9. Functional Data Integration in a Distributed Mediator System

Tore Risch[1], Vanja Josifovski[2], and Timour Katchaounov[1]

[1] Dept. of Information Technology, Uppsala University, Uppsala, Sweden
    email: Tore.Risch@dis.uu.se
[2] IBM Almaden Research Institute, San Jose, USA

Summary.
Amos II (Active Mediator Object System) is a distributed mediator system that uses a functional data model and has a relationally complete functional query language, AmosQL. Through its distributed multi-database facilities many autonomous and distributed Amos II peers can interoperate. Functional multi-database queries and views can be defined where external data sources of different kinds are translated through Amos II and reconciled through its functional mediation primitives. Each mediator peer provides a number of transparent functional views of data reconciled from other mediator peers, wrapped data sources, and data stored in Amos II itself. The composition of mediator peers in terms of other peers provides a way to scale the data integration process by composing mediation modules. The Amos II data manager and query processor are extensible so that new application oriented data types and operators can be added to AmosQL, implemented in some external programming language (Java, C, or Lisp). The extensibility allows wrapping data representations specialized for different application areas in mediator peers. The functional data model provides very powerful query and data integration primitives which require advanced query optimization.

9.1 Introduction

The mediator/wrapper approach, originally proposed by [9.42], has been used for integrating heterogeneous data in several projects, e.g. [9.16, 9.41, 9.14, 9.5]. Most mediator systems integrate data through a central mediator server accessing one or several data sources through a number of "wrapper" interfaces that translate data to a common data model (CDM). However, one of the original goals for mediator architectures [9.42] was that mediators should be relatively simple distributed software modules that transparently encode domain-specific knowledge about data and share abstractions of that data with higher layers of mediators or applications. Larger networks of mediators would then be defined through these primitive mediators by composing new mediators in terms of other mediators and data sources. The core of Amos II is an open, light-weight, and extensible database management system (DBMS) with a functional data model. Each Amos II server contains all the traditional database facilities, such as a storage manager, a recovery manager, a transaction manager, and a functional query language named

AmosQL. The system can be used as a single-user database or as a multi-user server to applications and to other Amos II peers.

9.1.1 Distribution

Amos II is a distributed mediator system where several mediator peers communicate over the Internet. Each mediator peer appears as a virtual functional database layer having data abstractions and a functional query language. Functional views provide transparent access to data sources from clients and other mediator peers. Conflicts and overlaps between similar real-world entities being modeled differently in different data sources are reconciled through the mediation primitives [9.18, 9.17] of the multi-mediator query language AmosQL. The mediation services allow transparent access to similar data structures represented differently in different data sources. Applications access data from distributed data sources through queries to views in some mediator peer.

Logical composition of mediators is achieved when *multi-database views* in mediators are defined in terms of views, tables, and functions in other mediators or data sources. The multi-database views make the mediator peers appear to the user as a single virtual database. Amos II mediators are composable since a mediator peer can regard other mediator peers as data sources.

9.1.2 Wrappers

In order to access data from external data sources Amos II mediators may contain one or several *wrappers* which process data from different kinds of external data sources, e.g. ODBC-based access to relational databases [9.11, 9.4], access to XML files [9.28], CAD systems [9.25], or Internet search engines [9.22]. A wrapper is a program module in Amos II having specialized facilities for query processing and translation of data from a particular class of external data sources. It contains both interfaces to external data sources and knowledge of how to efficiently translate and process queries involving accesses to a class of external data sources. In particular, external Amos II peers known to a mediator are also regarded as external data sources and there is a special wrapper for accessing other Amos II peers. However, among the Amos II peers special query optimization methods are used that take into account the distribution, capabilities, costs, etc., of the different peers [9.20].

9.1.3 The Name Server

Every mediator peer must belong to a group of mediator peers. The mediator peers in a group are described through a meta-schema stored in a mediator server called *name server*. The mediator peers are autonomous and there is no central schema in the name server. The name server contains only some

general meta-information such as the locations and names of the peers in the group while each mediator peer has its own schema describing its local data and data sources. The information in the name server is managed without explicit operator intervention; its content is managed through messages from the mediator peers. To avoid a bottleneck, mediator peers usually communicate directly without involving the name server; it is normally involved only when a connection to some new mediator peer is established.

9.1.4 AmosQL

AmosQL is a functional language having its roots in the functional query languages OSQL [9.31] and DAPLEX [9.37] with extensions of mediation primitives [9.18, 9.17], multi-directional foreign functions [9.29], late binding [9.13], active rules [9.38], etc. Queries are specified using the select–from–where construct as in SQL. AmosQL furthermore has aggregation operators, nested subqueries, disjunctive queries, quantifiers, and is relationally complete.

9.1.5 Query Optimization

The declarative multi-database query language AmosQL requires queries to be optimized before execution. The query compiler translates AmosQL statements first into *object calculus* and then into *object algebra* expressions. The object calculus is expressed in an internal simple logic based language called ObjectLog [9.29], which is an object-oriented dialect of Datalog. As part of the translation into object algebra programs, many optimizations are applied on AmosQL expressions relying on its functional and multi-database properties. During the optimization steps, the object calculus expressions are re-written into equivalent but more efficient expressions. For distributed multi-database queries a multi-database query decomposer [9.20] distributes each object calculus query into local queries executed in the different distributed Amos II peers and data sources. For better performance, the decomposed query plans are rebalanced over the distributed Amos II peers [9.17]. A cost-based optimizer on each site translates the local queries into procedural execution plans in the object algebra, based on statistical estimates of the cost to execute each generated query execution plan expressed in the object algebra. A query interpreter finally interprets the optimized algebra to produce the result of a query.

9.1.6 Multi-Directional Foreign Functions

The query optimizer is extensible through a generalized foreign function mechanism, *multi-directional foreign functions*. It gives transparent access from AmosQL to special purpose data structures such as internal Amos II

meta-data representations or user defined storage structures. The mechanism allows the programmer to implement query language operators in an external language (Java, C, or Lisp) and to associate costs and selectivity estimates with different user-defined access paths. The architecture relies on extensible optimization of such foreign function calls [9.29]. They are important both for accessing external query processors [9.4] and for integrating customized data representations from data sources.

9.1.7 Organization

Next the distributed mediator architecture of Amos II is described. Then the functional data model used in Amos II is described along with its functional query language followed by a description of how the basic functional data model is extended with data integration primitives. After that there is an overview of the distributed multi-mediator query processing. Finally, related work is discussed followed by a summary.
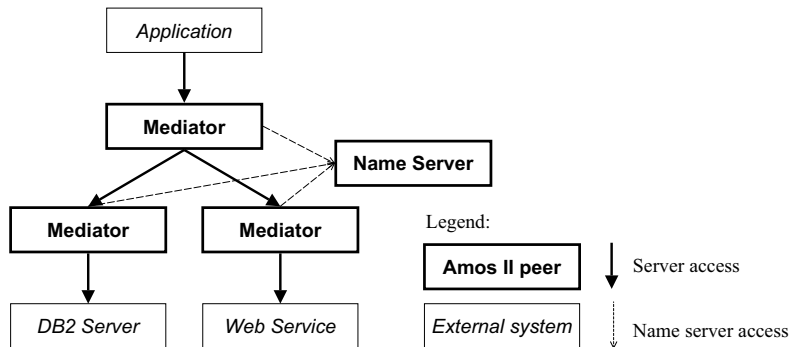
9.2 Distributed Mediation

Groups of distributed Amos II peers can interoperate over a network using TCP/IP. This is illustrated by Figure 9.1 where an application accesses data from two distributed data sources through three distributed mediator peers. The thick lines indicate communication between peers where the arrows indicate peers acting as servers.

