2009 21st IEEE International Conference on Tools with Artificial Intelligence

Design and Implementation of a Query Planner for Data Integration

Neeraj Koul, Vasant Honavar
Department of Computer Science,
Iowa State University, Ames -IA 50014, USA

Abstract

[1] *Many applications require integrated access to multiple distributed, autonomous, and often semantically disparate data. Hence there is a need for bridging the semantic gap between the user and the data sources and for answering user queries based on the contents of multiple data sources. This paper describes a query planner that solves these two problems.*

1 Introduction

Recent advances in high throughput data acquisition technologies in many areas have led to a proliferation of a multitude of physically distributed, autonomous, and often semantically disparate data sources. Effective use of such data in data-driven knowledge acquisition and decision support applications e.g., in health informatics, security informatics, social informatics, etc. presents a *data integration* challenge. Addressing this data integration challenge requires techniques for bridging the semantic gap between the user and the data sources with respect to both the data schema and the data content (see [4] for a survey). In a distributed setting, this also requires techniques for coping with *horizontal* and/or *vertical* data fragmentation. In the case of horizontal fragmentation, each data source (e.g., economic data for different states) contains a subset of data tuples that make up the data source of interest (e.g., economic data for the nation). In the case of vertical fragmentation, the different data sources contain subtuples of data tuples make up the data source of interest. Solving the data integration problem in such a setting presents us with a *query planning* problem, that is, the problem of decomposing a user query *q* into queries that can be processed (in the order specified by a *query plan*) by the individual data sources $D_1, D_2 \cdots D_p$ and for combining the results to produce the answer to the user query *q*. In general, there can be multiple query plans for answering a query query *q* from a collection of data sources $D_1, D_2 \cdots D_p$, with some plans being more optimal than others (e.g., the *cost* of execution). This paper describes a query planner for data integration that solves these two problems.

[1]This research was supported in part by the grant IIS 0711356 from the National Science Foundation.

2 The Data Integration Problem

We associate with each data source, a data source description (i.e., the schema and ontology of the data source) yielding an *ontology extended* data sources (OEDS). Formally An OEDS is a tuple $\mathcal{D} = \{D, S, O\}$, where D is the actual data set in the data source, S the data source schema and O the data source ontology [3]. The formal semantics of OEDS are based on ontology-extended relational algebra [2]. Let $\mathcal{D}_1 = \{D_1, S_1, O_1\}, \mathcal{D}_2 = \{D_2, S_2, O_2\} \ldots \mathcal{D}_p = \{D_p, S_p, O_p\}$ be a set of p ontology extended data sources. Let $\mathcal{D}_U = \{D_U, S_U, O_U\}$ be a (virtual) integrated data source from the user's perspective. A user's perspective is specified by $P_U = \{D_U, M_U, \Psi_U\}$ where M_U is the set of mappings from the user schema S_U to the data source schema $S_1 \ldots S_p$ and Ψ_U a set of semantic correspondences from user ontology O_U to the data source ontologies $O_1, O_2 \ldots O_p$. For simplicity, we consider case when the schema mappings are one to one (i.e. every attribute in S_U has a corresponding attribute in schema S_i), the ontologies are attribute value hierarchies and the semantic correspondences take the form of $x < y$ (x is semantically subsumed by y i.e. a subclass relationship), $x > y$ (x semantically subsumes y i.e. a superclass relationship), $x = y$ (x is semantically equivalent to y i.e. equivalent class relationship) and the individual data sources are either horizontal or vertical fragments of the data from the user's perspective. Let Q be the set of all possible queries that a user can pose against $\mathcal{D}_U$. The queries are expressed using an SQL like syntax (called SQL_{indus}). More precisely, an SQL_{indus} query q is of the form $\langle s, f, w \rangle$ where s is the set of attributes in the *Select* clause, f is the set corresponding to the tables in the *From* clause and w is an expression representing the *Where* clause. The syntax of the clause w conforms to the grammar $w = \epsilon | w_{atomic} | w$ *AND* $w | w$ *OR* w and $w_{atomic} = column.name\ op_1\ column.value$ where $op_1 \in \{> \mid < \mid = \mid ! =\}$. When the *column.name* has an ontology (attribute value hierarchy) associated with it, we

