# Comparison of Amos II with Other Data Integration Projects (working paper)

Vanja Josifovski and Tore Risch
Laboratory for Engineering Databases
Linköping University, Sweden

vanja@ida.liu.se, torri@ida.liu.se

April 16, 1999

**Abstract**

This working document gives an overview of some research projects whose aims are similar to the Amos II project. Amos II is related to research in the areas of OO views, data integration, distributed databases and general query processing. We have surveyed the literature on a number of multi-database integration and OO view projects and compared their approaches to Amos II .

## 1 Multi-database integration

The main purpose of Amos II project is development of a system for integration of data in multiple data sources. This section compares the architecture and implementation techniques used in Amos II with other multi-database integration projects [6, 22, 4, 20, 7, 45, 34, 25, 4, 28, 16]. In summary, the unique features of Amos II are:

To aid the comparison, we first summarize some major features of the Amos II data integration capabilities:

- A distributed mediator architecture where query plans are generated using a distributed compilation in several communicating mediator and wrapper servers.
- Data integration by reconciled OO views spanning over multiple mediators and specified through declarative OO queries. These views are *capacity augmenting* views, i.e. locally stored attributes can be associated with them.
- Processing and optimization of queries to the reconciled views using OO concepts such as overloading, late binding, and type aware query rewrites.

- Query optimization strategies for efficient processing of queries over a combination of locally stored and reconciled data from external data sources.

## 1.1 Disco

DISCO (Distributed Information SEarch Component) system is based on a centralized mediator-wrapper architecture. Although its primary focus is not on query performance, but on extensibility and partial query evaluation in presence of unavailable data sources, it has many principles in common with the Amos II system.

A special mediator called the *Catalog* keeps information about the available DISCO mediators and wrappers on the network. This service corresponds closely to the name services in Amos II . DISCO is based on the ODMG [6] standard data model and uses OQL and ODL as the query and data definition languages, respectively. One of the central concepts in the ODL is the concept of *type* which has associated an *interface* (structural type description) and an *extent*. In DISCO the concept of a type is extended with these facilities:

- Associating an interface with one or more extents stored in the data sources. The extents contain objects that have structure as described by the interface.
- Type mappings between types defined in the mediator and the types with extents stored in the data sources, in order to overcome structural differences.

The first extension allows for a type defined in the mediator to draw its extent from a set of data sources. The resulting extent is an union of all the instances in all the sources. The second extension is used to transform the data in the data sources into a common interface.

Data sources are defined by instantiation of the type *Repository*. Repositories are classified into *repository types*. To access a repository of a particular type, a wrapper must be implemented for it. For example, the following expressions define two repositories $r1$ and $r2$ of a same type to be wrapped with the wrapper $w0$. The data in both repositories has a format described by the interface $Person$. The query retrieves the names of all the persons with salary greater than 10.

```
extent person0 of Person wrapper w0 repository r0;
extent person1 of Person wrapper w0 repository r1;

select p.name
from p in union(person0, person1)
where p.salary > 10;
```

In this example, the extents are named explicitly. Alternatively, they can be specified by meta-data queries that dynamically determine the number of extents to be scanned.

For conflict resolution the user can use the OQL view definition capabilities. Compared to the approach used in Amos II this has the following disadvantages:

1. The reconciliation is performed in the mediator, while Amos II can push the conflict resolution code to the translators and the data sources when favorable.

2. The view mechanism (named *sets* in OQL) does not provide OIDs for the view instances and therefore the optimizations based on locally stored data in Amos II are not applicable.

3. The ODL/OQL language does not have conflicts resolution constructs as the integrated union types (IUTs). This requires from the user to manually specify the resolution in case of a data overlap.

The query processing in DISCO is performed over plans described in a formalism called universal abstract machine (UAM) which contains the relational algebra operators extended with primitives for executing parts of the plans in the wrapper. The mediator communicates with the wrapper by using a grammar describing the operator sequences accepted by the wrapper. It can also (in some cases) ask for the cost of a particular operator sequence. This method is more elaborate than the method for description of data source capabilities in Amos II , but it is more complex and time consuming, due to the combinatorial nature of the problem of constructing the subplans executed in the wrappers.

Finally, to our knowledge, an implementation of a prototype has been planned, but there are no experimental results have been reported.

## 1.2 Garlic

The Garlic [22, 23, 37] system, developed at the IBM Almaden Research Center, also has a centralized wrapper-mediator architecture. The system is based on ODMG's OO data model. The data from the wrapped data sources is represented as objects. The OIDs of these objects are constructed from the data source name, the object type, and a set of keys specified for each type retrieved from a data source. Except for the system data and intermediate results, Garlic does not provide facilities for storing local user-defined data, even though it has a fully functional query processor. The primary goals of the Garlic project are:

- To explore how the query optimization techniques based on exhaustive dynamic programming, developed in earlier IBM research prototypes and products, can be used in a data integration scenario.

- To expand these techniques, so that wrappers for different data source types can be easily specified, modified, and added to the system.

At the hart of the Garlic system is the *query service* facility. This facility is divided into two units: (i) a query language processor, and (ii) a distributed query execution engine. The query language processor performs tasks that correspond to some of the the calculus related phases in the query processor of Amos II (semantic checking, rewrite, etc.). The second unit performs cost based optimization and outputs an executable query execution plan.

The query optimizer in Garlic is based on dynamic programming. The optimizer builds plans of gradually increasing sizes, by adding POPs (Plan OPerators) to already built partial plans. The POPs can be relational algebra operators, operators for storing and retrieving data in temporary tables, and operators for accessing the wrappers. During the search, the optimizer prunes the plans that are more expensive than other plans representing the same

subquery. In addition to the composition of the plans, Garlic takes in account the location of the result of the plan execution. Two plans computing the same subquery, but placing the result in two different sites are not pruned from the system.

POPs are added to already constructed partial plans using STARS - (STrategy Alternative Rules). Each rule describes how a new plan is constructed from one or more partial plans. Each rule has attached a condition that guards its triggering. The rules create POPs that are executed locally, or the *PushDown* POP which executes a subquery in a wrapper. For the *select-project-join* queries there are three main STAR types (named *STAR roots*) for: *access* (scan), *join*, and *finish* (plan completions as e.g. projections). The two first STARs can produce plans that execute either in the mediator or in the data sources, while the third one is always executed in Garlic. An illustrative example of a STAR root is the *join* root which is translated into three different POPs: *ReproJoin* - executed in a wrapper, *NestedLoopJoin* - executed in Garlic over materialized operands, and *BindJoin* which is a semi-join-like POP where the outer table is sent one tuple at a time to the site of the inner table, retrieving the matching tuples.

The functionality of Garlic roughly corresponds to the multidatabase query engine in Amos II . An important difference is that Garlic does not store local data. The generated OIDs are used only to access the data in the data sources. This makes the techniques for optimization of queries over a combination of locally stored and imported data in Amos II not applicable. It seems that the Garlic OIDs are used only internally in the query processor, and possibly as object handles for user requests over individual objects. Furthermore, Garlic has no facilities equivalent to the IUTs in Amos II .

