PERGAMON

Information Systems 0 (■■■■) ■■■–■■■

www.elsevier.com/locate/infosys

Interactive Example-driven Integration and Reconciliation for Accessing Database Federations☆

Kai-Uwe Sattler[a,]*, Stefan Conrad[b], Gunter Saake[a]

[a] University of Magdeburg, Department of Computer Science, P.O. Box 4120, D-39016 Magdeburg, Germany
[b] University of Munich, Department of Computer Science, Oettingenstr. 67, D-80538 München, Germany

Received 15 April 2001; received in revised form 15 January 2002; accepted 29 March 2002

Abstract

The integration of heterogeneous databases affects two main problems: schema integration and instance integration. At both levels a mapping from local elements to global elements is specified and various conflicts caused by the heterogeneity of the sources have to be resolved. For the detection and resolution of instance-level conflicts we propose an interactive, example-driven approach. The basic idea is to combine an interactive query tool similar to query-by-example with facilities for defining and applying integration operations. This integration approach is supported by a multidatabase query language, which provides special mechanisms for conflict resolution. The foundations of these mechanisms are introduced and their usage in instance integration and reconciliation is presented. In addition, we discuss basic techniques for supporting the detection of instance-level conflicts. © 2002 Published by Elsevier Science Ltd.

Keywords: Data integration; Database federation; Instance integration; Reconciliation; Conflict detection; Conflict resolution

1. Introduction

Integrating heterogeneous data sources is still a current problem, particularly with regard to the numerous available sources in the Internet. No matter if we consider virtual integration based on multidatabase languages, federated database systems and mediator systems or materialization in data warehouses, two main tasks have to be solved as part of the integration process: schema integration and instance integration. During schema integration the relevant elements from the local schemata are identified, homogenized and mapped into an integrated global schema. In this context, several conflicts have to be resolved, which are caused by the heterogeneity of the data sources with respect to data model, schema or modeling concepts. Schema integration mainly treats object types with attributes and relationships as well as extensional relationships of the local schemata.

In contrast, integration on instance level considers the concrete data in the sources. Here, the mapping between entities from different sources representing the same real-world objects has to be defined. Furthermore, data conflicts caused e.g., by contradictory values or different units of measurement have to be resolved. While several

---

☆ Recommended by Maurizio Lenzerinl

*Corresponding author.

E-mail address: kus@iti.cs.uni-magdeburg.de (K.-U. Sattler).

0306-4379/02/$ - see front matter © 2002 Published by Elsevier Science Ltd.
PII: S0306-4379(02)00023-6

methods for schema integration have been proposed in the past, the problem of instance integration is addressed only partially.

In this paper we present an approach focusing on conflict resolution and data reconciliation in federated databases. It is based on the multidatabase query language FraQL [1], which extends SQL by advanced conflict resolution mechanisms. In conjunction with an interactive query and design tool we are able to support a technique, which we call in the following *example-driven integration*. The main idea is identifying relationships and conflicts at instance level by exploring the existing, non-integrated data, applying necessary integration operations and conflict resolutions and receiving direct feedback from the resulting integrated data. This approach is intended as supplement — not replacement — for schema integration methods. The example-driven integration strategy takes into consideration the iterative and interactive nature of the data integration process.

Basically, a wide spectrum of supporting mechanisms for instance-level integration is possible, ranging from simple facing of data and tagging potential conflicts to applying statistical data analysis or even machine learning methods. Because of this, we will focus in this paper on techniques, which can be realized as part of a query language.

The remainder of the paper is structured as follows. In Section 2 we give a short overview to the FraQL language and its data model. Section 3 defines the semantics of the supported integration operations and Section 4 introduces the overall integration process. A detailed discussion of the various kinds of conflicts is given in Section 5. The resolution of these conflicts and the reconciliation is described in Section 6. Section 7 presents a interactive tool, which is based on the FraQL language and supports the example-driven integration. Related work is discussed in Section 8. Finally, Section 9 concludes the paper.

2. The Federation query language FraQL

Realizing an example-driven integration approach requires performing integration operations and querying integrated data in an alternating fashion. Our approach is based on FraQL, a query language for object-relational database federations. It extends SQL by features for defining federations, accessing meta-data in queries, restructuring query results, and resolving integration conflicts. This is comparable with other multidatabase languages like MSQL [12] or SchemaSQL [3], but in contrast to these proposals FraQL is extensible by user-defined data types and functions. FraQL is not primary intended as an end user language, but an intermediate language for specifying integrated views. Therefore, users can query the global integrated relations with usual SQL operations without knowledge of the FraQL language features.

In FraQL a federation or multidatabase is a set of data sources consisting of relations. A data source can be provided by a full-featured DBMS or even by a Web source encapsulated by a wrapper [4]. FraQL is based on a simple object-relational data model: it supports the definition of object types and object views derived from types in the spirit of SQL-99 as well as a built-in type `ARRAY` for arrays of atomic values. Using object-relational features simplifies the integration of post-relational data sources (e.g., ODBMS-based sources or XML data stores) and provides more advanced modeling concepts for schema definition. This simple data model is appropriate for our intended application domain — the analysis and fusion of distributed and heterogeneous data [5], because in this scenario data is mostly available in relational structures.

Object types describe the structure of objects as sets of attributes and their domains. Types can be organized in a specialization hierarchy. An object type is defined following the SQL-99 standard:

```
CREATE TYPE product (
    vendor VARCHAR(30),
    price FLOAT,
    prodName VARCHAR(20)
);
CREATE TYPE bike_type UNDER product (
    orderNo INT,
    year INT,
    stock INT
);
```

Object views represent global virtual relations of the federation, i.e., data from the sources is not materialized, except for caching purposes in order to speed up query evaluation. Here we distinguish between import and integration views.

An *import view* is a projection of a local relation of a data source. The import view is defined by specifying the source relation and, if required, a mapping between local attributes (i.e., attributes of the source relation) and global attributes (attributes of the view).

```
CREATE VIEW global_name OF type_name
    AS IMPORT FROM source.local_name
    [ ( mapping_definitions ) ];
```

In the view definition given above, the attribute mapping can be described in the following variants:

- If there exists a local attribute with the same name as any of the global attributes and both are type compatible, an implicit mapping between the is established.
- The notation *g_name* `IS` *l_name* means renaming the local attribute to *g_name*. This requires type compatibility.
- The notation *g_name* `IS` *func(l_name)* defines that the global attribute value is calculated by using the user-defined conversion function *func* on the local attribute value.
- The definition *g_name* `IS` `@`*tbl (l_name, src, dest, default)* means that the database table *tbl* is used for mapping the values from the local attribute *l_name*. This value of the global attribute is obtained by looking for the value of attribute *l_name* in column *src* and retrieving the corresponding value of column *dest*. The field *default* denotes a default value, either as literal or as local attribute, which is assigned to the global attribute, if the value of *l_name* is not found in the table. In fact, this kind of attribute mapping is evaluated by a left outer join, where the `NULL` value is replaced by the default value *default*.
- Local attributes without a corresponding global attribute are ignored.
- A constant literal value for a global attribute is defined by the notation *g_name* `IS` *literal*, e.g., for the case where no corresponding attribute exists in the local relation.

The following example illustrates the usage of these mapping concepts: Given a source relation `bike (vendor, price, product, orderNo, stock)` an import could be defined, where the local attribute `product` is mapped to `prodName` and the prices are converted into dollar prices:

```
CREATE VIEW bikes OF bike_type AS IMPORT
FROM src.bike (
    prodName IS product,
    price IS euro2dollar(price),
    year IS 2002
);
```

The attributes `orderNo` and `stock` appear both in the type of the view and in the local relation. Thus, they are imported implicitly. Furthermore, the `year` value is set to `2002` for all tuples, because this attribute does not exist in the relation `bike`. An alternative solution could involve an computation of a new value for `year` using a conversion function or even a mapping table, e.g., based on other attributes.

A data source referenced in an import view definition is specified by the required database adapter and additional connection information:

```
REGISTER SOURCE source_name AT
    'DSN=db;UID=user;PWD=password'
    USING 'adaptor_name';
```

An *integration view* is a SQL-like view on other global relations defined by using the standard SQL operations as well as extended FRAQL operations. Such a view is defined as follows, where the term `table_expression` denotes a SQL query with extensions explained later.

```
CREATE VIEW global_name OF type_name
    AS table_expression;
```

Furthermore, FRAQL supports user-defined functions (UDF) as well as aggregate functions (UDA), which are stored in the database of the federation layer (i.e., in the query processing server) and are callable in queries. These functions are implemented in Java or C+ + and registered in the query system.

Another feature of FRAQL that can be utilized for resolving integration conflicts is the combination of extended grouping and user-defined aggregates. The `GROUP BY` operator supports grouping on arbitrary expressions: for each tuple of the input relation a value is computed from the grouping expression and based on this, the tuple is assigned to a group. As an example let us assume for simplicity that the model year of a product is encoded in the order number as the last two digits. In this case, we could group products of the same type independently from the year using the following query:

```
SELECT order_id
FROM bikes
GROUP BY floor(orderno / 100) AS or-
der_id;
```

As result of the `GROUP BY` operation each group consists of tuples representing the same real-world entity. After this, all the tuples of a group have to be merged in one item, for example by computing a value from conflicting attributes or by using the most up-to-date information. This can be implemented with the help of UDA. A UDA function is implemented in FRAQL as a Java or C+ + class with a predefined interface consisting of the methods:

- *init* for initialization purposes,
- *iterate* invoked for each tuple of the input relation and
- *result* for obtaining the final result.

