CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2001; **13**:933–953 (DOI: 10.1002/cpe.607)

Distributed data integration by object-oriented mediator servers

Tore Risch[*,†] and Vanja Josifovski[‡]

Department of Information Science, Uppsala University, P.O. Box 513, SE-751 20 Uppsala, Sweden

SUMMARY

Integration of data from autonomous, distributed and heterogeneous data sources poses several technical challenges. This paper overviews the data integration system AMOS II based on the wrapper-mediator approach. AMOS II consists of: (i) a mediator database engine that can process and execute queries over data stored locally and in several external data sources, and (ii) object-oriented (OO) multi-database views for reconciliation of data and schema heterogeneities among sources with various capabilities. The data stored in different types of data sources is translated and integrated using OO mediation primitives, providing the user with a consistent view of the data in all the sources. Through its multi-database facilities many distributed AMOS II systems can interoperate in a federation. Since most data reside in the data sources, and to achieve high performance, the core of the system is a main-memory DBMS having a storage manager, query optimizer, transactions, client–server interface, disk backup, etc. The AMOS II data manager is optimized for main-memory access and is extensible so that new data types and query operators can be added or implemented in some external programming language. The extensibility is essential for providing seamless access to a variety of data sources. Copyright © 2001 John Wiley & Sons, Ltd.

KEY WORDS: heterogeneous data integration; mediator systems; distributed databases; object-oriented databases

1. INTRODUCTION

The computing environments have become increasingly distributed through the use of Internet and other computer communication networks. Today we are experiencing an ever-increasing access to more or less structured information which is furthermore very dynamic and is continuously changing. In this environment it is getting more and more critical to develop tools for building systems that

[*]Correspondence to: Tore Risch, Department of Information Science, Uppsala University, P.O. Box 513, SE-751 20 Uppsala, Sweden.
[†]E-mail: Tore.Risch@dis.uu.se
[‡]Current address: IBM Almaden Research Center, San Jose, CA 95120, USA. E-mail: vanja@us.ibm.com

Contract/grant sponsor: Swedish Foundation for Strategic Research; contract/grant number: A3 96:34

combine relevant data from many sources and present them in a form which is comprehensible for users. These tools must be maintainable and scalable to highly dynamic and distributed environments.

Several technical problems arise in the design and implementation of data integration systems that provide the user with a unified view of data in multiple data sources. First, due to the distribution of the data sources, such a system has to operate in a distributed environment. Second, the data sources might use different data models and languages, and might contain equivalent, conflicting or complementary data, requiring reconciliation before it is presented to the user. Finally, the data sources are not under control of the data integration system, and the integration process should not affect their functionality or require modifications.

The *wrapper-mediator* approach introduced in [1], divides the functionality of a data integration system into two kinds of subsystems. The *wrappers* provide access to the data in the data sources using a *common data model* (CDM) and a *common query language*. The *mediators* provide coherent views of the data in the data sources by performing semantic reconciliation of the CDM data representations provided by the wrappers.

AMOS II[§] is a distributed mediator system where several AMOS II *mediator servers* communicate over the Internet using an internal TCP/IP socket based protocol. This protocol is used only for inter-AMOS II communication while wrappers interface the data sources, such as, e.g., wrappers for accessing XML data sources through HTTP [2] and for accessing relational databases through ODBC [3,4]. Each mediator server is also a DBMS of its own containing all the traditional database facilities, such as a storage manager, a recovery manager, a transaction manager, a disk backup manager, and a query processor for an object-oriented (OO) query language, AMOSQL [5]. AMOSQL is similar to the OO parts of SQL:99 [6] and based on the functional data model DAPLEX [7] and OSQL [8].

Each mediator server appears as a virtual OO database layer having OO data abstractions and query language. OO views provide transparent access to the data sources from clients and other mediator servers. Conflicts and overlaps between similar real-world entities being modelled differently in different data sources are reconciled through the mediation primitives [9,10] of AMOS II. The mediation services allow transparent access to similar object structures represented differently in different data sources.

AMOS II mediators are *composable* since the OO views in a mediator server can be based on the OO views in other mediator servers and data sources. The composition of mediators allows for modularity and reuse of the view definitions while avoiding the administrative and performance bottleneck of having a single mediator system with a global schema. Different interconnecting topologies can be used to compose mediator servers depending on the integration requirements of the environment.

Every mediator server must belong to a group (federation) of mediator servers. The mediator servers in a federation are described through a meta-schema stored in a mediator server called *name server*. The mediator servers 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 mediator servers in the federation while each mediator server has its own schema describing its local data and data sources. The information in the name server is managed without explicit operator intervention; its contents is managed through messages from the mediator servers. To avoid a bottleneck, mediator

[§]Active Mediator Object System.

servers usually communicate directly without involving the name server; it is normally involved only when a connection to some mediator server is established.

The AMOS II kernel in an object-oriented, open, and extensible DBMS with a relatively small footprint (about 2 MB on Windows NT 4.0). It is a main-memory (RAM) DBMS with high performance. This does not limit the capacity of AMOS II as a mediator system as it provides a *virtual* database layer on top of external data sources with limited amounts of data stored in the mediator servers. The vast bulk of the data is stored in the data sources while mainly meta-information, temporary data and data that is relevant only to particular applications is stored in the mediator servers. To minimize memory requirements during interpretation of queries over large data sets, the queries are compiled into execution plans that are interpreted in an iterative tuple-by-tuple style materializing data in the mediator only when favourable [10]. The query optimizer then inserts in the plans operators that materialize temporary data. Nevertheless, each mediator server can also store local data, e.g. for associating properties with mediated data. For example, a sales person that uses a mediator extracting prime customers from a set of data sources would store customer information in the mediator. A recovery system for the RAM storage has been developed [11] based on logging of database updates and saving complete database images. The system can be used as a single-user database or as a multi-user server to applications and to other AMOS II systems. The data manager is designed for high performance in main-memory [12] and is optimized for efficient execution when the entire database fits in main-memory.

