Global Query Processing in the AutoMed Heterogeneous Database Environment

Edgar Jasper
School of Computer Science and Information Systems
Birkbeck College, University of London.
edgar@dcs.bbk.ac.uk

April 22, 2002

1 Motivation

One of the requirements for implementing a heterogeneous database system is *global query processing*. In outline, the functionality of the global query processor is as follows. The global query processor takes as input a query expressed on the constructs of a global schema of the heterogeneous database system. This query must be translated so that it is expressed over the constructs of the local database schemata. Appropriate local sub-queries are then sent to the local data sources for processing there. The global query processor must then take the results it receives from the local data sources and perform any necessary post-processing to obtain the result of the originally submitted global query.

In a heterogeneous database system the local databases may employ different data models and use different query languages from each other and from the global schema(s). Thus the global query processor will, in all likelihood, have to perform translations between different query languages and between queries expressed on different data models in addition to the translation from global schema constructs to local schema constructs. Furthermore, to ensure satisfactory performance, the global query processor will also need to perform query optimisation possibly at several points in the process.

These are some of the sub-goals involved in global query processing. My research is concerned with how to achieve them within the AutoMed Heterogeneous Database Environment.

2 Related Work

My research is part of the EPSRC-funded AutoMed project at Birkbeck College and Imperial College (see http://www.doc.ic.ac.uk/automed). The aim of this project is to implement and evaluate the framework for schema integration and query processing developed by P. McBrien and A. Poulovassilis (see [13, 7, 8, 9]). In this framework, a common data model called the hypergraph data model (HDM) is used. The HDM is a low level model where schemata consist of nodes, edges and constraints. The edges are directed edges that can link multiple nodes and other edges. A schema expressed in another data model (eg. relational, ER, OO, XML) may be regarded as a high level representation of an underlying HDM schema. The advantage of such a low level common data model is that it simplifies the mapping process between the constructs of different high level data models and the common data model. There is not usually a simple correspondence between the constructs of different high level models. Thus specifying one high level modelling language in terms of another, which using a high level common data model requires, is complicated. This problem is avoided by using a low level common data model such as the HDM [8].

It is possible to translate a high level schema into an HDM schema (see [8]). Given knowledge of how a given data model's constructs are expressed in the HDM this schema translation can be done automatically.

In this framework, component HDM schemata are integrated into a global HDM schema by means of applying a list of primitive schema transformations to each component schema that transforms it into the global schema. The primitive transformations add, delete and rename nodes, edges and constraints. These transformations are automatically reversible (see [7]). This is significant for global query processing because it means that transformations used to transform component schemata into a global schema can be used to translate queries expressed on the global schema into queries expressed on the component schemata.

In AutoMed, queries on HDM schemata are written in a functional Intermediate Query Language (IQL) [12]. Thus a global query processor working in this framework will need to translate high level query languages (such as SQL, OQL) into and out of IQL.

3 Preliminary Ideas and Results So Far

3.1 Architecture of the Global Query Processor

Figure 1 illustrates the overall architecture of the Global Query Processor. The Global Query Processor consists of one 'Global Translator', one 'Global Evaluator' and several Wrappers. The functionality of these modules is as follows.

The Global Translator takes a query on the global schema expressed in some high level query language (HLQL) using some high level data model (HLDM). It performs necessary translations and optimisations to produce an equivalent IQL query with minimal newly introduced redundancy. It sends to the Global Evaluator an IQL query expressed on the constructs of the HDM representation of the global schema. From the Global Evaluator it receives an HDM result set that answers this query. It then converts this into an appropriate form for the data model of the high level global schema and returns this as its answer to the original global query.

The Global Evaluator takes as input an IQL query expressed on the HDM representation of the global schema. It performs transformations on this query and then sends to each wrapper an IQL sub-query expressed using only constructs from the component schema corresponding to the wrapper. From each wrapper the global evaluator receives an HDM result set. The Global Evaluator then processes these HDM result sets and returns results that answer the query it was sent.

A wrapper takes as input an IQL query expressed using constructs for the corresponding component schema. It translates this query into the high level query language and high level data model appropriate for the corresponding local data source (eg. a relational SQL query or an object-oriented OQL query). This query is then sent to the local data source for evaluation. The results obtained from the local data source must then be converted to an HDM result before being returned to the Global Evaluator.

3.2 The Translators

The global translator receives a query written in a high level query language (HLQL) such as SQL employing a high level data model (HLDM) such as the relational model. This HLQL HLDM query is translated into an IQL HDM query in two steps. It is first translated into an IQL HLDM query. Then this IQL HLDM query is translated into an IQL HDM query.

