Integrating Heterogeneous Overlapping Databases Through Object-Oriented Transformations

Vanja Josifovski
Laboratory for Engineering Databases
Linköping University, Sweden
vanja@ida.liu.se

Tore Risch
Laboratory for Engineering Databases
Linköping University, Sweden
torri@ida.liu.se

Abstract

Integration of data from autonomous and heterogeneous data sources often requires means to mediate and reconcile overlaps and conflicts between the integrated data. It is also desirable that the mediator system stores local data associated with the data from the sources. Queries over such a combination of local and mediated data must be guaranteed to be consistent and complete in presence of updates to any of the data sources. Due to these complex requirements, achieving acceptable query response time for a mediator system has been a known research problem. This work presents a mediator query processing framework based on a novel representation of the data mediation and reconciliation by a number of auxiliary system-defined object-oriented (OO) views and overloaded functions (queries). The framework is supported by defining an overloading and late binding mechanism for the OO views through declarative queries. A query over the mediated OO views will have late bound subquery invocations which are transformed into disjunctive query expressions. Consistency and completeness of the queries are guaranteed by expanding the queries with validation subqueries. Performance improvements are achieved by optimization of the query expressions using type aware query rewrites and selective OID generation in the mediators. Experiments show that the proposed query optimization dramatically improves the query execution time compared to a naive instance-oriented query strategy or partial strategies.

Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.

1 Introduction

Modern organizations often need to combine heterogeneous data from different data sources. Tools and infra-structures for *data integration* are required. Data integration using the *wrapper-mediator* architecture [19] with an Object-Oriented (OO) data model is a popular approach to integration of heterogeneous databases [2, 4, 5, 9, 10, 12, 18]. With this approach, the data sources are encapsulated in *wrappers* which interface the data sources using a common query language and a common data model (CDM). The role of the *mediators* is to provide a semantically coherent CDM view of the combined data from the wrapped data sources.

The data and the meta-data (schema) in the sources can have conflicting and overlapping portions. For example, two universities can each have employee databases organized in different ways with corresponding entities bearing different names. Also, there might exist employees employed by both universities. A comprehensive classification of these conflicts can be found in [3]. In this work we will concentrate on a framework for mediating a coherent view of databases in presence of *structural conflicts*, where attributes modeling the same real world property does not match in name and/or value, and *entity overlap conflicts* where there is an overlap of the sets of real-world entities represented by the data in the sources.

In particular, this paper deals with managing OO mediator views defined as unions of real-world entities from other mediators and data sources. Our mediating union views are modeled by a mechanism called *integrated union types* (IUTs) based on OO queries and views. The IUTs model unions of real-world concepts similar to [4, 5], and opposed to unions of type extents from different databases as in [18, 10]. IUTs have *reconciliation* facilities which allow the user to specify how overlaps and conflicts between data from different sources are resolved.

Users and applications using a mediator often need to associate some locally relevant data to the data integrated from the data sources. We call such mediators, permitting local methods and attributes in the OO views, *capacity augmenting mediators.* Capacity augmentation for the IUTs is achieved by making the instances of the IUTs first class objects with their own OIDs that can be used in locally stored attributes and methods as ordinary OIDs.

The data sources are autonomous and can be updated outside the control of the mediators. The system must therefore guarantee the consistency and completeness of queries to the capacity augmented mediators in presence of updates to the data sources. Our framework for IUTs guarantees that queries to the mediators are consistent and complete when the data sources are updated without any need for a notification mechanism. The queries over the integrated views always return *all* answers that qualify the query condition, and *only* those answers that qualify, based on the *current* state of the data in the data source, regardless of any database state materialized in the mediator.

It is challenging to achieve acceptable performance of OO queries over IUTs, in particular when the integrated extents have overlaps [4, 5]. Such overlaps require outerjoin-based query processing techniques having increased complexity compared to inner joins. Furthermore, queries involving both local and remote data should take advantage of the fast access to local data to improve performance.

This work presents a combination of query processing strategies that significantly improve the performance of queries over IUTs in capacity augmented mediators. The main principles of these strategies are:

1. The IUTs are internally represented as a set of *auxiliary views*, over which the reconciliation is specified by a set of overloaded auxiliary methods (queries).
2. The queries over the IUTs containing outerjoins and reconciliation are translated into late bound queries over the auxiliary views and methods.
3. In order to permit further query rewrites, the late bound queries are translated into disjunctive query expressions. These model the original query by joins and anti-semi-joins which are easier to rewrite and optimize.
4. Novel, type-aware query rewrite techniques remove inconsistent disjuncts and simplify the transformed disjunctive queries.
5. To efficiently support consistent and complete query answers the system uses a novel technique for selective OID generation and validation of the OO view instances, based on declarative queries.
6. Finally, local main-memory indexes created on-the-fly in mediators eliminate repeated accesses to data sources.

Experimental results show that the combination of the above methods has drastically better performance than a naive CORBA-like integration that resolves late binding on an object instance level at run time. The performance is drastically reduced even if only some of the combined optimization methods are relaxed.

2 Background

As a platform for our research we use the AMOS*II* mediator database system [11] developed from WS-Iris [15]. The core of AMOS*II* is an open light-weight and extensible DBMS. AMOS*II* is a distributed mediator system where both the mediators and wrappers are fully functional AMOS*II* servers. For good performance, and since most the data reside in the data sources, AMOS*II* is designed as a main-memory DBMS.

AMOS*II*'s CDM is an OO extension of the DAPLEX [17] functional data model. It has three basic constructs: *objects*, *types* and *functions*. Objects model entities in the domain of interest. An object can be classified into one or more types 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). Object attributes, queries, methods, and relationships are modeled by functions.

The non-literal types are divided into *stored*, *derived*, *translated*, and *proxy* types:

- The instances of *stored* types are explicitly stored in the mediator and created by the user.

- The extent of a *derived* type (DT) is a subset of the extents of one or more *constituent* supertypes specified through a declarative query over the supertypes. Its extent is a subset of the *intersection* of the extents of the constituent types. Each DT has an associated *extent function* defining its extent; an *OID generation function* for creating its OIDs; and a *validation function* which for a given DT object checks if the object is still valid, based on the state of the objects of its constituent types and the declarative condition given in the DT definition. The DTs are described in greater detail in [13].

- The *proxy* types represent objects stored in other AMOS*II* servers or in some of the supported types of data sources. In this work we will only use ODBC data sources.