AMOS II's predecessor, Amos [13], was built on top of the workstation version of the Iris system, WS-Iris [14], running on Unix platforms. AMOS II has a completely new kernel developed on a Windows NT/95 platform and ported to Unix platforms. AMOS II provides OO multi-database queries and reconciliation of heterogeneous data not present in Amos. Furthermore, AMOS II is designed for multi-layered distribution of mediator servers where distributed query optimization [15] allows queries to be passed through many layers of mediators without any performance degradation.

The AMOSQL query language has it roots in the functional query languages OSQL [8] and DAPLEX [7] with extensions of mediation primitives [9,10], multi-directional foreign functions [14], late binding [16], active rules [17], etc. AMOSQL is relationally complete. Queries are specified using the `select - from - where` construct as in SQL.

Due to its declarative nature, queries expressed in AMOSQL require optimization before they are executed. The query compiler translates AMOSQL statements into object calculus and algebra expressions in an internal simple logic based language called ObjectLog [14], which is an OO dialect of Datalog [18]. As part of the translation into object algebra programs, many optimizations are applied on AMOSQL expressions relying on their OO 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 *query decomposer* [19] distributes each object-calculus query into local queries to be executed in the different distributed mediator servers and data sources. A cost-based optimizer on each site translates the local queries into procedural execution plans in an OO algebra, based on statistical estimates of the cost to execute each generated query execution plan expressed in the OO algebra. A query interpreter finally interprets the optimized algebra to produce the (partial) result of a query.

The query optimizer is *extensible* through plug-ins using a generalized foreign function mechanism, multi-directional foreign functions [14]. This mechanism provides 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 [14]. They are important both for building wrappers that access external general query processors [4] and for integrating customized data representations from data sources.

The rest of this paper first describes the architecture of an AMOS II mediator server (Section 2) and how federations of distributed mediator servers are formed. In Section 3 the basic data model used in the mediator servers is described. Section 4 then describes the mediation primitives of the system. Section 5 gives an overview of the query processing. Related work is discussed in Section 6 followed by a summary.

2. ARCHITECTURE

The multi-database architecture of AMOS II allows mediator servers to connect and communicate over a network using an internal TCP/IP socked based protocol. Figure 1 illustrates how AMOS II systems in a federation can communicate and how they can be configured in different modes with respect to how they interact with other systems. The lines indicate communication between sub-systems where the arrows indicate the servers. A federation of mediator servers is managed by a *name server* which is a mediator server whose local database contains the names, locations and other general data about the mediators in the federation. The dotted lines illustrate how mediators communicate with the name server. The name server can be queried as any other mediator server which makes it possible to find and query the mediators in the federation.

Figure 1 illustrates how the system can be configured in two dimensions (Table I).

- On the *accessibility* dimension it can be a *single-user*, a *server* or an *embedded* system, where a single-user AMOS II system is a private database, a server is servicing several other AMOS II systems and an embedded system is linked to some application.
- On the *mediation* dimension it can be a *stand-alone* or a *mediator* system, where a stand-alone system is an isolated database and a mediator accesses data from some mediator(s) or data source(s).

The grey-shaded AMOS II systems in Figure 1 illustrate the following modes of operation along the two dimensions.

- (A) is an embedded mediator linked to an application program. In this configuration the system can mediate data from mediator servers, but not be used as a mediator server itself. The small footprint of an embedded AMOS II system makes it easy to link it to applications. The system has interfaces to application programs in Java, C and Lisp. Applications always access meditator servers by AMOSQL commands which are passed through an embedded AMOS II mediator.
- (B) is a single-user mediator importing and integrating data from mediator servers through the multi-database facilities, but not servicing other systems.
- (C) is a single user stand-alone database where the user can enter AMOSQL commands to populate, search and update a private database.

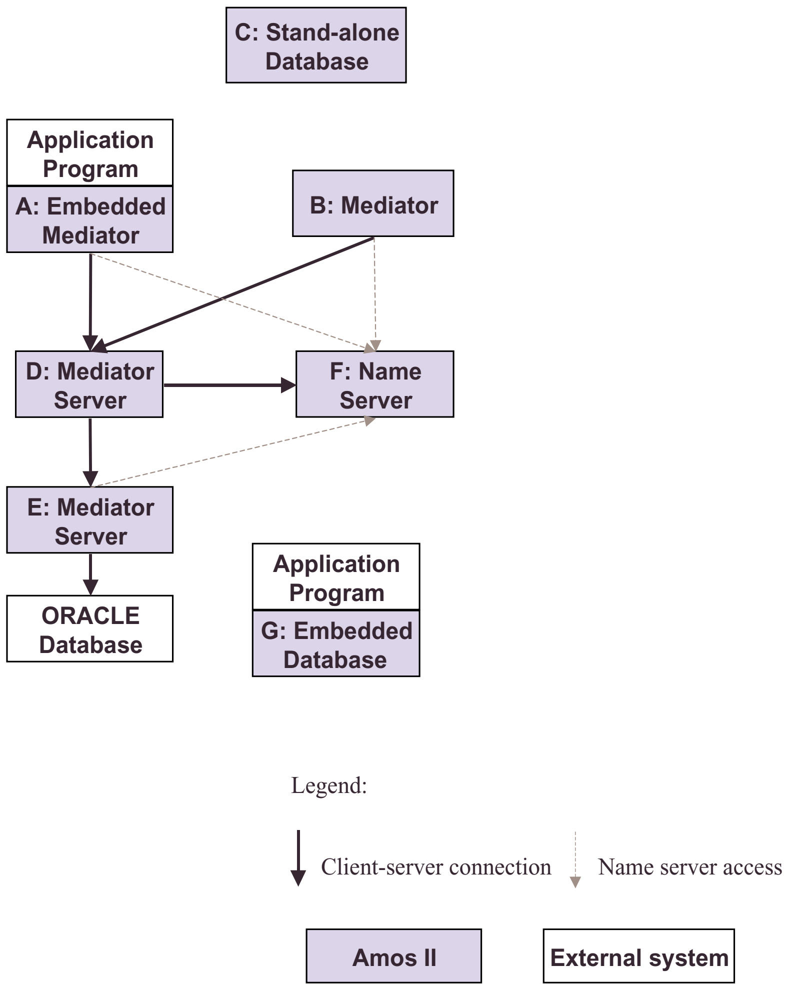

Figure 1. Distributed AMOS II configurations.