1082-3409/09 $26.00 © 2009 IEEE
DOI 10.1109/ICTAI.2009.37

overload the operators $>, <$ and $=$ to imply superclass, subclass and equivalent class respectively. The query $\langle s, f, \epsilon \rangle$ corresponds to the case in which the *where* clause is absent. A data integration problem in this setting can be seen as the triple $\langle \mathcal{D}^*, P_U, Q \rangle$ where $\mathcal{D}^* = \{\mathcal{D}_1, \mathcal{D}_2 \ldots \mathcal{D}_p\}$, $P_U = \{D_U, M_U, \Psi_U\}$ is the user perspective and Q is the set of possible queries that can be posed by the user. We say that the data integration problem is *well-specified* if it is possible to combine the data sources to form a user view of the data using the mappings specified. In this paper we restrict ourselves to *well-specified* data integration problems. We proceed to describe a solution to the data integration problem using a data structure called *DTree*. A *DTree* is a binary tree in which the leaf nodes correspond to the actual data sources. Each internal node is a virtual data source that combines information from its two children. The structure of the *DTree* specifies the constraints on the order in which the data from the individual data sources are combined with the root node denoting a virtual data source that corresponds to D_U.

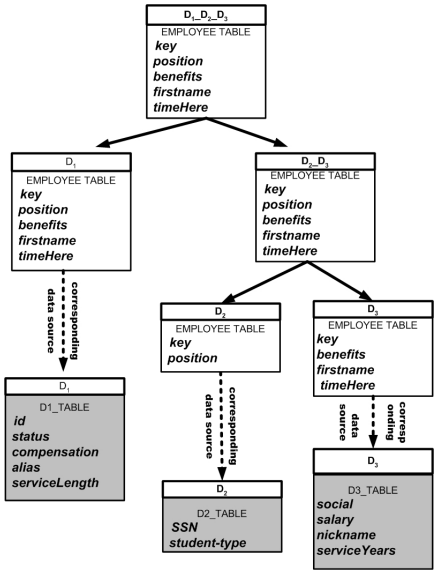

Figure 1. DTree for the Introduced Example

A query against D_U is submitted to the root of the *DTree* and recursively divided into sub-queries such that the leaf nodes receive the queries to be executed against the respective individual data sources. The results from the leaf nodes are recursively combined up the tree with the root node receiving the answer to the query. The leaf nodes in *DTree* cope with the problem of bridging the semantic gap between the user and the data sources (see below) whereas the internal nodes cope with data fragmentation (by combing the results from their respective children). Consider an example where D_1, D_2, D_3 are data sources and D_U represents the integration of the aforementioned data sources. Let each of the datasources contain a single table. Table 1 specifies the attributes present in the data sources along with the schema mappings from the attributes in D_U to the attributes in the data sources. A *DTree* for this example is shown in Figure 1.

D_U EMLOYEE TABLE	$\mapsto D_1$ D1_Table	$\mapsto D_2$ D2_Table	$\mapsto D_3$ D3_Table
key	id	SSN	social
position	status	type	-
benefits	compensation	-	salary
firstName	alias	-	nickName
timeHere	serviceLength	-	serviceYears

Table 1. Schema Mappings

Given a well-specified data integration problem $\pi = \langle \mathcal{D}*, P_U, Q \rangle$, let *DTree* $\tau(\pi)$ represent a *DTree* that solves a well-specified data integration problem π. Let $\Phi^{-1}(S_i)$ denote the set of attributes in S_U that are mapped to corresponding attributes in S_i (recall the schema mappings between S_U and S_i are one to one). The procedure in *Algorithm 1* outlines the steps to construct a *DTree* $\tau(\pi)$ for a *well-specified* integration problem π.

