Fusionplex: Resolution of Data Inconsistencies in the Integration of Heterogeneous Information Sources

Philipp Anokhin Amihai Motro

Department of Information and Software Engineering
George Mason University

Abstract

Fusionplex is a system for integrating multiple heterogeneous and autonomous information sources, that uses *data fusion* to resolve factual inconsistencies among the individual sources. To accomplish this, the system relies on source *features*, which are meta-data on the quality of each information source; for example, the recentness of the data, its accuracy, its availability, or its cost. The fusion process is controlled with several parameters: (1) With a vector of feature weights, each user defines an individual notion of *data utility*; (2) with thresholds of acceptance, users ensure minimal performance of their data, excluding from the fusion process data that are too old, too costly, or lacking in authority, or data that are too high, too low, or obvious outliers; and, ultimately, (3) in naming a particular fusion function to be used for each attribute (for example, *average*, *maximum*, or simply *any*) users implement their own interpretation of fusion. Several simple extensions to SQL are all that is needed to allow users to state these resolution parameters, thus ensuring that the system is easy to use. Altogether, Fusionplex provides its users with powerful and flexible, yet simple, control over the fusion process. In addition, Fusionplex supports other critical integration requirements, such as information source heterogeneity, dynamic evolution of the information environment, quick ad-hoc integration, and intermittent source availability. The methods described in this paper were implemented in a prototype system that provides complete Web-based *integration services* for remote clients.

1 Introduction and Background

The subject of this paper is the *data integration problem*. Given a collection of heterogeneous and autonomous information sources, provide a system that allows its users to perceive the entire collection as a single source, query it transparently, and receive a single, unambiguous answer. *Heterogeneous* information sources are sources with possibly different data models, schemas, data representations, and interfaces. *Autonomous* information sources are sources that were developed independently of each other, and are maintained by different organizations, that may wish to retain control over their sources.

An important issue in the data integration problem is the possibility of *information conflicts* among the different information sources. The sources may conflict with each other at three different levels: (1) *Schema level*: The sources are in different data models or have different schemas within the same data model. (2) *Representation level*: The data in the sources is represented in different natural languages or different measurement systems. (3) *Data level*: There are factual discrepancies among the sources in data values that describe the same objects.

Each level of conflict can only be observed after the previous level has been resolved. That is, different attribute names in the schemas of different information sources must be mapped to each other before discrepancies in measurement systems can be observed. Similarly, different attribute values have to be within the same measurement system to conclude that these values indeed contradict each other.

The resolution of inconsistencies at the *schema level* is often performed in the process of interpreting user queries. Numerous approaches exist that define a *global* schema that encompasses the entire collection of sources, and then attempt to enumerate all the possible *translations* of a global query to a query over the source descriptions [18, 5, 16, 2, 12, 10, 17, 7, 24, 3, 14]. The optimal translation (optimal in either the cost of retrieval or the number of participating sources) is then chosen and materialized as an answer to the original query. To perform such a translation, one needs to *map* attributes in the schemas of *local* sources (the sources being integrated) to the attributes of the global schema.

A good number of approaches also resolve inconsistencies at the *representation level*. This is often done by providing means for aggregation and conversion of values to a uniform standard (e.g., [23, 18]).

Yet, to our knowledge, few systems deal with the issue of possible inconsistencies at the *data level*: the discrepancies among the values obtained from different sources for the same data objects.

HERMES [25] uses software modules called *mediators* to assemble global objects from local objects provided by different data sources. Yet, this system does not attempt to detect data conflicts. The mediators are created manually, and the mediator author must specify a conflict resolution method wherever a conflict might occur. Also, the available resolution methods are limited to a number of predefined policies.

Multiplex [20] both detects data inconsistencies and attempts to resolve them. However, inconsistency is approached at the "record level": inconsistency occurs when a global query results in two or more different *sets of records*. Multiplex then proceeds to construct an *approximation* of the true set of records, with a *lower bound* set of records (a *sound* answer) and an *upper bound* set of records (a *complete* answer). The two estimates are obtained from the conflicting answers through a process similar to voting. The lower bound set is contained in the upper bound set, and the true answer is estimated to be "sandwiched" between these two approximations.

A significant limitation of this approach to inconsistency resolution is that Multiplex regards two records as describing entirely different objects, even if they are "almost identical" (e.g., identical in all but one "minor" field). Consequently, when two such records are suggested by two information sources, there is no attempt to recognize that these might be two descriptions of the same object, and therefore no attempt to reconcile their conflicting values. The two records are simply both relegated to the upper bound estimate.

A few approaches exist that resolve data level conflicts based on the content of the conflicting data and possibly some probabilistic information that is assumed to be available. They either detect the existence of data inconsistencies and provide their users with some additional information on their nature (e.g., [1]), or they try to resolve such conflicts probabilistically by returning a *partial value*: a set of alternative values with attached probabilities [9, 26, 19, 6].

There is elegance in the probabilistic approach, because probabilistic values are more general than simple values, and the type of output of their resolution process is the same as the types of its input (probabilistic values). But the benefit of a probabilistic value to the database user is often in doubt. Another drawback is that probabilistic information must be provided for every data item in an information source. This is relatively rare, especially for Web-based sources. Fundamentally, rather than resolve inconsistencies by concluding *data* from the conflicting data values, probabilistic methods focus on concluding *probabilities* from the conflicting probability values. In other words, these approaches fuse probabilities not data, and can be viewed as managing *uncertainty*, rather than *inconsistency*.

A drawback common to all the methods discussed is that they disregard the fact that the information provided by their participating sources is often very different in its quality, reliability or availability. They ignore any such differences among their sources and simply assume that all sources are equally good.

Another common drawback is that these methods (with the possible exception of HERMES) do not provide their users with any control over the inconsistency resolution process. Users cannot influence the resolution process to arrive at the "best" answer (where "best" is defined by the particular user). Often, there are multiple ways to resolve data inconsistencies, and their suitability can only be judged by the user. For example, in one situation a conflict in a set of values may be deemed to be resolved best by choosing the most recent value; in another, with the same set of values, the most frequent value (the mode) may well be preferred.

The Fusionplex system that is the subject of this paper takes a different approach to inconsistency. The system is a development of the earlier Multiplex system discussed above (and adopts many of its definitions and basic principles). Thus, it is a general data integration system, with support for information source heterogeneity, dynamic evolution of the information environment, quick ad-hoc integration, and intermittent source availability.[1] But the focus of Fusionplex is the resolution of data inconsistencies by means of true *data fusion*.

[1]These aspects are explained in detail in Section 2.3.

The main principle behind Fusionplex is that "all data are not equal." The data environment is not "egalitarian," with each information source having the same qualifications (as assumed by Multiplex and other systems). Rather, it is a diverse environment, in which information providers have their individual advantages and disadvantages. Some data is more recent, whereas other is more dated; some data comes from authoritative sources, whereas other may have dubious pedigree; some data may be inexpensive to acquire, whereas other may be costlier. To resolve conflicts, Fusionplex looks at the qualifications of its individual information providers. Thus, it uses meta-data to resolve conflicts among data.[2]

Every Internet user is often confronted with the need to choose between alternatives: Which is the most trustworthy source? Which is the most reliable download site? Which is the least expensive newswire service? Some of these meta-data may be provided by the source itself (e.g., date of last update, cost), other meta-data may be obtained informally from other Internet users, and there are also Web sites that are dedicated to calculating the quality of information and services provided by other sites (often through the evaluations of fellow users). So it is not far-fetched to assume that in the near future, given the Internet's continuing, fast-paced expansion, such meta-data will become commonplace, possibly even in a standard format. In a more restricted information environment, comprising perhaps only a few dozen sources (perhaps with a focus on a particular subject, such as business or medicine), it is quite conceivable that the multidatabase administrator will assign meta-data scores to its sources, and will keep updating these scores.

Our term for such meta-data is information *features*. Examples of features include: (1) *Timestamp*: The time when the information in the source was validated. (2) *Cost*: The time it would take to transmit the information over the network and/or the money to be paid for the information. (3) *Accuracy*: Probabilistic information that denotes the accuracy of the information. (4) *Availability*: The probability that at a random moment the information source is available. (5) *Clearance*: The security clearance level needed to access them information.

The other guiding principle of Fusionplex is that inconsistency resolution is a process in which users must be given a voice. Depending on their individual preferences (which are subject to individual situations), users must be allowed to decide how inconsistencies should be resolved. Decisions are made in two phases: Some decisions are made ad-hoc at query-time, other decisions are more enduring and control all subsequent queries. One important query-specific decision is what constitutes "good" data. This decision is implemented by means of a vector of feature weights. In essence, this information constitutes this query's definition of *data quality* and allows the system to *rank* the competing values according to their *utility* to the user. Other query-specific parameters are thresholds of acceptance with which users can ensure minimal performance of the data, excluding from the fusion process data that are too old, too costly, or lacking in authority. It also allows users to reject data that are too high, too low, or obvious outliers. Yet another decision is the particular fusion

[2] A somewhat similar approach can be seen in the area of Internet search engines, where Google, one of the prominent search engines, weighs heavily the *authority* or *importance* of individual links in the ranking of its search results [11].

function to be used for a particular attribute; for example, *average*, *maximum*, or simply *any*. Several simple extensions to SQL are all that is needed to allow users to state these resolution parameters. Altogether, Fusionplex provides its users with powerful and flexible, yet simple, control over the fusion process. For more details on this project, see [4].

The formal model of this work is the relational model. This model and the fundamental concepts of multidatabases are reviewed in Section 2. These fundamental concepts have been extended to include features. The extensions involve modifications to the basic relational structures, the algebra, SQL, and the definition of multidatabases. They are discussed in Section 3.