Table I. AMOS II configurations.

	Single-user	Server	Embedded
Stand-alone	C	F	G
Mediator	B	D, E	A

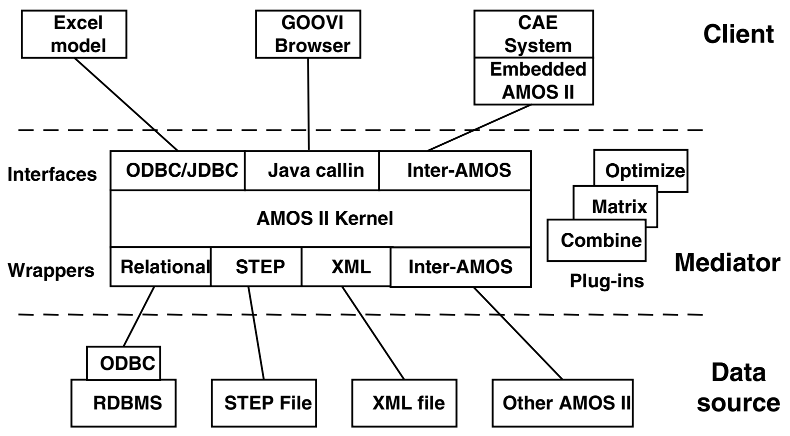

Figure 2. The architecture of AMOS II.

- (D) is a mediator server servicing inter-mediator requests from other mediators and defining mediating OO views integrating data from other mediator servers.
- (E) is a mediator server that translates data from a wrapped relational database. It has knowledge of how to translate AMOSQL queries to SQL [3] and interfaces to call SQL through ODBC [4]. It can use the facilities of AMOSQL for semantic reconciliation of data from its data source and its local database into views presented to other systems.
- (F) is a stand-alone database server accessed from mediator (D) by TCP/IP. It is also a *name server* which keeps track of the mediators in this group of mediators. Every AMOS II mediator belongs to a group of mediators and must be given a unique name within the group. The name server is an ordinary mediator server having the special task to store locally information about names, locations and other meta-properties of the mediators in a group. A name server thus identifies a group of mediators and all mediators in the group can there access meta-data for information about the federation of mediators (dotted lines in Figure 1).
- (G) is a stand-alone embedded AMOS II system which provides database facilities for an application, e.g. for Finite Element Analysis [20].

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 stand-alone databases or mediators can be done. By issuing a system command with the location of the name server the system requests there to become a mediator in the federation. Another system command makes the mediator a server that accepts incoming commands from other mediators in the federation.

Figure 2 illustrates the three level architecture of each AMOS II server. The top level contains applications and the lowest contain various data sources. Mediator servers constitute the middle mediator level in the figure. Each mediator server has a *kernel* containing the basic DBMS facilities. The lowest level contains the plug-ins implemented as foreign functions.

In order to access data from external data sources AMOS II mediators may contain one or several *wrappers* which interface and process data from external data sources, e.g. ODBC based access to relational databases [3,4] or access to XML files [2]. A wrapper is a program module in a mediator server having specialized facilities for query processing and translation of data from a particular kind of external data sources. It contains both interfaces to external data repositories and knowledge of how to efficiently translate and process queries involving accesses to the different kinds of external data sources. More specifically the wrappers perform the following functions.

- Schema importation: the explicit or implicit schema information from the sources is translated into a set of AMOS II types and functions.
- Query translation: object calculus is translated into equivalent API calls or query language expressions executable by the source.
- OID generation: when OIDs are required for the data in the sources, the query language expressions to be executed in the source are augmented with code or API calls to extract the information needed to generate these OIDs.

Once a wrapper has been defined for a particular kind of source, e.g. ODBC or XML, the system knows how to process any AMOSQL query or view definition for such sources. When integrating a new instance of that source the mediator administrator can define a set of views in AMOSQL that provide good abstractions of the source. View definitions must also be added to reconcile differences between abstracted data from different sources. We will explain this further in Section 4.

Meditator servers known to a mediator are also regarded as external data sources and there is a special wrapper for accessing other mediator servers. However, among the mediator servers special query optimization methods are used that take into account the distribution, capabilities, costs, etc. of the different servers [21].

Analogously, 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. Figure 2 illustrates three such *call-in* interfaces. The call-in interface for Java has been used for developing a Java-based multi-database object browser, GOOVI. Furthermore, there are also ODBC- and JDBC-based call-in interfaces that allow application programs to interact with AMOS II using those standards. A special problem in this case is that ODBC/JDBC assumes a non-OO relational model and the call-in interface modules for these standards therefore translate between SQL and AMOSQL.

It is even possible to closely embed AMOS II with applications, e.g. a Computed Aided Engineering (CAE) system [20]. The AMOS II kernel is then directly linked with the application. In this case the inter-Amos interface can be used for communication between the embedded database and the mediator servers.

Figure 2 furthermore illustrates that the AMOS II kernel can 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 integrating 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 also possible to add new query transformation rules (rewrite rules) for optimizing queries over the new domain.

To achieve good performance we have carefully optimized the representation of critical kernel data structures, e.g. the storage manager, object representation, type information and the representation of function definitions. We use tailored main memory data structure representations of system objects [12], rather than, for example, storing them in relational tables. For example, our object identifiers are represented as variable length records with pointers to data structures representing type-information, function definitions, dependent objects, etc. It is crucial that system information is represented efficiently, since it is extensively looked up during both compilation and interpretation of AMOSQL queries and functions. The storage manager has an incremental garbage collector for removing unused data.