The functions in AMOS*II* are divided by their implementations into four groups. The extent of a *stored* function is physically stored in the mediator (c.f. object attributes). *Derived* functions are implemented by queries in the query language AMOSQL (c.f. views and methods). *Foreign* functions are implemented in some other programming language, e.g. C++ or Java (c.f. methods). To help the query processor, a foreign function can have associated cost and selectivity function. The *proxy* functions are implemented in other AMOS*II* servers.

The AMOSQL query language is similar to OQL and based on OSQL [16] with extensions of multi-way foreign functions, active rules, late binding, overloading, etc. For example, assuming three stored function *parent*, *name* and *hobby*, the query on the left retrieves the names of the parents of the persons who have 'sailing' as a hobby:

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

create derived type sailors
 subtype of person
  where hobby(person)='sailing'
  properties (yachtType string);
```

The DT definition on the right defines a type representing persons having 'sailing' as a hobby, and defines a stored function 'yachtType' over this type.

The query processing in AMOS*II* first translates the AMOSQL queries into a type annotated *object calculus* representation. For example, the result of the calculus generation phase for the query from the example above is given by the following calculus expression:

$$\begin{aligned}
&\{\, p, nm \mid \\
&\quad p = Person_{nil \rightarrow person}() \wedge \\
&\quad d = parent_{person \rightarrow person}(p) \wedge \\
&\quad nm = name_{person \rightarrow charstring}(d) \wedge \\
&\quad 'sailing' = hobby_{person \rightarrow charstring}(p) \}
\end{aligned}$$

The first predicate in the expression is inserted by the system to assert the type of variable p. It defines the variable p to be member of the result of the extent function for type the $Person$. In case of a DT, the extent function contains a query defining the extent in terms of predicates over the supertypes. The extent function can be used to generate the extent of a type, as well as to test if a given instance belongs to a type. Therefore, a predicate containing a reference to an extent function is called a *typecheck predicate*. An extent function accesses the *deep extent* of the type, i.e. it includes the extents of all the supertypes. By contrast, the *shallow extent function* considers only the immediate instances of the type. By convention, the shallow extent functions are named by prefixing the type name by the prefix $Shallow$, e.g. $ShallowPerson_{nil \rightarrow Person}()$.

In the second processing phase, the calculus optimizer applies type-aware rewrite rules to reduce the number of predicates. For the example query, this produces the expression below by removing the type check predicate:

$$\begin{aligned}
&\{\, p, nm \mid \\
&\quad d = parent_{person \rightarrow person}(p) \wedge \\
&\quad nm = name_{person \rightarrow string}(d) \wedge \\
&\quad 'sailing' = hobby_{person \rightarrow string}(p) \}
\end{aligned}$$

This transformation is correct because p is used in a stored function (e. g. *name*) with an argument or result of type $person$. The referential integrity system of the stored functions constrains the stored instances to the correct type [15].

After the rewrites, queries operating over data outside the mediator are decomposed into distributed subqueries expressed in an *object algebra*, to be executed in different AMOS*II* servers and data sources. The decomposition uses a combination of heuristic and dynamic programming strategies. At each site, a single-site *cost-based optimizer* generates optimized execution plans for the subqueries. These system features are not the focus of this paper and will be a topic of a forthcoming paper.

An interested reader is referred to [11] for more detailed description of AMOS*II* and to [15], [6] [13] and [8] for more comprehensive descriptions of the query processing used in AMOS*II*.

3 Integration Union Types

The integration union types (IUTs) provide a mechanism for defining OO views capable of resolving semantic heterogeneity among meta-data and data from multiple data sources. Informally, while the DTs represent restrictions (selections) and intersections of extents of other types, the IUTs represent reconciled unions of data in one or more mediators or data sources.

The description of the IUTs in this section is from a perspective of a database administrator who models and defines a mediating view used later by the users. From the users' perspective, there is no difference between querying IUTs and ordinary types. The view definition process will be illustrated by an example of a computer science department (CSD) formed from 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 the both universities. The full-time members of a department are assigned an office in the department.

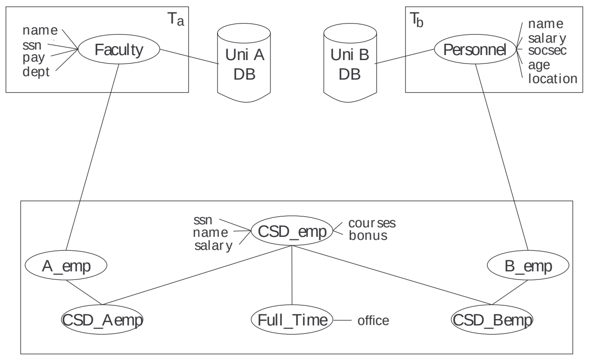


Figure 1: An Object-Oriented View for the Computer Science Department Example

One possible system architecture for the data integration problem described above is presented in Figure 1. In this figure, the mediators and translators are represented by rectangles; the ovals in the rectangles represent types; and the solid lines represent inheritance relationships between the types. The two AMOS*II* servers that provide a CDM representation of the data in the sources are labeled T_A and T_B. To distinguish between the wrapper subsytem in AMOS*II*, and an AMOS*II* server having a *role* of wrapping a data source, the second is named *translator*. The term wrapper will be used to represent the wrapper subsystem.

In T_A, there is a type *Faculty* and in T_B a type *Personnel*. A mediator is setup in the CSD to provide the integrated view. Here, the types *CSD_A_emp* and *CSD_B_emp* are defined as subtypes of the types in the translators:

```
create derived type CSD_A_emp
  subtype of Faculty@Ta
    where dept(A_emp) = ''CSD'';

create derived type CSD_B_emp
  subtype of Personnel@Tb
    where location(B_emp) = ''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 no implementation. In this example, the extents of the DTs are specified as subsets of the extents of their supertypes by using simple selections, but in general the subtyping condition can also be joins. During the query decomposition process, predicates containing proxy functions are scheduled for execution in the mediator or wrapper they originate from.

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, 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 1 are derived from the functions defined in the constituent types. These *reconciled* functions have more than one overloaded implementation, one for each possible combination of constituent types instances, matching an IUT instance. The functions on the right are locally stored functions.