Another difference between the two approaches is that Garlic is based on a centralized query compilation and execution architecture, while Amos II is based on a query processor which performs distributed query compilation in the network of translators and other mediators. Therefore, we cannot directly compare the POP formalism of Garlic with the decomposition tree (DcT) formalism of Amos II which is designed to distribute queries over multiple Amos II servers. By contrast, a Garlic system treats another Garlic system as a (relational) data source. Therefore strategies as achieved by the DcT distribution are not explored. Although, it can be guessed that STAR rules can be formed to take advantage of the intermediate result materialization capabilities of the Garlic mediators in order to employ similar strategies as in Amos II in a network of Garlic mediators, this approach has not been pursued in the reported work.

One limitation of the current implementation of Amos II is that it always pushes the joins to a data source where all its operands are available. Garlic, on the other hand, also considers plans which perform the join in the mediator. Our on-going work includes expansion of Amos II to consider such query execution plans.

## 1.3 Pegasus

The goal of the Pegasus project is to develop a heterogeneous information and process flow management system (HP-MS). This project was started in the early 1990s at the HP Labs in Palo Alto.

Pegasus is a fully fledged database management system. The focus of the project is

on integration of relational databases, multimedia databases, and legacy applications. The three main goals of the Pegasus project are:

- seamless integration of external schemas with the local database
- efficient query processing
- workflow management

Pegasus originates in the same data model as Amos II : the Iris OO data model [15], an OO extension of DAPLEX [42]. Earlier versions of Pegasus used the language HOSQL which is an extension of the language OSQL [30] used in the Iris system. OSQL has also served as a basis for Amos II 's query language AMOSQL. More recently Pegasus has been shifted to a SQL3 based language SQL3+. This language extends the SQL3 standard with data integration facilities.

Although the terminology differs greatly, the architecture of the Pegasus system is similar to the mediator-wrapper architecture used in Amos II , but it is not distributed. The core of Pegasus corresponds to the mediator services. External data sources are named External Data Resource Management Systems (EDRMSs). The interaction with the EDRMSs is performed using a module named *Pegasus Agent* (PA) which also has some processing capabilities. The PA process is intended to run on the same machine as the EDRMS it serves. The functionality of the a PA is similar to a functionality of the translators in the Amos II architecture. Nevertheless, a PA is not a fully fledged Pegasus server in the centralized architecture of Pegasus. This is one of the most important differences with the Amos II architecture.

The data integration facilities of Pegasus are named using the distributed database terminology. *Foreign tables* are imported from declared data sources. In the following example, summarized from [4], first a DB2 relational data source named $DB1$ is defined and then a table is imported and bound to the type *Programmer* in Pegasus.

```
REGISTER RELATIONAL DB2 DATASOURCE db1 AT 'smith@host1' AS Pdb;

CREATE TYPE Programmer WITH OID VISIBLE
  ( Prog_id INTEGER,
    Ssn     INTEGER,
    Name    CHAR,
    Salary  INTEGER);

CREATE TABLE ProgrammerTable (Dcn: Programmer) AS IMPORTED
   FROM RELATIONAL DATASOURCE Pdb RELATION Programmer
   WITH OID PRODUCING BY (Prog_id)
   (Prog_id AS MATCHING  Prog_id,
    Ssn     AS MATCHING  Ssn,
    Salary  AS MATCHING  Salary,
    Name    AS MATCHING  Name);
```

The resulting table has one-to-one correspondence with the table in the relational database. The OIDs of the new type are formed using the *Prog_id* column from the relational database.

Imported tables can be integrated with locally defined tables, as well as tables imported from other data sources, by two mechanisms:

- integrated views
- adding columns of one table to another

The first mechanism allows for merging horizontally fragmented tables, while the second is used for merging vertically fragmented tables. As opposed to the classical distributed database work, here the fragments are maintained by autonomous database systems. The following example from [4] defines first a supertype over the types *Programmer* and *Engineer*, and then an integrated view over the tables corresponding to this types:

```
CREATE TYPE Employee
   OVER Programmer WITH Prog_id AS Emp_id,
        Engineer   WITH Eng_id  AS Emp_id,
   (Emp_id INTEGER,
    Ssn     INTEGER,
    Name    CHAR,
    Salary  INTEGER);

CREATE VIEW EmployeeTable (Dcn Employee)
   AS SELECT * FROM ProgrammerTable UNION ALL
      SELECT * FROM EngineerTable;

DEFINE ROW EQIVALENCE FOR EmployeeTable ON
   (Tp ProgrammerTable, Te EngineerTable)
     BY Tp.Dcn->Ssn = Te.Dcn->Ssn;

DEFINE RECONCILER ON EmployeeTable.Dcn->Ssn
   (ProgrammerTable, EngineerTable)
      RETURNS INTEGER USING DISAMB_SUM
```

The ROW EQUIVALENCE clause defines the equality condition for the rows of the defined view. Rows that satisfy this condition will be treated as one in the resulting tables. Possible conflicts in the values of the other columns are resolved by RECONCILER definitions. These can be system specified as, for example DISAMB_SUM which returns the sum of the input values, or user defined derived functions. Although the view definition uses the UNION operator, the ROW EQUIVALENCE clause enforces outer-join semantics and processing.

The processing of the queries over the integrated view proceeds in three phases. The first phase performs query rewrites to transform the query from using the integrated view to the imported tables. Then, the query processor identifies portions of the query tree which can be evaluated in a single EDRMS and converts them into *Virtual Tables* (VT) which encapsulate the operations performed in the EDRMS. The resulting query tree has VTs or locally

stored tables as leafs and internal nodes which represent operators of extended relational algebra. The extensions deal, among other, with object-oriented concepts, constraints and reconciliation. Some of the query rewrite rules applied in this phase are: [4]:

- Push as much as possible of the operations into the EDRMS.
- Push up the reconcile operations in order to place the join operations close to the outerjoins.
- Combine joins with the outerjoins in order to make the inputs to the outerjoin smaller.
- Transform the outerjoins to left- or right-outerjoins, or to ordinary joins when some of the other query predicate use some attributes which are not present is the both of the joined tables. Since the language is null-intolerant (a predicate evaluates to false when a part of it is null), this eliminates the parts of the outerjoin where this predicate is not present.

The second query processing phase builds a left-deep query tree using a cost based method. The costs of the VTs are obtained using elaborate cost model for the operations performed in the EDRMS and calibration of the data sources [11].

The left-deep query tree generated in the first two phases is rebalanced in the third phase. The rebalancing operations are performed at certain points of the tree and are based on the associativity and commutativity properties of the join and cross-product operators.