3. BASIC DATA MODEL

The data model of AMOS II [5] is an OO extension of the DAPLEX [7] functional data model. Everything in an AMOS II database 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 literal objects are self-described system maintained objects which do not have explicit object identifiers (OIDs), e.g. numbers and strings. Literal objects can also be *collections* of other objects, e.g. *vectors* (1-dimensional arrays of objects) and *bags* (unordered sets with duplicates). The surrogates have associated explicit OIDs managed by the system. Surrogate objects are explicitly created and deleted by the user or the system. Examples of surrogates are objects representing real-world entities such as persons.

An object can be classified into one or more *types* (classes) making the object an *instance* of those 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 the extent of a supertype of that type (extent-subset semantics).

Surrogates also represent *meta-objects* (system objects) such as types and functions. The meta-objects are first class and can be queried as any other objects. For example, there are meta-types named *type* and *function* whose extents are the types and functions, respectively, defined in the mediator server. The transparent representation of meta-objects in the mediator servers allows powerful queries about the structure of each mediator server. In the name server even AMOS II mediators are represented as meta-mediator objects. Thus one can query the name server for the mediator server in the federation having, e.g. a specific name, then send meta-queries to that mediator server to explore its structure, and then finally query its data. Meta-queries are extensively used in the graphical object-oriented multi-database browser, GOOVI, which is written as an application that communicates with AMOS II solely through queries.

3.1. Types

There are four kinds types, called *stored*, *derived*, *proxy* and *integration union* types. The derived, proxy and integration union types are used for data integration purposes and are described in the next section.

Stored types are regular types (classes) having their extents explicitly stored locally in the mediator server. Their instances are maintained by the user. They are defined with a `create type` statement.

For example, the following statements submitted to a mediator server will define two types, *person* and *student*:

```
create type person;
create type student under person;
```

Inheritance from one or several stored supertypes is supported by the keyword `under`.

The general syntax for queries is

```
select <result>
   from <domain specifications>
   where <condition>
```

Each *domain specification* associates a query variable with a type where the variable is universally quantified over the extent of the type.

For example, the following query retrieves the names of the parents of all persons having sailing as a hobby:

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

Functions are allowed in projections and selections as in SQL:99. In this simple example it is assumed that every person can have just one hobby. It is also possible to represent that a person can have more than one hobby by declaring the value of the function *hobby* as an unordered collection (bag).

3.2. Functions

Object attributes, queries, methods and relationships are modelled by functions. Depending on their implementation the basic functions can be classified into *stored, derived, foreign* and *proxy* functions; also *database procedures*.

Stored functions represent properties of objects (attributes) stored in the mediator database. A stored function $S(X)$ returns the value of attribute S for an object X. Stored functions represent facts stored in the databases, as tables do in relational databases. For example,

```
create function age(person)->integer
       as stored;
create function name(person)->char
       as stored;
```

Stored functions are updatable, i.e. there are AMOSQL statements for updating the extent of stored functions explicitly. For example,

```
set age(:bob) = 45;
```

sets the age of object `:bob` to 45.

Derived functions are functions defined in terms of queries over other AMOSQL 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 OO models and views in relational databases. For example,

```
create function age(person p)->integer as
       select current_year() - born(p);
```

Functions may have any number of arguments; in the example *current_year* is a function with no argument, thus representing a constant object.

As for relational views derived functions are updatable only in special cases.

Foreign functions are implemented through an external programming language (Java, C or Lisp). Foreign functions correspond to methods in OO databases and provide access to external storage structures similar to data 'blades', 'cartridges' or 'extenders' in object-relational databases. As a simple example, assume we have an external disk-based hash table on strings to be accessed from AMOS II. We can then implement the following foreign function to access it:

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

Here the function *get_string* is externally defined in Java as a method *get_hash* of the public class *Foreign*. The code is dynamically loaded from some Java library when the function is defined. The Java Virtual Machine is interfaced with the AMOS II kernel through the Java Native Interface to C.

Foreign functions provide the basic primitives to access external systems from AMOS II. For example, data structures stored in external storage managers can be manipulated through foreign functions. Foreign functions can then also be defined to update the external data structures.

The access of an external source can be expensive, and, to help the query processor, a foreign function can have associated costing information defined as user functions. The foreign functions are furthermore *multidirectional*, allowing the definition of inverse functions. For example, our hash table could not only be accessed, but also scanned, allowing queries finding all the keys and values stored in the external table. We can generalize it by defining:

```
create function get_string(char x)->char y
   as multidirectional
      ("bf" foreign "JAVA:Foreign/get_hash"
            cost {10,1})
      ("ff" foreign "JAVA:Foreign/scan_hash"
            cost scan_cost);
```

Here, the JAVA function *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 (e.g. *'bf'*) indicate whether the argument or the result of the AMOSQL function must be bound (*b*) or free (*f*) when the external function is called. The `cost` specifications indicate both estimated *execution costs* in internal cost units and *result fanouts* (result sizes) of calls to the external functions. In the example the cost specifications are constant for *get_hash* and computed through the external function *scan_cost* for *scan_hash*. The cost specifications are used for comparing different execution strategies (e.g. scanning versus accessing) and need only be approximate.

Thus multi-directional foreign functions allow the specification of several access paths to external data structures. When wrapping external data sources with a mediator server the multi-directional foreign function facility provides the primitive to specify access paths and capabilities of the sources.

To handle external data as objects (rather than literals as in the example) it is possible to map external values to/from AMOS II proxy objects through system functions.

The basis for the multi-directional foreign functions was developed in [14], where the mechanisms are further described.

Proxy functions internally represent functions in other mediators when making multi-database queries (Section 4).

Database procedures are functions defined using a procedural sublanguage of AMOSQL. They correspond to methods with side effects in OO models. The syntax is similar to the stored procedures in SQL:99.