The *name server* is a mediator peer storing names, locations, and other general data about the mediators in a group. As illustrated by the dashed lines, mediators in a group communicate with the name server to register themselves in the group or obtain information about other peers.

The figure furthermore illustrates that several layers of mediator peers can call other mediator peers. Notice, however, that the communication topology is dynamic and any peer can communicate directly with any other peer or data source in a group. It is up to the distributed mediator query optimizer to automatically come up with the optimal communication topology between the peers for a given query. The query optimizers of the peers can furthermore exchange both data and schema information in order to produce an optimized distributed execution plan.

In the figure, the uppermost mediator defines mediating functional views integrating data from them. The views include facilities for semantic reconciliation of data retrieved from the two lower mediators.

The two lower mediators translate data from a wrapped relational database and a web server, respectively. They have knowledge of how to translate AmosQL queries to SQL [9.11] through JDBC and, for the web server, to web service requests.

Fig. 9.1. Distributed mediator communication

When an Amos II system is started, it initially assumes stand-alone single-user mode of operation in which no communication with other Amos II systems can be done. The stand-alone system can join a group by issuing a *registration* command to the name server of the group. Another system command makes the mediator a peer that accepts incoming commands from other peers in the group.

In order to access data from external data sources Amos II mediators may contain one or several *wrappers* to interface and process data from external data sources. A wrapper is a program module in a mediator having specialized facilities for query processing and translation of data from a particular kind of external data sources. It contains interfaces to external data repositories to obtain both meta-data (schema definitions) and data. It also includes data source specific rewrite rules to efficiently translate and process queries involving accesses to a particular kind of external data source. More specifically the wrappers perform the following functions:

– *Schema importation* translates schema information from the sources into a set of Amos II types and functions.
– *Query translation* translates internal calculus representations of AmosQL queries into equivalent API calls or query language expressions executable by the source.
– *Source statistics computation* estimates costs and selectivities for API calls or query expressions to a data source.
– *Proxy OID generation* executes in the source query expressions or API calls to construct *proxy OIDs* describing source data.
– *OID verification* executes in the source query expressions or API calls to verify the validity of involved proxy OIDs, in case they have become invalid between different query requests.

Once a wrapper has been defined for a particular kind of source, e.g. ODBC or a web service, the system knows how to process any AmosQL query or view definition for all such sources. When integrating a new instance of the source the mediator administrator can define a set of views in AmosQL that provide abstractions of it.

Different types of applications require different interfaces to the mediator layer. For example, there are call level interfaces allowing AmosQL statements to be embedded in the programming languages Java, C, and Lisp. The call-in interface for Java has been used for developing a Java-based multi-database object browser, GOOVI [9.6].

The Amos II kernel can also be extended with plug-ins for customized query optimization, fusion of data, and data representations (e.g. matrix data). Often specialized algorithms are needed for operating on data from a particular application domain. Through the plug-in features of Amos II , domain-oriented algorithms can easily be included in the system and made available as new query language functions in AmosQL. It is furthermore possible to add new query transformation rules (rewrite rules) for optimizing queries over the new domain.

9.3 Functional Data Model

The data model of Amos II is an extension of the Daplex [9.37] functional data model. The basic concepts of the data model are *objects, types,* and *functions*.

9.3.1 Objects

Objects model all entities in the database. The system is reflective is the sense that everything in Amos II is represented as objects managed by the system, both system and user-defined objects. There are two main kinds of representations of objects: *literals* and *surrogates*. The surrogates have associated object identifiers (OIDs), which are explicitly created and deleted by the user or the system. Examples of surrogates are objects representing real-world entities such as persons, meta-objects such as functions, or even Amos II mediators as meta-mediator objects.

The literal objects are self-described system-maintained objects which do not have explicit OIDs. Examples of literal objects are numbers and strings. Literal objects can also be *collections*, representing collections of other objects. The system-supported collections are *bags* (unordered sets with duplicates) and *vectors* (order-preserving collections). Literals are automatically deleted by an incremental garbage collector when they are no longer referenced in the database.

9.3.2 Types

Objects are classified into *types* making each object an *instance* of one or several types. The set of all instances of a type is called the *extent* of the type. The types are organized in a multiple inheritance, supertype/subtype hierarchy. If an object is an instance of a type, then it is also an instance of all the supertypes of that type; conversely, the extent of a type is a subset of all extents of the supertypes of that type (extent-subset semantics). For example, if the type `Student` is a subtype of type `Person`, the extent of type `Student` is also a subset of the extent of type `Person`. The extent of a type which is multiple inherited from other types is a subset of the intersection of its supertypes' extents.

There are two kinds of types, *stored* and *derived* types. Derived types are used mainly for data reconciliation and are described in the next section. Stored types are defined and stored in an Amos II peer through the `create type` statement, e.g.:

```
create type Person;
create type Student under Person;
create type Teacher under Person;
create type TA under Student, Teacher;
```

The above statements extend the database schema with four new types. A `TA` object is both a `Student` and a `Teacher`. The extent of type `Person` is the union of all objects of types `Person, Student, Teacher,` and `TA`. The extent of type `TA` is the intersection of the extents of types `Teacher` and `Student`.

All objects in the database are typed, including meta-objects such as those representing the types themselves. The meta-objects representing types are also stored types and instances of the meta-type named `Type`. In the example the extent of the type named `Type` is the meta-objects representing the types named `TA, Teacher, Student,` and `Person`.

The root in the type hierarchy is the system type named `Object`. The system type `Userobject` is the root of all user-defined types and the extent of type `Userobject` contains all user-defined objects in the database.

The major root types in the type hierarchy are illustrated by the *function diagram* on Figure 9.2 where ovals denote types, thin arrows denote functions, thick arrows denote type inheritance, and literal function result types are omitted for readability. The type `Datasource` and its subtypes and functions are explained later in section 9.4.2.

Every object has an associated *type set*, which is the set of those types that the object is an instance of. Every object also has one *most specific type* which is the type specified when the object is created. The full type set includes the most specific type and all types above the type in the type hierarchy. For example, objects of type `TA` have the most specific type named `TA` while its full type set is {`TA, Teacher, Student, Person, Userobject, Object`}.
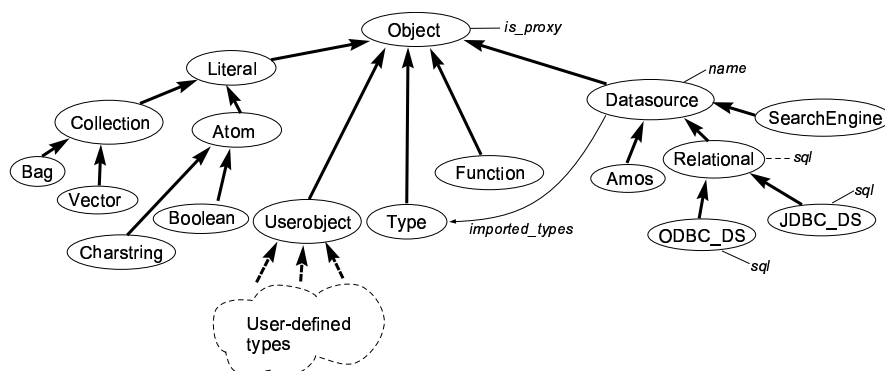
Fig. 9.2. System type hierarchy

The type set of an object can dynamically change during the lifetime of the object through AmosQL statements that change the most specific type of an object. The reason for such facilities is because the *role* of an object may change during the lifetime of the database. For example, a TA might become a student for a while and then a teacher.