Each of the three query processing phases in Pegasus can be related to a phase in the query processing in Amos II . The first phase corresponds to the calculus generation and rewrite, with a difference that the rewrites in Amos II reduce the number of predicates, while in Pegasus they perform reordering of the operators that influences their order of execution in the final execution plan. Techniques, as in Amos II , that take advantage of the types of the query variables to reduce the query size are not described.

In processing of queries over the integrated views, Pegasus keeps the outerjoins as a single operation, and later in the query it performs a correction of the result by reconciliation operators. This approach has an advantage in keeping the queries compact, but it does not take advantage of the selections stated over reconciled functions. We believe that this kind of selections appear often in queries over the integration views.

In Amos II , on the other hand, the outerjoin and the reconciliation is broken into up to three cases: one join and two anti-semi-joins, each processed separately. This allows selections specified over the reconciled functions to be pushed all the way down to the data sources in the two anti-semi-joins cases. In the join case, the optimizer might be able to push the selections down to the data sources when the reconciliation is defined using function values from only one of the data sources. Even when this is not the case, the join still generates smaller intermediate results than the full outerjoin, in particular when the overlap is small. The size of the result and the data shipped to perform the join has a maximum size proportional to the size of the smaller of the integrated extents. The outer join produces an intermediate result that is of size equal to the sum of the sizes of the integrated extents.

Another disadvantage of performing the reconciliation late in the query execution is that the reconciliation operator requires its whole input, which is in this case an outer join of the

integrated tables, to be materialized before the processing starts. This prevents streamed execution and might pose problems in cases when the intermediate results are too big to fit into the integration system memory.

Although the problem of parameterized queries to the data sources is noted in the context of tree rebalancing, Pegasus does not reduce the number of such queries by materializing temporary indices in the system's memory, as done in Amos II .

Due to its centralized architecture the rebalanced trees in Pegasus are constructed and stored in a single system. Distributed architecture is one of the future topics of the Pegasus project [4]

## 1.4 TSIMMIS

The TSIMMIS system - The Stanford-IBM Manager of Multiple Information Sources [20] is a continuation of the Light Weight Object Repository (LORE) project, and is aimed for integration of a large number of structured and unstructured data sources. The basis for the integration is a common data model named *object exchange model* (OEM). The idea behind the OEM is to provide as simple as possible, but complete facilities for data integration. Although OEM is not a fully fledged OO model, the basic entity in OEM is called "object". Each object is composed of four elements: $label$, $type$, $value$, and $object-id$. As opposed to other OO models, the OEM is *self-descriptive*. The type and the label of an object contains the information usually stored in a database schema. Actually, the notion of schema is absent in the OEM. The authors claim that the labels can be used not only for naming the objects, but also for inferring semantics that can be used in the data integration process. The value field of an object can contain a collection of literals or nested objects, thus creating a graph-like database structure.

To query a database described in OEM, a client can issue a query in a query language named OEM-QL. This query language adopts the OQL (and SQL) syntax style and is based on the **select-from-where** clause. The semantics, however is based on the OEM model. The path expressions in OEM-QL allow queries over the labeled graph that contain wildcards and other regular expressions that make the navigation easier. As a result, an OEM-QL query returns an OEM graph.

The TSIMMIS project uses a centralized mediator/wrapper data integration architecture. Mediators can fetch and combine data from wrapped data sources. However, unlike our translators, the wrappers do not have a complete query processor and data store. The emphasize in TSIMMIS has been to enable easy wrapper and mediator generation, using a *mediator specification language* (MSL), rather than on query performance as in our work. MSL is a rule based language where the input query is matched against a rule specification. In the wrapper definition, when a match is found, data source specific code specified within the rule is executed in order to retrieve the relevant data from the data sources. The data source capabilities mechanism in Amos II are more elaborate and perform cost based and heuristic optimizations that are not applied in TSIMMIS. Also, the OO transformations used in Amos II are, due to the differences in the CDM, are not applicable in TSIMMIS. The mediator generation system in TSIMMIS allows for joins, but does not consider integration operators for resolving conflicts in overlapping data as in Amos II . Furthermore, to the

extent of our knowledge, the TSIMMIS project has not reported performance evaluation of the execution of queries over views defined over data combined from the mediator and different data sources.

## 1.5 Multibase

The Multibase project [7, 8, 9, 10] is a pioneer work on integration of data in multiple databases. As in Amos II , the Multibase system is based on a derivative of the DAPLEX data model [42] extended with generalization. Data integration is performed by defining *generalized types* as supertypes of existing database types. For the generalized types, derived functions based on the functions of the subtypes can be defined to reconcile the data in the integrated databases. These features are closely related to the *functions* clause in the IUT definitions in Amos II .

Query transformations are used to transform a query over the generalized types into a set of queries over the local schemas. These query transformations break down the outerjoins and the reconciliation functions into queries the over disjoint parts of the integrated relations, using joins and anti-semi-joins. The approach allows for similar optimization techniques as the ones used in Amos II for optimizing queries over IUTs having no locally stored functions. Nevertheless, the method used in Multibase is not based on system predefined types and the properties of type hierarchies, which makes the query analysis and optimization more complicated. Furthermore, the project does not explore combining optimization by generalization with constructs as the DTs in Amos II .

Another important difference between the two systems is that the Amos II data model is OO, while the Multibase system lacks OIDs. The lack of OIDs disallows both materialization of the instances of the integrated types and seamless mixing of local data with data retrieved from various data sources. Locally stored data in not considered in this project.

In [8] it is also identified that in presence of selections over the reconciled functions, the two anti-semi-joins will be usually able to take advantage of these. The authors describe three optimization techniques to push the selections through the most common aggregations used in reconciliation of function values of overlapping data. Nevertheless, they note that these techniques apply to very limited number of cases. Therefore, we have chosen not to pursue this approach in Amos II .

Finally, to the extent of the reported work available to us, the benefits of the proposed optimization techniques have not been quantified by experimental results.

## 1.6 Data Joiner

The IBM's DataJoiner [46, 45] is a state-of-the-art commercial product targeted for integration of relational databases of different vendors. As opposed to the previous generation integration tools that provide only a gateway for retrieving data stored in multiple vendor databases, the DataJoiner has a full scale distributed query processor capable of pushing down whole subqueries in to the connected databases. DataJoiner also is a fully fledged DB2 database.

DataJoiner's query optimizer has a detailed knowledge of the strategies used in the

database engines supported as data sources. This meta-data, stored in a *Server Attribute Tables* (SATs) includes information as the vendors' join implementations, index usage, type of query trees used in the optimization, specifics of the SQL dialect etc.

As a complement to the SAT table information, the system uses sampling techniques or catalog queries to build locally stored statistics about the characteristics of the tables imported from the data sources. Using this information, the DataJoiner optimizer is capable of precise estimates of the costs of the subqueries pushed to the source databases.

The DataJoiner query optimizer is an extension of the DB2/CS Starburst optimizer. It enumerates all the possible plans using a dynamic programming approach, as in Amos II . The suboptimal plans are pruned. The generated plans explore both performing the operations in the integrator, or if possible, in the data sources.