Functions may be overloaded, i.e. have the same name for different argument(s). Queries over overloaded functions pose special optimization problems as it may not be known until query execution time what resolvent of an overloaded function to invoke. Special optimization methods have been developed [10,16] to handle such *late bound* function calls. The implementations of many system features utilize the optimization of queries with late bound function calls.

4. DATA INTEGRATION BY OBJECT-ORIENTED MODELLING

To provide data integration features, the basic data model is extended with *proxy* types and functions [9] for supporting multi-database queries. Reconciliation is supported through *derived types* (DTs) [9] defined as subtypes of other types, and *integration union types* (IUTs) [10] defined as supertypes of other types.

Proxy types represent objects stored in other mediators or in data sources. The instances of proxy types are called *proxy objects*. The proxy types are internally created by the system when the user makes multi-database queries, e.g.,

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

This query retrieves the names of all persons in a mediator server named T_b. 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 *name* function in T_b.

Proxy types allow general multi-database queries over a federation of mediator servers. 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 proxy objects so they can be identified properly. Two proxy objects are considered equal if they represent objects with equal OIDs that are created in the same mediator server. Views can be defined that hide the origin of types and functions and that reconcile semantic differences. There are two special types for reconciliation, *derived types* and *IUTs*, to be explained next.

Proxy types can be used in function definitions as any other type. In the example one can define a function (view) of the persons located in a certain location by the function definition

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

In this case the function *personnel_in* will return instances of the proxy type for *Personnel@Tb*. It can be used in queries and function definitions, and in multi-database queries from other mediators in the federation. Multi-database queries and functions are compiled and optimized through a distributed query decomposition process fully described in [19] and summarized in the next section. Notice again that there is no central federated schema and the query compilation and execution of multi-database queries are made by exchanging (meta-)data with the accessed mediator servers. If some schema of a mediator server is modified, the multi-database functions accessing the mediator server become invalid and must be recompiled.

Derived types (DTs) are defined implicitly in terms of one or more supertypes through a declarative query over the supertypes. The supertypes are called the *constituent* types of the DT. The extents (object instances) of DTs are thus subsets of the intersection of the extents of the constituent types restricted by the query. For example,

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

This statement creates a derived type *CSD_Emp* whose extent contains those people who work in the CSD department. When a derived type is queried the system will create those of its instance OIDs necessary to execute the query.

An important purpose of derived types 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 T_a while T_b has a corresponding type *Faculty*. The following statement executed in a third mediator, M, defines a derived type *Emp* in M representing those employees who work both in T_a and T_b.

```
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 objects from both sources. The function *ssn* is assumed to uniquely identify faculty members in T_a, while the function *id* in T_b identifies personnel by employee numbers. A (foreign) function *id_to_ssn* in M translates employee numbers to SSNs.

The system will internally use proxy types to represent objects imported from T_a or T_b when making queries over *Emp*. The system internally maintains the information necessary to map between OIDs of a derived type and its constituent types. For details on this mechanism see [9].

Any kind of type is allowed as constituent types of a derived type. However, stored types cannot inherit from derived types as this could allow explicit creation of objects to a stored subtype that does not have a corresponding instance in some of its derived supertypes. This violates the extent-subset semantics where the extent of a supertype is a superset of the union of the extents of the subtypes.

The DT instances are derived from the instances of their supertypes according to a declarative condition specified in the DT definitions. DT instances are assigned OIDs by the system, which allows their use in locally stored functions (attributes) 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 derived types and its use for data integration is fully described in [9].

The DTs 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

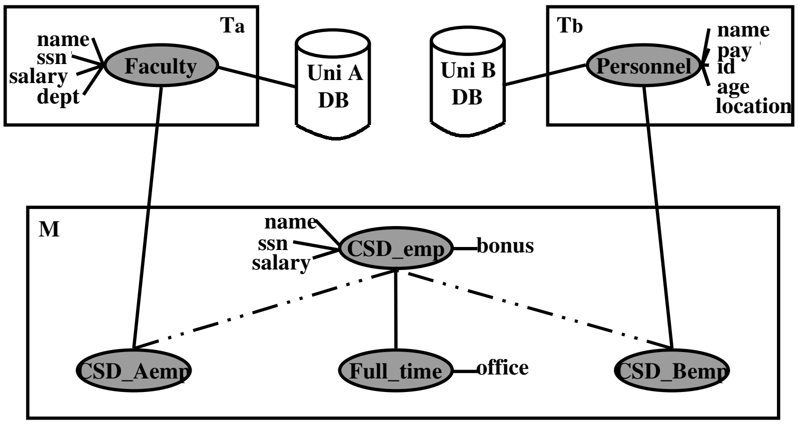

Figure 3. An object-oriented view for the computer science department.

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 *Integration Union Types* (IUTs) defined as supertypes of other types. IUTs are used to model unions of real-world entities represented by overlapping type extents. Informally, while the 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 3 illustrates the features and the applications of the IUTs.

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 3 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 T_A and T_B provide views in AMOS II's data model of the relational databases *Uni A DB* and *Uni B DB*. In mediator T_A there is a type *Faculty* and in mediator T_B a type *Personnel*.

The relational databases are accessed through an ODBC wrapper in T_a and T_b 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 [4]. The translation process is based on partitioning general queries into subqueries only using the capabilities of the data source, as fully explained in [19].

A third mediator *M* is setup 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 T_A and T_B.

```
create derived type CSD_A_EMP
  under Faculty@TA
    where dept(Faculty@TA) = 'CSD';

create derived type CSD_B_EMP
   under Personnel@TB
    where location(Personnel@TB) = 'G house';
```

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 types 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 3 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 modelled 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 [10] 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 *key* expression is given. The instances of different constituent types having the same key values will map into a single IUT instance. The key expressions can contain both local and remote functions.

The `FUNCTIONS` clause defines the reconciled functions of *CSD_emp*, derived from functions over the constituent types. For different subsets of the constituent types, 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 derived types. In Figure 3, 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 (50 000). 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)->char
       as stored;
```

5. QUERY PROCESSING

The functional data model is flexible and well suited for data integration, which actually was one of the motivations for the DAPLEX functional data model [7]. By describing type hierarchies and semantic heterogeneity using declarative functions and a functional Common Data Model there are many opportunities for the extensive query optimization needed in an OO mediation framework.