Unlike query translation approaches that are unconcerned with inconsistencies, Fusionplex must retrieve *all* relevant data from the information sources. Section 4 describes how Fusionplex concludes which of the available information is relevant, and how this information is assembled in a "raw" answer. This intermediate product is termed *polyinstance* and it contains all the relevant data, including all possible inconsistencies. This polyinstance is the input to the resolution process. The first step in the resolution process is the identification of inconsistencies. In this process the tuples of the polyinstance are clustered in *polytuples*. The members of each polytuple are the different "versions" of the same information. Essentially, the inconsistency resolution process fuses the members of each polytuple in a single tuple. This fusion process is multi-phased and is based on information provided either in the query itself or in the currently prevailing resolution policies. The detection and resolution of inconsistencies are the subject of Section 5.

The methods described in this paper were implemented in a prototype system called Fusionplex. The architecture and features of Fusionplex are described in Section 6. Finally, Section 7 summarizes the contributions of this work and reviews several possible directions for further research.

2 Multidatabase Concepts

In this section we provide a brief overview of fundamental multidatabase concepts used in this paper. These concepts are adopted from [20].

2.1 Relational Databases

The relational data model was adopted for this work. This choice was motivated by the fact that the relational model is widely used and standardized, most production-quality database management systems implement this model, and most of the information sources that require integration are relational. Our terminology is mostly standard [21]. Throughout the paper, we use both relational algebra and SQL notations to describe views and queries.

Our resolution methodology could introduce *null values* into relations. This requires appropriate extensions to the relational model to determine the results of comparisons that involve nulls. Codd's three-valued logic [8] is adopted for this purpose. In this logic, comparisons that involve nulls evaluate to the value *maybe*. Different interpretations of such *maybe* values can be provided. In general, a *permissive* interpretation will map *maybe* values to *true*, and a *restrictive* interpretation will map *maybe* values to *false*. In each situation, the interpretation of choice will be stated.

2.2 Schema Mappings

Consider a database (D, d), where D is the database schema and d is its instance. Let D' be a database schema whose relation schemas are defined as views of the relation schemas of D. The database schema D' is said to be *derived* from the database schema D. Let d' be the database instance of D' which is the extension of the views D' in the database instance d. The database instance d' is said to be *derived* from the database instance d. Altogether, a database (D', d') is a *derivative* of a database (D, d), if its schema D' is derived from the schema D, and its instance d' is derived accordingly from the instance d.

Let (D_1, d_1) and (D_2, d_2) be two derivatives of a database (D, d). A view V_1 of D_1 and a view V_2 of D_2 are *equivalent*, if for every instance d of D the extension of V_1 in d_1 and the extension of V_2 in d_2 are identical. Intuitively, view equivalence allows one to substitute the answer to one query for an answer to another query, although these are different queries on different schemas.

Assume two database schemas D_1 and D_2, that are both derivatives of a database schema D. A *schema mapping* (D_1, D_2) is a collection of view pairs $(V_{i,1}, V_{i,2})$, where $V_{i,1}$ is a view of D_1, $V_{i,2}$ is a view of D_2, and $V_{i,1}$ is equivalent to $V_{i,2}$, for every i.

As an example, the equivalence of attribute *Salary* of relation schema *Employee* in database schema D_1 and attribute *Sal* of relation schema *Emp* in database schema D_2 is indicated by the view pair

$$(\pi_{Salary}\, Employee,\ \pi_{Sal}\, Emp)$$

As another example, given the schemas *Employee* = (*Name,Title,Salary,Supervisor*) in database schema D_1, and *Manager = (Ename, Level, Sal, Sup)* in database schema D_2, the following view pair indicates that the retrieval of the salaries of managers is performed differently in each database:

$$(\pi_{Name,Salary}\, \sigma_{Title=manager}\, Employee,\ \pi_{Ename,Sal}\, Manager)$$

Schema mappings are instrumental in our definition of multidatabases.

2.3 Multidatabases

Assume that there exists a hypothetical database that represents the real world. This ideal database includes the usual components of schema and instance, which are assumed to be perfectly correct. The relationships between actual databases and this ideal database are governed with two assumptions.

The Schema Consistency Assumption (SCA). All database schemas are *derivatives* of the real world schema. That is, in each database schema, every relation schema is a view of the real world schema. The meaning of this assumption is that the different ways in which reality is modeled are all correct; i.e., there are no *modeling errors*, only *modeling differences*. To put it in yet a different way, all intensional inconsistencies among the independent database schemas are reconcilable.

The Instance Consistency Assumption (ICA). All database instances are *derivatives* of the real world instance. That is, in each database instance, every relation instance is derived from the real world instance. The meaning of this assumption is that the information stored in databases is always correct; i.e., there are no factual *errors*, only different *representations* of the facts. In other words, all extensional inconsistencies among the independent database instances are reconcilable.

In this work we assume that the Schema Consistency Assumption *holds*, meaning that all differences among database schemas are reconcilable. These schemas are related through a *multidatabase* schema, which is yet another derivative of this perfect database schema. On the other hands, it is assumed that the Instance Consistency Assumption *does not hold*, allowing the possibility of irreconcilable differences among database instances. This means that the database instances are *not* assumed to be derivatives of the real world instance.

Formally, a *multidatabase* is

1. A *global* schema D.
2. A collection $(D_1, d_1), \ldots, (D_n, d_n)$ of *local* databases.
3. A collection $(D, D_1), \ldots, (D, D_n)$ of schema mappings.

The first item defines the schema of a multidatabase, and the second item defines the local databases in the multidatabase environment. The third item defines a mapping from the global schema to the schemas of the local databases. The schemas D and $D_1, \ldots, D_n$ are assumed to be derivatives of the real-world schema, but the instances $d_1, \ldots, d_n$ are not necessarily derivatives of the real-world instance (see Figure 1). Note that there is no instance for the global database, and therefore a multidatabase is said to be a *virtual database*.

The "instance" of a multidatabase consists of a collection of global view extensions that are available from the local databases. Specifically, the views in the first position of the

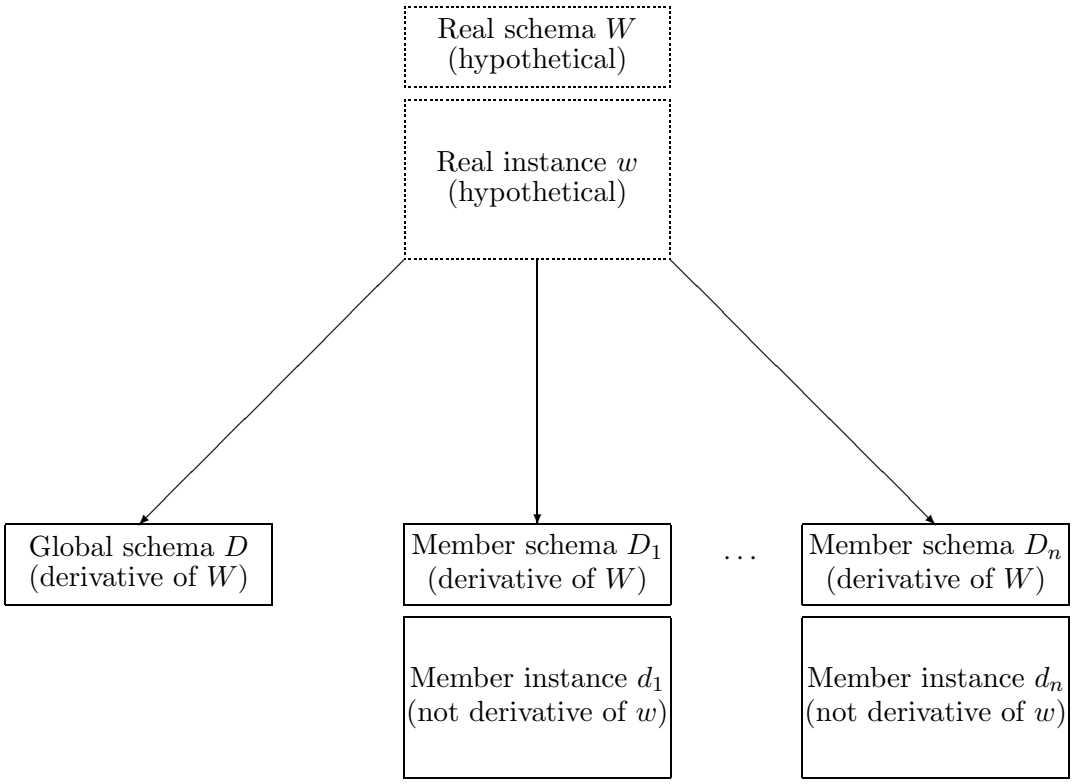

Figure 1: Consistency assumptions in multidatabases.

schema mappings specify the "contributed information" at the global level, and the views in the second position describe how these contributions are materialized.

As defined earlier, schema mappings allow to substitute certain views in one database with equivalent views in another database. In a multidatabase, the former database is the global database, and the latter is a local database.

The above definition of multidatabases provides four *degrees of freedom*, which reflect the realities of multidatabase environments.

First, the mapping from D to the local schemas is not necessarily *total*; i.e., not all views of D are expressible in one of the local databases (and even if they are expressible, there is no guarantee that they are mapped). This models the dynamic situation of a multidatabase system, where some local databases might become temporarily unavailable. In such cases, the corresponding mappings are "suspended," and some global queries might not be answerable in their entirety.