This order is chosen because a data model cannot depend on the use of a particular query language but it is possible that a query language could be specific to a particular data model. Therefore it is possible that an HLQL HDM query may be nonsensical. The order chosen avoids the existence of an HLQL HDM query at any point. It is expected that IQL is sufficiently general and extensible to work on any data model and cover the expressivity of any query language that may be required in practice.

The IQL HDM query is then sent to the global evaluator which returns an HDM result. This HDM result must be converted to a HLDM result before it is returned to the user.

A wrapper takes an IQL HDM query. It first translates this into an IQL HLDM query appropriate for the data source. This is then translated into an HLQL HLDM query that is sent to the local data source. This order is chosen for the same reasons as given above. The HLDM results that the local data source returns must be converted into HDM results before they are returned to the global evaluator.

Thus, the wrappers and the global translator are in essence mirror images of each other. They both receive a query, translate it, send it out for evaluation and then change the form of the results before returning them but they do it in opposite directions.

3.3 The Global Evaluator

The global evaluator takes as input an IQL HDM query. This query is expressed using the constructs of the HDM representation of the global schema. It needs to be rewritten to be expressed using the constructs of the component schemata. This is done using the transformation pathways between the component schemata and the global schema.

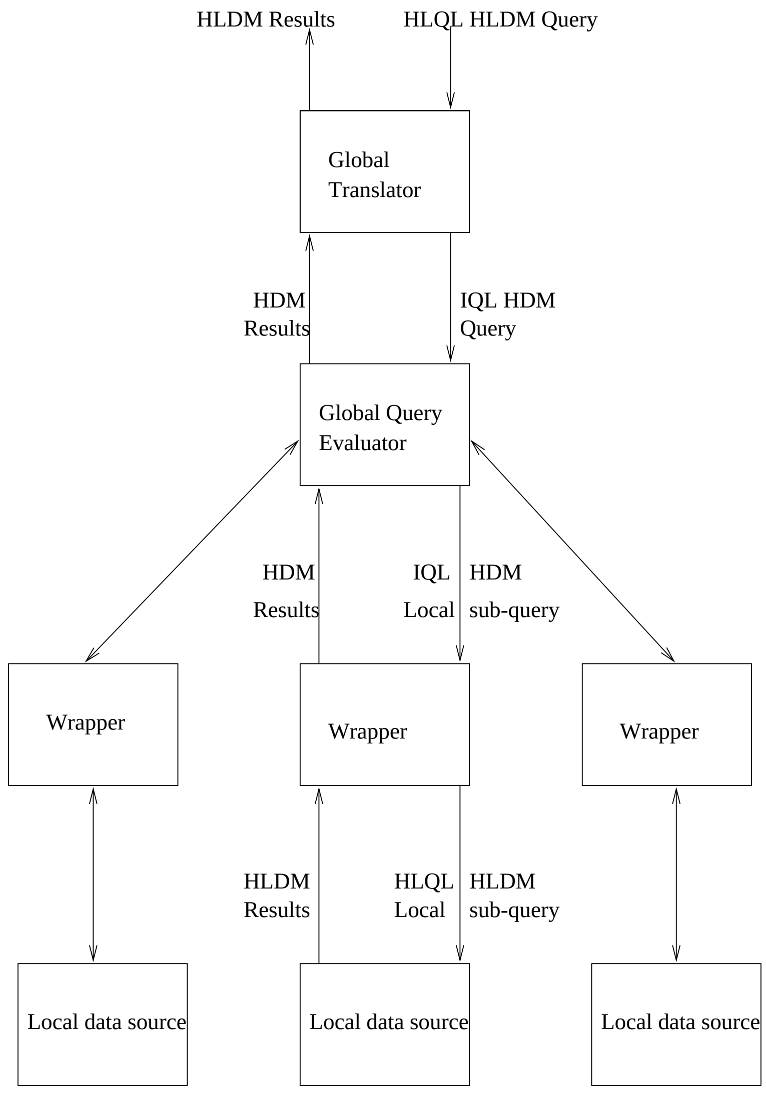

Figure 1: Architecture of the Global Query Processor

In the AutoMed system, these transformations will be retrieved from a repository - the Schemas and Transformations Repository or STR (see [1]).