Figure 4 illustrates the query processing of a distributed query in AMOS II.

To illustrate the query compilation 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 [14]. The object calculus is a declarative representation of the original query.

The calculus generator translates the example query into the expression

$$\begin{array}{l} \{p, nm \mid \\ \quad p = Person_{\text{nil}\rightarrow\text{person}}() \wedge \\ \quad pa = parent_{\text{person}\rightarrow\text{person}}(p) \wedge \\ \quad nm = name_{\text{person}\rightarrow\text{character}}(pa) \wedge \\ \quad \text{'}sailing\text{'} = hobby_{\text{person}\rightarrow\text{character}}(p)\} \end{array}$$

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 unoptimized object calculus is then transformed by the *query optimizer* into an execution plan represented in an *object algebra*. The *execution plan interpreter* will finally interpret the execution plan to yield the result of the query.

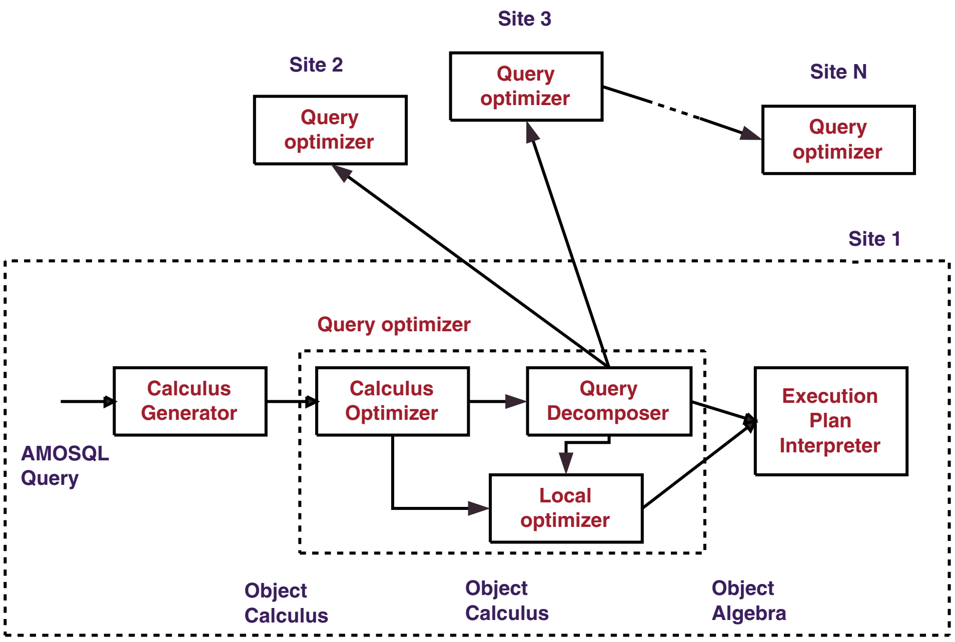

Figure 4. Multi-database query processing in AMOS II.

The query optimizer has several submodules, as indicated by the inner dashed box. In case the query is over more than one mediator the query optimizer will interact with other mediator servers to produce several distributed execution plans that are interacting during query interpretation, as indicated by the arrows.

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

$$\begin{aligned}&\{p, nm \mid\\ &\quad pa = parent_{\text{person}\to\text{person}}(p) \wedge\\ &\quad nm = name_{\text{person}\to\text{character}}(pa) \wedge\\ &\quad \text{'}sailing\text{'} = hobby_{\text{person}\to\text{character}}(p)\}\end{aligned}$$

This transformation is correct because p is used in a stored function (*parent* or *hobby*) with an argument or result of type *person*. The referential integrity system of stored functions constrains the stored instances to the correct type [14].

Queries over DTs are expanded by system-inserted predicates performing the DT system support tasks [9]. 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 analysed and, where needed, the appropriate functions 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. [9] 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* [10]. 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 modelled 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 [10].

In case the query is not distributed, the *Local Optimizer* will then use cost-based optimization to produce the executable object algebra plan from the transformed query calculus expression. The system has a built-in cost model for the 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 then chooses the cheapest for execution. The system has the options of using dynamic programming, hill climbing, or random search to find the final execution plan with a minimal cost. The user can instruct the system to choose one of these strategies.

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 servers in distributed execution plans.

The *Query Decomposer* [15,19] is invoked whenever a query is posed over data from more than one mediator server. As the local optimizer, it uses a combination of heuristic and dynamic programming strategies to produce a set of distributed object algebra plans. The query decomposer thereby interacts with the local optimizer as well as with the query optimizers of the other mediator servers involved in the query. Examples of interactions with other mediator servers are requests to estimate costs and selectivities of subqueries and requests to compile subqueries into local execution plans in remote mediator servers. The generated local execution plan interacts with the execution plans produced by the other mediator servers.

The details of the query decomposer is described in [19]. Here we will overview its main steps. Given a query over multiple data sources, the goal of the query decomposition is to explore the space of possible distributed execution plans and choose a ‘reasonably’ cheap one. The distributed nature of AMOS II requires a decomposition framework that allows cooperation of a number of distinct mediator servers for query processing.

The Query Decomposer has five phases.

(1) **Predicate grouping.** This phase attempts to reduce the problem of finding an optimal execution plan by reducing the query fragments to optimize at each site. Predicates executed at the same

data source are grouped into one or more composite predicates which are treated afterwards as a single predicate.