The data definition facilities of AMOSQL include constructs for defining IUTs as described above. 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
      courses BAG OF STRING;
      bonus integer;
END;
```

The IUT *csd_emp* definition reveals some details not apparent from the graphical representation of the integration scenario. The first clause defines a set of *keys* and their types. In the example, the key is single valued of type *integer*. For each of the constituent subtypes, a key expression is given to calculate the value of the key from the instances of this subtype. 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 the values of the functions over the constituent types. For different subsets of the constituent types, a reconciled function of an IUT can have different implementations specified in 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 she 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. If no single most specific case exists (e.g. *name*), the system assumes "any" semantics and chooses one based on a heuristic to improve the performance of the queries over these functions.

Finally, the `PROPERTIES` clause defines the two stored functions over the IUT *CSD_emp*. At any time after the definition of an IUT, the user can add stored or derived functions. The derived functions can be based on any functions already defined in the mediator, regardless whether they are implemented locally or in some other AMOS*II* server.

The IUTs can be subtyped by DTs as any other types. In the example in Figure 1, the type *Full_Time* representing the full time employees is defined as a subtype of the type *CSD_emp*. The locally stored function *office* stores the information about the offices of the full time CSD employees.

4 Modeling and Querying the Integration Union Types

Every instance of an IUT corresponds to either an instance in one of the two constituent types, or to one instance in both of them. Therefore, the extent of an IUT can be divided into three subsets (Figure 2a). Two sets contain the IUT instances corresponding to

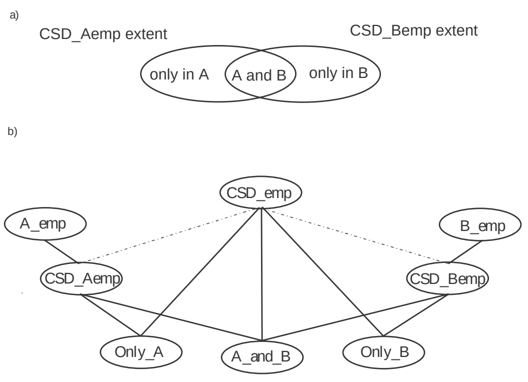


Figure 2: IUT implementation by ATs

an instance in a single constituent type. The third set contains the IUT instances corresponding to instances in both constituent types. Since the extent subsets can be defined by declarative queries, we can define each of them as a DT, named an *auxiliary type* (AT). The three ATs generated for each IUT form an inheritance hierarchy as shown in Figure 2b.

A function f defined over an IUT can have a different implementation for each of these three subsets (i.e. for each `CASE` clause). It can be thus defined for the whole extent of the IUT by being *overloaded* on the ATs. A call to f for an IUT will then result in a late bound function call, to be discussed below.

The ATs are generated by the system and are not visible to the user. Each AT corresponds to a `CASE` clause in the IUT definition. By using the specifications from the `KEYS` clause of the IUT definition, two functions are generated for each constituent type. The overloaded function $key_{CT \rightarrow key\ types}$ calculates the key of an instance of a constituent type CT. The function *Allkeys*`CT`*()* returns all the keys for the type CT. With these functions defined, the AT definitions for the example are:

```
create derived type Only_A
 subtype of CSD_Aemp ae
   where key(ae) not in
         AllkeysCSD_Bemp();

create derived type Only_B
  subtype of CSD_Bemp be
    where key(be) not in
          AllKeysCSD_Aemp();

create derived type A_and_B
  subtype of CSD_Aemp ae, CSD_Bemp be
     where key(ae) = key(be);
```

The first two subtypes represent keys based on anti-semi-joins of the integrated types. The third is a join of the integrated types.

Next, the system creates the IUT and makes the ATs its subtypes. The overloaded function resolvents are then defined over the IUT and each of the ATs. The AT resolvents are generated from the *FUNCTIONS* clause in the IUT definition. The resolvent for the IUT itself is defined as *false* since all the instances of the IUT belong to one of the ATs, giving the optimizer a hint to reduce the execution plans.

The extents of the ATs represent mutually exclusive sets of real world entities. The union of these extents forms the extent of the IUT which therefore contains one instance for each entity. From the user's point of view, the only difference between the IUTs and the ordinary types is that no objects can be explicitly created in the IUTs. The extent of the IUTs are completely derived from the extents of the ATs.

4.1 Late Binding Over Derived Types

To process queries over the system-generated OO views having overloaded functions, we developed a novel late binding mechanism for efficient handling of declarative view definitions in a multiple mediators environment. A late bound function call $f(a)$ is first translated into a calculus *late binding operator* (LBO) whose first argument is a tuple of the possible resolvents of f sorted with the least specific type first, and the second argument is a. For functions used when an IUT is modeled by ATs, the late binding calculus expression is:

$$LBO(< f_{iut}, f_{at1}, ..., f_{atn} >, a)$$

where the ATs $at1 \ldots atn$ are subtypes of iut. Based on the types of the argument a, LBO chooses the most-specific resolvent, executes it over the argument, and returns the result(s).

In our previous work, we have developed a corresponding algebraic late binding operator for the ordinary types, the Dynamic Type Resolver (DTR) [8]. DTR, as most late binding mechanisms described in the literature (e.g. [7]), processes one tuple at a time and selects the query plan of a resolvent based on the type of a. This mode of processing is not suitable for the IUT queries for the following reasons. First, because the resolvents are functions defined over data in multiple sources, processing a tuple at a time results in calling remote functions in an RPC manner. Second, it requires the instances to have assigned OIDs, leading to OID generation for *all* the instances processed in a query, and not only for the ones requested by the user. Furthermore, such a late binding mechanism assumes that the type information of the argument object is explicitly stored with its OID. By contrast, the types in the IUT are defined *implicitly* by queries, and IUT instances can obtain and drop a type dynamically and outside the control of the mediator, based on the state of the data in the sources. Therefore, the use of late binding as above leads into partitioning the query into three separate subqueries: the resolvent function bodies (i.e. the expressions in the `CASE` clauses), the AT subtyping conditions, and the predicate in the query. This separation will prohibit query rewrite techniques to eliminate common subexpressions and other query reduction methods as described in [13] and [6].

In order to overcome these limitations, the LBO is translated into an equivalent disjunctive object calculus predicate, which is then combined and optimized with the rest of the query. AMOS*II* supports multimethods and overloading on all function arguments and the translation algorithm can handle this too. Due to space limitations, here we only present a simplified version of the algorithm that handles overloading on a single argument. The full version of the algorithm will be presented in a forthcoming technical report.

In the translated disjunctive calculus expression every branch (disjunct) is a conjunction of a typecheck for an AT and a call to the overloaded function f corresponding to the AT. The translation algorithm is:

```
generate_lb_calculus( resolvents ) − > disjunctive predicate
  result = {$res$ |}; /*empty disjunction predicate */
  while resolvents != ∅      do
    head = first(resolvents);
    /* the argument type for the head function */
    $t_h$ = arg_type(head);
    if $\nexists f \in resolvents \mid subtype\_of(argtype(f), t_h)$ then
      result = append(result,
                      $\vee\{arg = t_h() \ \wedge res = f_{t_h}(arg)\}$);
    else
      wset = $\{t_p \mid subtype\_of(t_p, t_h) \wedge$
                $\nexists f \in resolvents \mid$
                    $subtype\_of(t_p, argtype(f))\}$
      for each $t_p$ in wset
        result = append(result,
                        $\vee\{arg = Shallow\_t_p(),$
                          $res = f_{t_h}(arg)\}))$;
    end if
    resolvents = resolvents - head;
  end while
  return result;
end;
```

First, *append*, and − perform the usual set operations, and *arg_type* returns the argument type of a function. The algorithm traverses the sorted list of resolvents. If the type hierarchy rooted in the argument type of a resolvent does not intersect with the hierarchies of the argument types of some resolvents in the rest of the list, then a conjunction of an ordinary (deep) typecheck and the resolvent call is added as a new disjunct to the result. Otherwise the new disjunct will instead contain a shallow typecheck. Notice that for IUTs there will be no shallow typechecks, because there are never any subtypes of the system-generated ATs. Since the type checks are mutually exclusive, only one resolvent will be evaluated.

To illustrate the translation process we examine the translation of the LBO for the function *salary* over the IUT *CSD_emp*:

$$LBO(< salary_{csd\_emp \rightarrow int}, salary_{Only\_A \rightarrow int}, salary_{Only\_B \rightarrow int}, salary_{A\_and\_B \rightarrow int} >, arg)$$

is translated into:

$$\{ s \mid (arg = only\_A_{nil \rightarrow only\_a}() \ \wedge \ s = salary_{only\_A}(arg)) \ \vee$$
$$(arg = only\_B_{nil \rightarrow only\_b}() \ \wedge \ s = salary_{only\_B}(arg)) \ \vee$$
$$(arg = A\_and\_B_{nil \rightarrow a\_and\_b}() \ \wedge \ s = salary_{a\_and\_b}(arg))\}$$

The expression is a disjunction of only three disjuncts. No disjunct is generated for the first resolvent $salary_{csd\_emp \rightarrow int}$ since it is defined as *false*.

After the query normalization, the extent functions of the ATs are expanded by substituting them with their bodies containing the expressions from the `CASE` clauses of the IUT definition. These expressions in turn reference the extent functions of the constituent types, which are DTs and the expansion continues until no DT extent functions are present. This process makes visible to the query decomposer i) the query selections defined by the user, ii) the conditions in the IUT, and iii) the DT definitions. The query decomposer combines the predicates, divides them into groups of predicates executable at a single mediator, translator or data source, and then schedules their execution. As opposed to dealing with parametric queries over multiple databases, as would have been the case with a tuple-at-the-time implementation of the late binding, the strategy ships and processes data among the mediators and the data sources in bulks containing many tuples. The size of a bulk is determined by the query optimizer to maximize the network and resource utilization. The results in the next section demonstrate how the bulk-processing allows for query processing strategies with substantially better performance than the instance-at-the-time strategies. Furthermore, this strategy allows the optimizer to detect and remove unnecessary OID generations for the instances not in the query result.

4.2 Normalization of Queries Over the Integration Types

If there are disjunctive predicates, we need to normalize the query to disjunctive normal form in order to separate the subqueries for the individual data sources. One drawback of the query normalization is that it duplicates predicates in several different disjuncts of the normalized disjunctive predicate. To avoid some of the unnecessary duplication, we use a query normalization which is aware of the multidatabase environment. The normalization algorithm is based on the principle that as many as possible of the normalization decisions should be delegated to the sites where the predicates are executed. Therefore the query decomposer analyzes the elements of a disjunctive predicate and groups together the disjuncts executed in the same mediator or data source capable of processing disjunctions.

Another source of disjunctions in queries over IUTs are the late bound functions from above, which are translated to disjunctions. A full disjunctive normalization would then produce a cross product of the disjuncts in all the late bound IUT functions. For example the query:

```
select salary(e), ssn(e) from csd_emp e;
```

produces the calculus expression:

$\{ sal, ssn \mid$
$(arg = only\_A_{nil \rightarrow only\_a}() \ \wedge \ sal = salary_{only\_A}(arg)) \vee$
$(arg = only\_B_{nil \rightarrow only\_b}() \ \wedge \ sal = salary_{only\_B}(arg)) \vee$
$(arg = A\_and\_B_{nil \rightarrow a\_and\_b}() \wedge$
$sal = salary_{a\_and\_b}(arg)) \wedge$

$(arg = only\_A_{nil \rightarrow only\_a}() \ \wedge \ ssn = ssn_{only\_A}(arg)) \vee$
$(arg = only\_B_{nil \rightarrow only\_b}() \ \wedge \ ssn = ssn_{only\_B}(arg)) \vee$
$(arg = A\_and\_B_{nil \rightarrow a\_and\_b}() \wedge$
$ssn = salary_{a\_and\_b}(arg))\}$

The expression is then normalized into 9 disjuncts, one for each combination of the disjuncts in the two disjunctive predicates above. This expression shows the first two disjuncts:

$\{ sal, ssn \mid$
$(arg = only\_A_{nil \rightarrow only\_a}() \ \wedge \ sal = salary_{only\_A}(arg) \wedge$
$arg = only\_A_{nil \rightarrow only\_a}() \ \wedge \ ssn = ssn_{only\_A}(arg)) \vee$

$(arg = only\_B_{nil \rightarrow only\_b}() \ \wedge \ sal = salary_{only\_B}(arg) \wedge$
$arg = only\_A_{nil \rightarrow only\_a}() \ \wedge \ ssn = ssn_{only\_a}(arg)) \vee$
$\ldots\}$

We can see that each disjunct contains two typecheck predicates for the variable *arg*. This will also be the case in the remaining six disjuncts not shown above. Based on the presence of more than one typecheck over the same variable in a conjunctive predicate and on the properties of the type hierarchy, the disjuncts generated by the query normalization can be rewritten into a simpler form or eliminated.

Since an object can have only one most specific type, two typecheck predicates for a single variable of two unrelated types are always rewritten to $false$, and the disjunct is removed. When the types are related, depending on whether the typechecks are deep or shallow, the result of the rewrite is either $false$ or the more specific typecheck predicate.

These rewrite rules eliminate in the example above all six disjuncts in which the typecheck is not performed over the same type (they remove the second of the two disjuncts shown above). In the remaining three it leaves just a single typecheck predicate transforming the query into the following predicate which will be shown to be significantly faster than the original query:

$\{ sal, ssn \mid$
$(arg = only\_a_{nil \rightarrow only\_a}() \wedge$
$sal = salary_{only\_a}(arg) \ \wedge ssn = ssn_{only\_a}(arg)) \vee$
$(arg = only\_b_{nil \rightarrow only\_b}() \wedge$
$sal = salary_{only\_b}(arg) \ \wedge ssn = ssn_{only\_b}(arg)) \vee$
$(arg = a\_and\_b_{nil \rightarrow a\_and\_b}() \wedge$
$sal = salary_{a\_and\_b}(arg) \ \wedge ssn = ssn_{a\_and\_b}(arg))\}$

4.3 Managing OIDs for the IUTs

The IUT instances are assigned OIDs when used in locally stored functions. For example, a query giving a bonus of $1000 to all employees in the department with salary lower than $1000 can be specified as:

```
set bonus(csde) = 1000 from CSD_emp csde
    where salary(csde) < 1000;
```

In order to manipulate the IUT OIDs we have generalized the framework developed for handling OIDs of DT instances [13] to the IUTs. As noted in the introduction, the DT functionality is modeled with three functions: OID generation function, extent function, and validation function. Next we describe how the system generates each of these functions for the IUTs.

Since an IUT is a supertype of the corresponding ATs, every AT instance is also an instance of the IUT. Each distinct real world entity is always represented by an instance in exactly one of the ATs. Therefore, the extent of an IUT is a non-overlapping union of the extents of the ATs and the extent function of an IUT is a disjunction of the extent functions of its ATs.

The OID generation function assigns an OID to a DT instance. In the case of DTs, the OID generation function is called by the extent function. Since the extent function of an IUT only references the extent functions of its ATs, there is no need for OID generation functions for IUTs. The IUT instances are thus assigned OIDs by the OID generation functions of the ATs.

If the ATs were treated as ordinary DTs, the assignment of OIDs to the AT instances would be made independently of the other ATs of an IUT. On the other hand, due to the nature of the conditions used in the ATs definition, instances 'drift' from one AT to another. For example, let's assume that John Doe is an employee of University A, and also a member of the CSD in the example above. When his bonus is assigned, the system will generate an OID for the instance representing John Doe in the AT *Only_A* and use this OID in the stored function *bonus* to relate John with his bonus. If John now gets an appointment at University B, he still belongs to the *CSD_emp* IUT, but an instance representing him appears in the type *A_and_B*, while the instance in the type *Only_A* is removed. If the newly created instance in *A_and_B* has a different OID from the old instance in *Only_A*, then John cannot be matched with his bonus stored in the database using the old OID.

The example shows that the OID assignment for instances of the ATs must be coordinated, so the instances representing the same real-world entity can move from one AT to another, while preserving their identity. An instance is related to a real world entity through its key, so to solve the problem, the OID assignments of the ATs are controlled by a function storing the generated OIDs along with the keys. When a new AT OID is to be generated, the OID generation function first checks if there is a stored OID with a matching key. If so, it adjusts the type of the stored OID and returns it as result. Otherwise, it generates a new OID. We notice here that, because the selections are pushed to the data sources and due to the OID generation removal mechanism described in [13], only a subset of the whole IUT extent is assigned OIDs in queries containing selections. Very often, queries require function values and not the OIDs of the queried types. In these cases no OIDs will be generated at all.

In Section 2 an example was presented on how the typecheck predicate of a variable can be removed from a query when the variable is used in a predicate with a locally stored function of that type. This mechanism, described in greater detail in [15], is extended to apply over the IUTs. An advantage of removing the typecheck is that the costly generation of the IUT extent is not needed, but instead only the already generated OIDs stored in the local function are used. However, when dealing with stored DT or IUT instances, we need to make sure that they are still valid, i.e. that the data sources still contain the corresponding instances.

A straightforward solution to the problem of validating an IUT instance is to test which of the three IUT ATs it belongs to. It is, however, sufficient to validate an IUT instance by testing the existence of a corresponding instance having the same key in one of the two integrated sources; the intersection AT need not be tested. This condition can be expressed by a two-branch disjunctive predicate instead of a three-branch one in the straightforward solution. The gain is due to the fact that we are not interested in exactly which AT an IUT instance belongs to, but if it belongs to *any* of the ATs. As an example we present the calculus representation of the validation function body for the *CSD_emp* type from the example above:

$$\begin{array}{l} validate_{csd\_emp}(e) \leftarrow \\ (ssn = skey_{csd\_emp \rightarrow integer}(e) \wedge \\ \ ssn = ssn_{csd\_a\_emp \rightarrow integer}(csda)) \ \vee \\ (ssn = skey_{csd\_emp \rightarrow integer}(e) \wedge \\ \ id = id_{csd\_b\_emp \rightarrow string}(csdb) \wedge \\ \ ssn = id\_to\_ssn_{string \rightarrow integer}(id)) \end{array}$$

The variables *csdb* and *csda* are local variables.

The validation method described above suffices when a query contains only locally stored functions over an IUT, while not containing late bound functions over the same IUT. When a query contains both locally stored and late bound functions, the system needs to determine which AT an IUT instance belongs to, in order to execute the right resolvent. Since an instance can drift between the ATs, the system must determine the AT membership for the IUT instances at query time. In order to do this, a disjunctive predicate similar to the one described earlier in this section is used. The only difference is that here the typecheck predicates are replaced with the corresponding validation predicates.

5 Performance Measurements

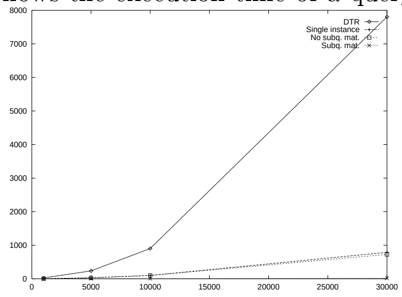
The AMOS*II* system with the mediation features described in this paper is implemented on Windows NT. We will present an overview of some experimental results obtained from running the system over 10Mb Ethernet and ISDN networks. The results demonstrate how the techniques presented above drastically reduce the response times.

The experiments are performed for a scenario similar to the running example above. We used two Compaq Professional Workstation 5000 with 200MHz Pentium processors and 64 MB memory, connected through a 10Mb Ethernet network. We also performed the same tests using a 64kb ISDN connection over the public telephone network in Sweden.

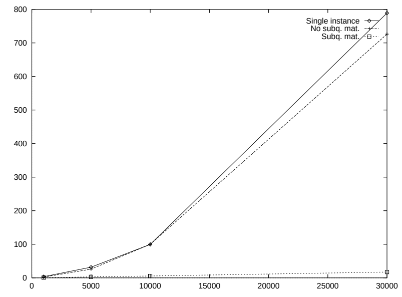
