Introduction to Amos II

Tore Risch
Department of Information Technology
Uppsala University
Sweden
2012-02-13
Amos II Object-Relational DBMS

- Amos II developed at UDBL/LiTH
- Amos II runs on PCs under Windows and Linux
- Amos II uses functional query language AmosQL: Use functions to model data.
- Amos II system is a fast main-memory DBMS
- Amos II has single user or optional client-server configuration
- Amos II is extensible, i.e. foreign functions in e.g. C
- Amos II is a mediator system for wrapping and integrating data from other databases, files, streams, etc.
- Highly parallel data stream management system (DSMS) on top of Amos II, SCSQ
- Applications: Science, industry, etc.
Amos II Data Model

Data Model

Basic elements in data model

OBJECTS

FUNCTIONS

TYPES

classify

belong to

relate

participate in

constrain

defined over
Amos II Data Model, Types

- For example:

  ```
  create type Person;
  create type Student under Person;
  create type Instructor under Person;
  create type TA under Student, Instructor;
  create type Dept;
  create type Course;
  ```

- Part of type hierarchy
Amos II Data Model, Objects

- AmosQL Example:
  
  ```amosql
  create Person instances :tore;
  ```

  creates new object, say OID[1145], and binds temporary environment variable :tore to it. Environment variables disappear when transaction finished.

- The DBMS manages the objects.

- Surrogate type objects have unique object identifiers (OIDs), e.g. OID[1145]

  Explicit creation and deletion, e.g.
  ```amosql
  delete :tore;
  ```

- Objects belong to one or more types where one type is the most specific type

- Regard database as collection of objects

- Built-in atomic types, literals: String, Integer, Real, Boolean, Date, e.g.
  ‘This is a string’, 2.34, |2008-10-13/18:11:43|

- Collection types:

  Bag, e.g. `bag("Tore","Kjell","Thanh")`

  Vector, e.g. `{1,2,3}`

  Record, e.g. `{“Name”:”Tore”, “Dept”:”IT”}`

  Tuple, e.g. ("Tore","IT")
Amos II Data Model, Types

- *Classification* of objects:
- Every object is an instance of *at least one type*
- Objects are grouped by the types to which they belong (are instances of)
- Organized in type/subtype Directed Acyclic Graph
- Defines that OIDs of one type is *subset* of OIDs of other types
- *Type set* is associated with each OID
- Each OID has one *most specific type*
- *Multiple inheritance* supported
- Each surrogate type has an *extent* which is the set of objects having that type in its type set.
- Objects of user-defined types are instances of type *Type* and subtypes of *UserObject*
- Types and functions are objects too
  Of types ‘Type’ and ‘Function’
Amos II Data Model, Functions

Examples of function definitions:

create function name (Person p) -> Charstring nm
    as stored;
create function name (Dept) -> Charstring as stored;
create function name(Course) -> Charstring as stored;
create function dept(Person) -> Dept as stored;
create function teacher(Course) -> Instructor
    as stored;
create function teaches(Instructor i)-> Bag of Course c
    as /* Inverse of teacher */
    select c where teacher(c) = i;
create function instructorNamed(Charstring nm)
    -> Instructor p
    as select p where name(p) = nm;
Populate and query the database

- create Dept(name) instances :it ('IT');
  create Instructor(name, dept) instances
    :tr ('Tore', :it),
    :ko ('Kjell', :it);
create Course(name, teacher) instances
  ('Databases', :tr), ('Data Mining', :ko), ('ETrade', :ko);
Equivalent formulation using update command set:
create Instructor instances :tr, :ko;
set name(:tr) = 'Tore';
set dept(:tr) = :it;
set name(:ko) = 'Kjell';
set dept(:ko) = :it;
etc.