Second, the mapping is not necessarily *surjective*; i.e., the local databases may include views that are not expressible in D (and even if they are expressible, there is no guarantee that they are mapped). For example, a large database may share only one or two views with the multidatabase. This enables quick ad-hoc (i.e., not necessarily comprehensive) integration.

Third, the mapping is not necessarily *single-valued*; i.e., a view of D may be found in several local databases. This models the realistic situation, in which information is found in several overlapping databases, and provides a formal framework for dealing with multidatabase inconsistency. Since there is no assumption that the Instance Consistency Assumption holds, the local instances need not be derived from a single instance. Therefore, the fact that view pairs (V, V_1) and (V, V_2) participate in schema mapping of a multidatabase does not imply that the extensions of V in the local databases are identical.

Fourth, while the definition appear to require that the local databases comply with the relational model, in practice they need not be relational, and the views in the second position of the schema mappings need not be relational model expressions. The only requirement is that they compute tabular answers. In other words, this definition allows for heterogeneity in the participating sources. The relational model provides only a convenient global view, and a communication protocol.

3 Formal Framework

To achieve our goal of resolving data conflicts, we introduce only one significant addition to the model described in Section 2, which we call *features*. This addition requires that we extend the notations for relational algebra queries (Section 3.2) and SQL queries (Section 3.3) as well.

3.1 Features

With the growth of the Internet, the number of alternative sources of information for most applications has increased enormously. To choose the most suitable source among the alternatives, users often evaluate information *about* the sources. These meta-data — whether provided by the sources themselves, or by third-party sites dedicated to the ranking of information sources — help users judge the suitability of each source for the intended use. Examples of such meta-data include:

- **Timestamp**: The time when the information in the source was validated.
- **Cost**: The time it would take to transmit the information over the network and/or the money to be paid for the information.
- **Accuracy**: Probabilistic information that denotes the accuracy of the information.
- **Availability**: The probability that at a random moment the information source is available.
- **Clearance**: The security clearance level needed to access the information.

These meta-data are referred to as source *features*. Each feature is associated with a domain of possible values, and a total order is assumed to be defined on the domain. For each information source that possesses a particular feature, a value from that domain is available. For example, the domain of *availability* could be the interval [0, 1], the values of *cost* could range between 0 and M, where M is an arbitrary number, and the domain of *clearance* could be {*top-secret*, *secret*, *confidential*, *unclassified*}.

To facilitate comparisons between different feature values, all features are *normalized*. Each feature value is linearly mapped to a number in the interval [0, 1]. The mapping is done so that high feature values are always more desirable that low values; that is, the worst feature value is mapped to 0, and the best to 1. For example, higher *availability* value means higher probability of the source being available. Similarly, higher *timestamp* value means the data is more recent. But notice that higher *cost* value means the data is cheaper to obtain.

Every information source has a set of features associated with it, and in practice, different sources may have different features. Therefore, a *global* set of features $\mathcal{F}$ is defined for the entire multidatabase, as the *union* of the features of the participating information sources. Each source feature set is then augmented to the features in $\mathcal{F}$ by adding null values for the features it does not possess. For example, assume the global set of features is $\mathcal{F}$ ={*timestamp*, *cost*, *availability*}. An information source with the current timestamp, zero cost and no data about availability would have the feature set {*timestamp*=1, *cost*=1, *availability*=*null*}.

Our definition of features associates the same feature value with the *entire* information source. That is, in this work all features are assumed to be *inherited* by all individual tuples and all their attribute values. For example, in case of *timestamp*, it is assumed that the source-wide *timestamp* value is also the *timestamp* value for every attribute in every tuple of the source. Clearly, this assumption is restricting as it implies that the data in every source are homogeneous with respect to every feature. However, in situations where an information source is heterogeneous with respect to a given feature, it might be possible to partition the source into several disparate parts that would be homogeneous with respect to that feature. These parts would consequently be treated as separate information sources.

We now update the definition of multidatabases offered in Section 2. The only change is in the definition of schema mappings. Recall that mappings were made of pairs, where the first element of each pair was a view of the global database D, and the other a view of a local database D_i. These pairs are now extended to triplets with the addition of the set of source features F. With this third element, every information source now provides its meta-data along with its data.

3.2 Extended Relational Model and Algebra

To take advantage of feature meta-data in the processing of global queries, we extend the standard relational model. We offer extensions to the definitions of relation schemas, relation instances, and the relational algebra operations that manipulate these structures.

Each relation schema is extended with all the features in $\mathcal{F}$. Assuming $\mathcal{F} = F_1, \ldots, , F_k$, a relation scheme $R = (A_1, \ldots, A_m)$ is now extended to $R = (A_1, \ldots, A_m; F_1, \ldots, F_k)$. Correspondingly, the tuples in each relation instance are extended with the appropriate feature values. Recall that feature columns of source instances have the same value for all their tuples, and that *null* is an appropriate value for database attributes as well as features.

The extension to the relational algebra consists of modifications to three basic operations (selection, projection, and Cartesian product), and two new operations: feature-select and resolve. The other two basic operations (union and difference) remain unchanged.

The extended selection operation allows only predicates that involve database attributes (not features). In the case of null attribute values, the selection predicate has two interpretations: a *restrictive* interpretation in which it evaluates to $false$ and *permissive* interpretation in which it evaluates to $true$. The extended projection operation is also limited to relation attributes (it always retains all the feature columns).

The extended Cartesian product *concatenates* the database values of the participating relations, but *fuses* their feature values. The fusion takes the *minimum* value. Recall that higher feature values imply better performance. Thus, our worst case approach is intended to guarantee a minimal performance of the combined information. When one of the two feature values is *null*, the resulting feature value is set to *null* as well, a choice consistent with the worst case approach. To wit, a null value may be interpreted as any value in the interval [0,1]. Assume now that the other value is α. The worst case approach implies that their fusion is in the interval $[0,\alpha]$. Although more specific, this information is still represented by the value *null*.

The first new operation ω is for selecting tuples by feature values. Let ψ be a selection predicate over the features of an (extended) relation R with (extended) instance r. Then $\omega_\psi(r)$ is the set of (extended) tuples from r that satisfy ψ. In the case of null feature values, ψ has two interpretations: a *restrictive* interpretation in which ψ evaluates to $false$ and *permissive* interpretation in which ψ evaluates to $true$.

The other new operation ρ resolves inconsistencies in relation instances. This operation is discussed in Section 5.4. We note, however, that the operation depends on a vector of feature weights $\vec{w}$ as well as a set of system-specified resolution policies. Thus, $\rho_{\vec{w}}(r)$ transforms r to an instance that is free of inconsistencies.

The following example illustrates these extensions to the relational algebra. The relations are *R=(Salary, EmpID)* and *S=(ID,Name)* and the features are *timestamp* (abbreviated *time*), *cost*, and *availability* (abbreviated *avail*). The example follows the construction of an answer to the query "Salaries and names of employees, where cost is not less then 0.5, and the importance of timestamp and cost are 0.3 and 0.7, correspondingly. The (extended) relational algebra expression for this query is $\pi_{Salary,Name}\, \rho_{(0.3,0.7,0)}\, \omega_{cost \geq 0.5}\, \sigma_{EmpID=ID}\, (R \times S)$. In the interest of generality, the feature values in r and s are not uniform; this may be the case if these relations were created from different sources. Admittedly, at this point the ρ operator is still unexplained, but note that it identifies *Johnson* and *Johansen*.

r

Salary	*EmpID*	*time*	*cost*	*avail*
10000	1002	1	0.5	*null*
50000	1003	1	1	0.5
20000	1001	0.8	0.2	*null*

s

ID	*Name*	*time*	*cost*	*avail*
1002	*Johnson*	0.7	1	*null*
1002	*Johansen*	0.8	0.8	*null*
1001	*Nguyen*	1	1	0.1
1004	*Smith*	1	*null*	1

$t_1 = r \times s$

Salary	*EmpID*	*ID*	*Name*	*time*	*cost*	*avail*
10000	1002	1002	*Johnson*	0.7	0.5	*null*
50000	1003	1002	*Johnson*	0.7	1	*null*
20000	1001	1002	*Johnson*	0.7	0.2	*null*
10000	1002	1002	*Johansen*	0.8	0.5	*null*
50000	1003	1002	*Johansen*	0.8	0.8	*null*
20000	1001	1002	*Johansen*	0.8	0.2	*null*
10000	1002	1001	*Nguyen*	1	0.5	*null*
50000	1003	1001	*Nguyen*	1	1	0.1
20000	1001	1001	*Nguyen*	0.8	0.2	*null*
10000	1002	1004	*Smith*	1	*null*	*null*
50000	1003	1004	*Smith*	1	*null*	0.5
20000	1001	1004	*Smith*	0.8	*null*	*null*

$t_2 = \sigma_{EmpID=ID}(t_1)$

Salary	*EmpID*	*ID*	*Name*	*time*	*cost*	*avail*
10000	1002	1002	*Johnson*	0.7	0.5	*null*
10000	1002	1002	*Johansen*	0.8	0.5	*null*
20000	1001	1001	*Nguyen*	0.8	0.2	*null*

$t_3 = \omega_{cost \geq 0.5}(t_2)$

Salary	*EmpID*	*ID*	*Name*	*time*	*cost*	*avail*
10000	1002	1002	*Johnson*	0.7	0.5	*null*
10000	1002	1002	*Johansen*	0.8	0.5	*null*

$t_4 = \rho_{(0.3,0.7,0)}(t_3)$

Salary	*EmpID*	*ID*	*Name*	*time*	*cost*	*avail*
10000	1002	1002	*Johansen*	0.8	0.5	*null*

$t_5 = \pi_{Salary,Name}(t_4)$

Salary	*Name*	*time*	*cost*	*avail*
10000	*Johansen*	0.8	0.5	*null*

3.3 Queries

For practical purposes, the relational algebra extensions defined earlier have also been incorporated into SQL. The extended SQL query statement is

select [**restrictive**] ...
from ...
where ...
using ***use_stmt***
with ***weight_stmt***
;

The extensions to the five standard relational algebra operations require no additional syntax (except for the optional keyword **restrictive**, to be explained below). The two new operations are handled by two additional clauses: **using** and **with**. The **using** clause is for specifying the desired features of the answer and corresponds to the new relational operation ω_{ψ}. In analogy with the **where** clause that restricts the answer set with a condition on *attributes*, the **using** clause restricts the answer set with a condition on *features*. The syntax of the **using** clause is

use_stmt ::= ***feature_ name comparison feature_value***
[**and** ***feature_name comparison feature_value***]
...