The portions of the plans pushed to the data sources are translated to SQL that closely resembles the execution strategy used by the local query processor. By this, the DataJoiner takes over the optimization decisions from the relational database used as data sources. The authors of the system claim that in many cases they generate queries that perform better than if the original query was executed directly in the system. An industry report [38] comparing the DataJoiner with two other products in the same area, sets the performance and the functionality of this product high above the other two products.

## 1.7 MIND

The MIND (Middle-East Turkish University Interoperable DBMS) prototype [34, 36, 13] is based on the OMG's distributed object management architecture. The system is implemented around DEC's ObjectBroker ORB. Various relational databases from different vendors are connected to the system using an interface defined in IDL. Two interfaces play a major role in the MIND integration architecture: the Global Database Agent (GDA) and the Local Database Agent (LDA). For each session with a client, the GDA is instantiated in a server CORBA object which handles the requests for the client. The CORBA architecture provides location transparency for the GDA objects (GDAO). A GDAO contains a Global Transaction Manager Object and a Global Query Processor Object (GQPO). The latter performs the query decomposition and sends an executable plan for execution to the former. The LDA objects (LDAOs) manage the submissions of the operations to the relational data sources and transaction management. The tasks of the GDAOs and LDAOs can be related to the tasks of the mediator and wrapper in the mediator-wrapper architecture.

The schema integration process is based on a typical four schema transformation layers: local, export, global and external schema in order from the individual data sources to the applications. The focus of these transformations is on resolution of the class structural conflicts and class extent conflicts, while preserving the autonomy of the sources.

The conflict resolution is specified by a *mapping* definition. The following example [34] illustrates an integration of departments tables from three databases (*dept@DB1, division@DB2 and department@DB3*):

```
/* global schema: department(dept_no, dept_name, address) */
/* local schemas: DB1: dept(dno, dname)
```

```
  DB2: division(dno, dname, location)
  DB3: department(dno, deptname, address) */


mapping department {
  origin
     DB1: dept d1,
     DB2: division d2,
     DB3: department d3;
  def_ext dept_ext  as
     select * from d1, d2, d3 where d1.dno *= d2.dno
                                and d2.dno*=d3.dno;
  def_att dept_no   as
     select d1.dno, d2.dno, d3.dno from d1, d2, d3;
  def_att dept_name as
     select d1.dname, d2.dname, d3.deptname from d1, d2, d3;
  def_att address   as
     select d2.location, d3.address from d2, d3; }
```

The *mapping* clause specifies the data sources, the extent of the new class, and the correspondence of the local tables' attributes to the attributes of the global table. Note that the $^*=$ operator denotes an outerjoin.

The goal of the query decomposition is to produce:

- A set of single data source queries that retrieve the needed data from each of the involved data sources
- A set of post-processing operations executed in the GDAO that produce the query result from the intermediate results sent by the data sources.

The set of single data source subqueries is produced by instantiating the global schema query for each of the data sources. One limitation of this process as described in [34] is that in presence of a join over two integrated tables, the generated single site queries assume that the joins are performed only over fragments (integrated tables) located at a same data source. For example, a query where the *dept* type defined above is joined with a *emp* table integrating data from the same three sources, will produce subqueries which explore only entries where the local employee tables join with the local department tables. Cross-source strategies (e.g. where an employee at DB1 works at a department stored at DB2) are not considered. Although this conforms with the probable intended semantics of the example above, in general, this kind of simplifications are application dependent and, in our opinion, should be inferred based on declared database constraints.

In the post-processing phase, the GDAO performs operations as outer-joins and joins to build the final result from the intermediate results returned by the data sources. The execution plan for this phase is generated by using a dynamic and heuristics-based query optimization approach that takes in account the actual load of the systems during the query execution time.

As the other systems that perform the reconciliation in the final phases of the query processing, the method used in MIND suffers from not being able to use the selections based on reconciled functions early in the query processing. Also, although the requests to the data sources can be executed in parallel, the reconciliation process in the mediator has to wait until all the inputs are materialized, before emitting the first result tuple.

Another difference between MIND and Amos II is that MIND's integration facilities do not provide means of materializing OIDs for the data from the data sources and augmenting the views over this data with locally defined attributes.

## 1.8 IRO-DB

The IRO-DB project (Interoperable Relational and Object-Oriented Databases - ESPRIT - III P8629) [14, 43, 17] developed tools for unified access to a number of relational and OO databases. The system is based on the ODMG standard data model and the query language OQL. The architecture of IRO-DB is divided into three layers:

- The **Local layer** represents the data sources wrapped by Local Database Adapters (LDA) that proovide ODMG/ODL mapping to the schema and OQL access to the data in the sources. This layer also generates OIDs for the instances in the OO CDM that correspond to the instances in the data sources.

- The **Communication Layer** performs the transfer of objects and OQL queries between the server and the client sites. The protocol used is an OO extension of the remote database access (RDA) standard, named OORDA. The main purpose of the communication layer is to allow the interoperable layer to communicate with the local layer, but it can also be used by the applications to directly access the data sources via an OO extension of SQL CLI.

- The **Interoperable Layer** provides the application with means of integrated access to multiple remote databases. Its functionality can be divided into two parts: an interoperable DBMS (IRO-DBMS) that supports use and maintenance of an interoperable (global) schema, and tools for aiding the building of an interoperable schema (Integrators Workbench).

Compared with the wrapper-mediator architecture, the interoperable layer provides services that correspond to the mediator services, while the local layer corresponds to the wrapper. In the following, a description of the IRO-DBMS is presented. The IRO-DBMS is also consisted of several functional units:

- The *API generator* generates an ODMG compliant C++ API from the integrated schema to be used by applications that access this schema.

- The *global transaction manager* implements the nested transaction protocol of the ODMG standard.

- The *global parser and processor* takes a text representation of an OQL query and returns the result of its execution over the interoperable schema. The features of this

unit in relation to the Amos II system will be explored in greater detail in the rest of this section.

- The *global data repository* stores and provides the rest of the system with an export schemas description, a description of the interoperable schema, schema localization information, and a description of the mappings between the export and the interoperable schemas.

The data integration schema in IRO-DB is specified by three layers of class mappings. Each class to be exported by a data source is named an *external class*. In the interoperable system each external class of interest has a corresponding *imported class* which serves a similar purpose as the proxy types in Amos II . The actual integration is performed by defining *derived classes*. The interoperable system can also host locally stored data organized into *standard classes*. The following example illustrates the use of the *mapping* construct used for defining derived classes. In the example, first two imported classes, *S1_PART* representing the table *part* at the source *S1*, and *S2_PART* representing the table *prt* at the source *S2*, are defined. The mapping clause defines the extent of the derived class *PART* and its attributes using query expressions [43]:

```
mapping imported S1_PART{
    origin S1::PART   orig;}

mapping imported S2_PART{
    origin S2::PRT    orig;}

mapping PART {
  origin S1_PART sorig;
  origin  S2_PART iorig;
  def_extent parts  as select PART(sorig: s_i, iorig: i_i)
                           from s_i in s1_parts, i_i in s2_ptrs
                           where s_i.part_id = i_i.prt_id;
  def_att part_id  as this.sorig.part_id;
  def_att upd_date as this.sorig.upd_date;
  def_att description as this.iorig.ptr_tpflg;}
```

Since the derived classes can use a general query to draw their extents from the *origin* classes, they can be used for functionality that corresponds to the DTs in Amos II . Extent definitions with outerjoin conditions could be used to define constructs similar to the IUTs, but this are not elaborated in the IRO-DB reports available to us, nor are special query processing techniques to support this type of operators presented. Also, the derived classes are not placed in the class/type hierarchy as are the DTs and IUTs in Amos II .

