P2P query reformulation over Both-as-View data transformation rules

Peter McBrien[1] and Alexandra Poulovassilis[2]

[1] Dept. of Computing, Imperial College, Univ. of London, pjm@doc.ic.ac.uk
[2] School of Computer Science and Information Systems, Birkbeck College, Univ. of London, ap@dcs.bbk.ac.uk

Abstract. The both-as-view (BAV) approach to data integration has the advantage of specifying mappings between schemas in a bidirectional manner, so that once a BAV mapping has been established between two schemas, queries may be exchanged in either direction between the schemas. In this paper we discuss the reformulation of queries over BAV transformation pathways, and demonstrate the use of this reformulation in two modes of query processing. In the first mode, public schemas are shared between peers and queries posed on the public schema can be reformulated into queries over any data sources that have been mapped to the public schema. In the second, queries are posed on the schema of a data source, and are reformulated into queries on another data source via any public schema to which both data sources have been mapped.

1 Introduction

In [17] we presented the **both-as-view** (**BAV**) approach to data integration, and compared it with **global-as-view** (**GAV**) and **local-as-view** (**LAV**) [13]. In BAV, schemas are mapped to each other using a sequence of schema transformations which we term a transformation **pathway**. These pathways are reversible, in that a pathway $S_x \to S_y$ from a schema S_x to a schema S_y may be used to automatically derive the pathway $S_y \to S_x$. Also, from BAV pathways it is possible to extract GAV, LAV and GLAV mapping rules [12]. The BAV approach has been implemented as part of the AutoMed data integration system (see http://www.doc.ic.ac.uk/automed).

One advantage of BAV is that it readily supports the evolution of global and local schemas, including the addition or removal of local schemas. An evolution of a schema S_x to S'_x is expressed as a pathway $S_x \to S'_x$, and then pathways of the form $S_x \to S_y$ may be 'redirected' to S'_x by prefixing the reverse of $S_x \to S'_x$ to derive a pathway $S'_x \to S_x \to S_y$. As we discussed in [18], this feature makes BAV well-suited to the needs of **peer-to-peer** (**P2P**) data integration, where peers may join or leave the network at any time, or may change their schemas or pathways between schemas.

Figure 1 illustrates via an example how centralised and P2P data integration differ in BAV. In Figure 1(a), standard centralised data integration of data sources $S_1, S_2, \ldots$ into a global schema S_p is specified by a set of pathways $S_1 \to S_p$, $S_2 \to S_p, \ldots$ managed centrally by the data integration system (some of the transformations used to specify the pathways A, B, C will be listed later in the paper). In P2P data integration, each peer P_x manages the integration of a data source S_x as a pathway $S_x \to S_p$, and there

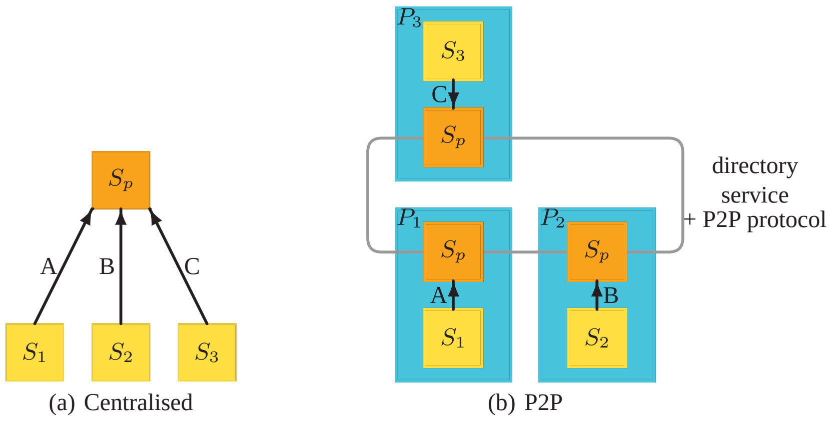

Fig. 1. Example of centralised versus P2P Data Integration in BAV

is a directory service and P2P protocol that allows the peers to interact[1]. The shared global schema is called a **public schema**, emphasising that no single peer controls the global schema but, by contrast, it is simply a publicly available schema definition that any peers may use. Note that the *same* BAV pathway specification is used to map $S_x \rightarrow S_p$ in both the centralised and the P2P systems. The directory service allows a peer to discover what public schemas S_p exist, and which peers support pathways to that public schema [1].

One contribution of this paper is that we specify how, given a pathway $S_x \rightarrow S_y$ and a query q posed on S_y, q can be reformulated using a combination of LAV and GAV techniques into a query q' posed on S_x. This is an advance on our previous work which only showed how GAV or LAV views individually could be derived from BAV pathways (we do not consider in this paper reformulation using in addition the GLAV rules that could be extracted from the BAV pathway, and leave that as an area of future work). A second contribution of this paper is that the P2P protocol combined with the reversibility of BAV pathways allows us to support two types of query processing:

– In **public schema querying** we simulate centralised data integration within a P2P environment: a user at a peer P_x poses a query on a public schema S_p, and P_x asks each other peer P_y supporting S_p to either (1) process the query and return the result back to P_x, or (2) send its pathway to S_p to P_x so that P_x can construct the centralised data integration model and process the query itself.
– In **data source querying** a user at a peer P_x poses a query q on data source S_x and wishes it to be reformulated into a query q' on some other data source S_y. This is achieved by using the pathway $S_x \rightarrow S_p$ to reformulate q into a query on S_p. Then P_x is able to interact with other peers supporting the public schema S_p, using the public schema querying techniques already described.

[1] For simplicity of presentation in this paper, we assume that each data source is accessed via one peer, and each peer accesses only one data source. In fact, our approach allows a many-many relationship between data sources and peers.

Previous work on P2P data integration in the Piazza system has used combinations of LAV and GAV rules between schemas, and a combination of GAV and LAV query processing techniques [10, 9]. Piazza differs from our approach in that mappings must be specified directly between peers. Whilst our approach does not preclude this, we also allow mappings to be specified to a public schema, making our approach more scalable.

Other related work is [19, 15] which uses a superpeer based network topology to provide better scalability than pure peer-to-peer networks. Routing indexes at superpeers store information about the data reachable from the peers directly connected to them, and aid in the forwarding of query requests only to relevant peers.

The need for a superpeer is avoided in the local relational model [2], where peers are directly related by a combination of a domain relation that specifies how the data types of the peers are related, together with coordination formulae that specify that if one predicate is true in one peer, then another predicate is true in another peer.

Our approach combines the respective advantages of these systems by having virtual public schemas — allowing peers to reuse the existing integration of other peers with public schemas — but having no physical superpeer nodes that may act as a bottleneck in the system — in particular, any peer can combine the integrations of other peers with public schemas in order to form direct pathways between peers for query and update processing.