As an extension to the UDA concept available in Oracle 9, Informix or PostgreSQL, the FRAQL aggregates can be defined with more than one parameter. This is particularly useful for reconciliation function, where the aggregated value of a column has to be computed depending on values of another column. There is a set of predefined reconciliation functions including the following:

- `pick_where_eq` (*v*, *col*) returns the value of column *col* of the first tuple, where the value of *v* is true, i.e., $\neq 0$. In case of a group consisting of only one tuple, the value of this tuple is returned independently of the value of *v*.
- `pick_where_min` (*v*, *col*) returns the value of column *col* of the tuple, where *v* is minimal for the entire relation or group, respectively.
- `pick_where_max` (*v*, *col*) returns the value of column *col* of the tuple, where *v* is maximal.

In these functions *v* means a value that is computed from an expression formulated on the attribute values of the current tuple. As an example the following query returns the name of the most expensive bike:

```
SELECT pick_where_max (price, prodName)
FROM bikes;
```

Obviously, this could be formulated also in standard SQL, but we will show later, that such aggregation functions simplify conflict resolution by allowing a kind of transposition operation on relations.

Another predefined aggregation function is `to_array` which "nests" the values of the given column and returns an array containing all values. In this way, the complete set of instances of an attribute belonging to a single real-world entity can be collected and passed to the user or application for further considerations.

Restructuring of relations is implemented in a way inspired by SchemaSQL [3]. Variables of a query can not only be bound to relations as tuple variables, but also to meta-data, like the set of attributes of a relation or the set of relations of a schema. But in contrast to SchemaSQL, where meta-data access in queries is implemented as a language extension, in our approach the schema catalog is used. So, the catalog relation `catalog.columns` contains information about attributes of all global relations, whereas the relation `catalog.tables` describes the global relations. Unlikely SchemaSQL, any global user relation with information about other relations can be used as meta-data source.

As an extension to standard SQL, attributes of tuple variables in queries can be obtained during evaluation. This means, while in SQL names of attributes and relations are constants, in FRAQL they can be constructed from current values of other tuple attributes. This *variable substitution* is written in the notation `$var` and can appear everywhere in a query, where names of attributes or relations are expected. For example, the expression `tbl1.$(tbl2.col)` means the attribute value of the current tuple of relation `tbl1`, whose name is obtained from the current value of `tbl2.col`.

In the same way, a relation in the `FROM` clause of a query can be dynamically determined. The following query selects product name and price information from all relations implementing the object type `bike_type`. So, it is equivalent to a union of all these relations.

```
SELECT t2.prodName, t2.price
FROM catalog.tables t1, $(t1.table-
_name) t2
WHERE t1.type_name = 'bike_type';
```

In summary, data integration in FraQL follows the *global as view* paradigm [6], where the global (integrated) view is defined by a query over a set of source relations. Thus, it inherits the advantages of this approach like a simplified query rewriting and decomposition. But in contrast to the *local as view* approach, adding or removing sources affects the global view definition and hence is more complicated. However, by using meta-data queries in combination with variable substitution we are able to mitigate this problem in a certain way: as demonstrated in the above query, if a new view or relation of a given type is created, it can be automatically included in a global view.

FraQL is implemented as part of a federated query system and consists of the following main components: the query parser, the decomposer and the global optimizer, the query evaluator, the Java VM for evaluating user-defined functions, and the catalog. The adapter layer contains the management component as well as the individual adapters providing a uniform access interface to the data sources. The interface to query processor is implemented using CORBA, the adapters are dynamic loadable libraries and thus can be plugged into the system at runtime. On top of the query interface we have developed a JDBC driver and an interactive query tool. Currently, adapters are available for full-fledged DBMS (e.g., Oracle) as well as for flat files, (relational) structured Web sources and XML documents. This permits particularly the integration of Web sources which are generated from relational databases.

The limited query capabilities of Web and file sources are taken into account during query rewriting. So it is possible to specify query constraints for individual source relations via a special `ALTER VIEW` statement. For instance, the constraints for an import view `bikes` supporting only queries on the attribute `prodName` using the '=' operator are specified as follows:

```
ALTER VIEW bikes SET QUERY CONSTRAINTS (
    PREDICATES (prodName, =),
    COMBINATIONS (prodName));
```

Based on this information queries accessing limited sources are rewritten in a way, that the specified query constraints are fulfilled [7].

3. Semantics of Integration operations

Due to the fact that FraQL is an extension of SQL we can build upon SQL and its semantics. In order to allow query optimization we base on an algebraic framework such that the well-known results for algebraic optimization can be used without restrictions. In the following, we show how to integrate advanced concepts of FraQL into the standard relational algebra, in particular we consider

- the application of user-defined function,
- the application of mapping tables,
- operators dealing with variable substitution, and
- the transposition operator.

We omit the description of the semantics of the extended `GROUP BY` operator, because allowing arbitrary expressions as grouping attributes as in a query like

```
SELECT a, aggr(b) FROM rel
GROUP BY func(a);
```

is a shortcut for the following query:

```
SELECT a, aggr(b) FROM ( SELECT func(a) AS
a, b FROM rel )
GROUP BY a;
```

Therefore, we can build upon the semantics of the standard `GROUP BY` operator of the relational algebra.

3.1. User-defined functions

For mapping local attributes onto global attributes when defining import relations, user-defined functions can be introduced. A typical application is the conversion of attribute values which are represented in a local data source in a different way than needed in the global system (e.g., using different units of measurement).

As introduced in the previous section, the declaration of an import relation in FraQL consists in principle of four parts (where the third and fourth part are alternatives not used together) when we translate it into relational algebra:

- There is a *projection* determining which attributes of the local relation are mapped onto attributes of the global import relation.
- Attribute names of the local relation are mapped onto attribute names of the global relation by *renaming*.
- The *application* of user-defined functions is used for transforming attribute values.
- Another way of transforming attribute values is the usage of *mapping tables* which are stored like usual tables.

Whereas projection and renaming are already basic operations of standard relational algebra, the application of user-defined functions is an additional concept for which we can fall back upon several approaches for extended relational algebras, e.g., for extended database models (cf. e.g. [8–13]). In the following we represent the application of (user-defined) functions as algebraic operation 'apply' (using the symbol α):

$$\alpha_{A,f}(r),$$

where r is a relation with schema R, A an attribute of R, and $f : T_A \rightarrow T$ a function which can be applied to values of the type T_A defined for the attribute A in R resulting in values of type T. Please note, that for the moment this is a rather restricted form for applying functions, which might later be extended towards functions producing other result types.

The operation $\alpha_{A,f}(r)$ then produces a relation r' with schema R' which is identical to R except of the type for attribute A in case $T \neq T_A$ ($R \equiv R'$ if $T = T_A$). The resulting relation r' contains all tuples of r except of the fact that for each tuple the value of the attribute A has been transformed by applying f.

For algebraic optimization a collection of rules expressing the equivalence of terms is needed. Examples for such rules are:

- $\alpha_{A,f}(\alpha B, g(r)) = \alpha B, g(\alpha_{A,f}(r))$ if $A \neq B$;
- $\alpha_{A,f}(\alpha_{A,g}(r)) = \alpha_{A,g \circ f}(r)$
- $\alpha_{A,f}(\pi_{A_1,\ldots,A_n}(r)) = \pi_{A_1,\ldots,A_n}(\alpha_{A,f}(r))$ if $A \in \{A_1, \ldots, A_n\}$

For short, we can omit the attribute to which the function is applied if it is clear from the context or if the function $f : R \rightarrow R$ transforms not only single attributes but entire tuples of type R. A function f which transforms only a single attribute A can always be extended to a function $f_R : R \rightarrow R$ where f_R change the attribute A in the same way as f does and all other attributes remain unchanged. We then may write $\alpha_f(r)$.

3.2. Mapping tables

A special way for transforming attribute values from local relations into global relations is the usage of mapping tables. A mapping table is a usual global relation which might have been imported from other local sources if the mapping information is derivable from some local data. Of course, we can also directly define a new global relation only for mapping purposes and explicitly store the needed mapping information there.

A table *tbl* is used as mapping table if in the mapping definition for some import relation an expression `@tbl (l_name, src, dest, default)` is given where *l_name* is the name of the local attribute which has to be transformed, *src* and *dest* are attributes of the mapping table *tbl* describing the mapping, and *default* is an optional default value which is used if no explicit mapping is provided for some local values. From an operational point of view, the transformation of local values works as follows: taking a value of the local attribute *l_name* we look for a tuple in *tbl* having this local value as value in the attribute *src*. If such a tuple exists its attribute *dest* contains the transformed value; otherwise the result is the NULL value – or the *default* value if given.

By means of the relational algebra this operation could be captured by a left or right outer join. The substitution of NULL values by the *default* value (if needed) is a little bit complicated due to the fact that we do not want to substitute NULL values which are already given as resulting value in the *dest* attribute. A complete algebraic description of this mapping applied to a local relation r is the following one:

$$\rho_{dest \to l\_name}(\pi_{R-r.l\_name+tbl.dest}(\sigma l\_name = src(r \times tbl))),$$
$$\cup\, \alpha l\_name, f_{default}(r - \pi R(\sigma l\_name = src(r \times tbl))),$$

where ρ is the renaming operation, $f_{default}$ is a function always resulting in the *default* value (if given). Although this description looks rather complex it should be clear that a really efficient implementation of the mapping operation can easily be found.

3.3. Operators for variable substitution

Variable substitution or dereferencing comes in FraQL in two fashions: as *column dereferencing* as part of a `SELECT` or `WHERE` clause and as *table dereferencing* in the `FROM` clause.

For the first form we define an operator $\nu_{B \leftarrow A_i}(r)$ that returns a relation r' with the relation schema R' comprising the attributes $A_1, \ldots, A_n$ from R as well as the newly introduced attribute B. Each tuple $t' \in r'$ is derived from a corresponding tuple $t \in r$ as follows: for each $t \in r$ there is one and only one tuple t' with

$$t'(A_j) = t(A_j)\ \forall j = 1, \ldots, n \text{ and } t'(B) = t(t(A_i)),$$

where we restrict the domain of A_i to alphanumeric values only. In case of $t(A_i) \notin r$ the `NULL` value is assigned to $t'(B)$.