(2) **Site assignment (predicate placement).** This phase uses cost-based heuristics to decide which composite predicate is executed where, eventually replicates some of the predicates, and assigns execution sites to those predicates that can be executed at more than one site (e.g. θ-joins specified by comparison operators). The output of this phase is a distributed query graph where all the nodes are assigned to some site.

(3) **Cost-based execution scheduling.** In order to translate the query graph from the previous phase into an executable query plan, the query processor decides on the order of execution of the predicates in the query graph nodes, and on the direction of the data shipping between the nodes. Execution schedules 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 an arbitrary execution plan. As the space of all execution plans is exponential to the number of participating databases, we examine only a subset of the family of left-deep decomposition trees using a dynamic programming approach.

(4) **Tree distribution.** In this phase, a distributed compilation is performed at the participating mediators to rebalance and distribute the decomposition tree produced by the previous phase. One deficiency of the plans produced by the previous phase is that all the data flows through the mediator. This can lead to many superfluous data flows when the data is to be transferred from one data source to another. The execution schedule resulting from the tree distribution can contain sequences where the data is shipped directly from one mediator to another, eliminating the bottleneck of shipping all data through a single mediator. This approach makes the whole set of mediators function as one distributed mediation system. As a result of this phase a set of decomposition trees representing multi-source queries are generated at a subset of the mediators participating in the query compilation and execution. See [15] for details.

(5) **Object algebra generation.** The input to this phase is an executable decomposition tree, which is translated into equivalent sets of inter-calling local object algebra plans.

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 integration of data in a multi-database environment [22–31]. The integration facilities of AMOS II are based on work in the area of OO views [32–39].

Most of the mediator frameworks reported in the literature (e.g. [26,31,40]) propose centralized query compilation and execution coordination. In [41] 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 DataJoiner, also provide centralized mediation features.

In the DIOM project [29], the importance of the mediator composability is also recognized. 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 [37] OO view system provides multiple inheritance and a capacity-augmented view mechanism implemented with a technique called Object Slicing [35] 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 OO view functionality.

There are few research reports describing the use of OO view mechanisms for data integration. The Multibase system [23] is also based on a derivative of the DAPLEX data model and does reconciliation similar to the IUTs in this paper. An important difference between Multibase and AMOS II is that the data model used in Multibase does not contain the concept of OIDs. The query optimization methods in AMOS II are also more elaborate than in Multibase.

The UNISQL [27] system also provides views for database integration. The virtual classes (corresponding to the DTs) are organized in a separate class hierarchy. 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, but rather a set of queries can be used to specify a virtual class as an union of other classes. This imposes relationships among the classes not included in the class hierarchy, resulting in two types of dependencies among the virtual classes.

[42] 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 [43] 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 [44] which also used a hill-climbing heuristics as the query decomposer in AMOS II.

7. SUMMARY

We have given an overview of the AMOS II mediator system where federations of distributed mediator servers are used to integrate data from several sources with different query processing capabilities. Each mediator server in a federation has DBMS facilities for query compilation and exchange of data and meta-data with other mediator servers. OO views can be defined where data from several mediator servers are 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 paper gave an overview of AMOS II's architecture with references to other published papers on the system for details.

We described the OO and 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 servers in a federation are autonomous without any central schema. A special mediator server, the name server, keeps track of what mediator servers are members of a federation. The name servers can be queried for the location of mediator servers in a federation. Meta-queries to each mediator server can be posed to investigate the structure of its schema.

Some unique features of AMOS II are as follows.

- A distributed mediator architecture where query plans are distributed over several communicating mediator servers. This direction has been noted as a promising future research direction in, e.g. [22].
- Modelling reconciled OO views spanning over multiple mediator servers and specified through declarative functional queries.
- Query processing and optimization of queries to reconciled views using OO concepts such as overloading, late binding and type-aware query rewrites.

REFERENCES

1. Wiederhold G. Mediators in the architecture of future information systems. *IEEE Computer* 1992; **25**(3):38–49.
2. Lin H, Risch T, Katchaounov T. Object-oriented mediator queries to XML data. *Proceedings of the 1st International Conference on Web Information Systems Engineering (WISE2000)*, 2000; 38–45.
3. Fahl G, Risch T. Query processing over object views of relational data. *The VLDB Journal* 1997; **6**(4):261–281.
4. Brandani S. Multi-database access from Amos II using ODBC. *Linköping Electronic Press* 1998; **3**(19), http://www.ep.liu.se/ea/cis/1998/019/ [December 1998].
5. Risch T, Josifovski V, Katchaounov T. *Amos II Concepts*. http://www.dis.uu.se/~udbl/amos/doc/ [2000].
6. Gulutzan P, Pelzer T. *SQL-99 Complete, Really*. Miller Freeman: KA, 1999.
7. Shipman D. The functional data model and the data language DAPLEX. *ACM Transactions on Database Systems* 1981; **6**(1):140–173.
8. Lyngbaek P. OSQL: A language for object databases. *Technical Report, HP Labs, HPL-DTD-91-4*, 1991.
9. Josifovski V, Risch T. Functional query optimization over object-oriented views for data integration. *Intelligent Information Systems* 1999; **12**(2–3):165–190.
10. Josifovski V, Risch T. Integrating heterogeneous overlapping databases through object-oriented transformations. *Proceedings of the 25th Conference on Very Large Databases (VLDB'99)*, 1999; 435–446.
11. Karlsson JS. An implementation of transaction logging and recovery in a main memory resident database system. *MSc Thesis LiTH-IDA-Ex-94-04*, Linköping University, Sweden. http://www.dis.uu.se/~udbl/publ/recovery.pdf [1994].
12. Garcia-Molina H, Salem K. Main memory database systems: An overview. *IEEE Transactions on Knowledge and Data Engineering* 1992; **4**(6):509–516.
13. Fahl G, Risch T, Sköld M. AMOS—an architecture for active mediators. *Proceedings Workshop on Next Generation Information Technologies and Systems (NGITS'93)*, Haifa, Israel, 1993; 47–53.
14. Litwin W, Risch T. Main memory oriented optimization of OO queries using typed datalog with foreign predicates. *IEEE Transactions on Knowledge and Data Engineering* 1992; **4**(6):517–528.
15. Josifovski V, Katchaounov T, Risch T. Optimizing queries in distributed and composable mediators. *Proceedings of the 4th Conference on Cooperative Information Systems*, CoopIS'99, 1999; 291–302.
16. Flodin S, Risch T. Processing object-oriented queries with invertible late bound functions. *Proceedings of the 21st Conference on Very Large Databases (VLDB'95)* 1995; 335–344.
17. Sköld C, Risch T. Using partial differencing for efficient monitoring of deferred complex rule conditions. *Proceedings of the 12th International Conference on Data Engineering (ICDE'96)*. IEEE, 1996; 392–401.
18. Ullman JD. *Principles of Database and Knowledge-Base Systems*, vol. I, II. Computer Science Press, 1988, 1989.
19. Josifovski V, Risch T. Query decomposition for a distributed object-oriented mediator system. *Distributed and Parallel Databases* 2001; to be published.