The **with** clause corresponds to the new relational operation $\rho_{\vec{w}}$. With this clause users assign weights to features. The features not mentioned in the **with** clause are assigned a weight of 0. The syntax of the **with** clause is

weight_stmt ::= ***feature_ name*** **as** ***feature_weight***
[, ***feature_ name*** **as** ***feature_weight***]
...

When **restrictive** is present, all comparisons to null values, either in attribute columns (the **where** clause) or in feature columns (the **using** clause) evaluate to $false$; otherwise, they are *true*.

Expressed using the extended SQL statement, the previous query is

select *Salary*, *Name*
from r, s
where $EmpID = ID$
using $cost \geq 0.5$
with *timestamp* **as** 0.3, *cost* **as** 0.7
;

All user queries are assumed to be restricted to relational algebra expressions that contain the aforementioned seven relational algebra operations (projection, positive selection, union, difference, Cartesian product, feature selection and resolution). Both SQL and relational algebra notations will be used for query formulation interchangeably.

4 Data Collection and Assembly

4.1 Contributions and Query Fragments

As discussed in Section 1, the most common approach to data integration is *query translation*. In this process, each user query, expressed in terms of the *global schema*, is translated to a query over the *global views* (the views that define the information available from the sources). The challenge in this translation is that some of the data requested by the user query may not be in *any* of the available information sources (although it is described in the global schema), or it may be in *several* of them.

Most translation methods deal with the former problem by translating the given query to the maximal possible query over the global views (i.e., a maximal subview of the original translation). Assuming that no data inconsistencies exist, the latter problem is solved by selecting one of the sources and ignoring the others. In the following, we extend this query translation method to handle data inconsistencies.

Recall that each schema mapping is a set of triplets. We amalgamate the triplets of the different mappings in a single set, and denote each triplet (V, URL, F), where V is a global view, URL is an expression that is used to materialize this view from one of the participating information sources, and F is the features of this source. Each such triplet is called a *contribution* to the virtual database.

Like most translation methods, we restrict contribution views (the V of each triplet) to relational algebra expressions that include only projections, selections and joins (PSJ views). Additionally, we assume that they do not contain comparisons *across* the view relations.

Obviously, not all contributions are needed for every query. To determine the contributions that are relevant to a given query, the following two-step process is applied. First, the sets of attributes of the query and a contribution are intersected. If the intersection is empty, the contribution is deemed not relevant. Next, the selection predicates of the query and the contribution are conjoined. If the resulting predicate is not *false*, then the contribution is considered relevant to the query.

From each relevant contribution we derive a unit of information suitable for populating the answer to the query. Such units are termed *query fragments*. Intuitively, to obtain a query fragment from a contribution C one needs to remove from C all tuples and attributes that are not requested in the query, and to add null values for the query attributes that are missing from the contribution.

Figure 2 illustrates the construction of query fragments for a query Q from two relevant contributions: $C_1 = (V_1, URL_1, F_1)$ and $C_2 = (V_2, URL_2, F_2)$. The left part of the figure shows v_1 and v_2, their intersection (the shaded area), and the "ideal" answer q (the dashed box). The shaded rectangle represents an area of possible conflicts. The right part of the figure shows the two resultant query fragments, q^{C_1} and q^{C_2}. The area marked *null* contains null values as a result of V_2 not including all the attributes of Q.

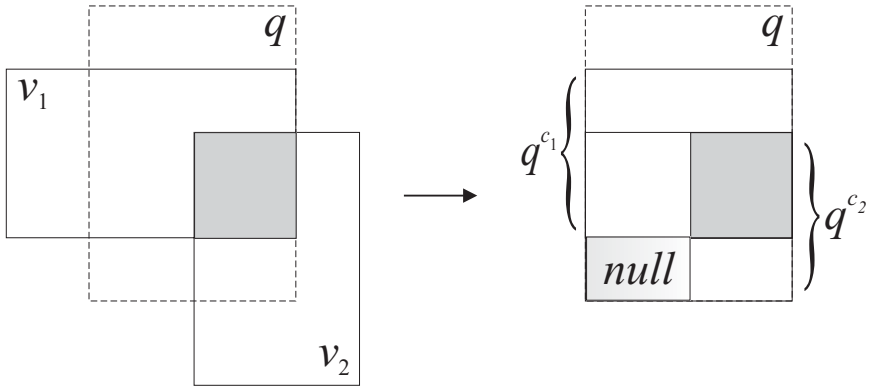

Figure 2: Constructing fragments for query Q from contributions C_1 and C_2.

At times, a source may provide values for one global attribute, yet through its definition, values of another global attribute may be inferred. Let V be a view. Assume that V includes the attribute A_i but removes the attribute A_j, and assume that its selection predicate contains the equality $A_i = A_j$. Clearly, although this contribution does not provide the attribute A_j, it can be enhanced automatically to include it. Therefore, we add to V the attribute A_j, and we add to its instance v a column of A_j values which replicates the A_i values. This process, which is repeated for all equalities in V's selection predicate, is called an *enhancement* of V and its result is denoted $\overline{V}$ (the enhanced instance is denoted $\vec{v}$).

This discussion is summarized in this definition of query fragments:

1. Assume a query $Q = \pi_{A_1,\ldots,A_q}\, \omega_\psi\, \sigma_\phi\, (R_1 \times \ldots \times R_p)$.
2. Assume a contribution $C = (V, URL, F)$.
3. Let $\overline{V}$ be the enhanced contribution view, and let $\overline{v}$ be its instance.
4. Denote $Y = \{A_1, \ldots, A_q\} - \overline{V}$. Y is the difference between the schemas of the query and the enhanced contribution view. Let y be an instance of Y that consists of a single tuple t_{null} composed entirely of null values.
5. Let $\mathcal{F}$ be the global feature set, and let p be the tuple of feature values of C. Let $\overline{p}$ be the extension of p to the global set $\mathcal{F}$ with null values.
6. A *query fragment* derived from C for a query Q, denoted Q^C, is a view definition whose schema is $\{A_1, \ldots, A_q\} \cup \mathcal{F}$, and for every instance v of V, the instance of Q^C, denoted q^c, is $\omega_\psi\, \sigma_\phi\, (\pi_{A_1,\ldots,A_q}(\overline{v}) \times y \times \overline{p})$.

The concepts of contribution, enhancement and query fragment are illustrated in the following example. Assume a multidatabase relation with five attributes and three features: $R = (A, B, C, D, E; timestamp, cost, availability)$, and consider a query

$$Q = \pi_{A,B,D,E}\, \omega_{timestamp>0.5}\, \sigma_{C<10}\, R$$

Further, assume a contribution $C = (V, URL, F)$ where

$$\begin{aligned} V &= \pi_{A,B,C}\, \sigma_{B>0 \wedge C=E}\, R \\ F &= (timestamp = 0.7, cost = 0.8, availability = 1) \\ URL &= \text{“http} : //\text{www.aname.com/smth.cgi?action} = \text{retrieve\&ID} = 517\text{”}. \end{aligned}$$

and let the instance v of V be

A	B	C
1	2	7
2	2	11
3	4	4

First, this instance is extended to include the features:

A	B	C	*time*	*cost*	*avail*
1	2	7	0.7	0.8	1
2	2	11	0.7	0.8	1
3	4	4	0.7	0.8	1

Next, because the selection predicate of V includes an equality $C = E$, V is enhanced to

A	B	C	E	*time*	*cost*	*avail*
1	2	7	7	0.7	0.8	1
2	2	11	11	0.7	0.8	1
3	4	4	4	0.7	0.8	1

Finally, the query fragment q^c formed from the contribution C is

A	B	D	E	*time*	*cost*	*avail*
1	2	*null*	7	0.7	0.8	1
3	4	*null*	4	0.7	0.8	1

Note that the column D contained in Q's projection set is not available from the contribution, and is therefore presented as a column of null values.

From each relevant contribution a single query fragment is constructed. Some of these fragments may be empty. The union of all non-empty query fragments is termed a *polyinstance* of the query. Intuitively, a polyinstance encompasses all the information culled from the data sources in response to a user query.

From a user's perspective, a query against the virtual database is supposed to return single consistent answer. By resolving all inconsistencies, the resolve operation (to be described in Section 5) will convert this polyinstance to a regular instance.

Recall that query translation is a process in which a query over the global relations is translated to a query over the global views. Since each query fragment is a view over a global view, the polyinstance is a view over the global views as well. Hence, the construction of the polyinstance is simply a query translation process. Note that in the absence of data conflicts (i.e., when the ICA holds), this polyinstance is equivalent to the output of a conventional query translation algorithm.

4.2 Defining Data Inconsistency

Before addressing issues of data inconsistency, it is necessary to understand the meaning of this term, because its common use in the literature on information integration and the wide range of concepts it describes can be misleading.