In [4] global-local-as-view (GLAV) rules [8, 16] are used to specify the constructs of each schema in terms of the constructs of some set of other peer schemas. There is no distinction between source and global schemas, and any number of GLAV rules may be specified between schemas. However, unlike BAV, [4] does not differentiate between sound, complete and exact rules, as the GLAV rules are always sound. CoDB [7] generalises this to allow sound and complete GLAV rules to be specified.

The remainder of the paper begins with a review of the BAV data integration approach in Section 2 together with details of the data integration example sketched in Figure 1. We then describe in Section 3 the process of query reformulation over BAV pathways, and illustrate how it supports public schema querying. In Section 4 we discuss how to improve support for data source schema querying, where a certain degree of pathway repair may be needed in order to fully support data source schema querying.

2 Overview of BAV data integration

The basis of the BAV approach to data integration is a low-level **hypergraph-based data model (HDM)**. Higher-level modelling languages are specified in terms of this lower-level HDM. An HDM schema consists of a set of nodes, edges and constraints, and each modelling construct of a higher-level modelling language is specified as some combination of HDM nodes, edges and constraints. For each type of modelling construct of a modelling language (*e.g.* Table, Column, Primary Key and Foreign Key in the relational model) there are available a set of primitive schema transformations for adding such a construct to a schema, removing such a construct from a schema and, in the case of constructs with textual names, renaming such a construct. Schemas are incrementally transformed by applying to them a sequence of primitive schema transformations, each primitive transformation adding, deleting or renaming just one schema construct.

In general, schema constructs may be extensional i.e. have a data extent associated with them (*e.g.* Table and Column in the relational model) or may be constraints (*e.g.* Primary Key and Foreign Key in the relational model). In this paper we will restrict our discussion to the relational model, and hence extensional schema constructs consist of sets of values. The general form of a primitive transformation that adds an extensional construct c of type T to a schema S in order to generate new schema S' is add$T(c, q_S)$, where q_S is a query over S specifying the extent of c in terms of the existing constructs of S. The semantics of this transformation are that

$$\forall x \,.\, x \in c \leftrightarrow x \in q_S$$

In the AutoMed implementation of BAV, q_S is expressed in a functional **intermediate query language** (**IQL**) (see Section 2.1).

When it is not possible to specify the exact extent of the new construct c being added in terms of the existing schema constructs, the primitive transformation extend$T(c,$ Range $q_l\ q_u)$ must be used instead of add. This adds a new construct c of type T to a schema S, generating a new schema S'. The query q_l over S states what is the minimum extent of c in S'; q_l may be the constant Void if no lower bound on the extent can be specified. The query q_u over S states what is the maximal extent of c in S', and may be the constant Any if no upper bound on the extent can be specified [2]. For non-Void q_l therefore, $\forall x \,.\, x \in c \leftarrow x \in q_l$; and for non-Any q_u, $\forall x \,.\, x \in c \rightarrow x \in q_u$. Also, add$T(c, q_S)$ is equivalent to extend$T$ $(c,$ Range $q_S\ q_S)$

In a similar fashion, the transformation delete$T(c, q_s)$ when applied to schema S' generates a new schema S with the construct c of type T removed. The extent of c may be recovered using the query q_S on S, and $\forall x \,.\, x \in c \leftrightarrow x \in q_S$. Note therefore that from a transformation delete$T$(c,q_S) used to transform schema S' to schema S we can automatically infer that add$T$(c,q_S) transforms S to S', and vice versa. When it is not possible to specify the exact extent of the construct c being deleted from S' in terms of the remaining schema constructs, the transformation contract$T(c,$ Range $q_l\ q_u)$ must be used instead of delete. This removes a construct c of type T from schema S' to form a new schema S. The query q_l over S states what is the minimum extent of c in S', while the query q_u over S states what is the maximal extent of c in S'. Again, q_1 may be Void and q_u may be Any. delete$T(c, q_S)$ is equivalent to contract$T(c,$ Range $q_S\ q_S)$. Also, from contract$T(c,$ Range $q_l\ q_u)$ used to transform schema S' to schema S we can infer that extend$T(c,$ Range $q_l\ q_u)$ transforms S to S', and vice versa.

Finally, the transformation rename$T(c, c')$ causes a construct c of type T in a schema S to be renamed to c' in a new schema S', where $\forall x \,.\, x \in c \leftrightarrow x \in c'$. Thus, from rename$T(c, c')$ used to transform S to S' we can infer that rename$T(c', c)$ transforms S' to S.

[2] Syntactically, Range, Void and Any are all examples of **constructors**, which in this case respectively take 2, 0 and 0 arguments. Constructors in functional languages are analogous to function symbols in logic languages.

2.1 AutoMed's IQL Query Language

IQL is a comprehensions-based functional query language[3]. It supports strings e.g. 'Computer Science', booleans True and False, real numbers, integers, tuples e.g. $\{1,2,3\}$, and sets, bags and lists. There are several polymorphic primitive operators for manipulating sets, bags and lists. The operator ++ concatenates two lists, and performs bag union and set union on bags and sets, respectively. The operator flatmap applies a collection-valued function f to each element of a collection and applies ++ to the resulting collections. For sets, it is defined recursively as follows, where [] denotes the empty set and (SCons x xs) denotes a set containing an element x with xs being the rest of the set (which may be empty):

$\mathsf{flatmap}\, \mathsf{f}\, [] = []$

$\mathsf{flatmap}\, \mathsf{f}\, (\mathsf{SCons}\, \mathsf{x}\, \mathsf{xs}) = (\mathsf{f}\, \mathsf{x}) ++ (\mathsf{flatmap}\, \mathsf{f}\, \mathsf{xs})$

Henceforth in this paper, we confine our discussion to collections that are sets.