- Examples of functional expressions, the simplest queries:
  sqrt(2)+3;
instructorNamed('Tore');
name(dept(instructorNamed('Tore')));
Amos II Data Model, Functions

- Functions define *semantics* of objects (entities):
  - *Attributes* of objects, e.g. `name(Person) -> Charstring`
  - *Relationships* among objects (mappings between arguments and results),
    `parents(Person) -> Bag of Person, dept(Person) -> Dept`
  - *Views* on objects, derived function, e.g. `square, grandparents`.
  - *Procedural function* over objects
- Bag valued results allowed, e.g. `Parents(...) -> Bag of Person`
- Multiple argument allowed, e.g.
  `1+2+3; children(Person x, Person u) -> Bag of Person`
- Tuple values allowed, e.g.
  `parents2(Person x) -> (Person m, Person f)`
Amos II Data Model, Functions

A function has two parts:

1. *Signature:*
   - Name and types or arguments and results
     
     - `name(Person p) -> Charstring n`
     - `teacher(Course c) -> Person`
     - `plus(Number x, Number y) ->Number r (infix ‘+’)`
     - `children(Person m, Person f) -> Bag of Person c`
     - `parents2(Person p) -> (Person m, Person f)`

2. *Body:*
   - Specifies how to compute outputs from valid inputs.
   - Usually *non-procedural* specifications, i.e. no side effects, *except* for *procedural functions.*
Amos II Data Model, Functions

Four kinds of functions:

1. **Stored** functions (c.f. relational tables, object attributes, facts)
   - Values stored explicitly in database, e.g. name, parents

2. **Derived** functions (c.f. relational views, object methods, rules)
   - Defined as *queries* over other functions using AmosQL,
     e.g. instructorNamed
   - Derived functions are immediately compiled and optimized by Amos II when defined

3. **Procedural functions** (c.f. SQL2003 UDFs, stored procedures, object methods)
   - For procedural computations over the database

4. **Foreign** functions (c.f. object methods, built in predicates)
   - Escape to programming language (Java, C, or Lisp), e.g. plus, sqrt
   - 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, e.g. name(Person), name(Dept), 1+2, “a”+“b”
AmosQL language, Schema Definition

- Model E/R relationships with cardinality constraints as functions:
  create function enrolled(Student e) -> Bag of Course c as stored;
  create function teacher(Course) -> Instructor as stored;
**AmosQL queries**

Select statements

- Power: *Relationally complete* and more
- General format
  
  ```
  select <expressions>
  from <variable declarations>
  where <predicate>;
  ```

- Example:
  
  ```
  select name(p), name(dept(p))
  from Person p
  where p = instructorNamed('Tore');
  =>
  ("Tore","IT")
  ```

  ```
  select p, name(p) from Person p;
  =>
  (#[OID 1440],"Tore")
  (#[OID 1441],"Kjell")
  (#[OID 1442],"Thanh")
  ```
AmosQL queries

- Function composition simplifies queries that traverse function graph, *Daplex semantics*: 
  
  ```sql
  name(teaches(instructorNamed('Kjell')));
  =>
  "ETrade"
  "Data Mining"
  ```

- *Daplex semantics*: `name` applied on each element in bag returned from `teaches`, a form of path specification over functions.

- Equivalent more SQLish flattened query:
  
  ```sql
  select n
  from Charstring n, Instructor i, Course c
  where n = name(c) and
    c in teaches(i) and
    i = instructorNamed('Kjell');
  Notice that literal types like Charstring can be used in from clause.
AmosQL aggregate functions

- An aggregate function is a function that coerces some value to a single unit, a bag, before it is called.
- ‘Bagged’ arguments are not ‘flattened’ for aggregate functions (no Daplex semantics for aggregate functions).

\[
\text{count}\left(\text{teaches(\text{instructorNamed('Kjell')})}\right); \\
\Rightarrow 2
\]

Signature of aggregate function count and sum:
\[
\text{count}(\text{Bag of Object}) \rightarrow \text{Integer} \\
\text{sum}(\text{Bag of Number}) \rightarrow \text{Number}
\]

- Nested queries, local bags:
\[
\text{sum(\text{select count(\text{teaches(p)}) from Instructor p})}; \\
\Rightarrow 3
\]
AmosQL quantification

Quantifiers

- *Existential and universal quantification* over subqueries supported through two aggregate functions:
  
  create function notany(Bag of Object) -> Boolean as foreign ...;
  
  create function some(Bag of Object) -> Boolean as foreign ...;

  **some** tests if there exists some element in the bag
  
  **notany** tests if there does not exist any element in the bag

- Example:

  ```sql
  select name(p)
  from instructor p
  where notany(name(teaches(p))="Data Mining");
  => "Tore"
  ```
Bag variables, Cursors

- Queries and function calls may return very large bags as results, for example:
  \[ \text{iota}(1,1,000000)+\text{iota}(1,1,000000) \]
  returns \(10^{12}\) numbers!

- Can assign query result to bag variable:
  ```
  set :b = \text{iota}(1,1,000000);
  count (:b);
  \rightarrow 1000000
  ```

- Can iterate over very large bags using cursors:
  ```
  open :c1 on \text{iota}(1,1,000000)+\text{iota}(1,1,000000);
  fetch :c1;
  \rightarrow 1
  fetch :c1;
  \rightarrow 2
  etc.
  ```
**Grouping**

SQL’s construct ‘group by’ of key-values is in AmosQL expressed using *second order* aggregate functions:

\[
\text{groupby}(\text{Bag of (Object, Object)} \ q, \ \text{Function} \ \text{aggfn}) \rightarrow \ \text{Bag of (Object} \ g, \ \text{Object} \ a)
\]

In subquery \(q\) returning bag of tuples \((g, v)\) for each different value of \(g\) form the bag \(b_g\) of corresponding \(v\):s and call the aggregate function \(\text{aggfn}(b_g)\). Return pair \((g, \text{aggfn}(b_g))\).

For example:

\[
\text{groupby}(\text{(select name(i), c from Course c, Instructor i where teaches(i)=c, #'count')});}
\]

\[
=>
\]

\(("K. Orsborn", 2)\n\(("T. Risch", 1)\)

The notation #’count’ denotes the function named *count*, a *functional constant*. 
Procedural functions

- E.g. to encapsulate database updates (constructors):
  
  ```
  create function newInstructor(Charstring nm, Charstring dept) -> Instructor i as
      create Instructor instances i;
      set name(i)=nm;
      set dept(i)=dept;
      return i;
  end;
  ```

- Iterative update:
  
  ```
  create function removeOverloadedInstructors(Integer th)->Bag of Charstring
      as for each Instructor i
          where count(select teaches(i)) >= th
          begin return name(i);
          delete i;
          end;
  ```

- *loop and while* statements as in PSM, if-then-else, cursors
AmosQL sequences

Vectors (ordered sequences of objects)

- The datatype *vector* stores ordered sequences of objects of any type, e.g. numerical vectors, tuples.
- Vector declarations can be parameterized by declaring
  Vector of *<type>*
  e.g.

```sql
create type Segment;
create function start(Segment)->Vector of Real as stored;
create function stop(Segment)->Vector of Real as stored;
create type Polygon;
create function segments(Polygon)->Vector of Segment as stored;
```

- Vector values have system provided *constructors*:

```sql
create Segment instances :s1, :s2;
set start(:s1)={1.1,2.3};
set stop(:s1)={2.3,4.6};
set start(:s2)={2.3,4.6};
set stop(:s2)={2.8,5.3};
create Polygon instances :p1;
set segments(:p1)={: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 r * r;
  create function length(Segment l) -> Real
      as euclid(start(l), stop(l));
  create function length(Polygon p) -> Real
      as sum(select length(s) from Segment s where s in segments(p));
  ```

- Extented ER notation:

```
Polygon \rightarrow segments \rightarrow Segment
```

- Queries, e.g.:
  ```sql
  length(:s1);
  count(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
- E.g. Find the names of the supertypes of TA:
  
  \[
  \text{name(supertypes(typenamed(“TA”)));}
  \]
  
  "STUDENT"
  "INSTRUCTOR"

- Find the types of the first argument of a resolvent:
  
  \[
  \text{name(resolventtype(#’teaches’));}
  \]
  
  "INSTRUCTOR"

- Find all functions whose single argument have type INSTRUCTOR:
  
  \[
  \text{signature(methods(typenamed(‘Instructor’)));}
  \]
  
  "TEACHES(INSTRUCTOR) –> COURSE"

- Notice that the schema can be browsed by calling `goovi();` when Amos II combined with Java (javaamos.cmd).
Amos II

How to run Amos II:

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

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

• Amos II Tutorials:
  http://user.it.uu.se/~torer/kurser/dbt/tutorial.amosql
  http://user.it.uu.se/~udbl/amos/doc/tut.pdf

• These slides:
  http://user.it.uu.se/~torer/kurser/dbt/amosQL.pdf
Summary

- AmosQL provides flexible Object-Relational DBMS capabilities
- Not hard wired object model, but dynamically extensible model
- Powerful object-oriented and functional data modeling
- Very good support for ad hoc queries
- Extensible query processor

The key is the functional data model of Amos II.