For table dereferencing we rely on the expansion operator of the extended algebra from Ross [14]. This operator[1] $\mu^k(r)$ expands a set of relation names obtained from the relational expression r into a union of the arity-k relation extensions. In order to apply this operator to relations with more than one column we add an attribute A_i as a parameter to this operator. Then, the operator can be defined as follows:

$$\mu^k_{A_i}(r) = \bigcup_{s \in \pi_{A_i}(r)} (\{s\} \times s(A_1, \ldots, A_k))$$

where $s(A_1, \ldots, A_k)$ is a relation with the name s of arity k. Obviously, this corresponds to a query like

```
SELECT∗FROMcatalog.tablest, $(t.table_name);
```

3.4. The transposition operator

Although transposing relations is not explicitly supported in FraQL by a dedicated operator but rather through a query pattern exploiting special aggregation functions, we will give in the following the semantics of this important restructuring operation.

For this purpose, we follow the idea of a *unfold* operation, that originally appeared in [3], and denote this operator $\tau_{A_i,A_j}(r)$. This operator produces a relation r' with the relation schema R' consisting of the following set of attributes:

$$R' = \{A_1, \ldots, A_n\} - \{A_i, A_j\} \cup S,$$

where $S = \pi_{A_i}(r)$. This means, the additional attributes in R' are derived from the set of distinct values of A_i in r. Let be $S = \{B_1, \ldots, B_m\}$ then the tuples of r' are obtained by grouping the tuples of r based on equal values for the attribute set $\{A_1, \ldots, A_n\} - \{A_i, A_j\}$. Thus, each of the resulting groups consists of $k \leq m$ tuples $t_1, \ldots, t_k \in r$ where

$$t_1(A_l) = t_2(A_l) = \cdots = t_k(A_l)\ \forall l = 1, \ldots, n, l \neq i, l \neq j.$$

Now, for each of these groups there is one and only one tuple $t' \in r'$ with

$$t'(A_l) = t_1(A_l) = \cdots = t_k(A_l)\ \forall l = 1, \ldots, n, l \neq i, l \neq j$$

$$\forall l = 1, \ldots, m : t'(B_l) = \begin{cases} t(A_j) & \text{where } t(A_i) = B_l \text{ and } t \in \{t_1, \ldots, t_k\} \\ \text{NULL} & \text{otherwise} \end{cases}$$

We will give an example for this operation in Section 6.

---

[1] In fact, in the original work α is used for denoting this operator.

4. The Integration Process

During data integration both levels — schema level as well as instance level — have to be taken into consideration. At both levels conflicts can occur, which are caused by the heterogeneity of the sources. In the following we sketch the overall integration process and point out to these steps, which are particularly supported by our approach.

The core concepts of the FraQL data model corresponds to the main steps of the integration process. In the first step — as part of schema integration — the global object types of the integrated schema have to be defined. This is done either top-down — from the requirements of the application domain — or bottom-up — by analyzing the local schemata. In case where local types are not explicitly available, e.g. in classical relational databases, the type definitions and their relationships have to be derived from the relation schema. The goal of the following steps is to map the local relations onto these types by applying various integration operations. In this context, schema-level as well as instance-level conflicts have to be resolved. But while most schema-level conflicts are resolvable by examining the local schemata only, the resolution of instance-level conflicts requires considering the concrete data from the sources and applying reconciliation techniques.

By examining this data and performing appropriate queries, the database integrator is able to identify instance conflicts and to resolve them with the help of user-defined conversion and resolution functions, which are applied as part of importing a relation as well as in form of aggegration functions. This procedure is shown in Fig. 1.

First of all, import relations are defined. Here, we resolve description conflicts by specifying the mappings of attributes. Second, import relations representing semantically overlapping extensions are combined into integration relations by applying join or union operations. These initially defined relations are examined now by special conflict checking queries, which we will describe later. The query results may indicate possible instance conflicts. Furthermore, special tools for data analysis could support this step. For very large datasets the query response time can be reduced by using sampling techniques for approximate answers [15].

With knowledge about existing instance conflicts the definitions of import and integration views are refined, i.e., transformation functions are introduced for attribute mapping, join predicates or grouping expressions are modified and reconciliation functions are applied. In principle, these steps are repeated until no conflict remains or can be detected. The final definitions of import and

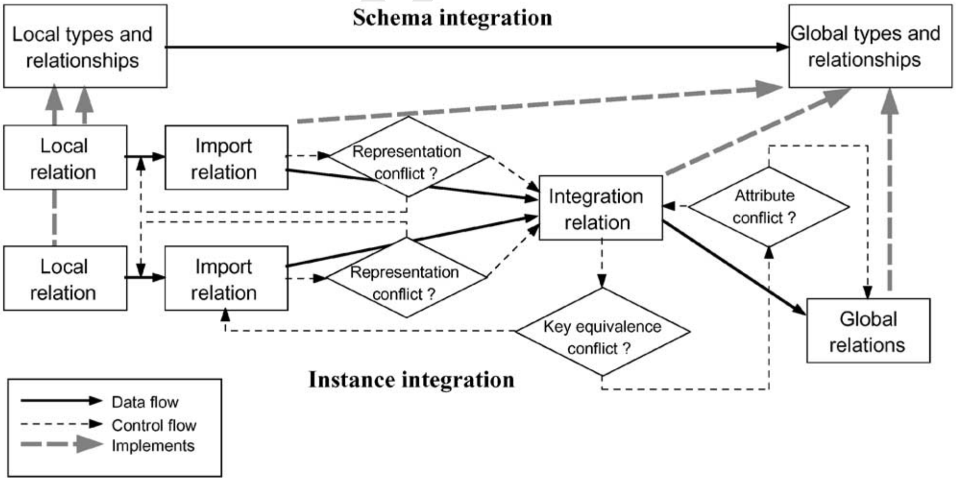


Fig. 1. Integration process.

integration views form the integrated schema of the database federation.

Considering the overall process we can state that the integration is driven by the current instances or example data. The available data is used for conflict identification as well as evaluating integration operations. Finally, success of the conflict resolution strategies is immediately visible in terms of this data.

However, please note that the absence of instance conflicts is valid only for the current instances in the data sources and not necessarily valid for all possible instances. Moreover, there may exist further data conflicts which are not resolvable in this way, because they do not follow some general rules. Typical conflicts of this class are for instance typo-errors or outdated values, which have to be treated separately. In addition, there could exist discrepancies in data which are not really conflicts rather representations of different facts, e.g., different prices for the same product sold in different shops [16].

5. Integration conflicts

In this section we classify those conflicts which are particularly addressed by our approach. Starting with schema-level conflicts for an overall view, we relate them to instance-level conflicts and discuss basic techniques for conflict detection.

5.1. Schema-level conflicts

Due to heterogeneities at data model, schema, and instance level, integration of existing data sources has to deal with various kinds of conflicts. For schema-level conflicts several classifications were proposed in the literature, e.g. [17,18].

As basic classification we use the one which was introduced in [18]. Following this classification integration conflicts are divided into four classes:

- semantic conflicts,
- description conflicts,
- heterogeneity conflicts, and
- structural conflicts.

In practice, we often have to face combined occurrences of these conflict types. In consequence, the conflict resolution needs to take into account different aspects at the same time. For a conceptual clarity we explain these four conflict classes in isolation.

5.1.1. Semantic conflicts

This class of conflicts deals with the semantic relationship between extensions (possible populations) of classes where the notion “class” stands for any modeling construct representing a collection of real-world objects (depending on the concrete data model we have to consider classes, relations, entity and relationship types, etc.).

Integrating existing schemata the overlapping parts of the local Universes-of-Discourse must be identified and within these overlapping parts we have to find out which classes correspond in which way to each other. Unfortunately, the correspondence between two classes is often not an exact correspondence in the sense that the two classes always represent the same set of real-world objects. If we consider two corresponding classes, we may find four different kinds of correspondences between them: equivalent extensions, including extensions, overlapping extensions, and disjoint extensions. For each of these kinds of correspondences there may be different ways to build corresponding classes in the integrated schema. An important aspect is to find an adequate mapping between the classes for which we have found such a correspondence and the corresponding classes in the integrated schema.

5.1.2. Description conflicts

The class of description conflicts comprises a large number of more specific conflicts. Here, we can only give some examples for typical description conflicts. A detailed discussion on description conflicts can be found e.g. [19].

Objects belonging to corresponding classes are often described by different sets of properties (attributes) in the local schema. This is due to different requirements of the local applications. In one system local applications need a certain property of the objects whereas in another system no application accesses this property.

Other often occurring description conflicts result from the usage of homonyms and synonyms for attribute names, class names, etc. In general, homonyms and synonyms cannot be resolved in a fully automated way.

Further examples for description conflicts are that corresponding attributes may have different data types or ranges in different component systems. Even if they have the same data type, different units of measurements or a different scaling can be used within the component systems. Furthermore, there can exist conflicts due to different integrity conflicts.

5.1.3. Heterogeneity conflicts

In this class we can find all conflicts which are due to the use of different data models for the local schemata in the participating database systems. The usage of different data models implies that different sets of modeling concepts are used. In particular, in data models having only very few modeling concepts (like the relational model) other modeling concepts are simulated by means of the existing ones. In general, the usage of different modeling concepts in different data models leads to the next class of conflicts, i.e., structural conflicts, which are usually not direct resolvable by transforming schemata from heterogeneous data models into a global data model.

5.1.4. Structural conflicts

This kind of conflicts is caused by the usage of different modeling concepts for expressing the same real-world fact. All data models offer several possibilities to model the same real-world fact. Thereby, database schemata expressed in the same data model can have different structures although they describe the same Universe-of-Discourse. In particular, data models offering a large number of modeling concepts allow numerous ways of description.

