Example-driven Integration of Heterogeneous Data Sources

Kai-Uwe Sattler\textsuperscript{1}  Stefan Conrad\textsuperscript{2}  Ingolf Geist\textsuperscript{1}  Gunter Saake\textsuperscript{1}

\textsuperscript{1} Department of Computer Science, University of Magdeburg
P.O. Box 4120, D-39016 Magdeburg, Germany
\{kus\mid geist\mid saake\}@iti.cs.uni-magdeburg.de

\textsuperscript{2} Department of Computer Science, University of Munich
Oettingenstr. 67, D-80538 München, Germany
conrad@dbs.informatik.uni-muenchen.de

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 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 is presented. In addition, we discuss basic techniques for supporting the detection of instance-level conflicts.

1 Motivation

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 the 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 treats mainly 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 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 focused on detection and resolution of instance-level conflicts. It is based on the multidatabase query language FRAQL \cite{SCS00}, which extends SQL by advanced conflict resolution mechanisms. In conjunction with the interactive query and design tool VI\textsc{bewerbe} we are able to support a technique, which we call in the following \textit{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 followed. 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. Next, in section 5 we discuss identification and resolution of instance-level conflicts. Section 6 presents the Vibe prototype and sketches the support for example-driven integration. Related work is discussed in section 7. Finally, section 8 concludes the paper.

2 FraQL: An Overview

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 [GLRS93] or SchemaSQL [LSS96], but in contrast to these proposals FraQL is extensible by user-defined data types and functions. FraQL is not 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 [RS97]. FraQL is based on an simple object-relational data model: it supports the definition of object types and object tables derived from types in the spirit of SQL99. 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.

Object types describe the structure of objects as sets of attributes and their domains. Types can be organized in a specialization hierarchy. Object tables represent global virtual relations of the federation, i.e. data form the sources are not materialized, except for caching purposes in order to speed up query evaluation. Here we distinguish between import and integration relations.

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

```
create table global_name of type_name
  as import from source.local_name
  [ mapping_definitions ];
```

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

- Without an explicit mapping, a local attribute corresponding to an attribute defined by the global object type in terms of identifier and type becomes an attribute of the global relation.
• The notation \( g_{name} \textit{ is } l_{name} \) means renaming the local attribute to \( g_{name} \). This requires type compatibility.

• The notation \( g_{name} \textit{ is } \text{func}(l_{name}) \) defines that the global attribute value is calculated by using the user-defined conversion function \( \text{func} \) on the local attribute value.

• The definition \( g_{name} \textit{ is } @tbl (l_{name}, src, dest, default) \) 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, whereas the NULL value is replaced by the default value \( default \).

An integration relation is a view on other global relations combined by using operators like union, \( \theta \)-join and outer join. In addition, the standard SQL operations selection and projection are provided, too. An integration relation is defined as follows, where the term \( \text{table_expression} \) denotes a SQL view definition with extensions explained later.

```
create table global_{name} of type_{name}
   as table_expression;
```

Furthermore, FRAQL supports user-defined functions, which are stored in the database of the federation layer (i.e., in the query processing server) and are callable in queries. They are used as conversion functions in import relations as well as reconciliation functions for resolving data conflicts during integration operations. These functions are implemented in Java and registered in the query system.

Another feature of FRAQL for resolving conflicts is restructuring of relations, which is implemented in a way inspired by SchemaSQL. Variables of a query can not only be bound to relations as tuple variables, but also to meta-data from the schema catalog, like the set of attributes of a relation or the set of relations of a schema. Naturally, any global user relation with information about other relations can be used as meta-data source, too.

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. A detailed description of restructuring techniques in FRAQL is presented in [SCS00].

### 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 extended join operator, and

• the extended union operator.
**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).

The declaration of an import relation in **FRAQL** consists in principle of three parts 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.

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 a bunch of work on extended relational algebras e.g. for extended database models (cf. e.g. [JS82, AB86, SZ89, CDLR90, SJS91, SST94]). In the following we represent the application of (user-defined) functions as algebraic operation:

\[
\alpha_{A,f}(r)
\]

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 \( R' \) with 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 \( \alpha_f(r) \).
The extended join operator.

FRAQL offers an extended version of the standard join operator for integrating two relations by joining and applying a reconciliation function for resolving possible conflicts between certain attributes. A reconciliation function is an user-defined function, which is called for each pair of tuples fulfilling the comparison condition. The affected tuples are passed as arguments to the function, the resulting tuple is inserted into the global relation. In this way, the value of a global attribute can be computed from the (possibly) conflicting values of the corresponding local attributes, e.g. as the average of both values.

The general form of this extended join operator is as follows:

\[ r \text{ join } s \text{ on } \phi \text{ reconciled by } f \]

where \( r \) and \( s \) are relations (with schema \( R \) and \( S \), respectively), \( \phi \) is a (nearly arbitrary) join condition, and \( f \) is a reconciliation function (type restrictions for \( f \) are introduced below).