Often, the term data inconsistency is used to describe *schematic* differences among heterogeneous databases. As an example, assume a global relation *Employee* = (*Name, Salary, Sex*) and two local sources S_1 = (*Name, Income, Status*) and S_2 = (*LastName, Salary*) that describe female and male employees correspondingly. The local attributes *Name* and *LastName* should be mapped to the global attribute *Name*, the local attributes *Income* and *Salary* should both be mapped to *Salary*, and the values of the global attribute *Sex* should be inferred from the source descriptions. Clearly, the inconsistencies resolved here are schematic.

The term data inconsistency is also used to describe *representation* inconsistencies. Even when global and local attributes are mapped successfully, they might still conflict in their representations of their data [23, 18]. A classic example is currency. One source can contain currency expressed in USA dollars, and the other in Swedish kronors. Another example of difference in data representations is when one attribute is calculated as a sum of two others. Although this type of conflict is seemingly based on the content and not the schema, it can be resolved by a simple conversion and therefore does not constitute a factual discrepancy among the sources.

Common to the approaches that handle the above two types of inconsistency is an implicit assumption that the *contents* of all the information sources are mutually consistent [20].

This is further illustrated in the following example. One source of data may be in English and contain the attribute *EyeColor* and specify an individual's eye color as "dark." Another

data source may be in Russian and contain the attribute *CvetGlaz* (eye color in Russian) and specify the eye color of *the same* individual as "korichnevyj" (brown in Russian). The two values are expressed in different languages and one subsumes the other, but the sources do not *contradict* one another. This paper addresses situations similar to one in which the former source would specify the eye color of this individual as "blue."

More formally, a data inconsistency exists when two objects (or tuples in the relational model) obtained from different information sources are identified as *versions* of each other (i.e., they represent the same real-world object) and some of the values of their corresponding attributes differ. Note that such identification is only possible when both schema incompatibilities and representation differences have been resolved. The process of data integration requires two steps: inconsistency *detection* and inconsistency *resolution*.

5 Data Inconsistency Detection and Resolution

5.1 Data Inconsistency Detection

Data inconsistency detection begins by identifying tuples of the polyinstance that are versions of each other. Several techniques have been suggested for identifying tuples or records originating from multiple sources [13, 22, 15]. The work described here assumes that any one of these methods. For simplicity, we assume that identification is by *keys*.

We assume that each global relation is fitted with a key. Subsequently, to find the tuples in the polyinstance that are versions of each other, one needs to construct the key of the answer and use it to cluster the polyinstance. The resulting clusters are termed *polytuples*. Each polytuple may be visualized as a table:

SSN	*Name*	*Age*	*Salary*	*time*	*cost*	*avail*
326435218	*Smithson*	38	75000	0.8	0.5	1
326435218	*Smith*	35	*null*	0.7	0.2	*null*
326435218	*Schmidt*	35	77000	0.7	0.8	1

It is possible to improve the efficiency and accuracy of polytuple clustering, by considering only "horizontal slices" of the polyinstance in which there is possible contention (recall the shaded areas in Figure 2). These slices may be determined from the selection predicates of the individual contributions. For example, if ϕ_1 and ϕ_2 are the selection predicates of two contributions, then only the slice defined by $\phi_1 \wedge \phi_2$ might have conflicts. In other words, the membership of polytuples cannot span across different slices. For brevity, we do not address this refinement here, and additional details may be found in [4].

Consider now a contribution $C = (V,\ URL,\ F)$ and the view instance v materialized from this contribution. There is always a possibility that v is not an acceptable instance of V. For example, V may be a Cartesian product of two global relations, but the set of tuples

materialized from the URL could not possibly be a Cartesian product. As another example, V may involve a selection $A = a$, yet v includes in its column A values different than a. In this paper, a contribution v that contradicts its definition V is ignored.

Once data inconsistencies have been detected, they need to be resolved; i.e., every polytuple should be reduced to a single tuple. This process is performed in two passes. In the first pass (Section 5.2), the members of each polytuple are ranked and purged according to user-specified preferences. In the second pass (Section 5.3), in each polytuple, in each attribute, remaining values are purged and then fused in a single value. Similarly, in each polytuple, in each feature, different values are fused in a single value. This procedure is the actual definition of the extended relational algebra operation *resolve*, introduced earlier.

5.2 Utility Function

Recall that the resolve operation requires the specification of a vector of feature weights $\vec{w}$. With these weights, users prescribe the relative importance of the features in the resolution process. To use this information, a *utility function* is calculated for each member of every polytuple. Assume the user assigns weights $w_1, \ldots, w_k$ to the features $F_1, \ldots, F_k$. Then a member with feature values $f_1, \ldots, f_k$ receives utility $u = \sum_{i=1}^{k} w_i f_i$. These utility values are used to *rank* the members of each polytuple. Using a pre-defined *utility threshold*, members of insufficient utility are discarded. Utility is calculated using only the features that are non-*null* for all members of the polytuple.

Since a polytuple may have several members of acceptable utility, this process is not sufficient for resolving polytuples. Actual resolution is achieved in a second pass, described next.

5.3 Resolution Policies

The resolution of inconsistencies among different values can be based either on their features, such as *timestamp*, *cost* or *availability* (*feature-based* resolution policies), or on the data themselves (*content-based* policies). Ideally, a conflict resolution policy should be provided whenever data inconsistency is possible. However, to keep the number of policies under control, a policy will be defined per each *global attribute*.

Consequently, within each polytuple, data inconsistency is resolved in each attribute separately (i.e., each column of values in the polytuple is fused in a single value). To perform such resolution, the polytuple is first split into several smaller structures: *mono-attribute polytuples*, each consisting of the polytuple key and one additional attribute. These structures are obtained from the polytuple by projections. Figure 3 shows these mono-attribute polytuples for the previous example. Next, each mono-attribute polytuple is resolved with a single attribute value and its corresponding unique feature values. Finally, the mono-attribute tuples are joined back together, resulting in a single tuple.

SSN	*Name*	*Salary*	*time*	*cost*	*avail*
326435218	*Smithson*	75000	0.8	0.5	1
326435218	*Smith*	*null*	0.7	0.2	*null*
326435218	*Schmidt*	77000	0.7	0.8	1

↓

SSN	*Name*	*time*	*cost*	*avail*
326435218	*Smithson*	0.8	0.5	1
326435218	*Smith*	0.7	0.2	*null*
326435218	*Schmidt*	0.7	0.8	1

SSN	*Salary*	*time*	*cost*	*avail*
326435218	75000	0.8	0.5	1
326435218	*null*	0.7	0.2	*null*
326435218	77000	0.7	0.8	1

Figure 3: A polytuple split into two mono-attribute polytuples.

As already noted, resolution is performed in each mono-attribute polytuple separately. This process consists of two steps. First, some of the rows in the mono-attribute polytuple are *eliminated*, based on attribute or feature values. Then, the remaining rows are *fused*, producing single values for both the attribute and the features. Correspondingly, a resolution policy is defined as a sequence of *elimination functions*, followed by a *fusion function*.

Elimination functions are either content-based or feature-based. Examples of elimination functions are *min* and *max*. In the previous example, consider the mono-attribute polytuple *Salary*. Possible eliminations include $max(timestamp)$, $max(availability)$, $min(cost)$, and $max(Salary)$. The former three are feature-based, whereas the latter is content-based. But other functions may also be used; for example, $over\_average$, $top\_five\_percent$ and $within\_standard\_deviation\_of\_the\_mean$. Each function is applied in its turn (according to its place in the sequence) to the corresponding column of the mono-attribute polytuple, eliminating all but one value. However, this step can still result in multiple values (e.g., several *Salary* values may share the maximal availability). Such polytuples are handled subsequently by the fusion function.

Fusion functions are always content-based. The fusion function is applied to the values of the global attribute and to the values of the features, resulting in a single resolved value for each of them. Examples of fusion functions are *any* and *avg*. Consider the mono-attribute polytuple *Salary* from the previous example. Applying *any* gives quick resolution by choosing a tuple at random, for example (326435218, 7000; 0.7, 1). Applying *avg* results in the *Salary* value 76000, but average feature values may not reflect the feature values of 76000. As with the extended Cartesian product, a conservative approach is used that adopts the minimum of the feature values, yielding (326435218, 76000; 0.7, 1) as the resolved mono-attribute polytuple. This guarantees that the resulting features are *at least* those specified. Functions other then *any* or *avg* may also be used. For example, *mode*, *average_without_extreme_values*, or any other function of the conflicting values.

To give users full control over the process of inconsistency resolution, we provide a powerful and flexible resolution statement. This statement implements the full set of features discussed in this section:

for	A_i
[keep [restrictive]	$e_1(F_1)$**, ...,** $e_n(F_n)$**]**
fuse	f

Here, A_i is a global attribute name, e_1, ..., e_n is a sequence of elimination functions for this attribute and f is a content-based fusion function. Each F_i is a feature, and $e_i(F_i)$ indicates feature-based elimination (e.g., *max*(*timestamp*) eliminates all but the most recent value). If F_i is not given, the elimination is content-based (e.g., *min*() retains the smallest value). Elimination functions are applied in the order in which they appear in the **keep** clause. Note that the entire **keep** clause is optional. Frequently, multiple attributes would share the same resolution policy. To facilitate the specification, the resolution statement allows multiple attributes in its **for** clause: **for** $A_{i_1}, \ldots, A_{i_m}$.