One of the workstations hosted an ODBC data source and an associated AMOS*II* system as a translator. For the experiments we used Microsoft Access as a relational data source because of its availability, but the results apply to any other ODBC data source. On the second workstation, another AMOS*II* server represented another data source. To be able to quantify the difference in the times between the processing in AMOS*II* and in the ODBC data source, the data was here stored directly in the AMOS*II*'s main-memory database. The second workstation also hosted the mediator system where the queries were issued. The three AMOS*II* servers just described will be referred to in the rest of this section as T_a (the ODBC translator), T_b (the AMOS*II* storing data locally) and the *mediator* for the AMOS*II* server where the queries are issued.

In the experiments, we scaled simultaneously the tables *Faculty* in the ODBC data source and the extent of the type *Personnel* stored in T_b from 1000 to 30000 tuples. From these tuples, 10% are selected to be members of each of the types *A_emp* and *B_emp* (i.e. members of the CSD), which are the constituent types for the integration type *CSD_emp*. Between these two types, we assume that half of the instances are overlapping (represent the same persons), meaning that the size of the extent of the type *CSD_emp* is 15% of the cardinality of the table. For example, when the size of both the *Faculty* table in the ODBC source and the extent of the type *Personnel* in T_b is 30000, there are 3000 instances of each selected as working in the CSD department by the conditions in the definition of the derived types *A_emp* and *B_emp*. From each of these two sets of 3000 instances, 1500 appear only in one of these types and 1500 appear in the both constituent types. The extent of integrated type *CSD_emp* therefore has 4500 instances.

The experiments are based on queries over the IUT *CSD_emp*. The queries are simple in order to analyze certain features of the system. Also, we have chosen queries that are the building blocks of most user-specified queries over the IUTs. More specifically the test cases can be divided into i) queries over reconciled IUT functions, and ii) queries calling locally stored functions over the IUT. In the former group we first investigate queries with no selection, exact match, and range selections. Then we present results when more than one function is used in the same query, to investigate the performance impact of the type-aware rewrites. Queries with locally stored functions are investigated in one example. We conclude the tests by comparing the times for some queries over the 10Mb network with the times obtained when the same queries were executed over an ISDN network. Notice that the y-axis in all the graphs represents response time in seconds and the x-axis represents the number of tuples in the test databases. All the measurements are performed with preoptimized queries. Figure 3 shows the execution time of a query retriev-



(a) DTR strategy



(b)Other strategies

Figure 3: Query: select salary(e) from csd_emp e;

ing the salaries of the CSD employees. We examine 4 different strategies. The graph on the left shows that the "DTR" strategy using pure late binding on an instance level is by orders of magnitude worse that the remaining three strategies. This strategy, first generates OIDs for all instances in the extent of the type CSD_emp. Then, DTR is executed over each of the OIDs, choosing the resolvent. Finally, the chosen resolvent is executed. The resolvent body also contains predicates to confirm the right AT of the argument, which causes the typecheck to be executed once again before the function value is calculated.

	Time distribution			
	Mediator	T_a	T_b	Net.
DTR	23%	69%	1%	7%
Single instance	5%	80%	3%	12%
No subq. mat.	3%	91%	3%	3%
Subq. mat.	27%	22%	32%	19%

Table 1: Query execution time distribution for the 4 evaluation strategies

Table 1 shows the percentage of the time spend in the three cooperating AMOS*II* servers, and the network time for each of the examined strategies. For the DTR strategy, the biggest portion of the query execution time is spent in T_a for accessing the relational data source. Table 2 presents the number of ODBC calls issued by the data source T_a for the different strategies. The DTR strategy issues by far the most such of calls. The number of calls is a linear function of the data sizes in the sources, but as the data volume grows, each of these calls demands more time, explaining the

	ODBC requests / DB size			
	1000	5000	10000	30000
DTR	251	1251	3001	9017
Single inst.	102	502	1002	3002
No subq. mat.	102	502	1002	3002
Subq. mat.	3	3	3	6

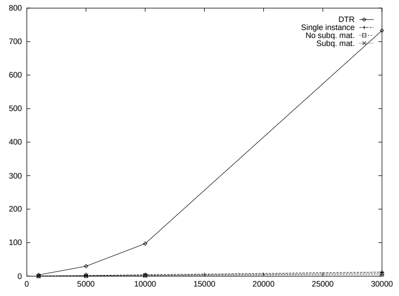
Table 2: Number of data source accesses for the 4 evaluation strategies

hyper-linear growth in the query execution time. We can also note that the DTR strategy spends 23% of the time in the mediator. This is due to OID generation, function resolution, and execution of the protocol for shipping instances among different AMOS*II* servers. The OID generation for IUT instances requires that OIDs are generated for the constituent types, which in turn triggers proxy objects generation for the instances imported from the translators. Since the DTR operator is executed over each instance individually, there is a large amount of computation involved.

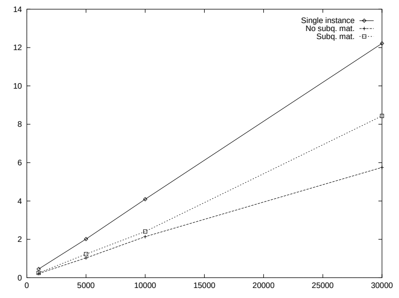
The lower part of the graph in Figure 3a is enlarged in Figure 3b. Here, we can see the remaining 3 query processing strategies. The uppermost curve represents a strategy in which the late bound function call is substituted by a disjunctive predicate, but the data shipment is still one instance at the time. This type of nested loop join over a network is named bind-join in [10]. Query rewrites eliminate OID generation, duplicate condition evaluations, and run-time function resolution. Also, the number of ODBC calls in T_a is reduced by two thirds. All of this reduces the query execution time by nearly 10 times. Nevertheless, the ODBC calls are still the main factor in the query execution cost. We can also note that the relative network cost has risen to 12%.

The first step into designing a better strategy is to pass the instances in bulks instead of an instance-at-a-time protocol. While this strategy, due to the fast networks used, does not radically improve the result (the next curve in the graph in Figure 3b), it does lower the relative network cost to 3% and makes the final query strategy possible.