9.3.3 Functions

Functions model the semantics (meaning) of objects. They model properties of objects, computations over objects, and relationships between objects. They furthermore are basic primitives in functional queries and views. Functions are instances of the system type `Function`.

A function consists of two parts, the *signature* and the *implementation*.

The *signature* defines the types, and optional names, of the argument(s) and the result of a function. For example, the signature of the function modeling the attribute `name` of type `Person` would have the signature:

        name(Person)->Charstring

Functions can be defined to take any number of arguments, e.g. the arithmetic addition function implementing the infix operator "+" has the signature:

        plus(Number,Number)->Number

The *implementation* specifies how to compute the result of a function given a tuple of argument values. For example, the function `plus` computes the result by adding the two arguments, and `name` obtains the name of a person by accessing the database. The implementation of a function is normally non-procedural, i.e. a function only computes result values for given arguments and does not have any side-effects. The exception is *database procedures* defined through procedural AmosQL statements.

Furthermore, Amos II functions are often *multi-directional*, meaning that the system is able to inversely compute one or several argument values if (some part of) the expected result value is known [9.29]. Inverses of multi-directional functions can be used in database queries and are important for specifying general queries with function calls over the database. For example, the following query, which finds the age of the person named "Tore", uses the inverse of function `name` to avoid iterating over the entire extent of type `Person`:

```
select age(p) from Person p where name(p)='Tore';
```

Depending on their implementation the basic functions can be classified into *stored*, *derived*, and *foreign* functions. In addition, there are *database procedures* with side-effects and *proxy* functions for multi-mediator access as explained later.

- *Stored functions* represent properties of objects (attributes) locally stored in an Amos II database. Stored functions correspond to attributes in object-oriented databases and tables in relational databases.
- *Derived functions* are functions defined in terms of functional queries over other Amos II functions. Derived functions cannot have side effects and the query optimizer is applied when they are defined. Derived functions correspond to side-effect free methods in object-oriented models and views in relational databases. AmosQL has an SQL-like *select* statement for defining derived functions and ad hoc queries.
- *Foreign functions* provide the low-level interfaces for wrapping external systems from Amos II. For example, data structures stored in external storage managers can be manipulated through foreign functions. Foreign functions can also be defined for updating external data structures, but foreign functions to be used in queries must be side-effect free.
  Foreign functions correspond to methods in object-oriented databases. Amos II furthermore provides a possibility to associate several implementations of inverses of a given foreign function, *multi-directional foreign functions*, which informs the query optimizer that there are several access paths implemented for the function. To help the query processor, each associated access path implementation may have associated cost and selectivity functions. The multi-directional foreign functions provide access to external storage structures similar to data "blades", "cartridges", or "extenders" in object-relational databases.
- *Database procedures* are functions defined using a procedural sublanguage of AmosQL. They correspond to methods with side-effects in object-oriented models and constructors. A common usage is for defining constructors of objects along with associated properties.

Amos II functions can furthermore be *overloaded*, meaning that they can have different implementations, called *resolvents*, depending on the type(s)

of their argument(s). For example, the salary may be computed differently for types `Student` and `Teacher`. Resolvents can be any of the basic function types[1]. Amos II's query compiler chooses the resolvent based on the types of the argument(s), but not the result.

The *extent* of a function is a set of tuples mapping its arguments and its results. For example, the extent of the function defined as

```
create function name(Person)-> Charstring as stored;
```

is a set of tuples $< P_i, N_i >$ where P_i are objects of type `Person` and N_i are their corresponding names. The extent of a stored function is stored in the database and the extent of a derived function is defined by its query. The extents are accessed in database queries.

The structure of the data associated with types is defined through a set of function definitions. For example:

```
create function name(Person) -> Charstring as stored;
create function birthyear(Person) -> Integer as stored;

create function hobbies(Person) -> Bag of Charstring
                                      as stored;
create function name(Course) -> Charstring as stored;

create function teaches(Teacher) -> Bag of Course
                                      as stored;
create function enrolled(Student) -> Bag of Course
                                      as stored;
create function instructors(Course c) -> Bag of Teacher t
      as
      select t
      where teaches(t) = c; /* Inverse of teaches */
```

The above stored function and type definitions can be illustrated with the function diagram of Figure 9.3.

The function `name` is overloaded on types `Person` and `Course`. The function `instructors` is a derived function that uses the inverse of function `teaches`. The functions `hobbies, teaches`, and `enrolled` return sets of values. If `Bag of` is declared for the value of a stored function it means that the result of the function is a bag (multiset)[2], otherwise it is an atomic value.

Functions (attributes) are inherited so the above statement will make objects of type `Teacher` have the attributes `name, birthyear, hobbies`, and `teaches`.

We notice here that single argument Amos II *functions* are similar to *relationships* and *attributes* in the entity–relationship (ER) model and that

---

[1] A resolvent cannot be overloaded itself, though.
[2] DAPLEX uses the notation `->>` for sets.
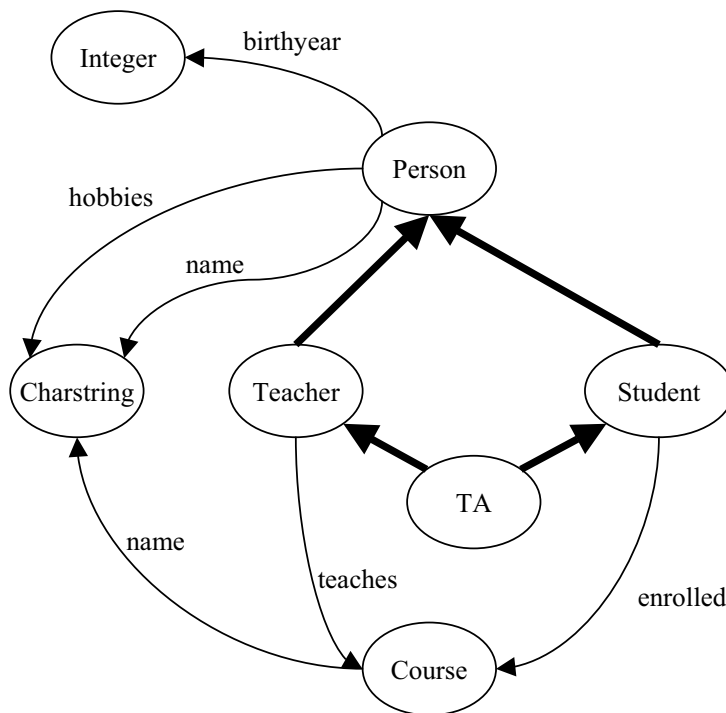
Fig. 9.3. Function diagram

Amos II *types* are similar to ER *entities*. The main difference between an Amos II function and an ER relationship is that Amos II functions have a logical *direction* from the argument to the result, while ER entities are direction neutral. Notice that Amos II functions normally are invertible and thus can be used in the inverse direction too. The main difference between Amos II types and the entities in the basic ER model is that Amos II types can be inherited.

Multi-Directional Foreign Functions

As a very simple example of a multi-directional foreign function, assume we have an external disk-based hash table on strings to be accessed from Amos II. We can then implement it as follows:

```
create function get_string(Charstring x)-> Charstring r
    as foreign "JAVA:Foreign/get_hash";
```

Here the foreign function `get_string` is implemented as a Java method `get_hash` of the public Java class `Foreign`. The Java code is dynamically

loaded when the function is defined or the mediator initialized. The Java
Virtual Machine is interfaced with the Amos II kernel through the Java Na-
tive Interface to C.