The operator flatmap can be used to specify **comprehensions** over sets. These are of the form $[h \mid q_1;\ \ldots;\ q_n]$ where h is an expression termed the **head** and $q_1, \ldots, q_n$ are **qualifiers**, with $n \geq 0$. Each qualifier is either a **filter** or a **generator**. A generator has syntax $p{<-}e$ where e is a set-valued expression and p is a **pattern** i.e. an expression involving variables and tuple constructors only. The variables of p are successively bound by iterating through e. Any variables appearing in the head, h, inherit these bindings. A filter is a boolean-valued expression, which must be satisfied by the values generated by the generators in order for these values to contribute to the final result of the comprehension. Comprehensions are a convenient high-level syntax and add no extra expressiveness to languages such as IQL since they translate into applications of flatmap. We give the translation below for a set comprehension, where Q denotes a sequence of qualifiers and [h] a set comprising a single element h:

$[\mathsf{h} \mid \mathsf{p} <- \mathsf{e};\ \mathsf{Q}] \equiv \mathsf{flatmap}\, (\mathsf{lambda}\, \mathsf{p}.[\mathsf{h} \mid \mathsf{Q}])\, \mathsf{e}$

$[\mathsf{h} \mid \mathsf{e};\ \mathsf{Q}] \equiv \mathsf{if}\ \mathsf{e} = \mathsf{True}\ \mathsf{then}\ [\mathsf{h} \mid \mathsf{Q}]\ \mathsf{else}\ []$

$[\mathsf{h} \mid] \equiv [\mathsf{h}]$

IQL supports unification of variables appearing in the patterns of generators within the same comprehension. For example, $[\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\} \mid \{\mathsf{a},\mathsf{b},\mathsf{c}\} <- \mathsf{r}; \{\mathsf{d},\mathsf{c},\mathsf{e}\} <- \mathsf{s}]$ is equivalent to $[\{\mathsf{a},\mathsf{b},\mathsf{c},\mathsf{d},\mathsf{e}\} \mid \{\mathsf{a},\mathsf{b},\mathsf{c}\} <- \mathsf{r}; \{\mathsf{d},\mathsf{c2},\mathsf{e}\} <- \mathsf{s}; \mathsf{c} = \mathsf{c2}]$

Several equivalences hold for these IQL operators, which follow from their definition and from the interpretation assigned to the Void and Any constants. We list an indicative subset in the Appendix, including specifically those equivalences that we refer to later in the paper. These equivalences assume that expressions are well-typed (which can be verified statically for IQL) and they are applied by AutoMed's query optimiser in order to simplify IQL queries before evaluation.

2.2 An Example

Figure 2 shows four schemas S_1, S_2, S_3, S_p. S_1, S_2, S_3 are data source schemas while S_p is what in a centralised data integration system would be called a **global schema**

[3] We refer the reader to [11] for full details of the language and confine our discussion here to just those aspects that are necessary for this paper. Such languages subsume query languages such as SQL-92 and OQL in expressiveness [3].

and in our P2P system is called a **public schema**. The semantics of the application domain are that a student with name sname may repeatedly sit the exam for a course (identified by ccode, and each having a title) over any number of semesters, and achieve an exam mark on each exam sitting. However, for all attempts of the course, the student will have the same tutor (tutors having been introduced at the start of 1994, along with a coursework mark cwmark that students can attempt only once per course). Each student studies for one degree. Each degree is identified by a dcode, has a title dname and has an associated qualification.

S_1 studies($\underline{\text{sname}}$,$\underline{\text{ccode}}$,$\underline{\text{sem}}$,mark,title,dname)

S_2 teach($\underline{\text{sname}}$,$\underline{\text{ccode}}$,$\underline{\text{sem}}$,mark,tname?)

S_3 degree($\underline{\text{dcode}}$,dname,qual)
ug($\underline{\text{sname}}$,dcode)
reg($\underline{\text{sname}}$,$\underline{\text{ccode}}$,cwmark,tutor)

S_p degree($\underline{\text{dcode}}$,dname)
student($\underline{\text{sname}}$,dcode)
course($\underline{\text{ccode}}$,title)
sit($\underline{\text{sname}}$,$\underline{\text{ccode}}$,$\underline{\text{sem}}$,mark,cwmark?)

Fig. 2. Three data sources S_1, S_2, S_3, and a public schema S_p

Schema S_p is a virtual schema modelling the application domain, omitting the information about tutors and about the qualification associated with degrees. The cwmark is shown as optional (by a '?' suffix) since it was only awarded from 1994 onwards. Schema S_1 represents a data source that holds information about courses with a ccode greater or equal to 500, and holds data in first normal form (since dname is dependent on just sname and title is dependent on just ccode). Schema S_2 represents a data source that holds information about courses with a ccode less than 500, and is also in first normal form, since it holds in tname the tutor's name (an optional attribute), which is dependent on just sname and ccode. Schema S_3 represents a data source that details students' tutors, the degrees students studied, and the coursework mark students gained for courses, and is held in third normal formal.

We consider below fragments of the pathways $S_1 \rightarrow S_p$ and $S_2 \rightarrow S_p$ in order to illustrate the BAV approach and the use of IQL queries within transformations. Within $S_1 \rightarrow S_p$ it is necessary to decompose the studies table in S_1 in order to produce the separate course table that is present in S_p. Here is the fragment of that pathway:

① extendTable(⟪course⟫, Range ([{c} | {s, c, t} <– ⟪studies⟫]) Any)
② extendColumn(⟪course,ccode⟫, Range [{c, c} | {c} <– ⟪course⟫] Any)
③ extendColumn(⟪course,title⟫, Range ([{c, ti} | {{s, c, t}, ti} <– ⟪studies,title⟫]) Any)
④ contractColumn(⟪studies,title⟫,
  Range Void [{{s, c, t}, ti} | {s, c, t} <– ⟪studies⟫; {c, ti} <– ⟪course,title⟫])

Transformation ① states that the course table in S_p contains as its set of keys at least those ccode attributes of studies in S_1 (the first argument of the Range constructor). We note here that the AutoMed representation of a relational table models the table itself by its set of primary key values, and models each attribute a of the table by the projection of the table onto the primary key attributes plus a (see [17]).

Transformations ② and ③ add the ccode and title columns to course. Again these are extend transformations with upper bound Any. The final transformation ④ removes the title attribute of the studies table and specifies the upper bound that the title attribute in S_p places on the extent of the title attribute in S_1.

The pathway $S_2 \rightarrow S_p$ needs to specify that the tutor tname has no representation in S_p, using transformation ⑤ below. The remainder of the pathway is not required for the examples that follow, and is therefore omitted from our discussion.

⑤ contractColumn(⟨⟨teach,tname⟩⟩, Range Void Any)

3 Query Reformulation over BAV Pathways

In this section, we discuss how query reformulation can be undertaken over BAV pathways. We first illustrate how BAV pathways can be used for GAV and LAV query reformulation, and hence can support GAV and LAV query processing. We then present a BAV-specific query reformulation algorithm which subsumes as special cases GAV and LAV query reformulation.

GAV query reformulation is based on query unfolding. For example, to evaluate a query q on S_p with respect to S_1, we traverse the pathway $S_p \rightarrow S_1$ (i.e. the *reverse* of the pathway $S_1 \rightarrow S_p$ described earlier) replacing each scheme in q that appears in an delete or contract transformation with the corresponding query of that transformation.