The final strategy, which again reduces the response time by a couple of orders of magnitude, is based on the observations that most of the ODBC queries are issued to compute the extents of the ATs which involve anti-semi-joins translated into nested subqueries inside a *not exists* operator. In order to avoid the cost of repeated data access using parametric queries, we execute a single non-parametric query and materialize an index over all the parameter values in T_a. In this example the index contains the *ssn* for the 10% employees of University A who are also in CSD. In this way, we reduce the ODBC requests to one per bulk sent from the mediator to the translator. Being a main-memory based database, AMOS*II* facilitates a very fast index build-up for data sizes which can fit into memory of the translator. For this type of query where the materialized index is used repeatedly, this strategy is clearly advantageous. We can also see that the distribution of the query execution time in the last strategy is balanced evenly among the participating AMOS*II* servers and the network. Note that there is one access to the data source per disjunction branch of the query. Therefore, the skew in the data distribution will not affect the query execution times. The cost of executing a non-parametric query



(a) DTR strategy

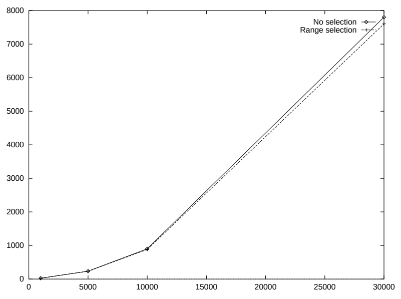


(b)Other strategies

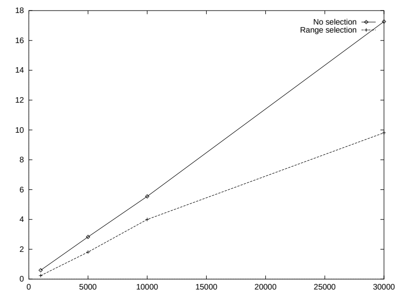
Figure 4: Query: select salary(e) from csd_emp e where ssn(e) = 1000;

and building an index on-the-fly has to be compared with the cost of completing the query without the index. In the next experiment, we executed a query containing an exact match selection using the same 4 strategies. The DTR strategy is again by far the worst, as shown in Figure 4a. On the other hand, the differences among the other strategies is not as large as in the previous experiment (Figure 4b). Also, here the strategy without index materialization for the nested subquery performs the best. This is due to the fact that the non-parameterized query used to compute the index has a larger cost than the parameterized query retrieving only the data matching a particular input tuple. In general the index materialization is favorable when: $size(input) * cost(parameterized\ query) > cost(non\ parameterized\ query) + cost(index\ generation)$.

In the next experiment we examine queries with non-equality selections, e.g. range selections. While the DTR strategy is able to apply the selections encapsulated in the DT condition, it is not efficient when the query contains non-equality conditions, since such conditions are then not pushed into the resolvents. In Figure 5a the execution times of a query containing a range selection is compared with the execution times of a query without any selection. It can be seen that the cost is about equal. In Figure 5b, on the other hand, there is clear difference between the execution times of the same queries using disjunctive predicates to model the late binding. This is due to the fact that the selection is pushed all the way down to the data sources.



(a) DTR strategies

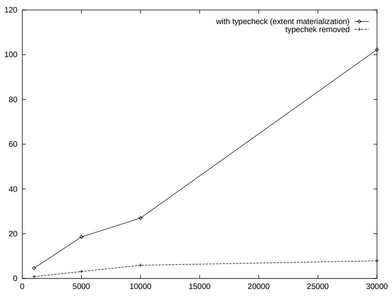


(b) disjunctive pred.with subquery materialization strategies

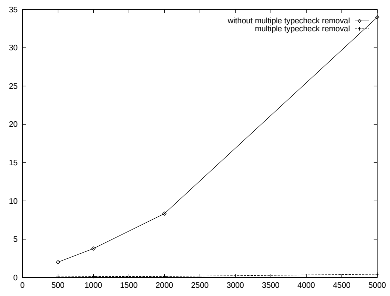
Figure 5: Selecting salary for the CSD employees with and without range selection (salary(e) > 2000)

Next, we measure the execution time for queries containing locally stored functions over an IUT. In this experiment, we created a locally stored function *office* over the type *CSD_emp* storing only 15 rows, and then executed a query to retrieve the offices stored in this function. Figure 6a compares the execution times of a naive strategy where the system generates the OIDs for the type extent and then applies the locally stored function with the strategy where the IUT instances of interest are retrieved from the locally stored function and then validated as described previously. Since the cardinality of a locally stored function is always smaller than the cardinality of the whole type extent, and the validation of an already generated OID is cheaper than a new OID generation, the validation strategy always outperform the naive strategy.

The graph in Figure 6b demonstrates the speedup obtained by typecheck removal using type-aware rewrites described in the previous section. The query is normalized to a disjunction with 9 branches, 6 of which are removed by the optimizer. The execution times on the other hand show greater than linear speedup and scalability as could be expected from the analysis of the number of the disjunctive branches. This is due to the fact that the 3 remaining branches after the query transformation are single type queries with a selection condition. The rest of the 6 queries are effectively join queries over different ATs. In these

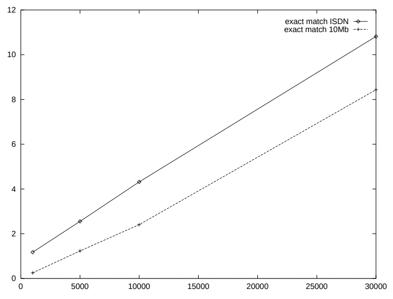


(a) select office(e) from csd_emp e;

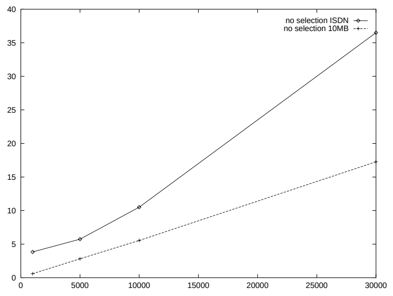


(b) select salary(e), name(e) from csd_emp e where name(e)="John";

Figure 6: a) Queries with locally materialized functions over IUTs. b) Queries calling several derived functions over IUTs.



(a) Exact match selection



(a) No selection

Figure 7: Comparison of execution times over 10Mb network with ISDN network.

cases, the AT extent functions and the extent functions of the constituent types are expanded for both the ATs appearing in the typecheck predicates. The optimizer cannot infer on the basis of these predicates that the whole disjunct will not produce any results. The resulting query execution strategy cannot therefore take advantage of the selections, and ships data proportional to the size of the extents of the constituent types. This leads to execution times with linear growth with the size of the extents, as opposed to the much slower growth of the execution time when the rewrite rule for removal of the typechecks is applied.

Finally, we briefly compare the execution times obtained over a 10Mb network with the results of the experiments using an ISDN connection over a public telephone network. Keeping all the parameters of the testing the same, the difference in the times can be attributed to the properties of the networks. The graph in Figure 7a shows that when the number of the manipulated tuples is low, the results are proportional. However, when the amount of shipped tuples increases, as with the query without selection used in Figure 7b, the execution times over ISDN rise faster than over the 10Mb network. Closer examination revealed that ISDN execution times follow the number of data bulks sent over the network. We can conclude that the unproportional increase is due to the fact that the message setup time compared to the transmission time per unit is higher in ISDN networks than it is in the 10Mb Ethernet. Probing the network to determine the bulking factor will be a topic of future investigations.

6 Related Work