Multi-directional foreign functions include declarations of inverse foreign
function implementations. For example, our hash table can not only be ac-
cessed by keys but also scanned, allowing queries to find all the keys and
values stored in the table. We can generalize it by defining:

```
create function get_string(Charstring x)->Charstring y
as multidirectional
    ("bf" foreign "JAVA:Foreign/get_hash"
          cost {100,1})
    ("ff" foreign "JAVA:Foreign/scan_hash"
          cost "scan_cost");
```

Here, the Java method `scan_hash` implements scanning of the external hash
table. Scanning will be used, for example, in queries retrieving the hash key
for a given hash value. The *binding patterns*, `bf` and `ff`, indicate whether the
argument or result of the function must be bound (`b`) or free (`f`) when the
external method is called.

The cost of accessing an external data source through an external method
can vary heavily depending on, e.g. the binding pattern, and, to help the query
optimizer, a foreign function can have associated costing information defined
as user functions. The `cost` specifications estimate both *execution costs* in
internal cost units and *result sizes* (fanouts) for a given method invocation. In
the example, the cost specifications are constant for `get_hash` and computed
through the Amos II function `scan_cost` for `scan_hash`.

The basis for the multi-directional foreign function was developed in
[9.29], where the mechanisms are further described.

9.3.4 Queries

General queries are formulated through the `select` statement with format:

```
select <result>
from   <type extents>
where  <condition>
```

For example:

```
select name(p), birthyear(p)
from   Person p
where  birthyear(p) > 1970;
```

The above query will retrieve a tuple of the names and birth years of all
persons in the database born after 1970.

In general the semantics of an AmosQL query is as follows:

1. Form the cartesian product of the *type extents*.
2. Restrict the cartesian product by the *condition*.
3. For each possible variable binding to tuple elements in the restricted cartesian product, evaluate the result expressions to form a result tuple.
4. Result tuples containing NIL are not included in the result set; queries are *null intolerant*.

It would be very inefficient to directly use the above semantics to execute a query. It is therefore necessary for the system to do extensive query optimization to transform the query into an efficient execution strategy. Actually, unlike in SQL, AmosQL permits formulation of queries accessing indefinite extents and such queries are not executable at all without query optimization. For example, the previous query could also have been formulated as:

```
select nm, b
from Person P, Charstring nm, Integer b
where b = birthyear(p) and
      nm = name(p) and
      b > 1970;
```

In this case, the cartesian product of all persons, integers, and strings is infinite so the above query is not executable without query optimization.

Some functions may not have a fully computable extent, e.g. arithmetic functions have an infinitely large extent. Queries over infinite extents are not executable, e.g. the system will refuse to execute this query:

```
select x+1 from Number x;
```

9.4 Functional Mediation

For supporting multi-database queries, the basic data model is extended with *proxy* objects, types, and functions. Any object, including meta-objects, can be defined by Amos II as a proxy object by associating with it a property describing its source. The proxy objects allow data and meta-data to be transparently exchanged between mediator peers.

On top of this, reconciliation of conflicting data is supported through regular stored and derived functions and through *derived types* (DTs) [9.18, 9.19] that define types through declarative multi-database queries.

9.4.1 Proxy Objects

The distributed mediator architecture requires the exchange of objects and meta-data between mediator peers and data sources. To support multi-database queries and views, the basic concepts of objects, types, and functions are generalized to include also *proxy objects*, *proxy types*, and *proxy functions*:

– *Proxy objects* in a mediator peer are local OIDs having associated descriptions of corresponding objects stored in other mediators or data sources. They provide a general mechanism to define references to remote objects.
– *Proxy types* in a mediator peer describe types represented in other mediators or data sources. The proxy objects are instances of some proxy types and the extent of a proxy type is a set of proxy objects.
– Analogously, *proxy functions* in a mediator peer describe functions in other mediators or sources.

The proxy objects, types, and functions are implicitly created by the system in the mediator where the user makes a multi-database query, e.g.:

```
select name(p) from Personnel@Tb p;
```

This query retrieves the names of all persons in a data source named `Tb`. It causes the system to internally generate a proxy type for `Personnel@Tb` in the mediator server where the query is issued, M. It will also create a proxy function `name` in M representing the function `name` in `Tb`. In this query it is not necessary or desirable to create any proxy instances of type `Personel@Tb` in M since the query is not retrieving their identities. The multi-database query optimizer will here make such an optimization.

Proxy objects can be used in combination with local objects. This allows for general multi-database queries over several mediator peers. The result of such queries may be literals (as in the example), proxy objects, or local objects. The system stores internally information about the origin of each proxy object so it can be identified properly. Each local OID has a locally unique OID number and two proxy objects are considered equal if they represent objects created in the same mediator or source with equal OID numbers.

Proxy types can be used in function definitions as any other type. In the example one can define a derived function of the persons located in a certain location:

```
create function personnel_in(Charstring l) -> Personnel@Tb
    as select p from Personnel@Tb p
        where  location(p) = l;
```

In this case the local function `personnel_in` will return those instances of the proxy type for `Personnel` in the mediator named `Tb` for which it holds that the value of function `location` in `Tb` returns 1. The function can be used in local queries and function definitions, and as proxy functions in multi-database queries from other mediator peers.

Multi-database queries and functions are compiled and optimized through a distributed query decomposition process fully described in [9.20] and summarized later. Notice again that there is no central mediator schema and the compilation and execution of multi-database queries is made by exchanging data and meta-data with the accessed mediator servers. If some schema of a mediator server is modified, the multi-database functions accessing that mediator server become invalid and must be recompiled.

9.4.2 Data Source Modeling

Information about different data sources is represented explicitly in the
Amos II data model through the system type `Datasource` and its subtypes
(Figure 9.2). Some subtypes of `Datasource` represent generic kinds of data
sources that share common properties, such as the types `Relational` and
`SearchEngine` [9.22] representing the common properties of all RDBMSs and
all Internet search engines, respectively. Other subtypes of `Datasource` like
`ODBC_DS` and `JDBC_DS` represent specific kinds of sources, such as ODBC and
JDBC drivers. In particular the system type `Amos` represents other Amos II
peers. Instances of these types represent individual data sources. All types
under `Datasource` are collectively called the *datasource types*.

Since wrappers and their corresponding datasource types interact tightly,
every wrapper module installs its corresponding types and functions whenever
initialized. This reflexive design promotes code and data reuse and provides
transparent management of information about data sources via the Amos II
query language.

Each datasource type instance has a unique name and a set of imported
types. Some of the (more specific) subtypes have defined a set of low-level
access functions. For example, the type `Relational` has the function `sql`
that accepts any relational data source instance, a parametrized SQL query,
and its parameters. Since there is no generic way to access all relational data
sources this function only defines an interface. On the other hand the type
`ODBC_DS` overloads this function with an implementation that can submit a
parametrized query to an ODBC source. These functions can be used in low-
level mediator queries, which roughly corresponds to the *pass-through* mode
defined in the SQL-MED standard [9.32]. However, normally the low-level
data access functions are not used directly by the users. Instead queries that
refer to external sources are rewritten by the wrapper modules in terms of
these functions. In addition datasource types may include other functions,
such as source address, user names, and passwords.

9.4.3 Reconciliation

Proxy objects provide a general way to query and exchange data between
mediators and sources. However, reconciliation requires types defined in terms
of data in different mediators. For this, the basic system is extended with
derived types (DTs), which are types defined in terms of queries defining
their extents. These *extent queries* may access both local and proxy objects.

Data integration by DTs is performed by building a hierarchy of DTs
based on local types and types imported from other data sources. The tra-
ditional inheritance mechanism, where the corresponding instances of an ob-
ject in the super/subtypes are identified by the same OID, is extended with
declarative query specification of the correspondence between the instances
of the derived super/subtypes. Integration by sub/supertyping is related to

the mechanisms in some other systems such as the integrated views and column adding in the Pegasus system [9.9], but is better suited for use in an object-oriented environment.