Example Query 1: To reformulate the query

$q_1 = [\{\mathsf{ti}\} \mid \{\mathsf{c}, \mathsf{ti}\} <\!- \langle\langle \mathsf{course}, \mathsf{title} \rangle\rangle; \mathsf{c} = 500]$

first ④ is ignored (since its reverse is an extend transformation), and then ③ unfolds ⟨⟨course,title⟩⟩ giving:

$[\{\mathsf{ti}\} \mid \{\mathsf{c}, \mathsf{ti}\} <\!- \mathsf{Range}([\{\mathsf{c}, \mathsf{ti}\} \mid \{\mathsf{s}, \mathsf{c}, \mathsf{t}, \mathsf{ti}\} <\!- \langle\langle \mathsf{studies}, \mathsf{title} \rangle\rangle])\ \mathsf{Any}; \mathsf{c} = 500]$

Using the equivalence in Appendix A (a) and the third equivalence in App. A (b) this simplifies to:

$\mathsf{Range}[\{\mathsf{ti}\} \mid \{\mathsf{c}, \mathsf{ti}\} <\!- [\{\mathsf{c}, \mathsf{ti}\} \mid \{\mathsf{s}, \mathsf{c}, \mathsf{t}, \mathsf{ti}\} <\!- \langle\langle \mathsf{studies}, \mathsf{title} \rangle\rangle]; \mathsf{c} = 500]\ \mathsf{Any}$

Using the last equivalence in App. A (d) this further simplifies to:

$\mathsf{Range}[\{\mathsf{ti}\} \mid \{\mathsf{s}, \mathsf{c}, \mathsf{t}, \mathsf{ti}\} <\!- \langle\langle \mathsf{studies}, \mathsf{title} \rangle\rangle; \mathsf{c} = 500]\ \mathsf{Any}$

Transformations ② and ① have no further effect on this query, and thus this is the transformed query that can execute on data source S_1[4].

As another example, consider table reg in S_3 that has sname and ccode as its key attributes. In the pathway $S_3 \rightarrow S_p$, reg is mapped to table sit of S_p that has sname, ccode and sem as its key attributes since students may (re)sit the examination part of any course once in any semester. Recall that the tutors for courses were only introduced from sem 1 of 1994. Below is the relevant fragment of the pathway $S_3 \rightarrow S_p$. We note that transformation ⑥ contains the expression Const1 s c in the head of the comprehension. Here, Const1 is an IQL **constructor** (Skolem function), used because it is not possible to derive the sem attribute of ⟨⟨sit⟩⟩ from ⟨⟨reg⟩⟩.

[4] We have used here equivalences from Appendix A to improve the readability of our example queries. In practice, the AutoMed logical optimiser applies these kinds of simplifications while the query is being reformulated and before it is executed.

$\textcircled{6}$ $\mathsf{extendTable}(\langle\langle\mathsf{sit}\rangle\rangle,$
$\quad \mathsf{Range}\ [\{\mathsf{s},\mathsf{c},\mathsf{Const1\ s\ c}\} \mid \{\mathsf{s},\mathsf{c}\} <\!- \langle\langle\mathsf{reg}\rangle\rangle]\ \mathsf{Any})$
$\textcircled{7}$ $\mathsf{extendColumn}(\langle\langle\mathsf{sit,sname}\rangle\rangle, \mathsf{Range}\ [\{\{\mathsf{s},\mathsf{c},\mathsf{t}\},\mathsf{s}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \langle\langle\mathsf{sit}\rangle\rangle]\ \mathsf{Any})$
$\textcircled{8}$ $\mathsf{extendColumn}(\langle\langle\mathsf{sit,ccode}\rangle\rangle, \mathsf{Range}\ [\{\{\mathsf{s},\mathsf{c},\mathsf{t}\},\mathsf{c}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \langle\langle\mathsf{sit}\rangle\rangle]\ \mathsf{Any})$
$\textcircled{9}$ $\mathsf{addColumn}(\langle\langle\mathsf{sit,cwmark}\rangle\rangle,$
$\quad [\{\{\mathsf{s},\mathsf{c},\mathsf{t}\},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \langle\langle\mathsf{sit}\rangle\rangle; \{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{reg,cwmark}\rangle\rangle])$
$\textcircled{10}$ $\mathsf{extendColumn}(\langle\langle\mathsf{sit,sem}\rangle\rangle, \mathsf{Range}\ [\{\{\mathsf{s},\mathsf{c},\mathsf{t}\},\mathsf{t}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \langle\langle\mathsf{sit}\rangle\rangle]\ \mathsf{Any})$
$\textcircled{11}$ $\mathsf{deleteColumn}(\langle\langle\mathsf{reg,sname}\rangle\rangle, [\{\{\mathsf{s},\mathsf{c}\},\mathsf{s}\} \mid \{\mathsf{s},\mathsf{c}\} <\!- \langle\langle\mathsf{reg}\rangle\rangle])$
$\textcircled{12}$ $\mathsf{deleteColumn}(\langle\langle\mathsf{reg,ccode}\rangle\rangle, [\{\{\mathsf{s},\mathsf{c}\},\mathsf{c}\} \mid \{\mathsf{s},\mathsf{c}\} <\!- \langle\langle\mathsf{reg}\rangle\rangle])$
$\textcircled{13}$ $\mathsf{deleteColumn}(\langle\langle\mathsf{reg,cwmark}\rangle\rangle, [\{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} \mid \{\{\mathsf{s},\mathsf{c},\mathsf{t}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{sit,cwmark}\rangle\rangle])$
$\textcircled{14}$ $\mathsf{contractTable}(\langle\langle\mathsf{reg}\rangle\rangle, \mathsf{Range\ Void}\ [\{\mathsf{s},\mathsf{c}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \langle\langle\mathsf{sit}\rangle\rangle; \mathsf{t} >= \text{‘1994-1’}])$

There are a family of constructors Const1, Const2, ... Any expression of the form Const$i$ $e_1 \ldots e_n$ is only comparable with an expression constructed using the same constructor i.e. with an expression of the form Const$i$ $e'_1 \ldots e'_n$. Thus, an expression of the form Const$i$ $e_1 \ \ldots \ e_n$ = Const$i$ $e'_1 \ \ldots \ e'_n$ evaluates to True if $e_j = e'_j$ evaluates to True for all j otherwise it evaluates to False, and similarly for the other comparison operators. Any other kind of comparison of Const$i$ returns the value Null, denoting "unknown". If Null is the value of a filter in a comprehension, then the result will be a Range expression i.e. the second rule of comprehension translation in Section 2.1 becomes:

$$\begin{aligned}[\mathsf{h} \mid \mathsf{e};\ \mathsf{Q}] \equiv\ & \mathsf{if\ e = True\ then}\ [\mathsf{h} \mid \mathsf{Q}]\ \mathsf{elseif\ e = False\ then}\ []\\ & \mathsf{else}\ (\mathsf{Range\ Void}\ [\mathsf{h} \mid \mathsf{Q}]\ )\end{aligned}$$