As Amos II , IRO-DB also uses proxy objects in the interoperable system to represent objects in the data sources. The same mechanism is used for the derived classes. This mechanism is similar to the coercion mechanism used for the Amos II DTs. However, the Amos II IUTs are different. When IUTs are used in Amos II , no new OIDs are created (and no coercion is used) since the extent of the IUT is an union of disjunctive sets of

object instances of the auxiliary subtypes. Another difference in the proxy manipulation is that in Amos II the proxy OIDs are generated in the mediator corresponding to the interoperable layer in IRO-DB, while in IRO-DB these are generated by the LDAs. This leads different internal representation of the OIDs of the standard class objects and the OIDs of the imported class objects. The objects of the later type have longer OIDs storing redundant class and source information that in Amos II is stored in the interoperable schema as a property of the imported classes.

The handling of the requests for object attribute values also differs considerably between the systems. In IRO-DB when a proxy object is used, the systems accesses the data source and materializes in the interoperable database (also named home database) all the attributes of the object. Possible references to other global objects are replaced by global OIDs, if these objects are already in the home database. Otherwise, these objects are retrieved first and then assigned global OIDs. The process proceeds until no unresolved object references exist in the materialized object graph. After this materialization, the queries using this object within a single transaction, access the local copy. The home database thus acts as an object cache of all integrated data in IRO-DB.

IRO-DB queries can be processed using two modes of operation: (i) ad-hoc queries can be processed by ignoring the current contents of the home-database and rematerializing there a superset of the object instances needed for the query evaluation before processing the query over the cache; (ii) long-transaction queries that are more likely to access the same objects more than once, and therefore the query processor tries to materialize only the objects missing in the home-database with the cost of more complicated processing.

Compared to this mechanism, Amos II uses selective retrieval of the proxy object function values that are used in the queries. This approach does not pay the penalty of retrieving some (possibly big) unused attributes and long chains of object referenced from the first object. In conjunctive Amos II queries, the calculus rewrites remove the common subexpressions that produce most of the repeated accesses to a single function. It is possible, in a rare case, that the same function values are retrieved twice within same conjunctive query that has two variables ranging over a single proxy type. This is a rare case and the penalty is big only when the function values are very large or the function invocation is very costly. Prefetching of proxy function values can be more useful in Amos II in the context of disjunctive queries as the one used when processing of queries over the IUTs. However, the analysis of these queries is much more complex than the analysis of conjunctive queries. Such features are one of the future research topics in the Amos II project.

Some issues that are addressed in Amos II , but to the extend of our knowledge, are not considered in IRO-DB are: (i) optimization of queries over combined local and imported data, (ii) queries with outerjoins and complex reconciliation functions, (iii) queries over hierarchies of derived classes and (iv) experimental study of the performance of the presented query processing strategies. The IRO-DB project is succeeded by the MIRO-Web project [17].

## 1.9 DIOM

The *Distributed Interoperable Model* (DIOM) project [31, 39] has developed a distributed mediation framework based on the ODMG-93 data model. The goal of this project is to

provide a scalable platform for uniform access to autonomous and heterogeneous systems based on evolving and composable mediators. A network of domain-specific mediators is deployed to support application access to the data in the data sources. Each mediator is instantiated from a meta-mediator by defining an integration schema. The meta-mediator architecture, *Diorama*, consists of two layers: a mediator layer and a wrapper layer. The mediator layer contains:

- *Interface manager:* provides a GUI interface and an API that expose the mediator functionality to the users.
- *Distributed query mediation services:* provides source selection, query decomposition, parallel access plan generation and result assembly.
- *Runtime supervisor:* executes subqueries in the wrappers.
- *Information source catalog manager:* manages the data source information and interface repository meta-data. Communicates with the local implementation repository in the wrapper layer in the management of the local wrappers data.

The wrapper layer has the following components:

- *Query wrapper service manager:* receives the requests from the runtime supervisor, translates the query in DIOM to a query in a local language using the data in the implementation repository, executes the subquery and returns results.
- *Implementation repository manager:* maintains the correspondence between the source data and its DIOM representation.

The unified view of the data in the repositories is built using *meta-operations* applied to *base interfaces* representing data in the data sources and *compound interfaces* built recursively by meta-operations. There are four meta operations in DIOM:

- *Aggregation* allows composition of a new interface based on a number of existing interfaces. The new interface can reference the existing interfaces when defining attributes. For example, a new interface *employment* can be defined that links employees from one database with departments from another.
- *Generalization* is used to merger several semantically similar interfaces into one. The new interface abstracts some common properties/attributes of the merged interfaces. An instance union semantics is used that does not provide for overlap resolution.
- *Specialization* creates a new interface by adding new attributes or operations to an existing interface.
- *Import/Hide* is used to import portions of schema from other DIOM mediators. It preserves the closure of the imported subschema by implicitly importing the types of the attributes and operations of the explicitly imported types. The *hide* clause can be used to exclude certain attributes from importing. The imported interfaces can also state their relationship in the exporting interface hierarchy using the *ISA* keyword. This meta-operation corresponds to the proxy type mechanism in Amos II .

Queries over integrated schemas are posed in a language named *interface query language* (IQL). The syntax of IQL is similar to the one proposed by the ODMG-93 OQL. One distinction is the *target* clause that is added to the *select-from-where* block to describe the possible data sources where the query is applied. The authors also propose a mechanism for automatic detection of equi-joins among the object types used in the *from* clause, to relieve the user of specifying obvious conditions in the *where* clause.

The IQL queries are processed in 5 phases:

- *Query routing* This phase selects the relevant information sources from the set of all available sources, by mapping the domain model terminology to the source model terminology.

- *Query Decomposition* Partitions a query expressed over a compound interface into queries over the basic interfaces used in the definition of the compound interface. Interfaces defined using aggregation and generalization meta-operations are substituted by n-ary join and union expressions respectively. Selections and projections are pushed down to the sources while joins that are performed at the same site are grouped together.