A special kind of structural conflicts are *meta conflicts* occurring when instances of a property are stored as specific values in one schema, whereas they are represented as schema objects (meta-data) in another schema.

Conflict detection at schema level requires knowledge about the problem domain, the schemata and the extensional correspondences. This task can be supported by thesauri or ontologies, but in general an automatic detection can only succeed in very restricted cases or application domains.

As we will see in Section 6, FraQL is able to deal with description, semantic and structural conflicts. Heterogeneity conflicts are resolved mainly by the adapters which map the modeling concepts and hide system-dependent differences.

5.2. Instance-level conflicts

Identifying and resolving schema-level conflicts does not mean that the instances are homogenizised as well. Different representations of data can result in different ways dealing with these conflicts, depending on the semantics and the further usage of the affected attributes. So, first we introduce a simple classification and discuss detection strategies for the individual conflict types.

The different kinds of instance-level conflicts arise not independently from each other. As the primary kind of conflicts we introduce the notion of *representation conflicts*. This refers to different representation of data values corresponding to the same real-world fact. This could be caused, e.g. by different units of measurements (e.g., Dollar vs. Euro), by different notations (e.g., “firstname lastname” vs. “lastname, firstname”) or simply different representations (e.g., ISBN with dashes vs. without dashes).

During integration representation conflicts can result in *key equivalence conflicts* as well as *attribute value conflicts*. Key equivalence conflicts arise when instances from different relations refer to the same real-world object but contain different object identifiers or keys. Attribute value conflicts occur when instances, which correspond to the same real-world object and share an equivalent key, differ in other attributes. One reason for this problem could be a situation, where two relations from different sources overlap semantically and one of the relation contains older or outdated data. For data models with richer expressive power we could add a further conflict class which refers to relationship conflicts [16].

In order to get a hint about the kind of conflicts in the current integration step we have to take into consideration the integration process discussed in Section 4. In the first step description conflicts at schema level are resolved by defining attribute mappings for import relations. We are also able to resolve instance-level representation conflicts with the help of conversion functions or mapping tables. However, there is no general solution for detecting these conflicts because we cannot compare the data values of this relation with others at this stage. Therefore, domain knowledge or application-specific plausibility checks are required for conflict detection.

5.3. Instance-level conflicts resulting from schema level conflicts

In the second step of the integration process semantically overlapping relations are combined. This overlapping could be horizontally or vertically. Here, two kinds of schema-level conflicts can occur: structural conflicts and semantic conflicts. The resolution of these conflicts is subject of schema integration. But based on the knowledge about the affected relations, i.e., the extensional correspondences, we are able to apply basic detection strategies for instance-level conflicts.

5.3.1. Structural conflicts

Representing a real-world fact by different modeling concepts results in structural conflicts. Depending on the variety of the data model several kinds of conflicts can arise, but the most frequent conflicts are partitioning and meta conflicts. Partitioning occurs, when the relations which have to be integrated overlap vertically, e.g., represent different aspects of the global relation, but still contain semantically equivalent attributes. Meta conflicts arise, when a concept is represented as data object in one schema, whereas it is modeled as schema object (attribute or relation) in another one. These conflicts are resolved at schema level by applying join operators for partitioning and restructuring for meta conflicts (cf. Section 6). However, on instance level we have to deal with key equivalence conflicts and attribute value conflicts.

The existence of key equivalence conflicts is recognizable by comparing the import relations with the integration result. If extensional correspondences between the relations are known, a first indicator could be the sizes of the individual relations. For the corresponding relations r_1, r_2 and the integrated relation r_i which is computed by $r_1 r_2$ we can define the following assertions regarding the size $|r|$:

- $r_1 \equiv r_2$ (equivalence): $|r_i| = |r_1| = |r_2|$
- $r_1 \subseteq r_2$ (inclusion): $|r_i| = |r_1| \leq |r_2|$
- $r_1 \cap r_2$ (overlapping): $0 \leq |r_i| \leq \min(|r_1|, |r_2|)$
- $r_1 \neq r_2$ (disjointness): $|r_i| = 0$.

Attribute value conflicts could arise when besides the key attributes additional common attributes exist and contain discrepancies. In this case we have to decide which of the two attribute values should occur in the integrated relations. This kind of conflict is detectable by comparing the attribute values. Obviously, for an given attribute A this can be checked by the following query expression:

$$\sigma_{r_1.A \neq r_2.A}(r_1 r_2)$$

This results in the set of tuples containing an attribute value conflict regarding A.

5.3.2. Semantic conflicts

Semantic conflicts arise, when the relations, which have to be integrated, overlap horizontally, i.e., there are tuples from both relations representing the same real-world entity. First of all, this kind of conflict is addressed by applying a union operation. This requires that the two relations are structurally equivalent, which is achieved by resolving structural and description conflicts. However, at instance level we have to deal again with key equivalence and attribute value conflicts. As discussed above a first statement about the existence of key equivalence conflicts can be formulated based on the knowledge about extensional correspondences between the relations:

- $r_1 \equiv r_2$: $|r_i| = |r_1| = |r_2|$
- $r_1 \subseteq r_2$: $|r_i| = |r_2|$
- $r_1 \cap r_2$: $\max(|r_1|, |r_2|) \leq |r_i| \leq |r_1| + |r_2|$
- $r_1 \neq r_2$: $|r_i| = |r_1| + |r_2|$

For detecting attribute value conflicts the approach of comparing attribute values is used. As shown above the two relations are joined and tuples containing discrepancies regarding a given attribute are selected.

Fig. 2 illustrates the dependencies between the different kinds and levels of integration conflicts. It should be made clear that there is a tight connection between schema-level and instance-level conflicts. This consideration should also motivate an interactive and iterative approach to data integration and reconciliation, which addresses both levels and is supported by an user-friendly tool for defining mappings and correspondences as well as a query system for exploring the integration results.

6. Conflict resolution and reconciliation

In section 2 we have introduced the language FraQL which provides mechanisms for resolving conflicts. In the following, we discuss the application of these features. Due to the tight relationship we describe the resolution of instance-level conflicts in context of the associated schema-level conflict.

6.1. Description and representation conflicts

As an example for representation conflict resolution in FraQL please consider the following scenario. The product database from two mountain-bike dealers shall be integrated. The relations are structured as shown in Fig. 3. The Relation for dealer A contains prices in dollar and a separate vendor attribute, whereas dealer B uses euro prices and a different order number schema.

Obviously, we can introduce a global type `bike_type` for both relations (cf. Section 2) which is structured as relation `bikes` from dealer A. But because dealer B uses its own schema for order numbers, a simple transformation is not possible. Therefore, we have to map the order numbers by using the mapping table from Fig. 4.

In addition, the mapping table provides the vendor information for each bike tuple based on the product number. With the help of this table and a conversion function `euro2dollar` for the price attribute which converts Euro to Dollar, the import views are defined as follows:

```
CREATE VIEW bikes_A OF bike_type
    AS IMPORT FROM dealerA.bikes;
CREATE VIEW bikes_B OF bike_type
    AS IMPORT FROM dealerB.bikes (
    price IS euro2dollar (price),
    vendor IS @map_orderNo(prodNo, pid,
    vendor, NULL),
    orderNo IS @map_orderNo(prodNo, pid,
    order, NULL)
);
```

As mentioned above, not all kinds of representation conflicts are identifiable in the early steps of the integration process. Therefore, later steps could require a refinement of the definitions of import views.

6.2. Structural conflicts

At schema level structural conflicts are resolved by applying a join operation (for partitioning) or by restructuring operations. Because the join operation is straightforward, we describe only the resolution of meta conflicts. As an example, please consider the relations shown in Fig. 5. In

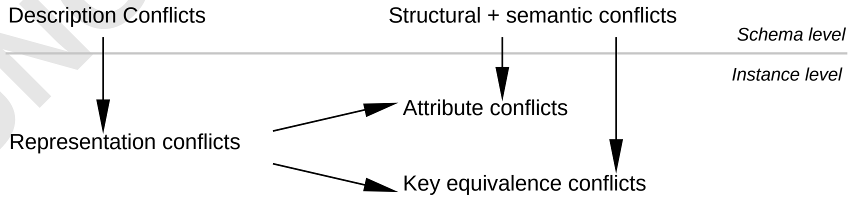


Fig. 2. Integration conflicts.

relation `bikes_1` the prices of the individual dealers are stored as an attribute value of the tuple. In contrast, relation `bikes_2` contains a separate tuple for each dealer.

So, in the first relation, the information about the dealers is represented as a schema element (an attribute), whereas it is represented as a data value in the second relation. Similar to the approach proposed in [22], the relation `bikes_1` can be transformed in order to match the structure of `bikes_2` with the following query:

```
SELECT b.prodName, b.orderNo, b.$(c.co-
lumn_name), c.column_name
FROM bikes_1 b, catalog.columns c
WHERE c.table_name = 'bikes_1' AND
    c.column_name < > 'prodName' and
    c.column_name < >
    'orderNo';
```

vendor	*year*	*orderNo*	*prodName*	*price* ($)	*stock*
Alpine Design	2000	AD-1234	XC-Shock	530	10
Bianchi	2000	B-6081	Grizzly	1750	4
Cannondale	1999	C-8193	Supler V 700	1908	1
Bianchi	1999	B-6070	Wannabee	751	10

(a) Relation for dealer A *bikes*

prodNo	*year*	*prodName*	*price* (Euro)	*stock*
0431-860	2000	Bianchi Wannabee	824	12
0431-871	1999	Bianchi Grizzly	1923	3
0431-241	2000	Raleigh M 8000	2097	2

(b) Relation for dealer B *bikes*

Fig. 3. Local dealer relations.

pid	*vendor*	*orderNo*
0431-871	Bianchi	B-6081
0431-860	Bianchi	B-6070
0431-241	Raleigh	R-4010

Fig. 4. Mapping relation *map_orderNo*.