Algorithm 1: *DTree* Construction

Input: A *well-specified* data integration problem $\pi = \langle \mathcal{D}^*, P_U, Q \rangle$
Output: A DTree $\tau(\pi)$ that solves π
$N \mapsto$ emptyset
$\forall \mathcal{D}_i = \langle D_i, S_i, O_i \rangle \in \mathcal{D}^*$ construct a node labeled $\mathcal{D}_i$ and associate with it the set $\Phi^{-1}(S_i)$ and add it to N .
repeat
  while $\exists \mathcal{D}_i, \mathcal{D}_j \in N$ *such that* $\Phi^{-1}(S_i) = \Phi^{-1}(S_j)$ **do**
    begin
      Replace nodes $\mathcal{D}_i$ and $\mathcal{D}_j$ in N by a node named $\mathcal{D}_i\_\mathcal{D}_j$ and associated it with the set $\Phi^{-1}(S_i)$
    end
  if $\exists \mathcal{D}_i, \mathcal{D}_j \in N$ *and* ***no*** $\mathcal{D}_k \in N$ *such that* $\Phi^{-1}(S_i) \cup \Phi^{-1}(S_k) \subseteq \Phi^{-1}(S_j)$ *OR* $\Phi^{-1}(S_j) \cup \Phi^{-1}(S_k) \subseteq \Phi^{-1}(S_i)$ **then**
    Replace nodes $\mathcal{D}_i, \mathcal{D}_j \in N$ by a node named $\mathcal{D}_i\_\mathcal{D}_j$ and associate it with the set $\Phi^{-1}(S_i) \cup \Phi^{-1}(S_j)$
until *until no change in* $|N|$

It is easy to show that a *DTree* exists for every well specified data integration problem; and that in general, given a well specified data integration problem, there can be more than one *DTree* that solves it. Our algorithm outputs one

of the possible *DTree*s that solves the given data integration problem. However, the algorithm can be modified to output an optimal *DTree* (based on some user-specified criteria). For lack of space we omit the proof of correctness for the *DTree* construction algorithm.

3 Query Planner

A query posed by the user against D_U is submitted to the root node of the *DTree* $\tau(\pi)$ that solves the data integration problem $\langle \mathcal{D}^*, P_U, Q \rangle$. A query planner is invoked at each non leaf node of $\tau(\pi)$ to compute the set of plans that can be used to answer the query submitted to the node. Suppose a query q is submitted to a node n in the *DTree* $\tau(\pi)$. The task of the query planner is to output a *set* of plans P such that each $p \in P$ is of the form $\{ q_n^l, q_n^r, \oplus \}$ where q_n^l and q_n^r are the queries submitted to the left and right child of node n and $\oplus$ is a binary operator applied to the results of q_n^l and q_n^r to obtain the results to q. The operator $\oplus$ specifies an *aggregation strategy*. For simplicity, we assume that their is no data overlap among the p data sources (the data overlap can be detected and handled by assuming the existence of an unique id for each instance). Denoting by r_l and r_r the results obtained from the left and the right child of a node, we specify the following aggregation strategies: (i) $\sqcup(r_l, r_r)$ denotes a multiset of all the rows/tuples in r_l and r_r. We overload the operator $\sqcup$ to denote the addition of counts in the case of count queries; (ii) $\bowtie_{id}$ (r_l, r_r) denotes an inner join of r_l and r_r on id; (iii) $\uparrow$ denotes that there is no need for aggregation which is the case when a query is submitted to only one child; (iv) $\langle \phi, q_{local} \rangle$ where $\phi \in \{\sqcup, \bowtie_{id}, \uparrow\}$ dentotes that obtain $r_{temp} = \phi(r_l, r_r)$ and then obtain the final results by running the query q_{local} on r_{temp}; and (v) $\langle \phi, q_{remote} \rangle$ where $\phi \in \{\sqcup, \bowtie_{id}, \uparrow\}$ denotes obtain $r_{temp} = \phi(r_l, r_r)$ and use r_{temp} to construct query q_{remote} from a template and generate a new plan for this q_{remote} yielding a *two step plan*. We now introduce some notation used in describing the Planner Algorithm. Given $\langle q, n, \tau(\pi) \rangle$ where $q = \langle s, f, w \rangle$, we define the following functions: (i) $sig(x)$ returns the set of attributes that appear in x where $x \in \{q, s, w\}$ (For n , $sig(n)$ returns $\Phi^{-1}(n)$); (ii) $\mathcal{F}_{join}(n)$ returns the join column for the children of n (applicable for vertical fragments only); (iii) $\mathcal{F}^l_{child}(n)$ and $\mathcal{F}^r_{child}(n)$ returns the left and right child of the node n respectively; (iv) $s^l_n = sig(s) \cap sig(\mathcal{F}^l_{child}(n))$ and $s^r_n = sig(s) \cap sig(\mathcal{F}^r_{child}(n))$ returns the select columns that are present in the left and right child of n respectively; (v) $q^+ = \langle s \cup \mathcal{F}_{join}(n), f, w \rangle$ adds a join column to the select clause of the query; (vi) $\mathcal{T}_{allData}(q, n, l) = \langle (sig(q^+) \cap sig(\mathcal{F}^l_{child}(n))), f, \epsilon \rangle$ retrieves the data corresponding to columns of the query q^+ that are present in the left child of n; (vii) $\mathcal{T}_{allData}(q, n, r) = \langle (sig(q^+) \cap sig(\mathcal{F}^r_{child}(n))), f, \epsilon \rangle$; (viii) $Result(q)$ represents the result of the query q; (ix) $singlePath(q, n)$ returns true if all the columns in q are present in one child of n (WLOG we assume when the function returns true, all the columns are present in $\mathcal{F}^l_{child}(n)$) ; (x) $horizontalFragmentation(n)$ returns true if $\mathcal{F}^l_{child}(n)$ and $\mathcal{F}^r_{child}(n)$ form the horizontal fragments of n.