The handling of null values during the elimination phase is controlled by the **restrictive** keyword. The decision whether the value *null* satisfies an elimination function (either content-based or feature-based) is similar to the decision on *null* comparison in the **using** and **where** clauses: If the keyword **restrictive** is present, a *null* is assumed to not satisfy the elimination function; otherwise, it is assumed to satisfy it. Regardless, in the fusion phase, attribute values that are *null* are discarded, under the common assumption that, whenever available, non-null values should be preferred (if all the attribute values are *null*, then the fusion value is *null*). When determining the features of the fusion value, it may be necessary to take the minimum of several feature values (for example, if the function was *avg*). In that case, if any of these values is *null*, then the feature of the fusion value is *null*.

A resolution statement is required for each attribute of the global schema. These statements may be supplied by the system, a group of domain experts, or defined by users at run-time.

When every mono-attribute polytuple has been resolved, the results are joined in a single tuple. Recall that the (extended) Cartesian product used in this join fuses the alternative feature values in their minimum. To illustrate, consider this example with attributes K, A and B (K is key) and features F, G, and H, and let $(k, a; f_a, g_a, h_a)$ and $(k, b; f_b, g_b, h_b)$ be the resolutions of two mono-attribute polytuples. The final tuple is $(k, a, b; min(f_a, f_b), min(g_a, g_b), min(h_a, h_b))$.

This resolution process yields a single tuple for every polytuple. Altogether, the polyinstance is reduced to a simple instance. This set of tuples is then presented to the user as an inconsistency-free answer to the query.

The following three examples illustrate different kinds of resolution policies. The examples all use the query

$$\pi_{Name,Age,Salary}\, \sigma_{Position=''Manager''}\, Employee.$$

and the polytuple

SSN	*Name*	*Age*	*Salary*	*time*	*cost*	*avail*
326435218	*Smithson*	38	75000	0.8	0.5	1
326435218	*Smith*	35	*null*	0.7	0.2	*null*
326435218	*Schmidt*	35	77000	0.7	0.8	1

1. Content-based policy. Consider this resolution policy that chooses any *Name*, the average *Age* and the minimal *Salary*:

for	*Name*			**fuse**	*any*
for	*Age*			**fuse**	*avg*
for	*Salary*	**keep**	*min*()	**fuse**	*any*

This policy uses only attribute values for elimination. Its result is

SSN	*Name*	*Age*	*Salary*	*time*	*cost*	*avail*
326435218	Smith	36	75000	0.7	0.2	1

2. Feature-based policy. Consider this resolution policy that chooses the *Name* that is most recent, and the *Age* and *Salary* that are least costly:

for	*Name*	**keep**	*max*(*timestamp*)	**fuse**	*any*
for	*Age*	**keep**	*max*(*cost*)	**fuse**	*avg*
for	*Salary*	**keep**	*max*(*cost*)	**fuse**	*any*

Note that least costly translates to maximal cost. This policy uses only feature values for elimination. Its result is

SSN	*Name*	*Age*	*Salary*	*time*	*cost*	*avail*
326435218	Smithson	35	77000	0.7	0.2	1

3. Mixed policy. Consider this resolution policy that chooses the *Name* that is least recent, the lowest *Age* and the *Salary* that is least costly:

for	*Name*	**keep**	*min*(*timestamp*)	**fuse**	*any*
for	*Age*	**keep**	*min*()	**fuse**	*avg*
for	*Salary*	**keep**	*max*(*cost*)	**fuse**	*random*

This policy uses both attribute values and feature values for elimination. Its result is

SSN	*Name*	*Age*	*Salary*	*timestamp*	*cost*	*availability*
326435218	Schmidt	35	75000	0.7	0.2	1

5.4 The Overall Resolution Procedure

Our inconsistency resolution methodology is summarized in the following procedure for resolving data inconsistencies. The procedure begins with a polyinstance: the "raw" answer, comprising the union of the query fragments that are extracted from the information sources, clustered into polytuples. Its output is a simple instance: the final, inconsistency-free answer to the query.

Input:

1. A feature-based selection predicate ψ, obtained from the query's **using** clause.
2. A vector of feature weights $\vec{w}$, obtained from the query's **with** clause.
3. A *utility threshold* $0 \leq \alpha \leq 1$ to be used in conjunction with the weights.
4. A resolution policy for every attribute of the global schema.

The first two items are specified by the user as part of the query. The last two items are assumed to be pre-specified (e.g., provided by either administrators or users).

Procedure:

1. In each polytuple, remove members that do not satisfy the predicate ψ.
2. In each polytuple, calculate the utility of each remaining member: $u = \sum_{i=1}^{k} w_i f_i$, and rank the members by their utility. Let u_0 denote the highest utility in the polytuple. Discard members whose utility is less than $(1 - \alpha)u_0$.
3. Decompose each polytuple to its mono-attribute polytuples.
4. In each mono-attribute polytuple, apply the resolution policy for that attribute: Eliminate attribute values according to the **keep** clause, and fuse the remaining values according to the **fuse** clause.
5. Join the resulting mono-attribute tuples in a single tuple, using the (extended) Cartesian product. This assigns each feature the minimum of the values of its mono-attribute components.
6. The tuples thus obtained for each polytuple comprise the final answer to the query.

Unless the keyword **restrictive** is specified in the query, polytuples may have members with null feature values. In such cases, features that include nulls are ignored, so that ranking is based on the features that are available for *all* members. For example, assume the utility function $u = 0.3 \cdot timestamp + 0.5 \cdot cost + 0.2 \cdot availability$ and the previous polytuple. The non-*null* features are *timestamp* and *cost*, and the utility function is changed to $u = 0.3 \cdot timestamp + 0.5 \cdot cost$.

If a global attribute is non-numeric (i.e., of type *string*), then the choice of possible elimination and fusion function is more limited. Some of the common functions may have to be specifically defined to operate on non-numerical values (for example, *min* or *max* may use lexicographical order). Other functions become meaningless; for example, *avg* is unlikely to be used as a fusion function for non-numeric attributes.

6 Implementation

The solutions presented in this paper were implemented in a prototype system. The system, called Fusionplex, is described in this section. Figure 4 illustrates the overall architecture of Fusionplex.

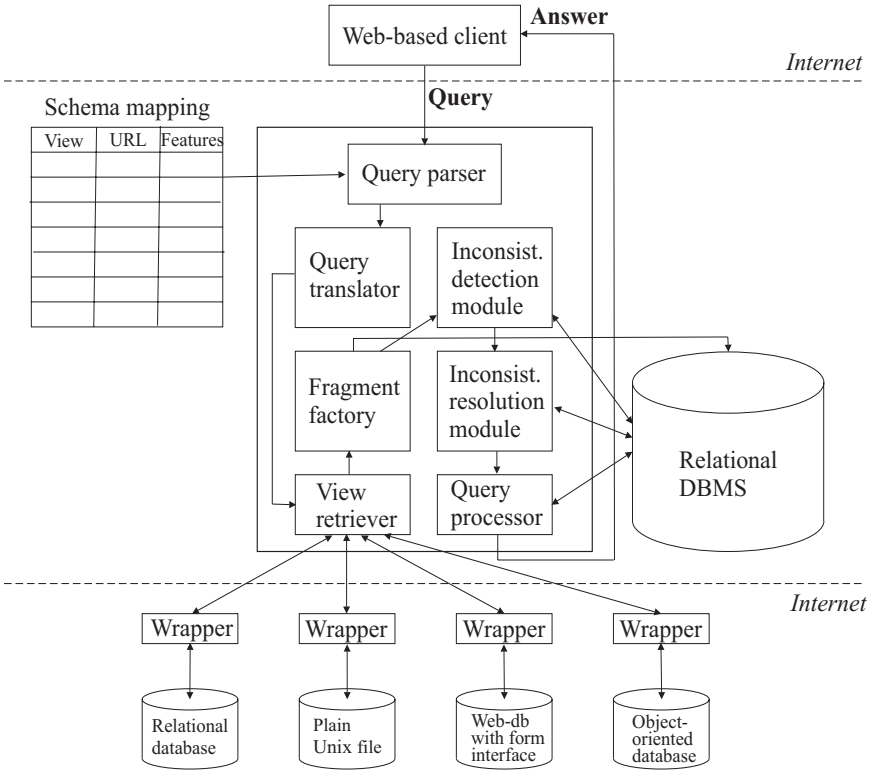

Figure 4: Architecture of Fusionplex.

Fusionplex conforms to a server-client architecture. The server is implemented in Java and contains the core functionalities described in this paper. At startup time, the server reads all configuration files, caches all source descriptions and creates temporary tables in a relational Database Management System. Then it starts listening for incoming connections from clients. Clients connect to the server using simple line-based protocol. Each client passes to the server the name of a database that the client wishes to query and the query itself. The server processes the query and returns its result to the client, which formats it and delivers it to its user.

The core of Fusionplex consists of the seven functional blocks:

1. The **query parser** parses the user query, checks its syntax and ensures that the relation and attribute names mentioned in the query are valid in the current virtual database.
2. The **query translator** determines the source contributions that are relevant to the given query by testing the intersection of the selection conditions of the query and of each contribution. It then calls the view retriever with the specifications of the contributions that were found relevant.
3. The **view retriever** consults the schema mapping and retrieves the relevant view instances from their corresponding URLs. It then attempts to enhance each instance with any attributes that participate in the view selection condition as part of an equality.
4. The **fragment factory** constructs the query fragments from the enhanced views and stores them in the relational database management system.
5. The **conflict detection module** assembles a polyinstance of the answer from the fragments, determines the possible areas of inconsistency by examining the selection predicates associated with the fragments and constructs the polytuples.
6. The **conflict resolution module** resolves data conflicts in each polytuple according to the appropriate resolution policies. First, each mono-attribute polytuple is resolved with a single value. Then the features of the resolved tuples are determined.
7. The **query processor** processes the union of all the resolved tuples, by applying further aggregating and ordering that may be specified in the query, and returns the query result.

