key is the functional model of (AM)OSQL