Given the transformation pathways from the global schema to each component schema the input query can be rewritten. In particular, from these transformation pathways a view definition expressed in component schema constructs can be derived for each global schema construct for each component schema. The process is as follows. Initially, we take the view definition of each global construct to simply be the identifier of that construct. Then, each primitive transformation in the reverse transformation pathway from the global schema to the component schema is examined in turn. If this is a delete transformation (of either a node or an edge) this means that the construct has been removed from the global schema and therefore will not be present in the component schema. Such a delete transformation will have as one of its arguments a query enabling the extent of the deleted node or edge to be reconstructed from the extents of constructs that still, at this point, exist. Thus, the query is a view definition of the deleted construct. If the deleted construct is referenced in any of the view definitions for global schema constructs then these references must be replaced with the view definition of the deleted constructs. So a set of view definitions for each global schema construct is derived for each component schema.

Now these view definitions are composed together and substituted for the global construct they represent in the input query. Thus the input query is rewritten to be expressed in terms of the component schema constructs.

Next, local sub-queries must be identified in the query. These are sub-queries that require only the constructs of one component schema. These sub-queries are then dispatched to the appropriate component translator and the results substituted into the query.

The result is a query consisting of data and function applications which will be evaluated and then returned by the global evaluator.

3.4 Query Optimisation

Query optimisation is of course essential for global query processing in heterogeneous database environments. In our approach this can take place at several points. In the global translator after the query has been translated there may be redundancy that can be eliminated (see the example given later).

There may be similar redundancy and general non-optimal structure in a wrapper after the query it receives has been translated. It may be possible not to optimise here as the local data source might do all the necessary optimisation itself.

In the global evaluator there will be some general optimisation work to be done - as in any DBMS query processor. More specifically, the IQL HDM query on component schema constructs should be rewritten prior to the identification of local sub-queries, in order to bring constructs from the same component schema together and make the local sub-queries as 'large' as possible. This should minimise the number of times any individual data source is queried. It should also reduce the volume of data that must be moved from local data sources to the global query processor. Also, this will maximise the amount of optimisation work that can potentially be passed down to the local data sources.

For the above query optimisation functionality, we expect to draw on previous techniques for optimising functional query languages [18, 14, 15, 4, 6].

4 An Example

The following example illustrates in outline how a query is processed by the global query processor from top to bottom.

Consider the following relational schemata where S_1 and S_2 are two local schemata and S_G the global schema:

S_1 pm : $\underline{\text{pname}}$, party, ybirth, ydeath, fullname
monarch : $\underline{\text{mname}}$, syear, eyear
stint : $\underline{\text{pname, syear, eyear}}$

S_2 reign : $\underline{\text{syear, eyear}}$, mname
gov : $\underline{\text{syear, eyear}}$, pname, party, fullname

S_G pm : $\underline{\text{pname}}$, party, ybirth, ydeath, fullname
term : $\underline{\text{pname, syear, eyear}}$
served : $\underline{\text{pname, mname}}$

Such relational schemata have equivalent HDM representations. In particular, each relation is represented by a node the extent of which is the extent of the primary key attributes of the relation (as a tuple). Each non-key attribute is also represented by a node which is linked to the relation node by an edge. The extents of these nodes and edges are derived from the extent of the original relation as projections. For example the HDM representation of the relation 'pm' in schema S_1 is as follows:

⟨⟨pm:KEY⟩⟩
⟨⟨pm:party⟩⟩
⟨⟨pm:ybirth⟩⟩
⟨⟨pm:ydeath⟩⟩
⟨⟨pm:fullname⟩⟩
⟨⟨_,⟨⟨pm:KEY⟩⟩,⟨⟨pm:party⟩⟩⟩⟩
⟨⟨_,⟨⟨pm:KEY⟩⟩,⟨⟨pm:ybirth⟩⟩⟩⟩
⟨⟨_,⟨⟨pm:KEY⟩⟩,⟨⟨pm:ydeath⟩⟩⟩⟩
⟨⟨_,⟨⟨pm:KEY⟩⟩,⟨⟨pm:fullname⟩⟩⟩⟩

The query translation algorithm makes use of information from the transformation pathways between the local schemata and the global schema. We give here the transformation pathway between the HDM representation of S_1 and the HDM representation of S_G:

1. *renameNode* ⟨⟨stint:KEY⟩⟩ term:KEY
2. *addNode* ⟨⟨served:KEY⟩⟩ [(p,m) | (m,ms)←_monarch:KEY_monarch:syear; (m2,me)←_monarch:KEY_monarch:eyear; m=m2; (p,ps,pe)←term:KEY; (if (pe<=me) pe me) > (if (ps>=ms) ps ms)]
3. *contractEdge* ⟨⟨_,⟨⟨monarch:KEY⟩⟩,⟨⟨monarch:syear⟩⟩⟩⟩
4. *contractNode* ⟨⟨monarch:syear⟩⟩
5. *contractEdge* ⟨⟨_,⟨⟨monarch:KEY⟩⟩,⟨⟨monarch:eyear⟩⟩⟩⟩
6. *contractNode* ⟨⟨monarch:eyear⟩⟩
7. *contractNode* ⟨⟨monarch:KEY⟩⟩

4.1 Translating an Example Query

Take the following SQL query expressed on the global schema S_G:

```
select syear, eyear, party
from pm, term
where pm.pname = term.pname
```

This query would be translated to the following IQL relational query by the global translator:

[(syear,eyear,party) |
(t:pname,syear,eyear)←term;
(p:pname, party, x1, x2, x3)←pm;
term:pname=pm:pname]

This in turn would then be translated into this IQL HDM query:

[(syear, eyear, party) |
(t:pname,syear,eyear)←term:KEY;
p:pname←pm:KEY;
(pn1,party)←_pm:KEY_pm:party;
p:pname=pn1;
(pn2,x1)←_pm:KEY_pn:ybirth;
p:pname=pn2;
(pn3,x2)←_pm:KEY_pn:ydeath;
p:pname=pn3;
(pn4,x3)←_pm:KEY_pn:yfullname;
p:pname=pn4;
t:pname=p:pname]

The generators involving relations in the IQL relational query are translated directly to their equivalent HDM based generators and comparisons without taking account that many of these constructs are not needed to evaluate the query. The above query can now be optimised to remove this redundancy, giving:

[(syear, eyear, party) |
(t:pname,syear,eyear)←term:KEY;
(p:pname,party)←_pm:KEY_pm:party;
t:pname=p:pname]

Next the global translator follows the transformation pathways to derive view definitions for the global constructs in the query. These view definitions are composed together and substituted into the query, giving:

[(syear,eyear,party) |
(t:pname,syear,eyear)←stint:KEY1 ++
[(p,s,e)|(s,e),p)←_gov:KEY_gov:pname2];
(p:pname,party)←_pm:KEY_pm:party1 ++
[(pn,pty)|(Ka,pn)←_gov:KEY_gov:pname2;
(Kb,pn)←_gov:KEY_gov:party2; Ka=Kb];
t:pname=p:pname]

Note that construct identifiers have a numeric suffix in this query. This number disambiguates which local schema, S_1 or S_2, the local construct is part of.

Now the global evaluator will rewrite this query to maximise the size of the local subqueries. For instance, the above query involves in effect a join that operates on two sets produced by unions. It is usually a good idea to distribute joins though unions in cases like these. Doing this would result in the following query:

[(syear,eyear,party) |
(t:pname,syear,eyear)←stint:KEY1;
(p:pname,party)←_pm:KEY_pm:party1;

```
   t:pname=p:pname ] ++
[(syear,eyear,party) |
   (t:pname,syear,eyear)←stint:KEY1;
   (p:pname,party)←
      [(pn,pty)|(Ka,pn)←_gov:KEY_gov:pname2;
         (Kb,pn)←_gov:KEY_gov:party2; Ka=Kb];
   t:pname=p:pname ] ++
[(syear,eyear,party) |
   (t:pname,syear,eyear)←
      [(p,s,e)|((s,e),p)←_gov:KEY_gov:pname2];
   (p:pname,party)←_pm:KEY_pm:party1;
   t:pname=p:pname ] ++
[(syear,eyear,party) |
   (t:pname,syear,eyear)←
      [(p,s,e)|((s,e),p)←_gov:KEY_gov:pname2];
   (p:pname,party)←
      [(pn,pty)|(Ka,pn)←_gov:KEY_gov:pname2;
         (Kb,pn)←_gov:KEY_gov:party2; Ka=Kb];
   t:pname=p:pname ]
```

The global evaluator will next identify local sub-queries and send them to the local data source wrappers. Consider the following local sub-query:

```
[(pn,pty) | (Ka,pn)←_gov:KEY_gov:pname2;
   (Kb,pn)←_gov:KEY_gov:party2;
   Ka=Kb ]
```

This query will be sent to a wrapper associated with S_2. This is a relational schema so this query must be translated, by the wrapper, into an SQL relational query. The wrapper will first translate the above IQL HDM query into an IQL relational query then into an SQL relational query. This will look something like the following:

```
select pn, pty
from
  (select syear as s1,
    eyear as e1,
    pname as pn
  from gov) as T1,
  (select syear as s2,
    eyear as e2,
    party as pty
  from gov) as T2
where (s1,e1)=(s2,e2)
```

This query will then be sent to the local data source for evaluation. The above query is clearly far from optimal but remaining optimisation can possibly be left to the local data source evaluator.

