Introduction to AmosQL

Tore Risch

Department of Information Technology

Uppsala University

Sweden

1/12/03

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

IRIS

- First 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/LiTH but has its roots in Iris
- Amos II runs on PCs under Windows NT/2000 and Linux
- 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 SQL:99 is close to AmosQL

MEDIATOR facilities

- Amos II also a multi-database (mediator) system for wrapping and integrating data from other databases
## 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 type/subtype Directed Acyclic Graph
  Defines that OIDs of one type is subset of OIDs of other types

Amos II Data Model (Types)

- Types and functions are objects too
  Of types ‘Type’ and ‘Function’
- Part of type hierarchy
Amos II Data Model (Types)

- Every object is an instance of at least one type
- 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 (Functions)

A function has two parts:

1. **Signature**:
   - Name and types or arguments and results
     - `name(Person p) -> Charstring n`
     - `name(Department d) -> Charstring n`
     - `dept(employee e) -> Department d`
     - `plus(Number x, Number y) -> Number r (infix syntax x+y)`
     - `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 a set of mappings from argument(s) to result(s), an *extent*
  
  For example:

  - `name(:tore) = 'Tore'`
  - `name(:d1) = 'Toys'`
  - `dept(:tore) = :d1`
  - `children(:tore,:ulla) = {:karl,:oskar}`
  - `marriages(:tore) = {<:eva, 1971>,:<ulla,1981>}`

- The extent of a function may be *complete*, i.e. defined for every possible argument, or *incomplete* or even *indefinite*

  - `plus(1,2) = 3` or `(1+2 = 3)` Plus has *indefinite* extent!
Amos II Data Model (Functions)

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. 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;
  ```

![Class Diagram](diagram.png)
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 /* Inverse of teaches */`
  
  `where teaches(i) = c;`

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:
  ```sql
  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:
  ```sql
  create function phone(Person) -> Charstring as stored;
  ```

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

![Diagram of E/R relationships with cardinality constraints]
**AmosQL language (Schema Manipulation)**

- Modeling *properties of relationships* by multi-argument *stored* functions:
  
  ```sql
  create function score(Student, Course) -> Integer s as stored;
  ```

- Modeling *properties of relationships* by multi-argument *derived* functions:

  ```sql
  create function instructors(Student s, Course c) -> Teacher t as
      select t
        where teacher(c) = t and enrolled(s) = c;
  ```

**AmosQL language**

**Data Definition and Manipulation**

- *Instance creation*

  ```sql
  create Person(name, birthyear) instances
      :risch ('T.J.M. Risch', 1949),
      :ketabchi ('M.A. Ketabchi', 1950);
  ```

  - Equivalent formulation:

    ```sql
    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*

  ```sql
  delete :risch;
  delete :ketabchi;
  ```
AmosQL language (Data Manipulation)

- **Calling** functions:
  
  ```
  name(:risch);
  'T.J.M. Risch'
  ```

  Equivalent formulation:
  ```
  select name(: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 Manipulation)

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

Queries

• Power: Relationally complete and more
• General format
  select <expressions>
  from <variable declarations>
  where <predicate>;
• Example:
  select name(p), birthyear(p)
  from Person p;

AmosQL queries

Queries

• 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.

E.g.:

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 grandparentsnetincomes(Person c) -> Integer as
  select netincome(sparents(parents(c)));
select name(c)
  from Person c
  where grandparentsnetincomes(c) > 100000 and
    income(c) <10000;

**AmoSQL aggregation functions**

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:

```sql
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:

```sql
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:

```sql
set salary(e) = s
from Employee e, Integer s
where s=salary(manager(e));
```

- Removing values from set-valued functions:

```sql
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
- E.g. to encapsulate database updates (constructors):
  
  ```
  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 `Vector of <type>`
  
  ```
  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(2.3,4.6);
  set stop(:s2)=Vector of Real(2.8,5.3);
  create Polygon instances :p1;
  set segments(:p1)=Vector of Segment(:s1,:s2);
  ```
**AmosQL sequences**

Vector types can be used as other types.

- Functions on sequences can be defined
  
  ```sql
  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);
  ```

- Extented ER notation:

  ```sql
  Polygon segments >
  Segment
  ```

- Vectors can be queried:
  
  ```sql
  length(:s1);
  length(:p1);
  select s from Segment s where length(s)>1.34;
  ```

**AmosQL schema queries**

Querying the schema

- System data is queryable as any other database data

  ```sql
  E.g. Find the names of the supertypes of Employee:
  name(supertypes(typenamed("EMPLOYEE")));
  "PERSON"
  ```

- Find the resolvents of an overloaded function:

  ```sql
  name(resolvents(functionnamed("AGE")));
  "DEPARTMENT.AGE->INTEGER"
  "PERSON.AGE->INTEGER"
  ```

- Find the types of the first argument of each resolvent of a function:

  ```sql
  name(resolventtype(functionnamed("AGE")));
  "DEPARTMENT"
  "PERSON"
  ```

- Find all functions whose single argument have type PERSON attributes(typenamed("PERSON"));

  ```sql
  "NAME"
  "AGE"
  ```
**Amos II**

How to run Amos II:

- Install system on your PC by downloading it from  
  http://user.it.uu.se/~udbl/amos/

- Run Amos II with:  
  amos2

- User’s Guide in  
  http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html

- Simple Amos II Tutorial:  
  http://user.it.uu.se/~udbl/amos/doc/tut.pdf

- These slides:  
  http://user.it.uu.se/~torer/kurser/dbt/amosql.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 SQL:99
- Very good support for ad hoc queries
- Good schema modification operations
- Object views

The key is the functional model of AmosQL.