The extents of derived subtypes are defined through queries restricting the intersection of the extents of the constituent supertypes. For example:

```
create derived type CSD_emp under Personnel p
    where location(p)=''CSD'';
```

This statement creates a DT CSD_emp whose extent contains those persons who work in the CSD department. When a DT is queried the system will implicitly create those of its instance OIDs necessary to execute the query.

An important purpose of DTs is to define types as views that reconcile differences between types in different mediator servers. For example, the type Personnel might be defined in mediator Tb while Ta has a corresponding type Faculty. The following statement executed in a third mediator, M, defines a DT Emp in M representing those employees who work both in Ta and Tb:

```
create derived type Emp
    under Faculty@Ta f, Personnel@Tb p
    where ssn(f)=id_to_ssn(id(p))
```

Here the where clause identifies how to match equivalent proxy objects from both sources. The function ssn uniquely identifies faculty members in Ta, while the function id in Tb identifies personnel by employee numbers. A (foreign) function id_to_ssn in M translates employee numbers to SSNs.

The system internally maintains the information necessary to map between OIDs of a DT and its supertypes.

An important issue in designing object views is the placement of the DTs in the type hierarchy. Mixing freely the DTs and ordinary types in a type hierarchy can lead to semantically inconsistent hierarchies [9.24]. In order to provide the user with powerful modeling capabilities along with a semantically consistent inheritance hierarchy, the ordinary types and DTs in Amos II are placed in a single type hierarchy where it is not allowed to have an ordinary type as a subtype of a DT. This rule preserves the extent–subset semantics for all types in the hierarchy. If DTs were allowed to be supertypes of ordinary types, due to the declarative specification of the DTs, it would not have been possible to guarantee that each instance of the ordinary type has a corresponding instance in its supertypes [9.24].

The DT instances are derived from the instances of their supertypes according to an extent query specified in the DT definition. DT instances are assigned OIDs by the system, which allows their use in locally stored functions defined over the DTs in the same way as over the ordinary types. A selective OID generation for the DT instances is used to avoid performance and storage overhead.

The concept of DTs and its use for data integration is fully described in [9.18]. The regular DTs, defined by subtyping through queries of their super-

types, provide means for mediation based on operators such as join, selection, and projection. However, these do not suffice for integration of sources having overlapping data. When integrating data from different mediator servers it is often the case that the same entity appears either in one of the mediators or in both. For example, if one wants to combine employees from different departments, some employees will only work in one of the departments while others will work in both of them.

For this type of integration requirements the Amos II system features a special kind of DTs called *Integration Union Types* (IUTs) defined as supertypes of other types through queries. IUTs are used to model unions of real-world entities represented by overlapping type extents. Informally, while the regular DTs represent restrictions and intersections of extents of other types, the IUTs represent reconciled unions of (possibly overlapping) data in one or more mediator server or data sources. The example in Figure 9.4 illustrates the features and the applications of the IUTs.
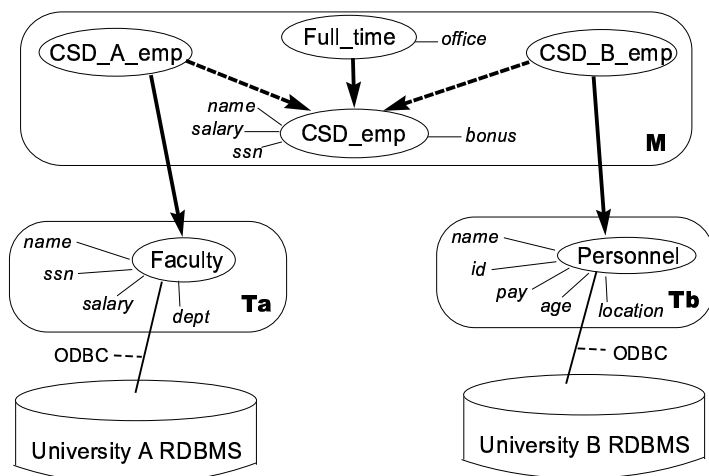
Fig. 9.4. An object-oriented view for the computer science department.

In this example, a computer science department (CSD) is formed out of the faculty members of two universities named *A* and *B*. The CSD administration needs to set up a database of the faculty members of the new department in terms of the databases of the two universities. The faculty members of CSD can be employed by either one of the universities. There are also faculty members employed by both universities. The full-time members of a department are assigned an office in the department.

In Figure 9.4 the mediators are represented by rectangles; the ovals in the rectangles represent types, and the solid lines represent inheritance relationships between the types. The two mediators `Ta` and `Tb` provide Amos II

views of the relational databases *University A DB* and *University B DB*. In mediator `Ta` there is a type `Faculty` and in mediator `Tb` a type `Personnel`.

The relational databases are accessed through an ODBC wrapper in `Ta` and `Tb` that translates AmosQL queries into ODBC calls. The ODBC wrapper interface translates AmosQL queries over objects represented in relations into calls to a foreign function executing SQL statements [9.4]. The translation process is based on partitioning general queries into subqueries only using the capabilities of the data source, as fully explained in [9.20].

A third mediator M is set up in the CSD to provide the integrated view. Here, the semantically equivalent types `CSD_A_emp` and `CSD_B_emp` are defined as derived subtypes of types in `Ta` and `Tb`:

```
create derived type CSD_a_emp
  under Faculty@Ta f
  where dept(f) = 'CSD';


create derived type CSD_b_emp
  under Personnel@Tb p
  where location(p) = 'Building G';
```

The system imports the external types, looks up the functions defined over them in the originating mediators, and defines local proxy types and functions with the same signature but without local implementations.

The IUT `CSD_emp` represents all the employees of the CSD. It is defined over the *constituent subtypes* `CSD_a_emp` and `CSD_b_emp`. `CSD_emp` contains one instance for each employee object regardless of whether it appears in one of the constituent types or in both. There are two kinds of functions defined over `CSD_emp`. The functions on the left of the type oval in Figure 9.4 are derived from the functions defined in the constituent types. The functions on the right are locally stored.

The data definition facilities of AmosQL include constructs for defining IUTs as described above. The integrated types are internally modeled by the system as subtypes of the IUT. Equality among the instances of the integrated types is established based on a set of key attributes. IUTs can also have locally stored attributes, and attributes reconciled from the integrated types. See [9.19] for details.

The type `CSD_emp` is defined as follows:

```
CREATE INTEGRATION TYPE CSD_emp
  KEYS ssn Integer;
  SUPERTYPE OF
    CSD_A_emp ae: ssn = ssn(ae);
    CSD_B_emp be: ssn = id_to_ssn(id(be));
  FUNCTIONS
    CASE ae
      name = name(ae);
```

```
        salary = pay(ae);
      CASE be
        name = name(be);
        salary = salary(be);
      CASE ae, be
        salary = pay(ae) + salary(be);
    PROPERTIES
        bonus Integer;
  END;
```

For each of the constituent subtypes, a `KEYS` clause is specified. The instances of different constituent types having the same key values will map into a single IUT instance. The key expressions can contain calls to any function.

The `FUNCTIONS` clause defines the reconciled functions of `CSD_emp`, derived from functions over the constituent subtypes. For different subsets of the constituent subtypes, a reconciled function of an IUT can have different implementations specified by the `CASE` clauses. For example, the definition of `CSD_emp` specifies that the `salary` function is calculated as the salary of the faculty member at the university to which it belongs. In the case when s/he is employed by both universities, the salary is the sum of the two salaries. When the same function is defined for more than one case, the most specific case applies. Finally, the `PROPERTIES` clause defines the stored function `bonus` over the IUT `CSD_emp`.