5 Progress to Date and Future Work

5.1 Implementation of Global Evaluator

The global evaluator has been partially implemented. So far (6 months into my PhD) I have produced software that, given an IQL HDM query expressed on global schema constructs and given some transformation pathways, can rewrite this query to be expressed on component schema constructs. The software can then send local sub-queries (consisting only of individual component schema identifiers) to HDM data sources and substitute the results into the query. The resulting query - an IQL HDM query containing only data and function applications - can then be evaluated and the data that is the result returned.

IQL evaluation with a number of predefined functions has been implemented. IQL is, as has been said, a functional language. Text IQL can be parsed into an Abstract Syntax Tree (AST) representation. The leaves of an IQL AST may be constant values (such as strings, numbers, and booleans), variable names, built-in function names (such as group, sort and so on) and constructors. The non-leaf nodes are either apply nodes or lambda nodes. If the root of a sub-tree is an apply node this indicates that to evaluate this sub-tree the left child should be applied to the right child. For this to be possible the left child must itself evaluate to a function or a constructor and the right child must evaluate to an appropriate argument for that function or constructor. If the root of a sub-tree is a lambda node this indicates that this sub-tree represents a lambda abstraction. Thus, new non-built-in functions can be defined using lambda abstractions. As will be seen, lambda abstractions are also used internally when evaluating comprehensions.

Certain elements are particularly important for understanding IQL evaluation. The constructors 'Cons' and 'Nil' enable IQL lists to be representing in the AST. 'Cons' is a constructor that defines a list. It has two arguments, a list head (the first element in the list) and a list tail (a list of all the other elements). The 'Nil' constructor defines an empty list. The built-in function flatmap is also important. It takes two arguments, a function and a list, and is defined recursively as follows:

flatmap f [ ] = [ ]

flatmap f (Cons x xs) = (f x) ++ (flatmap f xs)

An element of IQL that is very helpful for representing queries is the comprehension construct (see [17, 12]) consisting of a head and a list of generators and boolean valued queries. These are simplified by the global evaluator as follows before being evaluated:

[e | p←s; Q]⟹flatmap (lambda p.[e|Q]) s
[e | e'; Q]⟹if e' [e|Q] [ ]
[e |]⟹[e]

Note that comprehensions are only reduced to flatmaps and lambda abstractions as above when the global evaluator comes to evaluate the query after the wrappers have returned their results.

These IQL structures, comprehensions, list constructors, lambda abstractions and the flatmap function have been implemented in the global evaluator. Various useful built-in functions (group, group and compute, sort, remove duplicates, arithmetic, comparison and logical functions etc.) have also been implemented as have constructors for constructing tuples. The global evaluator as implemented is capable of evaluating IQL queries, in AST form, involving such structures and functions.

5.2 Work on the Translators

I have done some research on the global translator and wrappers that I have not yet implemented. How to transform between the HDM and various HLDMs had already been worked out by others (see [8]). I have been working on how to translate SQL to IQL and vice versa and have made some progress here. The example shown earlier uses this and aims to show how the whole process will proceed. I have also done some work on translating OQL to IQL.

5.3 Next Steps

The next step is to implement the global translators and wrappers to translate SQL/OQL relational queries to IQL HDM queries and vice versa. This software will be specific to SQL (or OQL). Its principal purpose will be to produce a complete working query processor. This will demonstrate that the AutoMed approach is fundamentally sound. It will also increase my understanding of translation between query languages in general.

After this I will consider translation between HLQLs and IQL in a general sense. Ultimately, it is desirable that there be some means of defining arbitrary HLQLs for use within the AutoMed architecture and that the translation software is capable of using this information to provide automatic translation between IQL and such an HLQL. (It is possible to define arbitrary data models within the AutoMed architecture so it seems natural that is should also be possible to define arbitrary high level query languages employing these data models.) Meta-translation techniques [2, 16] are an approach we will investigate for this.

Query optimisation will be a recurring theme throughout future work on the PhD. In the short term, rewrite rules for maximising the size of local sub-queries sent from the global evaluator to the wrappers and implementing them as software is the priority. In the longer term, further optimisation based on information about the local data sources themselves may well be possible [11, 5, 3]. This will be useful when, for example, overlapping information between local data sources means there is more than one data source the global evaluator could query for the same information.