- *Parallel access plan generation* The query scheduling strategy described in [39] first builds a join operator query tree (schedule) using a heuristics approach, and then assigns execution sites to the join operators using an exhaustive cost-based search. Amos II , on the other hand, performs a cost-based schedule composition and heuristic execution site assignment. Furthermore, the scheduling process in DIOM is centrally performed, and no distinction is made between the data sources and the mediators in the optimization framework, ignoring thus the problem of having sources with different capabilities. DIOM uses a parallel execution cost model. This is one of the current research issues in the Amos II project.

- *Subquery Translation and Execution* performs tasks similar to that of the wrapper layer in Amos II .

- *Query result packaging and assembly* This phase uses the results of the subqueries generated by the query decomposition to assemble the result required by the user.

DIOM does not specify constructs for resolving conflicts in an overlap among the data in the data sources. Also, no strategies to optimize queries over a combination of local and reconciled data are presented. Finally, no quantification of the benefits of the proposed strategies is presented in the available DIOM project reports.

## 1.10 UNISQL

The UNISQL [25] system is one of the first commercial products that provide views for database integration. The data integration views are build of *virtual classes* that correspond to the Amos II DTs, but are organized in a separate hierarchy. The virtual class instances inherit the OIDs from the ordinary class objects. This does not provide for definition of

stored functions over virtual classes defined by multiple inheritance, as in Amos II . In UNISQL there is no mechanism corresponding to the IUTs in Amos II , but rather a set of queries can be used to specify a virtual class as an union of other classes. This relationship is not included in the type hierarchy, imposing two different kinds of dependencies among the virtual classes.

## 1.11 Remote-Exchange

The remote exchange project at University of Southern California [16] uses a CDM similar to the one used Amos II to establish a framework for instance and behavior sharing. Three dimensions of freedom are explored for function application in a federated database environment: the location of the function (local or remote), the location of the arguments (local or remote), and the type of the function (stored or computed). Each case is elaborated and an abstract implementation description is given. Most of the cases correspond to the ones present in Amos II , although the terminology differs considerably. One case which is not covered in Amos II is the execution of remote derived functions over local objects. In this case the execution is performed at a remote site, where each function call in the definition of the derived function is trapped and a callback is issued to the local system for the needed argument values. This requires that the function calls used in the remote derived function evaluated over a local object have exactly the same name and semantics in the remote database as they have in the local one, limiting the use of the feature.

The disadvantages of the Remote-Exchange approach is that it forces late binding on every function execution that might need to be done remotely. Next, all remote operations are performed on an instance basis by performing a RPC for each individual instance. Another performance degradation is caused by the size of the surrogate identifiers for remote instances which are 300 bytes long and contain all the information needed to perform the remote function evaluation over the instance. Data integration features as the DTs and IUTs in Amos II are not described.

## 1.12 Myriad

Myriad [28, 29] is a federated database project developed at the University of Minnesota. The federation is defined as an integrated database with a global schema consisting of a set of global relations. This relational schema can be specified over data tables stored locally, as well as in other relational database systems accessed by gateways. An SQL-like language is used to query the integrated database schema. The goal of this project is to provide global query processing and transaction management over a set of autonomous and heterogeneous relational DBMS storing pre-existing data.

The global schema is generated from the export schemas by a specification based on outer-joins and the *generalized attribute derivation* GAD operator. The GAD operator is a reconciliation specification mechanism by which the local database attributes are mapped to a corresponding global schema attribute. The following example, based on a example in [28], defines a global *res* relation based on a three relations *res_A, res_B* and *res_C* stored in the relational databases *A, B* and *C* accordingly:

```
RES <-- GAD(
    OUTERJOIN({res_A, res_B, res_C},
       (res_A.rname = res_B.rname)
       (res_B.rname = res_C.rname)
       (res_C.rname = res_A.rname)),
  (rname   F_key(res_A.rname, res_B.rname, res_C.rname))
  (rating  F_avg(res_A.rating, res_B.rating))
  (cost    F_max(res_A.cost, res_B.cost)))
```

In the example, it is assumed that the three data sources have equal schema. The outerjoin is performed over the attribute *rname*. The reconciliation is performed by the system defined functions $F\_fn$. They perform the usual aggregation operations, except the $F\_key$ function which picks the first of the arguments with a non-null value.

Global queries are accepted by a *Federated Query Manager* (FQM), that performs the translation into executable plans executed by *Federated Transaction Management*. On the data source side, the function of the wrapper is performed by a *Federated Transaction Agent* which accepts the requests and invokes the *Federated Query Agent* which processes the requests and handles the communication with the data source.

The FQM translates a query over the global schema into a set of queries over individual export schemas and a set of result assembling operations executed in the FQM. Although a fully fledged query optimization module is not implemented [28], [29] presents an extensive study of optimization of queries including several outerjoins and GAD operators. This work manipulates the queries in a formalism named *constrained query trees* (CQT). CQTs are relational operator query trees extended with n-ary union, join and outerjoin operators. The authors note that a rigid interpretation of the definition of the outerjoin does not yield the expected result when more than two outerjoins are performed in sequence, and introduce operators to correct this problem. A graph of dependencies among the input relations is assigned to each n-ary operator node to allow transformations that, under certain conditions:

1. transform outerjoins into joins
2. split n-ary outerjoin nodes into an equivalent tree of two outer join nodes
3. distribute GADs over outerjoins
4. commute selections and projections over with GAD and outerjoins
5. distribute joins over outerjoins

Some of these transformations are similar or extend transformations described in other approaches (e.g. 5 in [12], 1 and 4 in [10], and [32, 47], etc.). The application of these transformations is subject to conditions defined over the attributes involved in the transformed nodes. It is not clear how this framework will perform in practice. There is no cost model defined to evaluate the benefits of the transformations and/or heuristics to determine which transformation is beneficial for a given tree, or how to choose a transformation that will lead to the best tree. The framework has not been experimentally evaluated.

For other research on optimization of sequences of outerjoins the reader is referred to [18, 19] where outer joins are treated as disjunctions of joins and ant-semi-joins (as in Amos II ); [2] uses hypergraphs for outerjoin reordering; and [21] describes how to push selections through outerjoins.

# 2 Object-Oriented Views

The integration facilities of Amos II are based on work in the area of OO views [1, 24, 40, 41, 44, 26, 3, 41, 33]. This section gives a short overview of two prototypes that has been most influential for the design of the OO views for data integration in Amos II .

## 2.1 Multiview

The Multiview [26, 27, 40] OO view system adds dynamically updateable materialized OO views on the top of the GemStone OO DBMS. The views are defined by defining *virtual classes*, which are placed in the same class hierarchy as as the ordinary GemStone classes. The virtual classes are *capacity-augmenting*, i.e. attributes and methods can be added to them, as to the ordinary GemStone classes. Virtual classes are defined using six object-preserving algebra-operators:

- **select**: Returns a subset of the input class based on a predicate expression.
- **hide**: Removes properties from a set of objects.
- **refine**: Casts a set of input objects downwards in the class hierarchy (i.e. changes the class of the input objects to a subclass of their original class).
- **union**: Makes a union of two input class extents. The equality condition is OID equality.
- **intersection**: Returns an intersection of the extents of two classes.
- **difference**: Returns an difference of the extents of two classes.