Example Query 2: Consider the following query posed on S_p:

$$q_2 = [\{\mathsf{s},\mathsf{c},\mathsf{cw}\} \mid \{\{\mathsf{s},\mathsf{c},\mathsf{t}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{sit,cwmark}\rangle\rangle; \mathsf{t} >= \text{‘1997-1’}]$$

Unfolding $\langle\langle\mathsf{sit,cwmark}\rangle\rangle$ using $\textcircled{9}$ we obtain:

$$\begin{aligned}&[\{\mathsf{s},\mathsf{c},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t},\mathsf{cw}\} <\!- [\{\mathsf{s},\mathsf{c},\mathsf{t},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \langle\langle\mathsf{sit}\rangle\rangle;\\ &\qquad \{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{reg,cwmark}\rangle\rangle]; \mathsf{t} >= \text{'1997-1'}]\end{aligned}$$

which by an equivalence in App. A (d) simplifies to

$$[\{\mathsf{s},\mathsf{c},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \langle\langle\mathsf{sit}\rangle\rangle; \{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{reg,cwmark}\rangle\rangle; \mathsf{t} >= \text{'1997-1'}]$$

Unfolding $\langle\langle\mathsf{sit}\rangle\rangle$ using $\textcircled{6}$ we obtain:

$$\begin{aligned}&[\{\mathsf{s},\mathsf{c},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- \mathsf{Range}[\{\mathsf{s},\mathsf{c},\mathsf{Const1\ s\ c}\} \mid \{\mathsf{s},\mathsf{c}\} <\!- \langle\langle\mathsf{reg}\rangle\rangle]\ \mathsf{Any};\\ &\qquad \{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{reg,cwmark}\rangle\rangle; \mathsf{t} >= \text{'1997-1'}]\end{aligned}$$

By the equivalences of App A (a) and (b), this simplifies to:

$$\begin{aligned}&\mathsf{Range}[\{\mathsf{s},\mathsf{c},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- [\{\mathsf{s},\mathsf{c},\mathsf{Const1\ s\ c}\} \mid \{\mathsf{s},\mathsf{c}\} <\!- \langle\langle\mathsf{reg}\rangle\rangle];\\ &\qquad \{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{reg,cwmark}\rangle\rangle; \mathsf{t} >= \text{'1997-1'}]\ \mathsf{Any}\end{aligned}$$

Swapping the last two qualifiers of the outer comprehension, and moving $\mathsf{t} >= \text{'1997-1'}$ into the inner comprehension (by equivalences in App A (d)) gives:

$$\begin{aligned}&\mathsf{Range}[\{\mathsf{s},\mathsf{c},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- [\{\mathsf{s},\mathsf{c},\mathsf{Const1\ s\ c}\} \mid\\ &\qquad\qquad \{\mathsf{s},\mathsf{c}\} <\!- \langle\langle\mathsf{reg}\rangle\rangle; (\mathsf{Const1\ s\ c}) >= \text{'1997-1'}];\\ &\qquad \{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{reg,cwmark}\rangle\rangle]\ \mathsf{Any}\end{aligned}$$

At run time this gives the same result as the following query, since $\mathsf{Const1\ s\ c} >=$ '1997-1' evaluates to Null:

$$\begin{aligned}&\mathsf{Range\ Void}\ [\{\mathsf{s},\mathsf{c},\mathsf{cw}\} \mid \{\mathsf{s},\mathsf{c},\mathsf{t}\} <\!- [\{\mathsf{s},\mathsf{c},\mathsf{Const1\ s\ c}\} \mid\\ &\qquad\qquad \{\mathsf{s},\mathsf{c}\} <\!- \langle\langle\mathsf{reg}\rangle\rangle]; \{\{\mathsf{s},\mathsf{c}\},\mathsf{cw}\} <\!- \langle\langle\mathsf{reg,cwmark}\rangle\rangle]\end{aligned}$$

i.e. it returns as an upper bound the student names, courses they have taken and coursework marks obtained from S_3.

Consider now the ⟨⟨studies,dname⟩⟩ attribute of S_1, which corresponds in S_p to some instances of the join between ⟨⟨student,dcode⟩⟩ and ⟨⟨degree,dname⟩⟩. This is expressed in BAV by the following fragment of the pathway $S_1 \rightarrow S_p$:

⑮ extendTable(⟨⟨student⟩⟩, Range [{s} | {s, c, t} <– ⟨⟨studies⟩⟩] Any)
⑯ addColumn(⟨⟨student,sname⟩⟩, [{s, s} | {s} <– ⟨⟨student⟩⟩])
⑰ extendColumn(⟨⟨student,dcode⟩⟩, Range Void Any)
⑱ extendTable(⟨⟨degree⟩⟩, Range [{d} | {s, d} <– ⟨⟨student,dcode⟩⟩] Any)
⑲ addColumn(⟨⟨degree,dcode⟩⟩, [{d, d} | {d} <– ⟨⟨degree⟩⟩])
⑳ extendColumn(⟨⟨degree,dname⟩⟩, Range [{d, dn} | {s, d} <– ⟨⟨student,dcode⟩⟩;
    {{s, c, t}, dn} <– ⟨⟨studies,dname⟩⟩] Any)
㉑ contractColumn(⟨⟨studies,dname⟩⟩, Range Void[{{s, c, t}, dn} | {s, c, t} <– ⟨⟨sit⟩⟩;
    {s, d} <– ⟨⟨student,dcode⟩⟩; {d, dn} <– ⟨⟨degree,dname⟩⟩])

Example Query 3: Consider the following query on S_p:

q_3 =[{s} | {s, d} <– ⟨⟨student,dcode⟩⟩; {d, dn} <– ⟨⟨degree,dname⟩⟩;
    dn = 'CS']

Using GAV, ⟨⟨degree,dname⟩⟩ would unfold using ⑳ and ⟨⟨student,dcode⟩⟩ would then unfold using ⑰, obtaining:

[{s} | {s, d} <– Range Void Any;
        {d, dn} <– Range[{d, dn} | {s, d} <– Range Void Any;
        {{s, c, t}, dn} <– ⟨⟨studies, dname⟩⟩] Any; dn = 'CS']

which simplifies to just Range Void Any, i.e. giving no answers.

However the query q_3 on S_p can yield answers using **LAV query processing**. There are two main techniques for this, the **inverse rule** algorithm [21, 6] and the **bucket** algorithm [14]. For simplicity we focus here on the former. Using the inverse rule approach, the definition of a construct c by a query of the form $[h \mid Q]$ is inverted in a two-step process. First, replace each variable in Q that does not appear in h by a distinct Const$i$ with arguments the variable(s) in h. For example, ⑮ has two such variables, c and t which are replaced by Const2 s and Const3 s respectively; while in ㉑, there is one such variable d, which is replaced by Const8 s dn (see below). Next, for each generator $p{<}{-}cs$ in Q, generate a query defining cs in terms of $[p \mid h{<}{-}c; Q']$ where Q' consists of all the filters from Q. To illustrate, we list below all the inverse rules derived from the fragment ⑮–㉑ of the BAV pathway $S_1 \rightarrow S_p$.