6 Contributions of the Research

The AutoMed approach to heterogeneous database systems is a novel one and so in turn my work on global query processing inevitably approaches the problem from a novel standpoint. In the AutoMed approach a low level data model - the hypergraph data model - is used as the common data model. Conventionally, a high level data model - such as the ER model - would be used. As discussed earlier, this requires other high level modelling languages be specified in terms of the high level common data model which is complicated because there is not often a simple correspondence between the constructs they use. Furthermore, the AutoMed schema integration approach is based on schema transformations rather than view definitions. The global schema is defined using a list of schema transformations from the local schemata or alternatively one may take the global schema as given and regard any individual local schema as being defined by a list of transformations from the global schema. The transformations are automatically reversible so both approaches are possible. This differs from the conventional approach where either the global schema constructs are defined as views over the local schemata (global-as-view) or where the local schema constructs are defined as views over the global schema (local as view). While some advantages of using the AutoMed automatically reversible schema transformation approach over view definition approaches are documented (see [10]) it is an open question as to what, if any, will be the benefits arising from a global query processing point of view. My PhD will aim to answer this question.

References

[1] M. Boyd and N. Tong. The automed repositories and api. Technical report, Imperial College, Dept. of Computer Science, March 2002.

[2] B. Czejdo and L. Gruenwald. *Schema and Language Translation*, pages 157–174. In Elmagarmid et al. [3], 1991.

[3] A. Elmagarmid, M. Rusinkiewicz, and A. Sheth, editors. *Management of Heterogeneous and Autonomous Database Systems.* Morgan Kaufmann, San Francisco, California, 1991.

[4] L. Fegaras. Query unnesting in object-oriented databases. In *Proc. ACM SIGMOD'98*, pages 49–60, 1998.

[5] Jean-Robert Gruser, Louiqa Raschid, Vladimir Zadorozhny, and Tao Zhan. Learning response time for websources using query feedback and application in query optimization. *VLDB Journal*, 9(1):18–37, 2000.

[6] T. Grust and M. H. Scholl. How to comprehend queries functionally. *JIIS*, 12(2–3):191–218, 1999.

[7] P.J. McBrien and A. Poulovassilis. Automatic migration and wrapping of database applications — a schema transformation approach. In *Proceedings of ER99*, volume 1728 of *LNCS*, pages 96–113. Springer-Verlag, 1999.

[8] P.J. McBrien and A. Poulovassilis. A uniform approach to inter-model transformations. In *Advanced Information Systems Engineering, 11th International Conference CAiSE'99*, volume 1626 of *LNCS*, pages 333–348. Springer-Verlag, 1999.

[9] P.J. McBrien and A. Poulovassilis. A semantic approach to integrating xml and structured data sources. In *Advanced Information Systems Engineering, 13th International Conference CAiSE2001*, volume 2068 of *LNCS*, pages 330–345. Springer-Verlag, 2001.

[10] P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. Technical report, Imperial College and Birkbeck College, February 2002.

[11] George A. Mihaila, Louiqa Raschid, and Maria-Esther Vidal. Using quality of data metadata for source selection and ranking. In *WebDB (Informal Proceedings) 2000*, pages 93–98, 2000.

[12] A. Poulovassilis. The automed intermediate query language. Technical report, Birkbeck College, Dept. of Computer Science, June 2001.

[13] A. Poulovassilis and P.J. McBrien. A general formal framework for schema transformation. *Data and Knowledge Engineering*, 28(1):47–71, 1998.

[14] A. Poulovassilis and C. Small. Algebraic query optimisation for database programming languages. *The VLDB Journal*, 5(2):119–132, 1996.

[15] A. Poulovassilis and C. Small. Formal foundations for optimising aggregation functions in database programming languages. In *Proc. DBPL'97*, volume 1369 of *LNCS*, pages 299–318. Springer-Verlag, 1997.

[16] A. Ramfos, N. J. Fiddian, and W. A. Gray. A meta-translation system for object-oriented to relational schema translations. In *BNCOD 1991*, pages 245–268, 1991.

[17] P. W. Trinder. Comprehensions, a query notations for DBPLs. In *Proc. DBPL'91*, pages 55–68, 1991.

[18] Limsoon Wong. Normal forms and conservative properties for query languages over collection types. In *Proc. ACM PODS'93*, pages 26–36, 1993.