The classes in GemStone and Multiview are organized in a multiple inheritance hierarchy. As Amos II , GemStone also requires that each object instance belongs to a single most specific class. In presence of declaratively specified virtual classes, it is impossible to guarantee that two virtual classes will not both contain a same instance of some of their common superclasses, which at the same time is not an instance of any of their common subclassess. For example, if a class *Person* is subclassed by two virtual classes *Student* and *Teacher* which do not have any common subclasses, there might be a person that satisfies both the conditions of being a student and a teacher. Such an instance would violate the requirements of belonging to a single most specific class. In [27] it is suggested that, to solve this problem, in a multiple inheritance OO views hierarchy the system must either generate automatically the intersection classes to classify this instances, or assign unique OIDs to the instances of the virtual classes. The second solution, which is applied in both Multibase and

Amos II , furthermore requires *single point of inheritance* property of the class hierarchy. This property guarantees that two classes having inherited the same property, inherit it from a single class in the class hierarchy.

The Multibase system uses an elaborate solution where each object is represented by a single *conceptual object* and a number of *implementation objects* for each of its superclasses. The graph of each conceptual object with its corresponding implementation objects mirrors the class hierarchy.

This idea has been simplified and adapted in Amos II where there is no distinction between implementation and conceptual objects. In Amos II , these relationships are stored in coercion tables. The benefit of this approach is that, in a data integration scenario, new view classes can be defined over already existing populated classes. The instances of the view classes can then have their own OIDs without affecting the classes they are derived from.

Within the Multibase system, a class restructuring strategy is proposed to avoid conflicts in the class hierarchy by preserving the single point of inheritance property when new view classes are added. Because of the complexity of that process, in Amos II we adopted some modeling constraints in order to prevent situations in which these transformations are needed.

Multiview is an implemented system with experimental results reported. However, it assumes active view materialization techniques and does not elaborate the consequences of the use of the applied techniques for data integration in a distributed heterogeneous environment.

## 2.2 O2 Views

The $O_2$ system is one of the first commercial OO systems to provide OO view functionality [35, 41]. Before the introduction of the OO view system, the $O_2$ system relied on named sets to provide some of the OO view features. Named sets however, do not provide some important features as: (i) description of the structure of the objects in the set, (ii) inheritance of methods from already defined classes (iii) attachment of new methods, etc..

The $O_2$ views are implemented on the top of the $O_2$ system. The views are defined using *virtual schemas* derived from *root schemas*. A root schema can either be another virtual schema or an $O_2$ schema. This allows for composition of views to an arbitrary degree of nesting. Corresponding to the root and virtual schemas there are a root and a virtual (data)base, representing the instances involved in the view mapping.

The views filter the data of the root base into the virtual base. Two modeling constructs are added to the $O_2$ data definition language to support the definitions of the filter mapping: *virtual classes* (VC) and *imaginary classes* (IC). A virtual class is defined as a subclass of a virtual or an ordinary $O_2$ class, named *root class*. A VC inherits the attributes of its root class, and can also have *virtual attributes* with functionality equivalent to the derived functions in Amos II . Some attributes of the root class can be declared *hidden* and therefore not accessible to the user of the VC. Other properties of the VCs are that they:

- have an extent selected by a declarative query form the root database.
- are connected to the class hierarchy.

- provide a named set representing the extent.
- provide OIDs for the class instances based on the one-to-one correspondence with instances in the root database.

The ICs have the following properties:

- The IC extent is selected by a declarative query from the root database.
- The ICs are not connected to the class hierarchy.
- The ICs assign OIDs to the instances based on a set of *core attributes*, corresponding to keys.

The following example, in which a VC *Adult* is defined as a specialization of the class *Person*, illustrates the language constructs used for the VC definition:

```
virtual class Adult from Person extension Adult
   virtual attributes
      age: integer has value self-> age;
   hide attribute date_of_birth
   includes
   (select p from p in People where p->age >= 21)
```

where *self* references the corresponding object of class *Person*, and the *includes* clause defines how is the extent of the VC selected from the extent of the root class.

The separation of the view definition facilities between the VC and IC constructs provide for a wide range of restructuring capabilities, while preserving the consistency of the class hierarchy. In comparison with Amos II , the IC approach is in Amos II used in the proxy types that retrieve their data from data sources other than Amos II mediators. The VCs are equivalent to Amos II DTs having a single supertype. In the query processing, Amos II relies as much as possible on OIDs, rather than on key values as the $O_2$ view system. When subtyping among Amos II mediators, OIDs are used and manipulated because they are at least as small as the shortest possible key of an object. We assume that there is a functional dependency between the keys and the OID of an object, and therefore key manipulation is not needed in intersection-based OO views, as the DTs.

The $O_2$ views mechanism does not provide multiple inheritance and integration facilities as the DTs and IUTs in Amos II . Therefore this approach can be classified as a class restructuring mechanism, or a selection-based view mechanism. For more advanced view definitions, the user is still limited to the named sets constructs.

# References

[1] S. Abiteboul and A. Bonner: Objects and Views. *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'91)*, pp. 238-247, ACM Press, 1991.

[2] G. Bhargva, P. Goel and B. Iyer: Hypergraph based reorderings of outerjoin queries with complex predicates *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'95)*, pp. 304-315, ACM Press, 1995.

[3] E. Bertino: A View Mechanism for Object-Oriented Databases. *3rd Intl. Conf. on Extending Database Technology (EDBT'92)*, Vienna, Austria, 1992.

[4] O. Bukhres, A. Elmagarmid (eds.): Object-Oriented Multidatabase Systems, Pretince Hall, Englewood Cliffs, NJ, 1996.

[5] A. Bouguettaya, B. Benatallah and A. Elmagarmid: Interconnecting Heterogeneous Information Systems. Kluwer Academic Pulishers, The Netherlands,1998

[6] R. Cattell: The Object Database Standard: ODMG-93 2.0, Morgan Kaufman Publishers, San Mateo, CA, 1996

[7] U. Dayal, N. Goodman, T. Landers, K. Olson, J. M. Smith and L. Yedwab: Local Query Optimization in MULTIBASE: A System for Heterogeneous Distribited Database, Techical Report CCA-81-11, Computer Corporation of America, 1981

[8] U. Dayal, T. Landers and L. Yedwab: Global Query Optimization in Multibase: A System for Heterogeneous Distribited Database, Techical Report CCA-82-05, Computer Corporation of America, 1982

[9] U. Dayal: Processing Queries Over Generalization Hierarchies in a Mutltidatabase System, *9th Conf. on Very Large Databases (VLDB'83)*, Florence, Italy, 1983

[10] U. Dayal, H. Hwang: View Definition and Generalization for Database Integration in a Multidatabase System, *IEEE Trans. on Software Eng.* 10(6), Nov. 1984.