The IUTs can be subtyped by DTs. In Figure 9.4, the type `Full_Time` is defined as a subtype of the `CSD_emp` type, representing the instances for which the salary exceeds a certain number (50000). The locally stored function `office` stores information about the offices of the full-time CSD employees. The type `Full_Time` and its property `office` have the following definitions:

```
    create derived type Full_Time under CSD_emp e
                              where salary(e)>50000;
    create function office(Full_Time)->Charstring
        as stored;
```

9.5 Query Processing

The description of type hierarchies and semantic heterogeneity using declarative multi-database functions is very powerful. However, a naive implementation of the framework could be very inefficient, and there are many opportunities for the extensive query optimization needed for distributed mediation.

The query processor of Amos II, illustrated by Figure 9.5, consists of three main components. The core component of the query processor is the *local query compiler* that optimizes queries accessing local data in a mediator. The
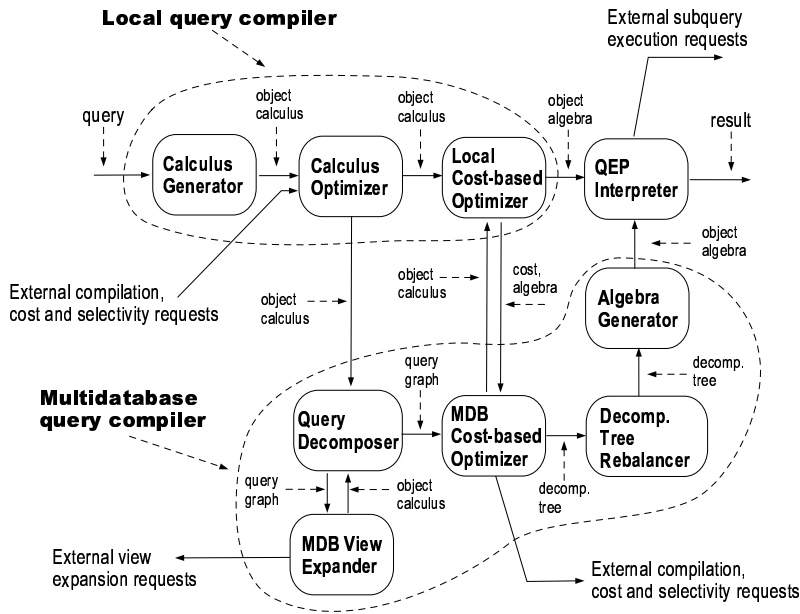
Fig. 9.5. Query processing in Amos II

Multi-database Query Compiler MQC, allows Amos II mediators to process queries that also access other mediator peers and data sources. Both compilers generate query execution plans (*QEPs*) in terms of an *object algebra* that is interpreted by the *QEP interpreter* component. The following two sections describe in more detail the subcomponents of the local and the multi-database query compilers.

9.5.1 Local Query Processing

To illustrate the query compilation of single-site queries we use the sample ad hoc query:

```
select p, name(parent(p))
  from person p
  where hobby(p) = 'sailing';
```

The first query compilation step, *calculus generation*, translates the parsed AmosQL query tree into an *object calculus* representation called ObjectLog [9.29]. The object calculus is a declarative representation of the original query and is an extension of Datalog with objects, types, overloading, and multi-directional foreign functions.

The calculus generator translates the example query into this expression:

$$\{\ p, nm\ |$$
$$p = Person_{nil \rightarrow Person}()\ \wedge$$
$$pa = parent_{Person \rightarrow Person}(p)\ \wedge$$
$$nm = name_{Person \rightarrow Charstring}(pa)\ \wedge$$
$${'sailing'} = hobby_{Person \rightarrow Charstring}(p)\}$$

The first predicate in the expression is inserted by the system to assert the type of the variable p. This *type check predicate* defines that the variable p is bound to one of the objects returned by the *extent function* for type Person, $Person()$, which returns all the instances (the extent) of its type. The variables nm and pa are generated by the system. Notice that the functions in the predicates are annotated with their type signatures, to allow for overloading of function symbols over the argument types.

The *calculus optimizer* of the query optimizer first transforms the unoptimized calculus expression to reduce the number of predicates, e.g. by exploring properties of type definitions. In the example, it removes the type check predicate:

$$\{\ p, nm\ |$$
$$pa = parent_{Person \rightarrow Person}(p)\ \wedge$$
$$nm = name_{Person \rightarrow Charstring}(pa)\ \wedge$$
$${'sailing'} = hobby_{Person \rightarrow Charstring}(p)\}$$

This transformation is correct because p is used in a stored function (parent or hobby) with argument or result of type Person. The referential integrity system constrains instances of stored functions to be of correct types [9.29].

The *local cost-based optimizer* will use cost-based optimization to produce an executable object algebra plan from the transformed query calculus expression. The system has a built-in cost model for local data and built-in algebra operators. Basically the cost-based optimizer generates a number of execution plans, applies the cost model on each of them, and chooses the cheapest for execution. The system has the options of using dynamic programming, hill climbing, or random search to find an execution plan with minimal cost. Users can instruct the system to choose a particular strategy.

The optimizer is furthermore extensible whereby new algebra operators are defined using the multi-directional foreign functions, which also provide the basic mechanisms for interactions between mediator peers in distributed execution plans.

The *query execution plan interpreter* will finally interpret the execution plan to yield the result of the query.

9.5.2 Queries over Derived Types

Queries over DTs are expanded by system-inserted predicates performing the DT system support tasks [9.18]. These tasks are divided into three mechanisms: (i) providing consistency of queries over DTs so that the extent–subset

semantics is followed; (ii) generation of OIDs for those DT instances needed to execute the query; and (iii) validation of the DT instances with assigned OIDs so that DT instances satisfy the constraints of the DT definitions. The system generates derived function definitions to perform these tasks. During the calculus optimization the query is analyzed and, where needed, the appropriate function definitions are added to the query. A selective OID generation mechanism avoids overhead by generating OIDs only for those derived objects that are either needed during the execution of a query, or have associated local data in the mediator database.

The functions specifying the view support tasks often have overlapping parts. Reference [9.18] demonstrates how calculus-based query optimization can be used to remove redundant computations introduced from the overlap among the system-inserted expressions, and between the system-inserted and user-specified parts of the query.

Each IUT is mapped by the calculus optimizer to a hierarchy of system-generated DTs, called *auxiliary types* [9.19]. The auxiliary types represent disjoint parts of the outerjoin needed for this type of data integration. The reconciliation of the attributes of the integrated types is modeled by a set of overloaded derived functions generated by the system from the specification in the IUT definition. Several novel query processing and optimization techniques are developed for efficiently processing the queries containing overloaded functions over the auxiliary types, as described in [9.19].

9.5.3 Multi-database Query Processing

The *Multi-database Query Compiler (MQC)* [9.20, 9.17] is invoked whenever a query is posed over data from more than one mediator peer. The goal of the MQC is to explore the space of possible distributed execution plans and choose a "reasonably" cheap one. As the local query compiler, the MQC uses a combination of heuristic and dynamic programming strategies to produce a set of distributed object algebra plans.

The distributed nature of Amos II mediators requires a query processing framework that allows cooperation of a number of autonomous mediator peers. The MQC interacts with the local optimizer as well as with the query optimizers of the other mediator peers involved in the query via requests to estimate costs and selectivities of subqueries, requests to expand the view definitions of remote views, and requests to compile subqueries in remote mediator peers. The generated local execution plan interacts with the execution plans produced by the other mediator peers.