For a node n we define a template for a where clause as $W^n_{temp} = \mathcal{F}_{join}(n)$ IN $\$values\$$. The function $Replace(W^n_{temp}, vals)$ replaces the place holder \$values\$ in the template W^n_{temp} by a comma separated list of values in $vals$. This template is used in a *two step plan*, where the results of the first step are used in the template to construct the query for the second step. Once a query q is submitted to a node n in the *DTree*, the plan(s) generated to answer the query depend on how the attributes in s and w clause are distributed (based on how the data is fragmented) among the children of node n. We specify the different data fragmentation scenarios using the function $\mathcal{M}(q, n) \mapsto \langle C_0, C_1, C_2, C_3 \rangle$ where C_0 is set to 1 when the attributes of the select clause are distributed among the two children of the node. C_1 is set to 1 when the attributes in the where clause are distributed among the two children of the node (in this case it has to be $w = w_l \ op \ w_r$ where $op \in \{AND, \ OR\}$). C_2 is set to 1 when $C_1 = 1$ and $w = w_l \ op \ w_r$ and the attributes in w_l and w_r occur individually in the two children (WLOG we assume the signature of the left child includes the signature of w_l whereas the signature of the right child includes the signature ofw_r). C_3 is set to 1 when C_1 is 1, $C_2 = 0$ and $w = w_l \ op \ w_{rl}$ and the attributes in w_l occurs completely in one child and attributes in w_{rl} are distributed over the two children. The value $\langle C_1, C_2, C_3 \rangle$ describe how the attributes in the *where* clause w are fragmented among the two children of the current node. The value $\langle 0, 0, 0 \rangle$ corresponds to *no fragmentation*, $\langle 1, 1, 0 \rangle$ corresponds to *clean fragmentation*, $\langle 1, 0, 1 \rangle$corresponds to *partial fragmentation* and $\langle 1, 0, 0 \rangle$ corresponds to *full fragmentation*. Since our goal is to effectively *minimize* data fragmentation, among the multiple equivalent ways of expressing w, we choose one that corresponds to the least amount of fragmentation. Let $DeFrag(n, w)$ return an equivalent *where* clause for w that has the least fragmentation for n. The *Query Planner* is outlined in *Algorithm 2*.

The correctness of the plans generated by the query planner follows from the manner in which the results are combined at each node of the *DTree*. It can further be shown that the plans produced by the query planner ensure that each data source is queried at most twice (proof omitted). Note that for a given user query *q*, the query planner is executed (and produces a *set* of query plans) at each node in the *DTree.* Answering the query *q* requires choosing one such plan at each of the non leaf nodes of the *DTree*. This choice can be made so as to optimize some desired criterion (e.g., estimated cost of answering the query *q* for each