Fusionplex also provides a client with a graphic user interface (GUI). This web-enabled client supports both direct input of queries as simple text and guided query construction through the use of its Query Assistant. The Query Assistant allows users to create queries using an intuitive visual interface. The client consists of a set of CGI-scripts written in Perl and allows for minimal correctness check at the client side. Figure 5 shows the client interface with an ongoing Query Assistant session.

As soon as the interaction with the Query Assistant is completed, a query in the SQL-like query language of Fusionplex is constructed and displayed in the query window. When the *Submit* button is pressed, this query is transmitted to the server, and the results returned from the server are displayed by the client (Figure 6).

Fusionplex also incorporates a database management tool for defining and maintaining virtual databases. Authorized users can create new virtual databases, and modify existing ones. To create a new virtual database, the user must define its global relations and plug-in any number of contributions from existing information sources. In each contribution, the user must specify a global view, a matching URL, and the appropriate source features. Existing virtual databases can be modified by adding or removing relations or contributions.

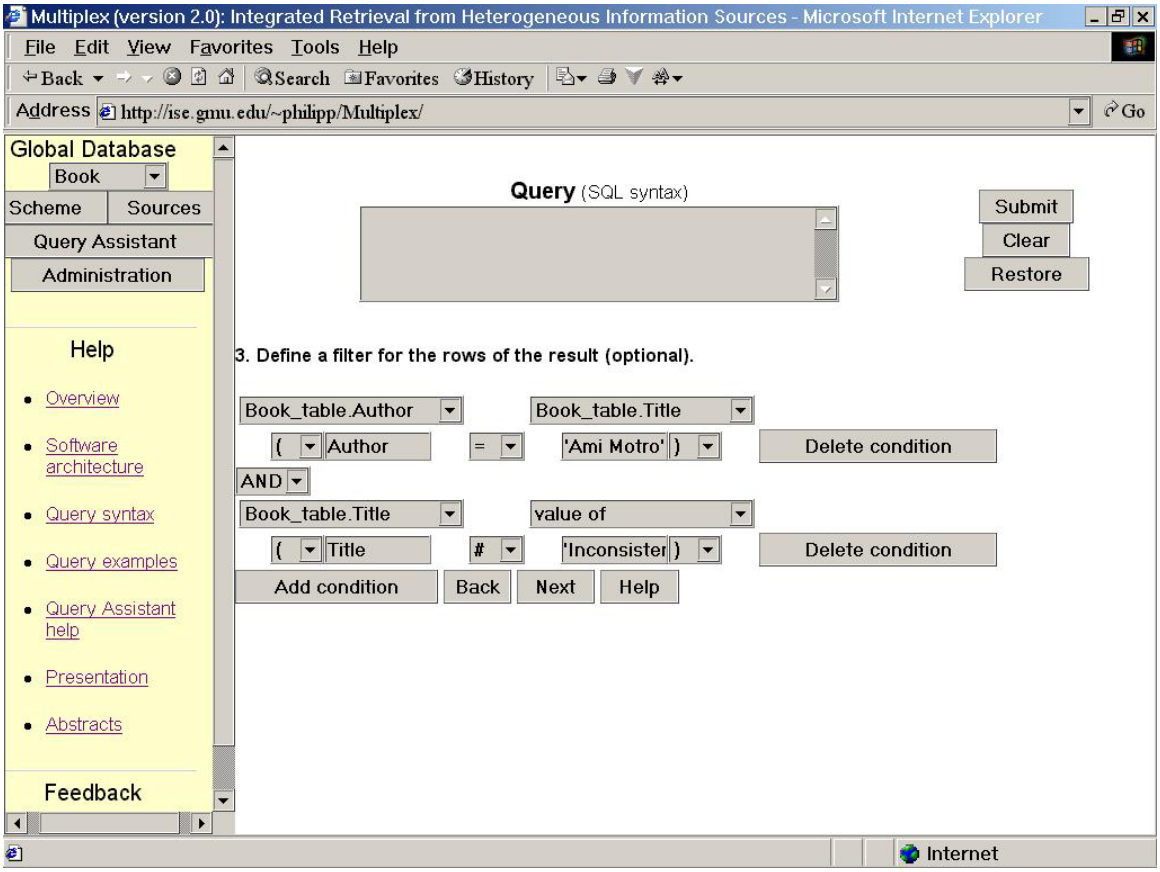

Figure 5: The Fusionplex GUI and a Query Assistant session

To experiment with the Fusionplex system, several information sources were constructed. For the purpose of heterogeneity, these sources were stored in four different types of systems: a relational database, an object-oriented database, a plain file in a Unix system, and a Web-based resource available only through a form interface. The latter format is typical of how information is retrieved from Web information sources. These sources were fitted with simple "wrapping" software that implements a relational model "communication protocol" between the sources and the system. In one direction, each wrapper translates Fusionplex queries to the local query language; in the opposite direction, it assembles the source response in the tabular format that is understood by Fusionplex.

The overall architecture of Fusionplex and its tools provide for a flexible *integration service*. A remote user wishing to integrate several information sources (possibly, sources from this user's own enterprise), logs into the server, provides it with the appropriate definitions, and can begin using its integration services right away. Future updates are fairly simple. For example, to integrate a new information source requires only a view definition (essentially, an SQL query) and a URL specification. Similarly, to expand the scope of the virtual database requires only a definition of a new virtual relation.

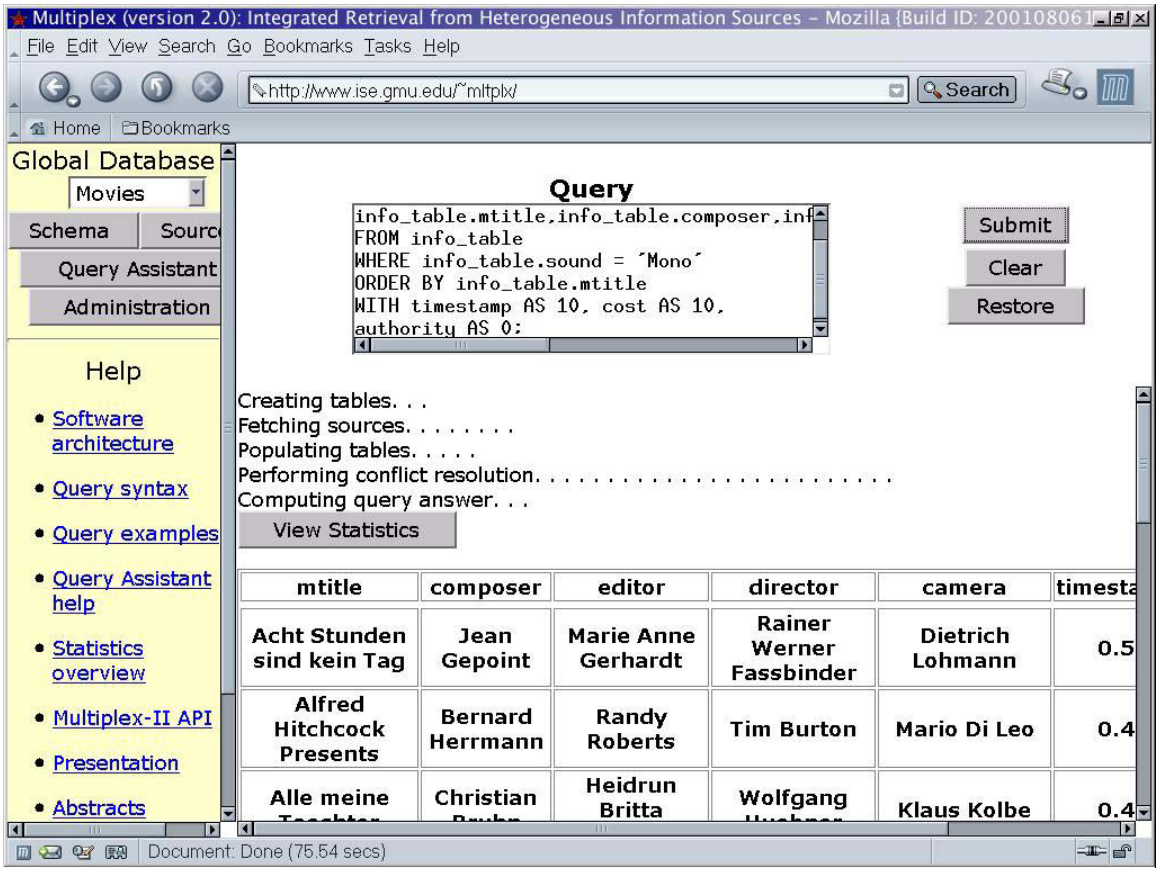

Figure 6: Query result in Fusionplex

7 Conclusion

In any realistic scenario of large-scale information integration, some of the information sources would be expected to "overlap" in their coverage. And in such situations, it is practically unavoidable that they would occasionally provide inconsistent information. From users' perspective, the desirable behavior of an integration system is to present them with answers that have been cleansed of all inconsistencies. This means that whenever multiple values contend for describing the same real world entity, they should be *fused* in a single value.

When humans are confronted with the need to choose a single value from a set of alternatives, they invariably consider the qualifications of the providers of these alternatives (and when they decide to take a simple average of the alternatives, or to choose one at random, it is usually because they conclude that the providers all have comparable qualifications).

The approach taken by the Fusionplex system formalizes these attitudes with the concept of source *features*, which quantify a variety of performance parameters of individual information sources.