$\textcircled{\scriptsize 15.1}$ ⟨⟨studies⟩⟩ =Range Void [{s, Const2 s, Const3 s} | {s} <– ⟨⟨student⟩⟩]
$\textcircled{\scriptsize 16.1}$ ⟨⟨student⟩⟩ = [{s} | {s, s} <– ⟨⟨student,sname⟩⟩]
$\textcircled{\scriptsize 18.1}$ ⟨⟨student,dcode⟩⟩ =Range Void [{Const4 d, d} | {d, d} <– ⟨⟨degree,dcode⟩⟩]
$\textcircled{\scriptsize 19.1}$ ⟨⟨degree⟩⟩ = [{d} | {d, d} <– ⟨⟨degree,dcode⟩⟩]
$\textcircled{\scriptsize 20.1}$ ⟨⟨student,dcode⟩⟩ =Range Void [{Const5 d dn, d} | {d, dn} <– ⟨⟨degree,dname⟩⟩]
$\textcircled{\scriptsize 20.2}$ ⟨⟨studies,dname⟩⟩ =Range Void [{{Const5 d dn, Const6 d dn, Const7 d dn}, dn} |
        {d, dn} <– ⟨⟨degree,dname⟩⟩]
$\textcircled{\scriptsize 21.1}$ ⟨⟨student,dcode⟩⟩ =Range [{s, Const8 s c t dn} | {{s, c, t}, dn}<–
        ⟨⟨studies,dname⟩⟩] Any
$\textcircled{\scriptsize 21.2}$ ⟨⟨degree,dname⟩⟩ =Range [{Const8 s c t dn, dn} | {{s, c, t}, dn}<–
        ⟨⟨studies,dname⟩⟩] Any
$\textcircled{\scriptsize 21.3}$ ⟨⟨sit⟩⟩ =Range [{s, c, t} | {{s, c, t}, dn} <– ⟨⟨studies,dname⟩⟩] Any

Query processing that requires to use a particular construct can now combine the direct definition of the construct within the BAV pathway with all the inverse rules for that construct derived from the BAV pathway. These definitions can be combined using a merge function defined as follows, where union and intersect are set union and set intersection:

merge (Range e1 e2) (Range e1′ e2′) = Range (union e1 e1′) (intersect e2 e2′)

Returning to our example, when a query is submitted to S_p and answers are required from S_1, the rules ⑮,⑯, ⑰,⑱,⑲,⑳,$\textcircled{21.1}$,$\textcircled{21.2}$, $\textcircled{21.3}$, can be used. In particular, for processing query q_3 above, we have:

⟨⟨student,dcode⟩⟩ = merge ⑰ $\textcircled{21.1}$ = $\textcircled{21.1}$ and
⟨⟨degree,dname⟩⟩ = merge ⑳ $\textcircled{21.2}$ = $\textcircled{21.2}$

Substitution now for ⟨⟨student,dcode⟩⟩ and ⟨⟨degree,dname⟩⟩ in q_3 gives:

```
[{s} | {s, d}<−Range[{s, Const8 s c t dn} | {{s, c, t}, dn}<−⟨⟨studies, dname⟩⟩] Any;
        {d, dn} <− Range[{Const8 s c t dn, dn} | {{s, c, t}, dn} <−
        ⟨⟨studies, dname⟩⟩] Any; dn = 'CS']
```

which simplifies to:

```
Range[{s} | {s, d} <− [{s, Const8 s c t dn} | {{s, c, t}, dn} <− ⟨⟨studies, dname⟩⟩];
        {d, dn} <− [{Const8 s c t dn, dn} | {{s, c, t}, dn} <− ⟨⟨studies, dname⟩⟩];
        dn = 'CS'] Any
```

which when evaluated would give the same set of answers as:

```
Range[{s} | {{s, c, t}, d} <− ⟨⟨studies, dname⟩⟩; dn = 'CS'] Any
```

3.1 BAV Query Reformulation

Following the examples presented above, we now summarise how combined GAV and LAV query reformulation can be carried out over a BAV pathway $S_x \rightarrow S_y$, with the aim of obtaining the maximal information that would be derivable from the BAV pathway by means of GAV and LAV query processing techniques.

Suppose we wish to reformulate a query q posed on S_x to be posed with respect to S_y. (We note that, due to the reversibility of BAV pathways, from a pathway $S_x \rightarrow S_y$ it is also possible to reformulate a query q posed on S_y to be posed with respect to S_x. The process is exactly as described below except that now it is with respect to the, automatically derivable, *reverse* pathway $S_y \rightarrow S_x$. This was the scenario illustrated in the examples above, where pathways $S_x \rightarrow S_p$ were used to reformulate queries on S_p so that they could be evaluated on S_x.)

The first step is to construct a set of view definitions, $\mathcal{V}$, defining constructs in S_x in terms of constructs in S_y. This is undertaken by traversing the pathway $S_x \rightarrow S_y$, and at each transformation step t taking one of the following actions:

– if t is of the form rename(c, c') the rule $c = c'$ is added to $\mathcal{V}$;
– if t is of the form delete(c, q) or contract(c, q), the rule $c = q$ is added to $\mathcal{V}$;
– if t is of the form add(c, q), where q is a comprehension referencing schema constructs $c_1, \ldots, c_n$ in its generators, then invert the rule $c = q$ (as described above) to obtain a set of rules of the form $c_i = q_i$ for $1 \leq i \leq n$ such that the only scheme referenced in each q_i is c; add these rules to $\mathcal{V}$;

- if t is of the form $\mathsf{extend}(c, \mathsf{Range\,Void}\, q_u)$, where q_u is a comprehension as in the case of $\mathsf{add}(c, q)$, then invert the rule $c = \mathsf{Range\,Void}\ q_u$ to obtain a set of rules of the form $c_i = \mathsf{Range}\, q_i\ \mathsf{Any}$; add these rules to $\mathcal{V}$;
- if t is of the form $\mathsf{extend}(c, \mathsf{Range}\, q_l\ \mathsf{Any})$, where q_l is a comprehension as in the case of $\mathsf{add}(c, q)$, then invert the rule $c = \mathsf{Range}\, q_l\ \mathsf{Any}$ to obtain a set of rules of the form $c_j = \mathsf{Range\,Void}\ q_j$; add these rules to $\mathcal{V}$;
- if t is of the form $\mathsf{extend}(c, \mathsf{Range}\, q_l\ q_u)$, where q_l and q_u are comprehensions as in the case of $\mathsf{add}(c, q)$, then invert the rule $c = \mathsf{Range}\, q_l\ q_u$ by inverting separately q_u and q_l, as in the previous two cases, to obtain from q_u a set of rules of the form $c_i = \mathsf{Range}\, q_i\ \mathsf{Any}$ and from q_l a set of rules of the form $c_j = \mathsf{Range\,Void}\ q_j$; add these rules to $\mathcal{V}$;

We note that the worst-case complexity of constructing $\mathcal{V}$ is $O(N \times M)$ where N is the number of primitive transformations in the pathway and M is the maximum number of schema constructs appearing in comprehension expressions.