Algorithm 2: Query Planner

Input: q posed to a node n in *DTree* $\tau(\pi)$
Output: Set of Plans to answer q
if *sig(q)* $\supset$ *sig(n)* **then**
  throw Exception(Query Can't be answered);
if *horizontalFragmentation(n)* **then**
  Add Plan $\mapsto q_l^n = q_r^n = q; \oplus = \sqcup$;
  Add Plan $\mapsto$ `DefaultPlan(`q, n`)`
else
  `/*VerticalFragmentation */`
  $\langle C_0, C_1, C_2, C_3 \rangle = \mathcal{M}(q, n)$
  $w = DeFrag(n, w)$
  if $C_0 == 0$ **then**
    `/*SelectClauseNotFragmented */`
    switch $\langle C_1, C_2, C_3 \rangle$ **do**
      case $\langle 0, 0, 0 \rangle$
        Plans $\mapsto$ `NoFragmentation(`q, n`)`
      case $\langle 1, 1, 0 \rangle$
        Plans $\mapsto$ `CleanFragmentation(`q, n`)`
      case $\langle 1, 0, 1 \rangle$
        Plans $\mapsto$ `PartialFragmentation(`q, n`)`
      case $\langle 1, 0, 0 \rangle$
        Plans $\mapsto$ `DefaultPlan(`q, n`)`
      otherwise
        throw Exception("Not a Possible Case")
  else
    `/*SelectClauseIsFragmented */`
    Plans $\mapsto$ `SelectFragmented(`q, n`)`

Function `DefaultPlan(`q, n`)`

$q_l^n = \langle \mathcal{T}_{allData}(q, n, l) \rangle$; $q_r^n = \langle \mathcal{T}_{allData}(q, n, r) \rangle$
if *horizontalFragmentation(n)* **then**
  $\oplus = \langle \sqcup, q_{local} \rangle$; where $q_{local} = q$;
else $\oplus = \langle \bowtie_{\mathcal{F}_{join(n)}}, q_{local} \rangle$ where $q_{local} = q$;

Function `NoFragmentation(`q, n`)`

if *singlePath(q,n)* **then**
  Add Plan $\mapsto q_l^n = q$; $q_r^n = null$; $\oplus = \uparrow$
else
  `/*WLOG assume attributes in` s `in right child and` w `in left child */`
  Add Plan $\mapsto$
  $q_l^n = \langle \mathcal{F}_{join}(n), f, w \rangle$; $q_r^n = null$;
  $\oplus = \langle \uparrow, q_{remote} \rangle$
  $q_{remote} = \langle s, f, Replace(W_{temp}^n, Result(q_l^n)) \rangle$;
  Add Plan $\mapsto$
  $q_l^n = \langle \mathcal{F}_{join}(n), f, w \rangle$; $q_r^n = \langle s \cup \mathcal{F}_{join}(n), f, \epsilon \rangle$;
  $\oplus = \{ \bowtie_{\mathcal{F}_{join(n)}} \}$;

Function `CleanFragmentation(`q, n`)`

$q = \langle s, f, w_l \; op \; w_r \rangle$
Add Plan $\mapsto$
$q_l^n = \langle \mathcal{F}_{join}(n), f, w_l \rangle$; $q_r^n = null$; $\oplus = \langle \uparrow, q_{remote} \rangle$
$q_{remote} = \langle s, f, w_r \; op \; Replace(W_{temp}^n, Result(q_l^n)) \rangle$
Add Plan $\mapsto$
Mirror of Plan Above (switch l and r in Plan Above)
Add Plan $\mapsto$
$q_l^n = \langle \mathcal{F}_{join}(n), f, w_l \rangle$; $q_r^n = \langle \mathcal{F}_{join}(n), f, w_r \rangle$;
if *op == AND* **then** $\phi = \sqcap$; **else** $\phi = \sqcup$;
$r_{temp} = \phi(Result(q_l^n), Result(q_r^n))$;
$\oplus = \langle \phi, q_{remote} \rangle$;
$q_{remote} = \langle s, f, Replace(W_{temp}^n, r_{temp}) \rangle$
Add Plan $\mapsto$ `/*applicable if` *op* `is AND */`
if *op == AND* **then**
  $q_l^n = q^+$; $q_r^n = \langle \mathcal{F}_{join(n)}, f, w_r \rangle$;
  $\oplus = \{ \bowtie_{\mathcal{F}_{join(n)}} \}$;
Add Plan $\mapsto$ `DefaultPlan(`q, n`)`;