This operation is sometimes called "transposition" because the relation `bikes_1` is transposed, i.e., the columns `dealer1` and `dealer2` become rows after applying this operation.

A transposition in the opposite direction which corresponds to the transposition operator described in Section 3 can be performed by the `GROUP BY` operator together with the `pick_where_eq` aggregation function described in Section 2. Here, the idea is to group tuples representing the same object and apply the aggregation function in order to project the different values to the corresponding columns. Assume we want to transpose relation `bike_2` according to the schema of relation `bike_1`, we can formulate the following query:

```
SELECT prodName, orderNo,
    pick_where_eq (dealer = 'dealer1',
    price) as dealer1,
    pick_where_eq (dealer = 'dealer2',
    price) as dealer2
FROM bikes_2
GROUP BY prodName, orderNo;
```

In fact, this query is an implementation of the operation $\tau_{\text{dealer,price}}$(`bikes_2`).

At instance level both key equivalence conflicts and attribute conflicts have to be taken into consideration. For resolving key equivalence conflicts, there are two strategies supported in FraQL: first the standard SQL facilities where the join operation can be refined, e.g., by defining additional join conditions or user-defined predicates. Second, the key values for one or both of the relations can be transformed by a conversion function or mapping table, which are specified as part of the definition of an import view as shown above for representation conflicts.

For resolving this kind of conflicts several alternatives are possible. The simplest way is to

prodName	*orderNo*	*dealer1*	*dealer2*
XC-Shock	AD-1234	530	500
Grizzly	B-6081	1750	1690
M 8000	R-4010	1238	1190

(a) Relation *bikes_1*

prodName	*orderNo*	*dealer*	*price*
Wannabee	B-6070	dealer1	825
Wannabee	B-6070	dealer2	850
M 8000	R-4010	dealer1	1238
M 8000	R-4010	dealer2	1190

(b) Relation *bikes_2*

Fig. 5. Two relations containing meta conflicts.

define a projection for the preferred attribute. However, this is a static solution because this applies to all tuples. A more viable way is to compute the value of the resulting attribute dynamically from the input values or other attribute values.

In the following example, we want to integrate the bike relation with a second relation containing further descriptions for the respective model as well as most up-to-date prices (Fig. 6). Therefore, if the entry in the dealer relation refers to an earlier model year, the dealer price should be used (perhaps it is a phase-out model), otherwise the more recently price value from the vendor relation appears in the integrated result. This query can be formulated easily using the standard SQL `CASE` clause:

```
SELECT b.prodName, b.orderNo
    CASE WHEN b.year < bm.year THEN
    b.price ELSE bm.price END AS price
FROM bikes_A b JOIN bike_models bm ON
b.orderNo = bm.orderNo;
```

6.3. Semantic conflicts

In general, semantic conflicts are resolved by applying the union operator. However, as already mentioned representation conflicts at instance level could result in tuple identity problems (i.e., key equivalence conflicts, if key attributes are affected) or data discrepancies (i.e., attribute conflicts, if remaining attributes are affected).

In FraQL key equivalence conflicts are resolvable in two ways: either by transforming the keys of one relation with the help of conversion functions or mapping tables or by using the extended grouping operator in combination with aggregate functions for reconciling/merging the different representatives of a real-world entity.

orderNo	*price* ($)	*description*
AD-1234	530	Head shock, rear shock
B-6081	900	Head shock, disc brake
R-4010	1150	Head shock, aluminum frame

Fig. 6. Vendor description relation *bike_models*.

Considering the bike dealer databases, the relations could be integrated by the following definition without conflict resolution for the moment:

```
CREATE VIEW bikes OF bike_type AS
    bikes_A UNION bikes_B;
```

However, the result contains several attribute value conflicts like different product names, years, prices or stock values. We could solve these for example by summing up the stock, choose the most current year and the corresponding price etc. All these reconciliation tasks can be performed using aggregation functions. So, for the attribute `stock` the usage of `sum` as well as using `max` for attribute `year` are straightforward. The price for the most current year is obtained via `pick_where_max` and for picking just any product name we could use the `to_array` function and take the first element of the resulting array:

```
CREATE VIEW bikes OF bike_type AS
    SELECT vendor, pick_where_max(year,
    price),
        element(to_array   (prodName),
        1), orderNo, max(year), sum(-
        stock)
    bikes_A UNION ALL bikes_B
    GROUP BY vendor, orderNo;
```

We can conclude that detection and resolution of instance-level conflicts comprises three phases:

1. Homogenization of representations, i.e. resolving representation conflicts by defining attribute transformations. But because at this stage the detection of these conflicts is often only possible for obvious cases, this step is repeated after conflict detection of the subsequent steps.
2. Dealing with key equivalence conflicts, which can be resolved by treating them as representation conflicts (going back to the previous step) or by refining the predicate for deciding equivalence (the on clause of the join operation or the grouping expression).
3. Resolution of attribute value conflicts either by going back to step 1 or by defining reconciliation functions for the integration operations.

We have briefly shown how a query language with special conflict resolution mechanisms supports data integration and reconciliation. Based on these facilities a tool for example-driven integration has been developed, which we present in the next section.

7. Tool support for an example-driven approach

In the previous sections we have shown, how the FraQL language extensions support the detection and the resolution of integration conflicts on schema level as well as on instance level and therefore enable data reconciliation. However, the process of integration is particularly for larger projects a complex task, so that one-shoot-strategies are not realistic. Rather we have to consider integration as an interactive and iterative process, which requires tool support enabling the definition and evaluation of integration operations as well as direct analysis of the — possibly intermediate — integration results. This includes especially features likes performing conflict detection automatically, providing hints on potential conflicts and applying resolution mechanisms in a semi-automatic manner, e.g., based on examples provided by the user or derived from the current data.

In this section we present the main principles and components of such a tool. The basic idea of this approach is the combination of interactive query features known from Query-by-Example (QBE) [20] and facilities for data integration and reconciliation. A prototype of this tool called VIBE has been developed by using the FraQL language for accessing different data sources in an homogeneous way, for defining and retrieving schema elements as well as for performing queries.

Integration and reconciliation with the VIBE system works according to the process discussed in Section 4: A first coarse application model represented as a set of object types and their relationships establishes the starting point of the process. This model could be developed either bottom–up — as result of a schema integration process — or top–down by using given concepts, e.g. from a standardized domain model [21]. Next, the database integrator selects the required sources, browses the available local relations and imports the appropriate relations by defining FraQL import views. For this purpose an existing pre-defined object type can be chosen or a new one has to be defined. If the structure of the local relation does not exactly match the type, a mapping between local and global attributes must be defined. The graphical representation for this step is shown in Fig. 7.

In this table view the imported relation is displayed together with the mapping information. In the heading the global attribute names defined by the specified type of the relation are shown. The second row contains the mapping definition. Here, the name of the corresponding local attribute is given. If a mapping function or a mapping table is required, the name is inserted into the appropriate column. In addition, an expression can be entered, that is automatically translated into an user-defined function. For example, for term "`* 0.91`" of column `price` the following Java class for a FraQL function is generated:

```
class Func
    public static double func (double p)
    { return p * 0.91; }
```

After compiling and registering this function in the FraQL system, it is automatically applied as part of the mapping.

In the rows following the mapping row the database integrator can specify QBE-like selection queries. These queries are evaluated and the mapping is applied to the results. In this way,

vendor	*year*	*orderNo*	*prodName*	*price*	*stock*	} *relation schema*
`@map(...)`		`@map(...)`		`* 0.91`		} *mapping row*
Bianchi	2000	B-6070	Bianchi Wannabee	750	12	} *data view*
Bianchi	1999	B-6081	Bianchi Grizzly	1750	3	
Raleigh	2000	R-4010	Raleigh M 8000	1908	2	

Fig. 7. Import table view.

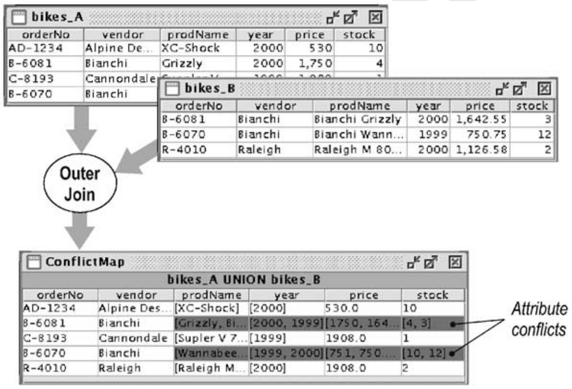
one receives direct feedback of the defined mapping by inspecting the data.

In the next step, the import relations are integrated using the ordinary join and union operators possibly in combination with grouping for resolving attribute value conflicts. This results in an integration graph specified as a view definition of an integration relation. A stepwise construction of this graph simplifies the detection and resolution of conflicts. So, based on the visualization of the integration graph, for each node the intermediate results can be inspected. There are two kinds of views for the results:

- a detailed data view in a tabular representation, where the data is displayed as result of a QBE-like query and
- a so-called conflict map — a special view which visualizes data discrepancies in a colored map (Fig. 8).

This map is constructed as follows: For a union as integration operation an outer join is computed and for each tuple appearing in both input relations (which is determined by comparing the primary keys) the corresponding attribute values are compared. Both values are presented in a single cell of the map, where the color depends on the comparison result. If both values are equal the color of the cell is white, otherwise red. Therefore, a red cell denotes an attribute conflict.