The details of the MQC are described in [9.20]. Here we will overview its main subcomponents:

– The *query decomposer* identifies fragments of a multi-database query, *subqueries*, where each subquery can be processed by a single data source. The decomposer takes as input an object calculus query and produces a *query*

graph with nodes representing subqueries assigned to an execution site and arcs representing variables connecting the subqueries. The benefit of decomposition is twofold. First, complex computations in subqueries can be pushed to the data sources to avoid expensive communication and to utilize the processing capabilities of the sources. Second, the multi-database query optimization cost is reduced by the partitioning of the input query into several smaller subqueries.

Query decomposition is performed in two steps:

1. *Predicate grouping* collects predicates executable at only one data source and groups them together into one or more subqueries. The grouping process uses a heuristic where cross-products are avoided by placing predicates without common variables in separate subqueries.
2. *Site assignment* uses a cost-based heuristics to place those predicates that can be executed at more than one site (e.g. θ-joins), eventually replicates some of the predicates in the subqueries to improve the selectivity of subqueries, and finally assigns execution sites to the subqueries.

– The *multi-database view expander* expands remote views directly or indirectly referenced in queries. This may lead to significant improvement in the query plan quality because there may be many redundancies in large compositions of multi-database views.

The multi-database view expander traverses the query graph to send expansion requests for the subqueries. In this way, all predicates defined in the same database are expanded in a single request. This approach allows the remote site to perform calculus simplifications of the expanded and merged predicate definitions as a whole and then return the transformed subquery. However, when there are many mediator layers it is not always beneficial to fully expand all view definitions, as shown in [9.21]. The multi-database view expander therefore uses a heuristic to choose the most promising views for expansion, a technique called *controlled view expansion*. After all subqueries in the query graph have been view expanded the query decomposer is called again for predicate regrouping.

– The *multi-database (MDB) query optimizer* decides on the order of execution of the predicates in the query graph nodes, and on the direction of the data shipping between the peers. Execution plans for distributed queries in Amos II are represented by *decomposition trees*. Each node in a decomposition tree describes a join cycle through a client mediator (i.e. the mediator where the query is issued). In a cycle, first intermediate results are shipped to the site where they are used. Then a subquery is executed at that site using the shipped data as input, and the result is shipped back to the mediator. Finally, one or more post-processing subqueries are performed at the client mediator. The result of a cycle is always materialized in the mediator. A sequence of cycles can represent any execution plan. As the space of all execution plans is exponential to the number of subqueries in the input query graph, we examine only the space of left-deep

decomposition trees using a dynamic programming approach. To evaluate the costs and selectivities of the subqueries the multi-database optimizer sends compilation requests for the subqueries both to the local optimizer and to the query compilers of the remote mediators.

– The *decomposition tree rebalancer* transforms the initial left-deep decomposition tree into a bushy one. To prevent all the data flowing through the client mediator, the decomposition tree rebalancer uses a heuristic that selects pairs of adjacent nodes in the decomposition tree, merges the selected nodes into one new node, and sends the merged node to the two mediators corresponding to the original nodes for recompilation. From the merged nodes, each of the two mediators generate different decomposition subtrees and the cheaper one is chosen. In this way, the input decomposition tree is rebalanced from a left-deep tree into a bushy one. The overall execution plan resulting from the tree rebalancing can contain plans where the data is shipped directly from one remote mediator to another, eliminating the bottleneck of shipping all data through a single mediator. See [9.17] for details.

– The *object algebra generator* translates a decomposition tree into a set of inter-calling local object algebra plans.

9.6 Related Work

Amos II is related to research in the areas of data integration, object views, distributed databases, and general query processing. There has been several projects on intergration of data in a multi-database environment [9.5, 9.8, 9.10, 9.12, 9.14, 9.16, 9.23, 9.27, 9.30, 9.40, 9.41]. The integration facilities of Amos II are based on work in the area of object-oriented views [9.1, 9.3, 9.15, 9.26, 9.33, 9.35, 9.36, 9.39].

Most of the mediator frameworks reported in the literature (e.g. [9.16, 9.41, 9.14]) propose centralized query compilation and execution coordination. In [9.9] it is indicated that a distributed mediation framework is a promising research direction, but to the best of our knowledge no results in this area are reported. Some recent commercial data integration products, such as IBM's Federated DB2, also provide centralized mediation features.

In the DIOM project [9.30] a framework for integration of relational data sources is presented where the operations can be executed either in the mediator or in a data source. The compilation process in DIOM is centrally performed, and there is no clear distinction between the data sources and the mediators in the optimization framework.

The Multiview [9.35] object-oriented view system provides multiple inheritance and a capacity-augmented view mechanism implemented with a technique called Object Slicing [9.26] using OID coercion in an inheritance hierarchy. However, it assumes active view maintenance and does not elaborate on the consequences of using this technique for integration of data in

autonomous and dislocated repositories. Furthermore, it is not implemented using declarative functions for the description of the view functionality.

One of the few research reports describing the use of functional view mechanisms for data integration is the Multibase system [9.8]. It is also based on a derivative of the DAPLEX data model and does reconciliation similar to the IUTs in this chapter. An important difference between Multibase and Amos II is that the data model used in Multibase does not contain the object-oriented concept of OIDs and inheritance. The query optimization and meta-modeling methods in Amos II are also more elaborate than in Multibase.

The UNISQL [9.23] system also provides views for database integration. The virtual classes (corresponding to the DTs) are organized in a separate class hierarchy. However, the virtual class instances inherit the OIDs from the corresponding instances in the ordinary classes, which prohibits definition of stored functions over virtual classes defined by multiple inheritance as in Amos II. There is no integration mechanism corresponding to the IUTs.

Reference [9.34] gives a good overview of distributed databases and query processing. As opposed to the distributed databases, where there is a centralized repository containing meta-data about the whole system, the architecture described in this paper consists of autonomous systems, each storing only locally relevant meta-data.

One of the most thorough attempts to tackle the query optimization problem in distributed databases was done within the System R* project [9.7] where, unlike Amos II, an exhaustive, cost-based, and centrally performed query optimization is made to find the optimal plan. Another classic distributed database system is SDD-1 [9.2] which used a hill-climbing heuristic like the query decomposer in Amos II.

9.7 Summary

We have given an overview of the Amos II mediator system where groups of distributed mediator peers are used to integrate data from different sources. Each mediator in a group has DBMS facilities for query compilation and exchange of data and meta-data with other mediator peers. Derived functions can be defined where data from several mediator peers is abstracted, transformed, and reconciled. Wrappers are defined by interfacing Amos II systems with external systems through its multi-directional foreign function interface. Amos II can furthermore be embedded in applications and used as stand-alone databases. The chapter gave an overview of Amos II's architecture with references to other published papers on the system for details.

We described the Functional Data Model and query language forming the basis for data integration in Amos II. The distributed multi-mediator query decomposition strategies used were summarized.

The mediator peers are autonomous without any central schema. A special mediator, the name server, keeps track of what mediator peers are members

of a group. The name servers can be queried for the location of mediator peers in a group. Meta-queries to each mediator peer can be posed to investigate the structure of its schema.

Some unique features of Amos II are:

– A distributed mediator architecture where query plans are distributed over several communicating mediator peers.
– Using declarative functional queries to model reconciled functional views spanning multiple mediator peers.
– Query processing and optimization techniques for queries to reconcile views involving function overloading, late binding, and type-aware query rewrites.

The Amos II system is fully implemented and can be downloaded from http://user.it.uu.se/∼udbl/amos. Amos II runs under Windows and Unix.