Function `PlansSelectFragmented(`q, n`)`

$\langle C_0, C_1, C_2, C_3 \rangle = \mathcal{M}(q, n)$
`/*Function called when` $C_0 == 1$ `*/`
switch $\langle C_1, C_2, C_3 \rangle$ **do**
  case $\langle 0, 0, 0 \rangle$
    `/*no fragmentation */`
    Add Plan $\mapsto$
    $q_l^n = \langle s_n^l, \cup \mathcal{F}_{join(n)}, f, w \rangle$;
    $q_r^n = \langle s_n^r, \cup \mathcal{F}_{join(n)}, f, \phi \rangle$;
    $\oplus = \langle \bowtie_{\mathcal{F}_{join(n)}}, q_{local} \rangle$; $q_{local} = q$;
    Add Plan $\mapsto$
    Assuming w occurs completely in left child.
    $q_l^n = \langle \mathcal{F}_{join}(n), f, w \rangle$; $q_r^n = null$;
    $\oplus = \langle \uparrow, q_{remote} \rangle$; $q_{remote} =$
    $\langle s, f, Replace(W_{temp}^n, Result(q_l^n)) \rangle$;
    Add Plan $\mapsto$ `DefaultPlan(`q, n`)`
  case $\langle 1, 1, 0 \rangle$
    `/*clean fragmentation */`
    Add Plan $\mapsto$
    $q = \langle s, f, w_l op w_r \rangle$
    if *op == AND* **then** $\phi = \sqcap$; **else** $\phi = \sqcup$;
    $q_l = \langle \mathcal{F}_{join}(n), f, w_l \rangle$;
    $q_r = \langle \mathcal{F}_{join}(n), f, w_r \rangle$; $\oplus = \langle \phi, q_{remote} \rangle$;
    $r_{temp} = \phi\left(Result(q_l^n, Result(q_r^n))\right)$;
    $q_{remote} = \langle s, f, Replace(W_{temp}^n, r_{temp}) \rangle$;
    Add Plan $\mapsto$ `DefaultPlan(`q, n`)`;
  case $\langle 1, 0, 1 \rangle$
    Add Plan $\mapsto$ `DefaultPlan(`q, n`)`;
  case $\langle 1, 0, 0 \rangle$
    Add Plan $\mapsto$ `DefaultPlan(`q, n`)`;
  otherwise
    throw Exception("Not Applicable");

Function `PartialFragmentation(`q, n`)`

Add Plan $\mapsto$ `DefaultPlan(`q, n`);`
Add Plan $\mapsto$
$q = \langle s, f, w_l \; op \; w_{rl} \rangle$.
if *($w_{rl} == w_{ll} \; op_2 w_{rr}$ such that w_{rr} occurs completely in right child and w_{ll} occurs completely in left child, AND s occurs completely in left child)* **then**
$w = (w_l \; op \; w_{ll}) op_2 (w_l \; op_1 \; w_{rr})$
`/*`op `and` op_2 `will be different */`
if $op == AND$ **then** $\phi = \sqcap$; **else** $\phi = \sqcup$;
$q_l^n = \langle \mathcal{F}_{join}(n), f, w_l \rangle$; $q_r^n = \langle \mathcal{F}_{join}(n), f, w_{rr} \rangle$;
$\oplus = \langle \phi, q_{remote} \rangle$;
$r_{temp} = \phi(Result(q_l^n), Result(q_r^n))$; $q_{remote} = \langle s, f, (w_l \; op w_{ll}) \; op_2 \; Replace(W_{temp}^n, r_{temp}) \rangle$;