For a join operation the map is constructed by applying an outer join, too. In addition to the coloring for the union operation, a further kind of conflict is considered. Null values of attributes appearing in only one input relation are presented as yellow cells. So, these cells indicate key equivalence conflicts. In fact, not the actual values of the resulting tuples are important, but the colors. Therefore, a compact representation of conflict spots is possible. The user can zoom into the overview map and select points of interests for further examinations.



bikes_A

orderNo	vendor	prodName	year	price	stock
AD-1234	Alpine De...	XC-Shock	2000	530	10
B-6081	Bianchi	Grizzly	2000	1,750	4
C-8193	Cannondale				
B-6070	Bianchi				

bikes_B

orderNo	vendor	prodName	year	price	stock
B-6081	Bianchi	Bianchi Grizzly	2000	1,642.55	3
B-6070	Bianchi	Bianchi Wann...	1999	750.75	12
R-4010	Raleigh	Raleigh M 80...	2000	1,126.58	2

ConflictMap

bikes_A UNION bikes_B

orderNo	vendor	prodName	year	price	stock
AD-1234	Alpine Des...	[XC-Shock]	[2000]	530.0	10
B-6081	Bianchi	[Grizzly, Bi...	[2000, 1999]	[1750, 164...	[4, 3]
C-8193	Cannondale	[Supler V 7...	[1999]	1908.0	1
B-6070	Bianchi	[Wannabee...	[1999, 2000]	[751, 750....	[10, 12]
R-4010	Raleigh	[Raleigh M...	[2000]	1908.0	2

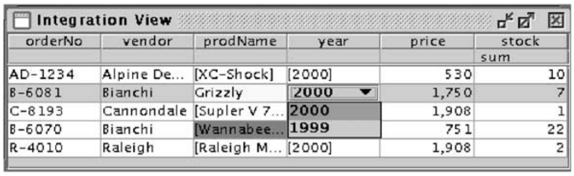
Fig. 8. Conflict map.

In addition, the conflict detection techniques described in Section 5 can be applied, if the extensional correspondences are known. In this case, comparing the cardinalities of input and result relations could give a hint about possible conflicts. Of course, more advanced techniques based on data analysis are possible, too.

Finally, if conflicts were detected, the integration operations have to be refined. As already shown in Section 6 there are two ways supported in FraQL: first modifying the comparison condition for joins or the grouping criteria for unions in combination with `GROUP BY` and second by adding a reconciliation function. Because specifying a condition is straightforward, we will focus in the following on support for applying reconciliation functions.

Basically, the integrated and possibly intermediate relation containing conflicts is visualized in a view similar to the conflict map, but with comboboxes in the cells where a conflict occurs (Fig. 9). The view contains an additional row for entering reconciliation functions for the respective columns. The specified functions are applied instantly to the integration operations and the view of the result relation is updated. There are three ways for defining these functions:

1. implementing and registering a function by hand and applying it as an aggregation function,



orderNo	vendor	prodName	year	price	stock
					sum
AD-1234	Alpine De...	[XC-Shock]	[2000]	530	10
B-6081	Bianchi	Grizzly	2000 ▼	1,750	7
C-8193	Cannondale	[Supler V 7...	2000	1,908	1
B-6070	Bianchi	[Wannabee...	1999	751	22
R-4010	Raleigh	[Raleigh M...	[2000]	1,908	2

Fig. 9. Integration table view.

2. entering expressions into the reconciliation row of the table,
3. deriving the reconciliation strategy from user-given examples.

For the second approach, a set of pre-defined primitives is available: `min`, `max`, `sum` and `avg` representing the standard SQL aggregate functions as well as `pmin`, `pmax` and `peq` (as short-cuts for `pick_where_min` etc.) for choosing a value depending on the minimum, the maximum or a certain value of another attribute. So, the term "`pmax(A2)`" in column `A1` has the following meaning, assuming $r_1 \ldots r_n$ are the input relations which are integrated in the view v by a `UNION ALL` operation and $r_1 \ldots r_m$ overlap semantically, i.e., share tuples with the same key:

$$t_v(A_1) := t_{r_i}(A_1) \text{ for } t_{r_i}(A_2) = \max(t_{r_1}(A_2), \ldots, t_{r_m}(A_2))$$

For example, this is used in the view `bikes` from Section 6 for resolving price conflicts, where `year` corresponds to `A2` and `price` to `A1`.

A third approach is to mark desired attribute values in tuples from the input relations. For the following basic but frequent cases the system is able to propose an appropriate reconciliation strategy. We assume $r_1 \ldots r_n$ to be the input relations with relation schema R which are integrated in the view v with the following properties: PK is the primary key and the source is explicitly given as an attribute SRC in each input relation. This can be achieved by defining a literal value for this attribute as part of the input view definition, e.g. as in the following example:

```
CREATE VIEW r1 OF R AS IMPORT FROM db.src1
(
    src IS 'src1',
    …
);
```

In fact, this approach simulates a source-aware model as proposed in [22].

Let be further $t \in r$ a tuple from relation r, $t(A)$ the value of attribute A for the tuple t, $s_{i,A} \subseteq r_i$ the set of tuples of r_i where examples for attribute A are selected by the user, cg_{A,t_i} a so-called conflict group for a tuple $t_i \in v$ with regard to attribute $A \in R$ where it holds: $cg_{A,t_i} = \{t_{i,r_1}(A), \ldots, t_{i,r_m}(A)\}$ and $\forall j,k = 1, \ldots, m, j \neq k : t_{i,r_j}(PK) = t_{i,r_k}(PK) \wedge t_{i,r_j}(A) \neq t_{i,r_k}(A)$. This means the set of conflicting values of tuple t_i for the attribute A that are caused by semantic overlapping of $r_1 \cdots r_m$. Finally, we denote the set of all conflict groups of an attribute A as CG_A.

Regarding an examined attribute A we can define the following heuristics:

1. If $|s_j| > 0 \wedge \forall i = 1 \ldots n, i \neq j : |s_i| = 0$
   Choose `peq`($SRC = j$)
   i.e., if all selected examples of attribute A are from relation r_j, then choose always the values from this relation in case of conflict.
2. If $\forall g \in CG_A : t_s(A) \in g$ is the selected example $\wedge\, t_s(A) = \min_j \in g\{t_j(A)\}$
   Choose `min`
   This means, if the given examples are from different relations and the selected values of attribute A is always the smallest of its conflict group, then choose the minimum of these values in case of conflict.
3. If $\forall g \in CG_A : t_s(A) \in g$ is the selected example $\wedge\, t_s(A) = \max_j \in g\{t_j(A)\}$
   Choose `max`
   This means the same as rule 2 but for the maximum.
4. Let $B_i \in R - \{A, PK\}$
   If $\forall g \in CG_A : t_s(A) \in g$ is a selected example $\wedge$
   for the corresponding value of B_i it holds: $t_s(B_i) = \min(cg_{t_s,B_i}) \wedge$
   $|CG_{B_i}| = \max_j\{|CG_{B_j}|\}$
   Choose `pmin`(B_i)
   i.e., check, if the value of an attribute B_i of a tuple, which is selected as an example for reconciling conflicts of A, is always the minimum of its own conflict group. If this condition is satisfied, choose `pmin`(B_i), where B_i is the attribute with the largest number of conflict groups.
5. Let $B_i \in R - \{A, PK\}$
   If $\forall g \in CG_A : t_s(A) \in g$ is a selected example $\wedge$
   for the corresponding value of B_i it holds: $t_s(B_i) = \max(cg_{t_s,B_i}) \wedge$
   $|CG_{B_i}| = \max_j\{|CG_{B_j}|\}$
   Choose `pmax`(B_i)

As shown in rule 4 but only for the maximum.

These rules are evaluated in the given order. If a precondition is fulfilled, the corresponding reconciliation primitive is applied to the column. As an example please consider Fig. 9. If we select “Grizzly” and “Wannabee” in the column `prodName`, the reconciliation expression `peq (src = 'bikes_A')` is derived according to rule (1) under the assumption of source-aware import views as described above. If we select “1750” and “750” in column `price`, the primitive `pmax(year)` is derived by rule (5). Obviously, these are rather simple cases, but for larger relations and more user-given examples the system is able to propose useful reconciliation expressions which can be directly mapped to FraQL aggregation functions.

The described steps of conflict detection and — if necessary — conflict resolution are performed for each node along the integration graph. The result of the integration process provided by the VIbE tool is the integrated schema for FraQL. This schema definition contains all required mapping information for schema translation and conflict resolution which are performed by the FraQL query processor. At this stage, an application can query the integrated and (hopefully) conflict-free data.

8. Related work

The problem of schema integration is addressed by several approaches, which are surveyed for example in [23,24]. For describing conflicts arising in the integration phase various classifications were developed, e.g. in [17,25,18].

Structural conflicts and resolution strategies are discussed in detail in [26]. Techniques for managing schematic heterogeneity (meta conflicts) based on SchemaSQL features are presented in [27]. Resolving description conflicts by using a rule-based data conversion language is described in [28,29] present a schema-based data translation solution. In [30] solving domain and schema mismatch problems with an object-oriented database language is discussed.

For problems of instance integration several solutions have been proposed. The work in [31] examines the entity identification problem, formulates it as a matching problem and defines soundness as well as completeness as important properties of the entity identification process.

An approach for resolving attribute value conflicts based on Dempster–Shafer theory, which assigns probabilities to attribute values is described in [32]. In [16] an object-oriented data model is introduced where each global attribute consists of the original value, the resolved value and the conflict type. These individual values are accessible by global queries. In addition, for each attribute a threshold predicate determining tolerable differences, and a resolution function for an automatic conflict resolution can be defined. In [33,22] approaches are proposed, where the origin of integrated data is included as an additional tuple attribute in order to improve the interpretation of global data. Another approach, presented in [34], introduces the notion of semantic values enabling the interoperability of heterogeneous sources by representing context information. In contrast, the intention of our approach is to support conflict detection and resolution based on the analysis of data in order to provide a conflict-free global view.