Acknowledgements

References

9.1 S. Abiteboul, A. Bonner: Objects and Views. *ACM Int. Conf. on Management of Data (SIGMOD'91)*, 238-247, 1991.
9.2 P. Bernstein, N. Goodman, E. Wong, C. Reeve, J. Rothnie Jr.: Query Processing in a System for Distributed Databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, 6(4), 602-625, 1981.
9.3 E. Bertino: A View Mechanism for Object-Oriented Databases. *3rd Int. Conf. on Extending Database Technology (EDBT'92)*, 136-151, 1992.
9.4 S. Brandani: Multi-database Access from Amos II using ODBC. *Linköping Electronic Press*, 3(19), Dec., 1998, http://www.ep.liu.se/ea/cis/1998/019/.
9.5 O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*. Prentice Hall, 1996.
9.6 K. Cassel and T. Risch: An Object-Oriented Multi-Mediator Browser. *2nd International Workshop on User Interfaces to Data Intensive Systems*, Zurich, Switzerland, 2001.
9.7 D. Daniels, P. Selinger, L. Haas, B. Lindsay, C. Mohan, A. Walker, P.F. Wilms: An Introduction to Distributed Query Compilation in R*. *2nd Int. Symp. on Distributed Data Bases*, 291-309, 1982.
9.8 U. Dayal, H-Y. Hwang: View Definition and Generalization for Database Integration in a Multidatabase System. *IEEE Transactions on Software Engineering*, 10(6), 628-645, 1984.

9.9   W. Du, M. Shan: Query Processing in Pegasus, In O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*. Prentice Hall, 449-471, 1996.

9.10 C. Evrendilek, A. Dogac, S. Nural, F. Ozcan: Multidatabase Query Optimization. *Distributed and Parallel Databases*, 5(1), 77-114, 1997.

9.11 G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. *The VLDB Journal*, 6(4), 261-281, 1997.

9.12 D. Fang, S. Ghandeharizadeh, D. McLeod, A. Si: The Design, Implementation, and Evaluation of an Object-Based Sharing Mechanism for Federated Database System. *9th Int. Conf. on Data Engineering (ICDE'93)*, IEEE, 467-475, 1993.

9.13 S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions. *21st Conf. on Very Large Databases (VLDB'95)*, 335-344, 1995

9.14 H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y.Sagiv, J. Ullman, V. Vassalos, J. Widom: The TSIMMIS Approach to Mediation: Data Models and Languages. *Journal of Intelligent Information Systems*, 8(2), 117-132, 1997.

9.15 S. Heiler, S. Zdonik: Object views: Extending the Vision. *6th Int. Conf. on Data Engineering (ICDE'90)*, IEEE, 86-93, 1990.

9.16 V. Josifovski, P. Schwarz, L. Haas, and E. Lin: Garlic: A New Flavor of Federated Query Processing for DB2, *ACM SIGMOD Conf.*, 2002.

9.17 V. Josifovski, T. Katchaounov, T. Risch: Optimizing Queries in Distributed and Composable Mediators. *4th Conf. on Cooperative Information Systems (CoopIS'99)*, 291-302, 1999.

9.18 V. Josifovski, T. Risch: Functional Query Optimization over Object-Oriented Views for Data Integration. *Journal of Intelligent Information Systems* 12(2-3), 165-190, 1999.

9.19 V. Josifovski, T. Risch: Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations. *25th Conf. on Very Large Databases (VLDB'99)*, 435-446, 1999.

9.20 V.Josifovski and T.Risch: Query Decomposition for a Distributed Object-Oriented Mediator System. *Distributed and Parallel Databases*, 11(3), 307-336, 2002.

9.21 T. Katchaounov, V. Josifovski, T. Risch: Distributed View Expansion in Object-Oriented Mediators, *5th Int. Conf. on Cooperative Information Systems (CoopIS'00), Eilat, Israel*, LNCS 1901, Springer Verlag, 2000.

9.22 T. Katchaounov, T. Risch, and S. Zürcher: Object-Oriented Mediator Queries to Internet Search Engines, *Int. Workshop on Efficient Web-based Information Systems (EWIS)*, Montpellier, France, 2002.

9.23 W. Kelley, S. Gala, W. Kim, T. Reyes, B. Graham: Schema Architecture of the UNISQL/M Multidatabase System. In W. Kim (ed.): *Modern Database Systems - The Object Model, Interoperability, and Beyond*, ACM Press, 621-648, 1995.

9.24 W. Kim and W. Kelley: On View Support in Object-Oriented Database Systems, In W. Kim (ed.), *Modern Database Systems - The Object Model, Interoperability, and Beyond*, ACM Press, 1995.

9.25 M.Koparanova and T.Risch: Completing CAD Data Queries for Visualization, *Int. Database Engineering and Applications Symp. (IDEAS 2002) Edmonton, Canada*, 2002.

9.26 H. Kuno, Y. Ra, E. Rundensteiner: *The Object-Slicing Technique: A Flexible Object Representation and Its Evaluation*. Univ. of Michigan Tech. Report CSE-TR-241-95, 1995.

9.27 E-P. Lim, S-Y. Hwang, J. Srivastava, D. Clements, M. Ganesh: Myriad: Design and Implementation of a Federated Database System. *Software - Practice and Experience*, John Wiley & Sons, 25(5), 533-562, 1995.

9.28 H. Lin, T. Risch, and T. Katchaounov: Adaptive data mediation over XML data. In special issue on Web Information Systems Applications of *Journal of Applied System Studies*, 3(2), 2002.

9.29 W. Litwin, T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *IEEE Transactions on Knowledge and Data Engineering*, 4(6), 517-528, 1992.

9.30 L. Liu, C. Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Distributed and Parallel Databases*, 5(2), 167-205, 1997.

9.31 P. Lyngbaek: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

9.32 J. Melton, J. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, and K. Zeidenstein: SQL and Management of External Data, *SIGMOD Record*, 30(1), 70-77, March 2001.

9.33 A. Motro: Superviews: Virtual Integration of Multiple Databases. *IEEE Transactions on Software Engineering*, 13(7), 785-798, 1987.

9.34 M.T. Özsu, P. Valduriez: *Principles of Distributed Database Systems*, Prentice Hall, 1999.

9.35 E. Rundensteiner, H. Kuno, Y. Ra, V. Crestana-Taube, M. Jones, P. Marron: The MultiView project: Object-Oriented View Technology and Applications, *ACM Int. Conf. on Management of Data (SIGMOD'96)*, 555, 1996.

9.36 M. Scholl, C. Laasch, M. Tresch: Updatable Views in Object-Oriented Databases. *2nd Deductive and Object-Oriented Databases Conf. (DOOD91)*, 189-207, 1991.

9.37 D. Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), 140-173, 1981.

9.38 M. Sköld, T. Risch: Using Partial Differencing for Efficient Monitoring of Deferred Complex Rule Conditions. *12th Int. Conf. on Data Engineering (ICDE'96)*, IEEE, 392-401, 1996.

9.39 C. Souza dos Santos, S. Abiteboul, C. Delobel: Virtual Schemas and Bases. *Int. Conf. on Extending Database Technology (EDBT'92)*, 81-94, 1994.

9.40 S. Subramananian, S. Venkataraman: Cost-Based Optimization of Decision Support Queries using Transient Views. *ACM Int. Conf. on Management of Data (SIGMOD'98)*, 319-330, 1998.

9.41 A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *IEEE Transactions on Knowledge and Data Engineering*, 10(5), 808-823, 1998.

9.42 G. Wiederhold: Mediators in the Architecture of Future Information Systems. *IEEE Computer*, 25(3), 38-49, 1992.