20. Orsborn K. Applying next generation object-oriented DBMS to finite element analysis. *Proceedings of the International Conference on Applications of Databases (ADB'94)*, Litwin W, Risch T (eds.). Springer, 1994; 215–233.
21. Josifovski V. Design, implementation and evaluation of a distributed mediator system for data integration. *PhD Thesis*, Linköping U., Sweden. http://www.dis.uu.se/~udbl/publ/vanjaphd.pdf [1999].
22. Bukhres O, Elmagarmid A (eds.). *Object-oriented Multidatabase Systems*. Pretince Hall, 1996.
23. Dayal U, Hwang H-Y. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering* 1984; **10**(6):628–645.
24. Evrendilek C, Dogac A, Nural S, Ozcan F. Multidatabase query optimization. *Distributed and Parallel Databases* 1997; **5**(1):77–114.
25. Fang D, Ghandeharizadeh S, McLeod D, Si A. The design, implementation, and evaluation of an object-based sharing mechanism for federated database system. *Proceedings of the 9th International Conference on Data Engineering Conference (ICDE'93)*. IEEE, 1993; 467–475.
26. Haas L, Kossmann D, Wimmers EL, Yang J. Optimizing queries across diverse data sources. *Proceedings of the 23th International Conference on Very Large Databases (VLDB'97)*, 1997; 276–285.
27. Kelley W, Gala S, Kim W, Reyes T, Graham B. Schema architecture of the UNISQL/M multidatabase system. *Modern Database Systems—The Object Model, Interoperability, and Beyond*, Kim W (ed.). ACM Press, 1995; 621–648.
28. Lim E-P, Hwang S-Y, Srivastava J, Clements D, Ganesh M. Myriad: Design and implementation of a federated database system. *Software—Practice and Experience* 1995; **25**(5):533–562.
29. Liu L, Pu C. An adaptive object-oriented approach to integration and access of heterogeneous information sources. *Distributed and Parallel Databases* 1997; **5**(2):167–205.
30. Subramananian S, Venkataraman S. Cost-based optimization of decision support queries using transient views. *Proceedings of the ACM International Conference on Management of Data (SIGMOD'98)*, 1998; 319–330.
31. Tomasic A, Raschid L, Valduriez P. Scaling access to heterogeneous data sources with DISCO. *IEEE Transactions on Knowledge and Date Engineering* 1998; **10**(5):808–823.
32. Abiteboul S, Bonner A. Objects and views. *Proceedings of the ACM International Conference on Management of Data (SIGMOD'91)*, 1991; 238–247.
33. Bertino E. A view mechanism for object-oriented databases. *Proceedings of the 3rd International Conference on Extending Database Technology (EDBT'92)*, 1992; 136–151.
34. Heiler S, Zdonik S. Object views: Extending the vision. *Proceedings of the 6th International Conference on Data Engineering (ICDE'90)*. IEEE, 1990; 86–93.
35. Kuno H, Ra Y, Rundensteiner E. The object-slicing technique: A flexible object representation and its evaluation. *Technical Report CSE-TR-241-95*, University of Michigan, 1995.
36. Motro A. Superviews: Virtual integration of multiple databases. *IEEE Transaction on Software Engineering*, 1987; **13**(7):785–798.
37. Rundensteiner E, Kuno H, Ra Y, Crestana-Taube V, Jones M, Marron P. The MultiView project: Object-oriented view technology and applications. *Proceedings of the ACM International Conference on Management of Data (SIGMOD'96)*, 1996; 555.
38. Scholl M, Laasch C, Tresch M. Updatable views in object-oriented databases. *Proceedings of the 2nd Deductive and Object-Oriented Databases Conference (DOOD91)*, 1991; 189–207.
39. Souza dos Santos C, Abiteboul S, Delobel C. Virtual schemas and bases. *Proceedings of the International Conference on Extending Database Technology (EDBT'92)*, 1994; 81–94.
40. Garcia-Molina H, Papakonstantinou Y, Quass D, Rajaraman A, Sagiv Y, Ullman J, Vassalos V, Widom J. The TSIMMIS approach to mediation: Data models and languages. *Intelligent Information Systems* 1997; **8**(2):117–132.
41. Du W, Shan M. Query processing in Pegasus. *Object-Oriented Multidatabase Systems*, Bukhres O, Elmagarmid A (eds.). Pretince Hall, 1996; 449–471.
42. Özsu MT, Valduriez P. *Principles of Distributed Database Systems*. Prentice Hall, 1999.
43. Daniels D, Selinger P, Haas L, Lindsay B, Mohan C, Walker A, Wilms PF. An introduction to distributed query compilation in R*. *Proceedings of the 2nd International Symposium on Distributed Data Bases*, 1982; 291–309.
44. Bernstein P, Goodman N, Wong E, Reeve C, Rothnie J Jr. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems (TODS)* 1981; **6**(4):602–625.