[11] W. Du, R. Krishnamurthy and M-C. Shan: Query Optimization in Heterogeneous DBMS. *18th Conf. on Very Large Databases (VLDB'92)*, Vancouver, Canada, 1992

[12] W. Du and M. Shan: Query Processing in Pegasus, *Object-Oriented Multidatabase Systems*, O. Bukhres, A. Elmagarmid (eds.), Pretince Hall, Englewood Cliffs, NJ, 1996.

[13] C. Evrendilek, A. Dogac, S. Nural, F. Ozcan: Query Optimization in Multidatabase Systems. *Journal of Distributed and Parallel Databases* Vol. 5 No. 1, January 1997, pp. 77-114.

[14] V. Smahi, J. Fessy and B. Finance: Query Processing in IRO-DB, *Int. Conf. on Deductive and Object-Oriented Databases (DOOD'95)* pp.299-319, 1995

[15] D. Fishman, D. Beech, J. Annevelink, E. Chow, T. Connors, J. Davis, W. Hasan, C. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M-A. Neimat, T. Risch, M-C Shan, W. Wilkinson: Overview of the Iris DBMS. In W. Kim and F. Lochovsky (eds.): *Research Foundations in OO and Semantic DBS* pp. 174-199, 1990

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

[17] P. Fankhauser, G. Gardarin, M. Lopez, J. Munoz and A. Tomasic: Experiences in Federated Databases: From IRO-DB to MIRO-Web. *24st Conf. on Very Large Databases (VLDB'98)*, New York, NY, 1998

[18] C. Galindo-Legaria: Outerjoins as Disjunctions, *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'94)*, pp. 348-358, ACM Press, 1994.

[19] C. Galindo-Legaria and A. Rosenthal: Outerjoin Simplification and Reordering for Query Optimization, *ACM Transactions on Database Systems*, Vol. 22, No. 1, March 1997

[20] 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 (JIIS)* Vol 8 No. 2 117-132, Kluwer Academic Pulishers, The Netherlands,1997

[21] P. Goel and B. Iyer: Query Optimization: Reordering for a general Class of Queries. *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'96)*, pp. 47-55, ACM Press, 1996.

[22] L. Haas, D. Kossmann, E. Wimmers, J. Yang: An Optimizer for Heterogeneous Systems with NonStandard Data and Search Capabilities. *Data Engineering Bulletin* Vol. 19 No. 4 pp 37-44, 1996

[23] L. Haas, D. Kossmann, E. Wimmers, J. Yang: Optimizing Queries accross Diverse Data Sources. *23th Int. Conf. on Very Large Databases (VLDB97)*, pp. 276-285, Athens Greece, 1997

[24] S. Heiler and S. Zdonik: Object views: Extending the Vision. *6th International Conf. on Data Engineering (ICDE'90)*, IEEE, pp. 86-93, 1990

[25] W. Kelley, S. Gala, W. Kim, T. Reyes, B. Graham: Schema Architecture of the UNISQL/M Multidatabase System, *Modern Database Systems - The Object Model, Interoperability, and Beyond*, W. Kim (ed.), ACM Press, New York, NY, 1995.

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

[27] H. Kuno and E. Rundensteiner: The MultiView OODB View System: Design and Implementation *University of Michigan Technical Report* CSE-TR-246-95, 1995.

[28] 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*, Vol. 25(5), 553-562, John Wiley & Sons, May 1995.

[29] E-P. Lim, J. Srivastava and S-Y. Hwang: An Algebraic Transformation Framework for Multidatabase Queries, *Journal of Distributed and Parallel Databases* Vol 3. No.3, pp 273-307, Kluwer Academic Pulishers, The Netherlands, 1995

[30] P. Lyngbaek et al: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[31] L. Liu and Calton Pu: An Adaptive Object-Oriented Approach to Integration and Access of Heterogeneous Information Sources. *Journal of Distributed and Parallel Databases* Vol 5. No. 2, pp. 167-205, Kluwer Academic Pulishers, The Netherlands, 1997.

[32] W. Meng, K-L. Liu and C. Yu: Query Decomposition in Multidatabase Systems. Techival Report CS-TR-93-9, Department of Computer Science, State University of New York and Binghampton, 1993

[33] A. Motro: Superviews: Virtual Integration of Multiple Databases. *IEEE Transaction on Software Engineering*, Vol. SE-13, No. 7, July 1987.

[34] S. Nural, P. Koksal, F. Ozcan, A. Dogac: Query Decomposition and Processing in Multidatabase Systems. *OODBMS Symposium of the European Joint Conference on Engineering Systems Design and Analysis*, Montpellier, July 1996.

[35] O2 Technology: O2 Views User Manual, version 1, Dec. 1993

[36] F. Ozcan, S. Nural, P. Koksal, C. Evrendilek, A. Dogac: Dynamic Query Optimization on a Distributed Object Management Platform *Fifth International Conference on Information and Knowledge Management (CIKM96)*, Maryland, USA, November 1996.

[37] M. Roth and P. Schwarz: Don't Scrap It, Wrap It. *23th Int. Conf. on Very Large Databases (VLDB97)*, pp. 266-275, Athens Greece, 1997

[38] F. Rendeze, K. Hergula: The heterogeneity problem and middleware thechonlogy: Experience and performance of database gateways. *24th Conf. on Very Large Databases (VLDB'98)*, New York, 1998

[39] K. Richine: *Distributed Query Scheduling in DIOM.* Tech. Report TR97-03, Computer Science Department, University of Alberta, 1997.

[40] E. Rundensteiner, H. Kuno, Y. Ra, V. Crestana-Taube, M. Jones and P. Marron The MultiView project: object-oriented view technology and applications, *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'96)*, pp. 555-563, ACM Press, 1996.

[41] C. Souza dos Santos, S. Abiteboul and C. Delobel: Virtual Schemas and Bases. *4th Intl. Conf. on Extending Database Technology (EDBT'92)*, Viena, Austria, 1992.

[42] D. Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), ACM Press, 1981.

[43] V. Smahi, J. Fessy and B. Finance: Query Processing in IRO-DB Technical Report PRiSM, Versailles University 1994/37, Versailles, France, 1994

[44] M. Scholl, C. Laasch and M. Tresch: Updatable Views in Object-Oriented Databases. *Second Deductive and Object-Oriented Databases Conference (DOOD91)*, Dec, 1991.

[45] S. Subramananian and S. Venkataraman: Cost-Based Optimization of Decision Support Queries using Transient Views. *ACM SIGMOD Intl. Conf. on Management of Data (SIGMOD'98)*, pp 329 - 330, 1998

[46] S. Venkataraman and T. Zhang: Heterogeneous Database Query Optimization in DB2 Universal DataJoiner *24th Conf. on Very Large Databases (VLDB'98)*, pp 685 - 689, New York, 1998

[47] C. Yu and W. Meng: Princliples of Database Query Processing for Advanced Applications, Morgan Kaufman Publishers, San Francisco CA, 1998