The research closely related to the work presented in this paper can be divided into roughly two groups: (i) integration systems that support constructs like the IUT as [5, 4, 14] and (ii) systems that provide basic multidatabase capabilities, but do not provide IUT-like capabilities to deal with overlap and reconciliation (e.g. [10, 18]) so the integration is to be born by the user, using classical select-project-join queries. In the latter, the user needs to simulate the IUT with a series of queries, loosing all the benefits of the coherent view representation and the capacity augmenting features. A comprehensive survey of the data integration field is presented in [2]. Due to space constraints, here we only overview the main differences between our approach and two different query processing strategies used in systems that support integration of overlapping entities in the data sources. A more elaborate comparison of AMOS*II* with other systems for data integration can be found in [13].

In the literature there prevail two overall strategies for processing queries in systems that provide integration of overlapping databases. A representative example of the first group (where also AMOS*II* belongs) is the Multibase system [4]. Here, outer-join based reconciliation is broken into join and anti-semi-join operators. However, there is neither a concept of OIDs, nor capacity augmentation in this system, yielding some of the techniques presented here not applicable. Also, the proposed query rewrites are based on analysis of the query attributes which is more complex than the simple type-based rewrite proposed in this work.

The strategies of the second group are exemplified by the strategy used in the Pegasus [5] system where the reconciliation is performed by a reconciliation operators. These are pushed "upwards" the query tree by the query heuristics, in order to be able to push down joins over small tables through the expensive outerjoin operation. This approach has an advantage of more compact queries in comparison with AMOS*II*. Some disadvantages are that selections based on the reconciled functions are not pushed to the data sources for the anti-semi-join. Also, the whole outer-join needs to be materialized in the mediator before application of the reconciliation operator, preventing streamed execution strategies as used in AMOS*II*. In the experiments above, we can note that the execution times of the queries with selections are about one third of the times without selection, corresponding to the portion of the integrated extents that overlap.

Late binding has been used for data integration in [7], but that system uses instance-level evaluation and no reconciliation facilities.

None of the compared systems provided an extended experimental assessment of the used strategies as the one provided in this work.

7 Summary

We presented a novel framework for data integration based on OO type hierarchies and late binding. Integration union types (IUTs) were introduced to model a coherent view of heterogeneous data in multiple repositories. IUTs allows for resolutions of conflicts in the metadata (e.g. naming, scaling etc.) and for dealing with overlaps in the extents of the integrated types. Furthermore, instances of the IUTs can be assigned OIDs used in locally stored and derived functions.

Each IUT is mapped by the system to a hierarchy of system generated derived types, called auxiliary types (ATs). The ATs represent disjoin parts (a join and two anti-semi-joins) of the outerjoin needed for the data integration. The reconciliation of the attributes of the integrated types is modeled by a system generated set of overloaded derived functions, The implementation of each function is inferred from the *CASE* clause in the IUT definition.

Several novel query processing and optimization technique were developed for efficiently processing queries containing overloaded functions over the system-generated OO views. Queries over such a type hierarchy contain late bound calls. The late bound calls are translated to disjunctive calculus expression which are suitable for application of techniques such as: bulk-oriented processing, type-aware query rewriting, selective OID generation, and dynamic generation of indexes for nested subqueries. The reported measurements compare the impacts of different query processing strategies showing that the combination of these techniques drastically lower the execution times, in some cases by several orders of magnitude.

Our current work includes methods to easily handle non-relational data sources and parallel execution strategies for integration of large number of sources.

References

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

[2] O. Bukhres, A. Elmagarmid (eds.): *Object-Oriented Multidatabase Systems*, Pretince Hall, 1996.

[3] M. Garcia-Solaco, F. Saltor, M. Castellanos: Semantic Heterogeneity in Multidatabase Systems, In [2].

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

[5] W. Du and M. Shan: Query Processing in Pegasus, *Object-Oriented Multidatabase Systems*, In [2].

[6] G. Fahl, T. Risch: Query Processing over Object Views of Relational Data. *VLDB Journal*, Nov. 1997.

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

[8] S. Flodin, T. Risch: Processing Object-Oriented Queries with Invertible Late Bound Functions, *VLDB95*, Zürich, Switzerland, 1995

[9] 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)* 8(2), 117-132, 1997

[10] L. Haas, D. Kossmann, E. Wimmers, J. Yang: Optimizing Queries accross Diverse Data Sources. *VLDB97*, pp. 276-285, Athens Greece, 1997

[11] S. Flodin, V. Josifovski, T. Risch, M. Sköld and M. Werner: *AMOS*II *User's Guide*, available at *http://www.ida.liu.se/labs/edslab*.

[12] V. Josifovski: *Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration*, Ph D. Thesis 582, Linköpings universitet, 1999. Available at *http://www.ida.liu.se/labs/edslab*.

[13] V.Josifovski, T.Risch: Functional Query Optimization over Object-Oriented Views for Data Integration To appear in *Journal of Intelligent Information Systems (JIIS)*, Vol. 12, No. 2-3, 1999.

[14] 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*, 25(5), 553-562, 1995.

[15] W. Litwin, T. Risch: Main Memory Oriented Optimization of OO Queries using Typed Datalog with Foreign Predicates. *ACM TKDE*, 4(6), pp. 517-528, 1992

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

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

[18] A. Tomasic, L. Raschid, P. Valduriez: Scaling Access to Heterogeneous Data Sources with DISCO. *ACM TKDE*, 10(5), 808-823, 1998

[19] G Wiederhold: Mediators in the Architecture of Future Information Systems, *IEEE Computer*, 1992.