possible choice of plans). Once a query plan is chosen for each node of the *DTree*, answering the user query reduces to recursively combining the answers to the sub queries that are passed to the leaf nodes of the *DTree*. The leaf nodes handle the semantic gap between the D_U and the individual data sources (arising due to difference in schema's and ontologies used) through the process of *Query Binding*. Query binding converts a sub query received by each leaf node in a form that can be executed against the corresponding data source and consists of three steps: *Translation*, *Renaming* and *Rewriting*. Let $\Phi^{\leftarrow}(S_i)$ be a renaming of the schema S_i by using the one to one schema mapping between S_U and S_i. *Translation* converts a query q in schema S_U and ontology O_U into a query against schema $\Phi^{\leftarrow}(S_i)$ and ontology O_i (refer [1] for details). *Renaming* converts the translated query q_1 (now in schema $\Phi^{-1}(S_1)$) to a query q_2 that is against the schema S_i of the corresponding data source D_i. After Renaming the query q_2 is still in SQL_{indus} syntax and may include ontological relations (e.g. subclass and superclass) in the *where* clause of the query. The process of *Rewriting* converts the query q_2 in SQL_{indus} syntax into a query q_3 in the language understood by the data source while preserving the query semantics. For RDBMS data source the only non trivial part is to convert the subclass, superclass and equivalentclass relations (within the corresponding data source ontology) that appear in the *where* clause w (if any) of the query in SQL_{indus} syntax into appropriate expressions in SQL. We do this as follows: $\forall w_{atomic} = column.name \; op_1 \; value$ where the attribute $column.name$ has an AVH associated with it, replace w_{atomic} by a SQL fragment of the form *column.name IN valueSet* where $valueSet = \{x | x \in O_i$ and $x \; op_1 \; value \; is \; true\}$.

4. Results and Discussion

The proposed approach to data integration has been implemented in Java as part of INDUS integration system. The prototype implementation of the system (with the planner) is open sourced at [7]. Our implementation when used with the Indus Learning Framework (ILF), an approach that learns classifiers from a *single* data source using SQL count queries [9] [8], enables ILF to learn classifiers from multiple semantically disparate data sources

The problem of data integration has received significant attention in literature (see [10] and [4] for overviews). Most of this work has focused on dealing with schema heterogeneity (see [11] for a survey). Aspects of data content heterogeneity are addressed in [13], [5]. Related systems of interest are SIRUP [14], COIN [5] and BUSTER [12]. In contrast our solution focuses on integrating ontology extended data sources that are fragmented and semantically heterogeneous from a user point view. It does not assume existence of a single global ontology and uses mappings to handle schema and data content heterogeneity.

References

[1] J. Bao, D. Caragea, and V. Honavar. Query translation for ontology extended data sources. AAAI Workshop on Semantic e-Science, 2007.

[2] P. Bonatti, Y. Deng, and V. Subrahmanian. An ontology-extended relational algebra. IRI , pp. 192–199, 2003.

[3] D. Caragea, J. Zhang, J. Bao, J. Pathak, and V. Honavar. Algorithms and software for collaborative discovery from autonomous, semantically heterogeneous information sources (invited paper). ALT, LNCS vol. 3734, pp. 13–44. Springer-Verlag, 2005.

[4] A. Doan and A. Y. Halevy. Semantic-integration research in the database community. *AI Mag.*, 26(1):83–94, 2005.

[5] C. H. Goh, S. Bressan, S. Madnick, and M. Siegel. Context interchange: new features and formalisms for the intelligent integration of information. *ACM Trans. Inf. Syst.*, 17(3):270–293, 1999.

[6] R. Hull. Managing semantic heterogeneity in databases: a theoretical prospective. PODS , pp. 51–61, ACM, 1997.

[7] N. Koul. Indus integration framework. http://code.google.com/p/indusintegrationframework/,2008.

[8] N. Koul. Indus learning framework. http://code.google.com/p/induslearningframework/, 2008.

[9] N. Koul, C. Caragea, V. Honavar, V. Bahirwani, and D. Caragea. Learning classifiers from large databases using statistical queries. Web Intelligence, 1:923–926, 2008.

[10] M. Lenzerini. Data integration: a theoretical perspective. PODS , pp. 233–246, ACM 2002.

[11] P. Shvaiko and J. Euzenat. A survey of schema-based matching approaches. *J. Data Semantics* , 4:146–171, 2005.

[12] U. Visser, H. Stuckenschmidt, H. Wache, and T. Vgele. Enabling technologies for interoperability. *TZI, University of Bremen*, pp. 35–46, 2000.

[13] H. Wache and H. Stuckenschmidt. Practical context transformation for information system interoperability. CONTEXT , pp. 367–380, Springer-Verlag, 2001

[14] P. Ziegler. *The SIRUP Approach to Personal Semantic Data Integration,*. PhD thesis, University of Zurich, 2007.