Once constructed, $\mathcal{V}$ can be used to reformulate a query q posed on S_x with respect to S_y. We term a schema construct c which appears in S_y **final** otherwise it is **non-final**. The query reformulation algorithm is as follows, where the function $NF(q)$ returns the set of non-final schemes occurring in an IQL query q:

```
$while\ NF(q) \neq \emptyset$
    $for\ each\ c \in NF(q)$
        $e :=$ Range Void Any
        $for\ each\ rule\ r \in \mathcal{V}\ such\ that\ head(r) = c$
            $e :=$ merge $e\ body(r)$
        $q := [c/e]q$
```

In other words, non-final constructs in q are successively replaced by their definition in $\mathcal{V}$ until there are no non-final constructs left. It is easy to see that this process terminates: Let $\mathcal{G}$ be the graph obtained from $\mathcal{V}$ by creating a node in $\mathcal{G}$ for each schema construct in the head of a rule in $\mathcal{V}$ and an arc $c \rightarrow c'$ in $\mathcal{G}$ if c' appears in a rule defining c. The acyclicity of $\mathcal{G}$ follows from the syntactic properties of BAV transformation sequences: an add or extend transformation can only add a construct that does not exist in the input schema, and the query within the transformation can only refer to constructs existing in the input schema; a delete or contract transformation can only delete a scheme that exists in the input schema and the query within the transformation can only refer to schemes existing in the output schema. By the acyclicity of $\mathcal{G}$ the query reformulation algorithm must terminate. The complexity of the query reformulation algorithm is again $O(N \times M)$. The resulting query would then be optimised, applying equivalences such as those listed in Appendix A and illustrated in the earlier examples, and evaluated.

4 Data Source Schema Query Processing

BAV pathways can in principle be used to map directly between peer schemas in a P2P data integration scenario, and the techniques we have described above can be used to reformulate queries with respect to a BAV pathway between two peer data source

schemas. However, in AutoMed we also support P2P BAV data integration via **public schemas**, as already described in the Introduction. A desirable property in data integration is that the mapping between a pair of schemas S_x and S_y should form a **complete mapping**, in the sense that it identifies all possible mappings between schema objects in S_x and S_y. In our P2P framework, we can construct mappings between S_x and S_y by finding some shared or public schema S_z for which we already know the pathways $S_x \rightarrow S_z$ and $S_z \rightarrow S_y$, and form a concatenation of these two pathways to form a pathway $S_x \rightarrow S_y$. However, this pathway may not in general represent a complete mapping, since S_z might not contain a schema object to represent data associated with schema objects that appear in S_x and S_y and for which a mapping could be specified in a *direct* pathway from S_x to S_y. Suppose that SO_x is a schema object in S_x and SO_y is a schema object in S_y for which a mapping between SO_x to SO_y could be established, but that it is currently absent due to the absence of a corresponding schema object in S_z. Then the pathway $S_x \rightarrow S_z$ must contain a transformation of the form
ⓐ contractObj$_x$(SO$_x$, Range Void Any)
expressing the fact that SO_x cannot be derived or represented in S_z, and similarly $S_z \rightarrow S_y$ must contain a transformation of the form
ⓑ extendObj$_y$(SO$_y$, Range Void Any)
expressing the fact that SO_y cannot be derived or represented in S_z.

Hence, we can use the presence of pairs of transformations of the form of ⓐ and ⓑ to extract pairs of schema objects that might be mappable between S_x and S_y, and feed such pairs into a **schema matching** process [22] in order to derive any mappings that exist between objects as yet unmapped in S_x and S_y. AutoMed supports a suitable schema matching tool [23], which automatically derives possible matchings between pairs of schema objects, and the transformations representing their mapping; the user is then asked to confirm or manually modify the matchings and generated transformations.

Thus, to construct a complete mapping $S_x \rightarrow S_y$ from two complete mappings $S_x \rightarrow S_z$ and $S_z \rightarrow S_y$, we can: (i) Form the set U_x of schema objects that appear in contract transformations in $S_x \rightarrow S_z$, and the set U_y of schema objects that appear in extend transformations in $S_z \rightarrow S_y$. (ii) Perform a pairwise match of objects in U_x against objects in U_y; for each positive match found, remove the transformation steps that contract/extend the matched pair of objects, and replace with the transformations that represent the match found. To illustrate, we return to our running example. Within the pathway $S_3 \rightarrow S_p$ there are two transformations:
㉒ contractColumn(⟨⟨degree,qual⟩⟩, Range Void Any)
㉓ contractColumn(⟨⟨reg,tutor⟩⟩, Range Void Any)

When deriving the pathway $S_2 \rightarrow S_3$ from $S_2 \rightarrow S_p$ (which will include transformation ⑤) and the reverse of $S_3 \rightarrow S_p$, a schema match table as follows is first formed (the filled in circles indicate that the reverse of a transformation is being used):

Data Source S_2		Data Source S_3	
Transformation	Schema Object	Transformation	Schema Object
⑤	⟨⟨teach,tname⟩⟩	●22	⟨⟨degree,qual⟩⟩
		●23	⟨⟨reg,tutor⟩⟩

The schema matching process should then discover that ⟨⟨teach,tname⟩⟩ and ⟨⟨reg,tutor⟩⟩ match (specifically, that they are equivalent, with the exception of the key used). Hence

transformations ⑤ and ㉓ can be removed and the following transformations added to the end of $S_2 \rightarrow S_3$:
㉔ $\mathsf{addColumn}(\langle\langle\text{reg,tutor}\rangle\rangle, [\{\{\mathsf{s},\mathsf{c}\},\mathsf{tu}\} \mid \{\{\mathsf{s},\mathsf{c},\mathsf{t}\},\mathsf{tu}\} <- \langle\langle\text{teach,tname}\rangle\rangle])$
㉕ $\mathsf{deleteColumn}(\langle\langle\text{teach,tname}\rangle\rangle, [\{\{\mathsf{s},\mathsf{c},\mathsf{Const1\,s\,c}\},\mathsf{tu}\} \mid \{\{\mathsf{s},\mathsf{c}\},\mathsf{tu}\} <- \langle\langle\text{reg}\rangle\rangle])$

5 Concluding remarks

The BAV approach has the advantage in a P2P data integration setting of allowing bidirectional logical mappings to be specified between peers. We have discussed how these mappings can be used to support two types of query processing in a P2P data integration system, where either queries are posed on the schema of a data source at a peer or on a virtual public schema. We have shown how GAV and LAV query reformulation can be combined over BAV pathways — specifically, for a comprehensions-based query language — thus obtaining the maximal information from BAV pathways that would be derivable by means of GAV and LAV query processing techniques.