Another observable behavior is that different individuals (or the same individuals in different situations) often apply different fusion policies. One individual might emphasize the importance of information recentness, whereas for another, cost might weight heaviest. As another example, in one situation, an individual would choose the average of the contending values, in another he would adopt the lowest, and in yet another he would pick the one that occurs most frequently.

Recognizing this, Fusionplex provides for powerful and flexible user control over the fusion process. In the two examples just mentioned, importance is conveyed by means of feature weights, and the preferred resolution method is stated in a policy that combines quality thresholds, elimination guidelines and appropriate fusion functions.

The practicality of these principles has been demonstrated in a prototype implementation of Fusionplex. Besides its fusion strengths, this implementation provides a sound overall information integration environment, with support for information source heterogeneity, dynamic evolution of the information environment, quick ad-hoc integration, and intermittent source availability. These aspects are delivered in a client-server architecture that provides remote users with effective integration services, as well as convenient virtual database management tools.

Research on Fusionplex is still continuing, with several issues currently under investigation, and other research directions being considered. We discuss briefly seven such issues and directions.

Suppose fusions are implemented as *linear combinations* of the conflicting values (i.e., a generalization of *average*), and suppose users express their preferences in a *utility function* that is a linear combination of the features (as they do now). Not only can the conflicting values be ranked according to their utility to the user (this is done now in the procedure described in Section 5.4), it should also be possible to determine whether the utility of the fusion value is indeed higher than the utility of the existing values; i.e., if fusion is indeed profitable. Moreover, it should also be possible to generate automatically the *optimal* fusion policy: the fusion coefficients that optimize the utility function.

The resolution algorithm described in Section 5.4 assumes a *utility threshold* (denoted α), which must be provided. This threshold determines the percentage of tuples of highest quality that will participate in the inconsistency resolution process. It may be possible to choose this threshold at run-time, based on the data to be integrated, so that the overall utility of the result is *optimized*. Both of the last two directions remove the need for user input, by choosing parameters that optimize utility.

Inconsistency detection and fusion are performed at the attribute level. This implies that each of the values of a result tuple could come from a different version of the information. In some situations, this may be undesirable. Consider this example in which two different versions of postal address information exist, each with *City* and *Postal_ code* attributes. Conceivably, the address generated by fusion could include a city from one version, and a postal code from another, resulting in incorrect information. In this example, *City* and

Postal_code should constitute a single fusion unit. The methodology should be extended to allow the specification of such units and to handle their fusion correctly.

During query translation, every contribution that is found to be relevant is materialized from its provider. Since providers are assumed to be able to deliver their contributions only as defined, inefficiencies may result. For example, a particular contribution may provide a large relation, of which the query might require only a few tuples. This is analogous to an Internet user downloading a large file, when in practice only a small portion of it is needed. The reason for this situation is that, in the interest of generality, we assumed that providers do not have capabilities for satisfying requests for *subsets* of their contributions. It would be useful to allow different classes of providers. Those that can only deliver their contributions verbatim, and those with capabilities of satisfying partial requests. Processing at the source would reduce considerably transmission and processing times for many queries.

As mentioned in Section 5.1, discrepancies might exist between the data "promised" in the view V of a contribution, and the data actually delivered when the URL is materialized. Fusionplex assumes that the information has been altered after the contribution has been plugged-in, and discards such contributions. It may be possible, however, to salvage some of the information by "repairing" contributions to correspond to their definitions.

To participate in Fusionplex virtual database, information providers must deliver their data in tabular format. XML [27] (the Extensible Markup Language) is quickly becoming a standard of data exchange. It would be beneficial to adopt XML as the communication protocol between Fusionplex and its information providers.

Finally, at times, users may benefit from having access to the entire set of alternative values when inconsistency occurs. This would allow them to monitor the performance of their fusion (and possibly to redefine it). Presently, Fusionplex does not have this ability to "explain" its behavior.

References

[1] S. Agarwal, A.M. Keller, G. Wiederhold, and K. Saraswat. Flexible Relation: An Approach for Integrating Data from Multiple, Possibly Inconsistent Databases. In *Proceedings of ICDE-95, the Eleventh International Conference on Data Engineering*, pages 495–504, 1995.

[2] R. Ahmed, J. Albert, W. Du, W. Kent, W. Litwin, and M-C. Shan. An Overview of Pegasus. In *Proceedings RIDE-IMS-93, the Third International Workshop on Research Issues in Data Engineering: Interoperability in Multidatabase Systems*, pages 273–277, 1993.

[3] J.L. Ambite, N. Ashish, G. Barish, C.A. Knoblock, S. Minton, P.J. Modi, I. Muslea, A. Philpot, and S. Tejada. ARIADNE: A System for Constructing Mediators for Internet

Sources. In *Proceedings ACM SIGMOD-98, International Conference on Management of Data*, pages 561–563, 1998.

[4] P. Anokhin. *Data Inconsistency Detection and Resolution in the Integration of Heterogeneous Information Sources*. Ph.D. thesis, School of Information Technology and Engineering, George Mason University, 2001.

[5] Y. Arens, C.A. Knoblock, and W. Shen. Query Reformulation for Dynamic Information Integration. *Journal of Intelligent Information Systems*, 6(2/3):99–130, 1996.

[6] D. Barbara, H. Garcia-Molina, and D. Porter. The Management of Probabilistic Data. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):487–502, 1992.

[7] M.L. Barja, T. Bratvold, J. Myllymaki, and G. Sonnenberger. Informia: A Mediator for Integrated Access to Heterogeneous Information Sources. In *Proceedings of the ACM CIKM-98, International Conference on Information and Knowledge Management*, pages 234–241, 1998.

[8] E.F. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions of Database Systems*, 4(4):397–434, 1979.

[9] L.G. DeMichiel. Resolving Database Incompatibility: An Approach to Performing Relational Operations over Mismatched Domains. *IEEE Transactions on Knowledge and Data Engineering*, 1(4):485-493, 1989.

[10] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J.D. Ullman, V. Vassalos, and J. Widom The TSIMMIS Approach to Mediation: Data Models and Languages *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

[11] Google. http://www.google.com.

[12] M.R. Genesereth, A.M. Keller, and O. Duschka Infomaster: An Information Integration System. In *Proceedings of ACM SIGMOD-97, International Conference on Management of Data*, pages 539–542, 1997.

[13] M.A. Hernandez and S.J. Stolfo. Real-world Data is Dirty: Data Cleansing and The Merge/Purge Problem. *Data Mining and Knowledge Discovery*, 2(1)9–37, 1998

[14] V. Josifovski, and T. Risch. Integrating Heterogeneous Overlapping Databases through Object-Oriented Transformations. In *Proceedings of VLDB-99, 25th International Conference on Very Large Data Bases* pages 435–446, 1999.

[15] L. Kaufman and P.J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis*. John Wiley and Sons, 1990.

[16] L.V. Lakshmanan, F. Sadri, and I.N. Subramanian. SchemaSQL — A Language for Interoperability in Relational Multi-database Systems. In *Proceedings of VLDB-96, 22th International Conference on Very Large Data Bases*, pages 239–250, 1996.

[17] T.A. Landers and R. Rosenberg. An Overview of MULTIBASE. In *Proceedings of the Second International Symposium on Distributed Data Bases*, pages 153–184, 1982

[18] A.Y. Levy, A. Rajaraman, and J.J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. In *Proceedings of VLDB-96, the 22nd International Conference on Very Large Data Bases*, pages 251–262, 1996.

[19] E.-P. Lim, J. Srivastava, and S. Shekhar. Resolving Attribute Incompatibility in Database Integration: An Evidential Reasoning Approach. In *Proceedings of ICDE-94, the Tenth International Conference on Data Engineering*, pages 154–163, 1994.

[20] A. Motro. Multiplex: A Formal Model for Multidatabases and Its Implementation. In *Proceedings of NGITS-99, 4th International Workshop on Next Generation Information Technologies and Systems*. Lecture Notes in Computer Science, Volume 1649, pages 138–158. Springer-Verlag, 1999.

[21] R. Ramakrishnan and J. Gehrke. *Database Management Systems (2nd Edition)*. McGraw-Hill, 1997.

[22] E.M. Rasmussen. Clustering Algorithms. In W.B. Frakes, R. A. Baeza-Yates, editors, *Information Retrieval: Data Structures and Algorithms*, pages 419–442. Prentice-Hall, 1992.

[23] A. Rosenthal and E. Sciore. Description, Conversion, and Planning For Semantic Interoperability. In *Proceedings of the DS-6, Sixth IFIP TC-2 Working Conference on Data Semantics*, pages 140-164, 1995.

[24] L.A. Shklar, A.P. Sheth, V. Kashyap, and S. Thatte. InfoHarness: A System for Search and Retrieval of Heterogeneous Information. In *Proceedings of ACM SIGMOD-95, International Conference on Management of Data*, page 478, 1995.

[25] V.S. Subrahmanian, S. Adali, A. Brink, R. Emery, J.J. Lu, A. Rajput, T.J. Rogers, R. Ross, and C. Ward. HERMES: Heterogeneous Reasoning and Mediator System. http://www.cs.umd.edu//projects/hermes/publications/abstracts/hermes.html, 1994.

[26] F.S-C. Tseng, A.L.P. Chen, and W-P. Yang. A Probabilistic Approach to Query Processing in Heterogeneous Database Systems, In *Proceedings of RIDE-TQP-92, Second International Workshop on Research Issues on Data Engineering: Transaction and Query Processing*, pages 176–183, 1992

[27] Extensible Markup Language (XML). http://www.w3.org/XML.