An advanced application of statistical data analysis for deriving mapping functions for numerical data is described in [35]. The integration of similar techniques in our tool could improve the usability for more complex scenarios. In [36] a data cleaning framework consisting of operators like mapping, view, matching, clustering, and merging is presented. These operators are embedded in a declarative language, which allows to specify the flow of logical transformations. Another data cleaning system is Potter’s Wheel [37], an interactive tool for building transformations to clean data. In contrast to these both systems our approach focuses on integration and basic cleaning operations as primitive of a multidatabase query language. However, we share the idea of an example-based approach for specifying data transformations in combination with a query system.

A totally different approach for dealing with instance heterogeneity during integration is presented in [38]. Here, textual similarity is used for computing joins between relations from different sources. This permits integration without normalization of values, but is restricted to textual data.

Several approaches are dedicated to declarative integration based on answering queries using views. In data integration systems like the one described in [39] the contents of the sources are specified as views over the global schema (the so-called *local-as-view* approach). Therefore, queries on the global schema have to be rewritten into queries referring to the source schemas. A good survey on this problem is given in [6]. In [40] several declarative language techniques for describing the content of sources are described. Another declarative approach addressing the integration problem in Data Warehousing is presented in [41]. It is based on the specification of reconciling correspondences between data in different sources which are used for query rewriting.

Query languages supporting the integration of heterogeneous sources are multidatabase languages like MSQL [2], SQL/M [42] and SchemaSQL [3]. MSQL provides basic features for accessing schema labels and converting them into data values. SQL/M addresses mainly description conflicts by providing mechanisms for scaling and unit transformation. More advanced conflict resolution is addressed for example by the restructuring techniques proposed in SchemaSQL supporting the specification of relations with data dependent output schemata. Our language FraQL extends these by additional resolution techniques for description and structural conflicts as well as instance-level conflicts. An algebra for data integration operations in federated database systems, which are similar to our language extensions is presented in [43].

Examples of system implementations addressing reconciliation of heterogeneous data are federated database system like Pegasus [44] or IBM DataJoiner [45] as well as mediator-based systems like TSIMMIS [46] or Information Manifold [39]. Pegasus uses a functional object-oriented data manipulation language called HOSQL with non-procedural features, DataJoiner is based on DB2 and therefore provides essentially standard SQL features for conflict resolution. In mediator systems such as TSIMMIS the mediator is specified by a set of rules. Each rule maps a set of source objects into a virtual mediator object. In this way, conflicts are resolved by defining appropriated rules. The special problem of combining objects from different sources (object fusion) in mediators is addressed in [47]. Another interesting mediator system concerning with conflict resolution is AURORA [48]. It supports so called conflict tolerant queries by allowing predicate evaluation parameters as part of the selection, e.g., "high confidence", which means that if an inconsistent value exists, the selection condition is satisfied only if all sources agree or "possible at all", if for at least one source the condition is satisfied. Conflict resolution is performed similar to our approach using aggregation functions.

Several tools supporting database integration are available, e.g. [49–51]. However, these mainly address schema integration and the resolution of schema-level conflicts. Our approach comprising the tool VIBE and the query system FRAQL supplements these systems by considering the instance level and providing an more interactive and data-centered method.

9. Conclusion

Modern information infrastructures are based on distributed systems with several independent data sources. In such an environment, the integrated access to distributed data stored in several more or less autonomous component databases remains an important problem. Experiences with building such information federations have shown that the integration process is the bottleneck for building a federation and that it is impossible to automatise all aspects of integration because of the involved semantic heterogeneity.

This paper proposes an interactive and example-driven integration process combining automatic support with interactive choice of integration steps. Such conflict resolution and data reconciliation steps are important aspects of integrating heterogeneous data sources. Our approach is based on the multidatabase query language FraQL providing advanced conflict resolution mechanisms being an upward compatible extension of standard SQL. The main issue of the presentation is the combination of this query language providing advanced conflict resolution mechanisms with an interactive query and definition tool with extensible support for conflict detection. For the advanced features of FraQL we present an integration into the framework of relational algebra, thereby, providing a basis for adopting and extending standard techniques for query optimization.

The integration process following our method is a tight coupling of data inspection on the instance level and conflict resolution on the schema level. The inspection of the instance level guides the schema level conflict resolution by presenting sample conflicts and partly generating proposals for conflict resolution functions. Such an interactive, example-based approach with immediate feedback may be better suited for smaller integration tasks than learning and using a new integration formalism as proposed by other approaches.

The FraQL query processing system has been implemented in C++. The current prototype supports all the features presented in this paper and provides access to Oracle and MySQL databases as well as to structured, file-based sources like XML documents and Web pages. The VIbE prototype as an interactive design and query frontend to the FraQL system has been developed in Java using JDBC. Besides the available basic facilities for conflict detection and reconciliation it is extensible by more advanced techniques which can be plugged into the system.

Current work includes the extension of the framework towards the full object-relational model of SQL-99, the handling of new conflict situations resulting from this extension and the support of more advanced data cleaning tasks. First results of this work are presented in [52]. The proposed integration methodology can be transferred to other canonical data models like ODMG or XML. Our approach is applied in a digital library scenario, where bibliographical data from heterogeneous libraries has to be integrated [53], as well as in a data preparation and analysis environment for information fusion.

References

[1] K. Sattler, S. Conrad, G. Saake, Adding conflict resolution features to a query language for database federations, Aus. J. Inform. Sys 8 (1) (2000) 116–125.

[2] J. Grant, W. Litwin, N. Roussopoulos, T. Sellis, Query languages for relational multidatabases, VLDB J. 2 (2) (1993) 153–171.

[3] L.V.S. Lakshmanan, F. Sadri, I.N. Subramanian, SchemaSQL — A language for interoperability in relational multi-database systems, in: T.M. Vijayaraman, A.P. Buchmann, C. Mohan, N.L. Sarda, (Eds.), VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, 1996, Mumbai (Bombay), India, Morgan Kaufmann, 1996, pp. 239–250.

[4] M.T. Roth, P.M. Schwarz. Don't scrap it, wrap it! a wrapper architecture for legacy data sources, in: M. Jarke, M.J. Carey, K.R. Dittrich, F.H. Lochovsky, P. Loucopoulos, M.A. Jeusfeld, (Eds.), VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, Athens, Greece, August 25–29, 1997, Morgan Kaufmann, 1997, pp. 266–275.

[5] K. Sattler, G. Saake, Supporting Information Fusion with Federated Database Technologies, in: S. Conrad, W. Hasselbring, G. Saake, (Eds.), Proceedings of secnd International Workshop on Engineering Federated Information Systems, EFIS'99, Kühlungsborn, Germany, May 5–7, 1999, infix-Verlag, Sankt Augustin, 1999, pp. 79–184.

[6] A. Halevy, Answering queries using views: a survey, VLDB J. 10 (4) (2001) 270–294.

[7] M. Endig, M. Höding, G. Saake, K. Sattler, E. Schallehn, Federation services for heterogeneous digital libraries accessing cooperative and non-cooperative sources, in: Proceedings of Kyoto International Conference on Digital Libraries: Research and Practice, IEEE Computer Society Press, 2000, pp. 314–321.

[8] G. Jaeschke, H.-J. Schek, Remarks on the algebra of non first normal form relations, in: Proceedings of the ACM Symposium on Principles of Database Systems, Los Angeles, CA, March 29–31 1982, ACM, 1982, pp. 124–138.

[9] S. Abiteboul, N. Bidoit, Non first normal form relations: an algebra allowing data restructuring, J. Comput. Syst. Sci. 33 (1986) 361–393.

[10] G.M. Shaw, S.B. Zdonik, An object-oriented query algebra, IEEE Data Eng. 12 (3) (1989) 29–36.

[11] S. Cluet, C. Delobel, C. Lecluse, P. Richard, RELOOP, an algebra based query language for an object-oriented database system, in: W. Kim, J.-M. Nicolas, S. Nishio, (Eds.), Deductive and Object-Oriented Databases, Proceedings of the 1st International Conference DOOD'89, Kyoto, Japan, December, 1989, Amsterdam. North-Holland 1990, pp. 294–313.

[12] G. Saake, R. Jungclaus, C. Sernadas, Abstract data type semantics for many-sorted object query algebras, in: B. Thalheim, J. Demetrovics, H.-D. Gerhardt (Eds.), Proceedings of the third Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems, MFDBS'91, Rostock, Germany, Lecture Notes in Computer Science, Vol. 495 Berlin, Springer, 1991, pp. 291–307.

[13] M.H. Scholl, H.-J. Schek, M. Tresch, Object algebra and views for multi-objectbases, in: M.T. Özsu, U. Dayal, P. Valduriez (Eds.), Distributed Object Management, Kaufmann Publishers, San Mateo, CA, Morgan, 1994, pp. 353–374.

[14] K.A. Ross, Relations with relation names as arguments: algebra and calculus, in: Proceedings of the 11th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, San Diego, CA, ACM Press, 1992, pp. 346–353.

[15] S. Chaudhuri, R. Motwani, and V. Narasayya, On random sampling over joins. in: A. Delis, C. Faloutsos, S. Ghandeharizadeh, (Eds.), SIGMOD 1999, Proceedings of the ACM SIGMOD International Conference on Management of Data, Philadelphia, PA, USA, June 1–3, 1999, ACM Press, New York, 1999, pp. 263–274.

[16] E.-P. Lim, R.H.L. Chiang, A global object model for accommodating instance heterogeneities, in: Tok Wang Ling, S. Ram, Mong-Li Lee, (Eds.), Conceptual Modeling – ER '98, 17th International Conference on Conceptual Modeling, Singapore, November 16–19, 1998, Proceedings, Lecture Notes in Computer Science, Vol. 1507, Springer, Berlin, 1998, pp. 435–448.