We have focused here on query processing along a single BAV pathway, which cannot generate cyclic relationships between schema objects and hence for which query answering is decidable c.f. [10]. The extension of P2P query processing along a network of arbitrary BAV pathways is an area of ongoing work, and in particular we wish to investigate the applicability of the epistemic semantics approach of [4, 5] to BAV.

References

1. Z. Bellahsène, C. Lanzanitis, P.J. McBrien, and N. Rizopoulos. Querying distributed data in a super-peer based architecture. In *Proc. IWI2006 (in conjunction with WWW06)*, 2006.
2. P.A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: a vision. In *Proc. WebDB'02*, pages 89–94, 2002.
3. P. Buneman et al. Comprehension syntax. *SIGMOD Record*, 23(1):87–96, 1994.
4. D. Calvanese, E. Damagio, G. De Giacomo, M. Lenzerini, and R. Rosati. Semantic data integration in P2P systems. In *Proc. DBISP2P, at VLDB'03*, 2003.
5. D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Logical foundations of peer-to-peer data integration. In *Proc. PODS*, pages 241–251, 2004.
6. O.M. Duschka and M.R. Genesereth. Answering recursive queries using views. In *Proc. PODS*, pages 109–116. ACM, 1997.
7. E. Franconi, G. Kuper, A. Lopatenko, and I. Zaihrayeu. Queries and updates in the coDB peer-to-peer database system. In *Proc. VLDB*, pages 1277–1280, 2004.
8. M. Friedman, A. Levy, and T. Millstein. Navigational plans for data integration. In *Proc. 16th National Conference on Artificial Intelligence*, pages 67–73. AAAI, 1999.
9. A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *Proc. WWW'03*, 2003.
10. A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *Proc. ICDE'03*. IEEE, 2003.
11. E. Jasper, A. Poulovassilis, and L. Zamboulis. Processing IQL queries and migrating data in the AutoMed toolkit. Technical Report No. 20, AutoMed, 2003.
12. E. Jasper, N. Tong, P.J. McBrien, and A. Poulovassilis. View generation and optimisation in the AutoMed data integration framework. In *Proc. Baltic DB&IS04*, volume 672 of *Scientific Papers*, pages 13–30. Univ. Latvia, 2004.

13. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. PODS'02*, pages 233–246. ACM, 2002.
14. A. Levy, A. Rajamaran, and J. Ordille. Querying heterogeneous information sources using source description. In *Proc 22nd VLDB*, pages 252–262, 1996.
15. A. Loser, W. Nejdl, M. Wolpers, and W. Siberski. Information integration in schema-based peer-to-peer networks. In *Proc. CAiSE'03*, LNCS. Springer, 2003.
16. J. Madhavan and A.Y. Halevy. Composing mappings among data sources. In *Proc. VLDB'03*, pages 572–583, 2003.
17. P.J. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. ICDE'03*, pages 227–238. IEEE, 2003.
18. P.J. McBrien and A. Poulovassilis. Defining peer-to-peer data integration using both as view rules. In *Proc. DBISP2P, at VLDB'03*, pages 91–107, 2003.
19. W. Nejdl *et al*. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *Proc. WWW'03*, 2003.
20. A. Poulovassilis and C. Small. Algebraic query optimisation for database programming languages. *The VLDB Journal*, 5(2):119–132, 1996.
21. X. Qian. Query unfolding. In *Proc. ICDE*, pages 48–55. IEEE, 1996.
22. E. Rahm and P.A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10:334–350, 2001.
23. N. Rizopoulos. Automatic discovery of semantic relationships between schema elements. In *Proc. of 6th ICEIS*, 2004.

Appendix A - Some IQL Equivalences

(a) The following equivalence states that $\mathsf{flatmap}$ propagates through expressions of the form $\mathsf{Range\,e1\,e2}$:
$\mathsf{flatmap\,f\,(Range\,e1\,e2) = Range\,(flatmap\,f\,e1)\,(flatmap\,f\,e2)}$
(b) For any f:
$\mathsf{flatmap\,f\,[\,] = [\,],\ flatmap\,f\,Void = Void,\ flatmap\,f\,Any = Any}$
$\mathsf{flatmap\,(lambda\,x.[\,])\,e = [\,],\ flatmap\,(lambda\,x.Void)\,e = Void}$
$\mathsf{flatmap\,(lambda\,x.Any)\,e = Any}$
(c) For any series of qualifiers Q and $\mathsf{Q'}$, pattern p and expression e:
$\mathsf{[e \mid Q; p <\!- [\,]; Q'] = [\,],\ [e \mid Q; p <\!- Void; Q'] = Void,\ [e \mid Q; p <\!- Any; Q'] = Any}$
(d) There are several other equivalences that can be used to simplify comprehension queries (see [20] for a comprehensive discussion). For example,

$\mathsf{[e \mid Q; p1 <\!- e1; e2; Q'] = [e \mid Q; e2; p <\!- e1; Q']}$
$\mathsf{[e \mid Q; e1\ and\ e2; Q'] = [e \mid Q; e1; e2; Q']}$
$\mathsf{[e \mid Q; p <\!- [p \mid p <\!- e']; Q'] = [e \mid Q; p <\!- e'; Q']}$

The first of these holds provided $\mathsf{FV(p1) \cap FV(e2) = \{\}}$, where $\mathsf{FV(e)}$ denotes the set of free variables in an expression e.

The following equivalence allows a filter to be moved inside a nested comprehension provided that $\mathsf{FV(e') \subseteq FV(p)}$ and the pattern $\mathsf{p'}$ is more specific than the pattern p i.e. $\mathsf{p'}$ can be obtained from p by means of some substitution. The filter $\mathsf{e''}$ is obtained from $\mathsf{e'}$ by substituting each variable from p in $\mathsf{e'}$ by its counterpart in $\mathsf{p'}$:
$\mathsf{[e \mid Q; p <\!- [p' \mid Q]; e'; Q'] = [e \mid Q; p <\!- [p' \mid Q; e'']; Q']}$
(e) The following equivalences govern the simplification of expressions involving Range:
$\mathsf{Range\,(Range\,e1\,e2\,)\,Any = Range\,Void\,(Range\,e1\,e2\,) = Range\,e1\,e2}$