[17] W. Kim, J. Seo, Classifying Schematic and Data Heterogeneity in Multidatabase Systems, IEEE Computer 24 (12) (1991) 12–18.

[18] S. Spaccapietra, C. Parent, Y. Dupont, Model independent assertions for integration of heterogeneous schemas, VLDB J. 1 (1) (1992) 81–126.

[19] G. Saake, S. Conrad, I. Schmitt, Database design, in: J.G. Webster (Ed.), Wiley Encyclopedia of Electrical and Electronics Engineering, Vol. 4, Wiley, New York, 1999, pp. 540–567.

[20] M.M. Zloof, Query by example: a data base language, IBM Sys. J. 16 (4) (1997) 324–343.

[21] W. Hasselbring, Top–down vs bottom–up Engineering of Federated Information Systems, in: S. Conrad, W. Hasselbring, G. Saake, (Eds.), Proceedings of the second International Workshop on Engineering Federated Information Systems, EFIS'99, Kühlungsborn, Germany, May 5–7, 1999, infix-Verlag, Sankt Augustin, 1999, pp. 131–138.

[22] E.-P. Lim, R.H.L. Chiang, Y. Cao, Tuple source relational model: a source-aware data model for multidatabases, Data & Knowledge Engineering 29 (1) (1999) 83–114.

[23] C. Batini, M. Lenzerini, S.B. Navathe, A Comparative analysis of methodologies for database schema integration, ACM Comput. Surv 18 (4) (1986) 323–364.

[24] E. Pitoura, O. Bukhres, A.K. Elmagarmid, Object orientation in multidatabase systems, ACM Computing Surveys 27 (2) (1995) 141–195.

[25] F. Saltor, M. Castellanos, M. Garcia-Solaco, Overcoming schematic discreprancies in interoperable databases, in: D.K. Hsiao, E.J. Neuhold, R. Sacks-Davis, (Eds.), Interoperable Database Systems, Proceedings of the IFIP WG 2.6 Database Semantics Conference DS-5, Lorne, Victoria, Australia, November, 1992, North-Holland, Amsterdam, 1993, pp. 191–205.

[26] W. Kim, I. Choi, S. Gala, M. Scheevel, On resolving schematic heterogeneity in multidatabase systems, in: W. Kim (Ed.), Modern Database Systems, ACM Press, New York, NJ, 1995, pp. 521–550 chapter 26.

[27] R.J. Miller, Using schematically heterogeneous structures, in: L.M. Haas, A. Tiwary (Eds.), SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 2–4, 1998, ACM Press, 1998, pp. 189–200.

[28] S. Cluet, C. Delobel, J. Siméon, and K. Smaga. Your mediators need data conversion!, in: L.M. Haas, A. Tiwary, (eds)., SIGMOD 1998, Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 2–4, 1998, ACM Press, 1998, pp. 177–188.

[29] T. Milo, S. Zohar, Using schema matching to simplify heterogeneous data translation, in: A. Gupta, O. Shmueli, J. Widom, (Eds.), VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, New York, NY, USA, August 24–27, 1998, Morgan Kaufmann, 1998, pp. 122–133.

[30] W. Kent, A rigorous model of object reference, identity, and existence, J. Object-Oriented Programming (1991) 28–36.

[31] E.-P. Lim, S. Prabhakar, Entity identification in database integration, in: Proceedings of the Ninth International Conference on Data Engineering (ICDE'93), Vienna, Austria, April 19–23, 1993, 1993, pp. 154–163.

[32] E.-P. Lim, J. Srivastava, S. Shekhar, Resolving attribute incompatibility in database integration: an evidential reasoning approach, in: Proceedings of the 10th IEEE International Conference on Data Engineering, ICDE'94, Houston, Texas, USA, Los Alamitos, CA, 14–18 February 1994, IEEE Computer Society Press, 1994, pp. 154–163.

[33] Y. Richard Wang, Stuart E. Madnick, A polygen model for heterogeneous database systems: the source tagging perspective, in: D. McLeod, R. Sacks-Davis, H-J Schek, (Eds.), Proceedings of 16th International Conference on Very Large Data Bases, Brisbane, Queensland, Australia, August 13–16, 1990, Morgan Kaufmann, 1990, pp. 519–538.

[34] E. Sciore, M. Siegel, A. Rosenthal, Using semantic values to facilitate interoperability among heterogeneous information systems, ACM Trans. Database Syst. 19 (2) (1994) 254–290.

[35] H. Lu, W. Fan, C. Goh, S. Madnick, D. Cheung, Discovering and reconciling semantic conflicts: a data mining perspective, in: Proceedings of the seventh IFIP 2.6 Working Conference on Data Semantics (DS-7), Leysin, Switzerland, 1997.

[36] H. Galhardas, D. Florescu, D. Shasha, E. Simon, C. Saita, Declarative data cleaning: language, model, and algorithms, in: VLDB'01, Proceedings of 27th International Conference on Very Large Data Bases, 2001, Roma, Italy, 2001, pp. 371–380.

[37] V. Raman, J.M. Hellerstein, Potter's wheel: an interactive data cleaning system, in: VLDB'01, Proceedings of 27th International Conference on Very Large Data Bases, 2001, Roma, Italy, 2001, pp. 381–390.

[38] W.W. Cohen, Integration of heterogeneous databases without common domains using queries based on textual similarity, in: L.M. Haas, A. Tiwary (Eds.), SIGMOD 1998, Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, Washington, USA, June 2–4, 1998, ACM Press, 1998, pp. 201–212.

[39] A.Y. Levy, A. Rajaraman, and J.J. Ordille, Querying Heterogeneous Information Sources Using Source Descriptions, in: T.M. Vijayaraman, A.P. Buchmann, C. Mohan, N.L. Sarda, (Eds.), VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, 1996, Mumbai (Bombay), India, Morgan Kaufmann, Los Alios, CA, 1996, pp. 251–262.

[40] A. Levy, Logic-based techniques in data integration, in: J. Minker (Ed.), Logic Based Artificial Intelligence, Kluwer Publishers, 2000.

[41] D. Calvanese, G. De Giacomo, M. Lenzerini, D. Nardi, R. Rosati, Data integration in data warehousing, Int. J. Cooperative Inform. Syst. 10 (3) (2001) 237–271.

[42] W. Kelley, S. Gala, W. Kim, T. Reyes, B. Graham, Schema architecture of the UniSQL/M multidatabase system, in: W. Kim (Ed.), Modern Database Systems, ACM Press, New York, NJ, 1995, pp. 621–648 chapter 30.

[43] C. Wyss, D. Van Gucht, A relational algebra for data/metadata integration in a federated database system, in Proceedings of the 2001 ACM CIKM International Conference on Information and Knowledge Management, Atlanta, Georgia, USA, 2001, pp. 65–72.

[44] R. Ahmed, P. De Smedt, W. Du, W. Kent, M.A. Ketabchi, W. Litwin, A. Rafii, M.-C. Shan, The Pegasus heterogeneous multidatabase system, IEEE Comput. 24 (12) (1991) 19–27.

[45] S. Venkataraman, T. Zhang, Heterogeneous database query optimization in DB2 universal dataJoiner, in: A. Gupta, O. Shmueli, J. Widom, (Eds.), VLDB'98, Proceedings of 24rd International Conference on Very Large Data Bases, 1998, New York, New York, USA, Morgan Kaufmann, 1998, pp. 685–689.

[46] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, . Sagiv, J.D. Ullman, V. Vassalos, J. Widom, The TSIMMIS approach to mediation: data models and languages, J. Intell. Inform. Syst. 8 (2) (1997) 117–132.

[47] Y. Papakonstantinou, S. Abiteboul, and H. Garcia-Molina, Object fusion in mediator systems, in: T.M. Vijayaraman, A.P. Buchmann, C. Mohan, N.L. Sarda, (Eds.), VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases, 1996, Mumbai (Bombay), India, Morgan Kaufmann, 1996, pp. 413–424.

[48] L.L. Yan, M. Tamer Özsu, Conflict Tolerant Queries in AURORA, in: Proceedings of the fourth IFCIS International Conference on Cooperative Information Systems (CoopIS), Edinburgh, Scotland, 1999, pp. 279–290.

[49] S. Castano, V. De Antonellis, A schema analysis and reconciliation tool environment for heterogeneous databases, in: Proceedings of 1999 International Database Engineering and Applications Symposium, IDEAS 1999, Montreal, Canada, 2 – 4 August, 1999, 1999, pp. 53–62.

[50] J.-L. Hainaut, P. Thiran, J.-M. Hick, S. Bodart, A. Deflorenne, Methodology and case tools for the development of federated databases, Int. J. Cooperative Inform. Sys. 8 (2–3) (1999) 169–194.

[51] K. Schwarz, I. Schmitt, C. Türker, M. Höding, E. Hildebrandt, S. Balko, S. Conrad, G. Saake, Design support for database federations, in: J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, E. Métais, (Eds.), Conceptual Modeling – ER'99 (18th International Conference on Conceptual Modeling, Paris, France, November 15–18, 1999, Proceedings), Lecture Notes in Computer Science Vol. 1728, Springer, Berlin, 1999, pp. 445–459.

[52] K. Sattler, E. Schallehn, A data preparation framework based on a multidatabase language, in: M. Adiba, C. Collet, B.P. Desai, (Eds.), Proceedings of International Database Engineering and Applications Symposium (IDEAS 2001), IEEE Computer Society, Grenoble, France. 2001, pp. 219–228.

[53] E. Schallehn, M. Endig, K. Sattler, Citation Linking in Federated Digital Libraries, in: M. Roantree, W. Hasselbring, S. Conrad, (Eds.), Proceedings of the third International Workshop on Engineering Federated Information Systems, EFIS'00, Dublin, Ireland, June, Verlagsgesellschaft, Berlin, Akadem, 2000, pp. 53–60.