The extended join operation of FRAQL bases on a standard join operation with a join condition\(^1\). Therefore, the \texttt{reconciled by} clause is optional.

Without a reconciliation function \( r \text{ join } s \text{ on } \phi \) is considered to be a Cartesian product of \( r \) and \( s \) with a subsequent selection using the selection condition \( \phi \), i.e. \( \sigma_{\phi}(r \times s) \) in terms of relational algebra (this can also be written as \( r \bowtie_{\phi} s \)). In consequence, the schema of the resulting relation is \( RS \), that is the collection of all attributes from \( R \) and \( S \) where the name of the original relation is added as prefix to all attribute names; please note that we assume that the order of the attributes has no relevance such that the relation schemata \( RS \) and \( SR \) cannot be distinguished.

We restrict reconciliation functions in a way that adding a reconciliation function to a join does not change the schema of the result. In consequence, we require reconciliation functions not to change the type. A reconciliation function \( f \) for a join between two relations \( r \) and \( s \) with schema \( R \) and \( S \), respectively, takes a tuple of the Cartesian product \( r \times s \) as input and produces a tuple of the same type as result, i.e. \( f : RS \rightarrow RS \). Thereby, reconciliation functions can be applied in the same way as (user-defined) functions to relations as described above.

In summary, the FRAQL join operation with a reconciliation function \( f \)

\[ r \text{ join } s \text{ on } \phi \text{ reconciled by } f \]

can algebraically be represented as follows:

\[ \alpha_f(\sigma_{\phi}(r \times s)) \]

The extended union operator.

Often, integration can also be achieved by some kind of union. However due to several possible kinds of conflicts the union operator of SQL is not able to deal with most of the interesting integration situations. Therefore, FRAQL provides an extended version of the union operator (in addition to the standard union operator in SQL) in order to incorporate conflict resolution. The syntax for the FRAQL union operator is as follows:

\[ r \text{ union } s \text{ on } (A_1, \ldots, A_k) \text{ reconciled by } f \]

\(^1\)In our current implementation, we allow as join conditions arbitrary logical conditions without quantification but with arithmetic expressions involving the attributes of the participating relations.
where \( r \) and \( s \) are relations with schema \( R \) and \( S \), respectively, such that \( R \) and \( S \) contain the same set of attribute names and for each attribute \( A \) the types of \( A \) in \( R \) and \( S \) are compatible\(^2\). \( A_1, \ldots, A_k \) is a list of attribute names of \( R \) (and as well of \( S \)). \( f : RS \to (R \cup S) \) is a reconciliation function which takes a tuple from the Cartesian product of \( r \) and \( s \) (a tuple of schema \( RS \) which in fact is composed from one tuple of \( r \) and one tuple of \( s \)) and produces a tuple of the type \( R \cup S \). The type \( R \cup S \) is the common super type being compatible to both original types. This type has exactly the same attributes (names) but each attribute obtains

- the same type as it has in \( R \) and \( S \) if it is the same \( R \) and \( S \), or
- the more general type from the two it has in \( R \) and \( S \) (please note that we require this two types to be compatible such that the more general type is always uniquely determined).

In contrast to the join operator the reconciled by clause is required for the extended FRAQL union operator. This reflects the intuition that a union of two sets of objects having the same or at least compatible types should always be one set of the same or a compatible type.

A union defined by means of the extended FRAQL union operator can in principle be computed in the following way:

1. By projection onto the attributes from the \( \text{on} \) list we obtain from \( r \) and \( s \) two derived relations \( r' \) and \( s' \), respectively.
2. By means of a standard union (in the set-theoretic sense, i.e. eliminating duplicates) we obtain \( r s' \).
3. The missing attributes from \( r \) and \( s \) are now (re)added by executing outer joins with \( r \) and \( s \) (the order of these outer joins does not matter; furthermore, instead of full outer joins it is sufficient to compute left (LOJ) or right outer joins (ROJ) such that only in case that \( r \) or \( s \) does not contain a corresponding tuple NULL values for the missing attributes are inserted).
4. By projection we take care that each attribute from the original relations \( r \) and \( s \) is only represented once, i.e. for each original attribute name we have now two attributes, one from \( r \) and one from \( s \).
5. By applying the reconciliation function to each tuple produced in the preceding step we compute from the pair of values for each attribute one resolved value. For the attributes in the \( \text{on} \) list the resolved value should be the common value or the value which is different from the null value (these are the only two possible cases).

Obviously, the extended union operation with a reconciliation function \( f \)

\[
r \cup s \text{ on } (A_1, \ldots, A_k) \text{ reconciled by } f
\]

has a rather complex algebraic representation:

\[
\alpha_f(\pi_{\text{att}(R)}, \pi_{\text{att}(S)}) (r \cup (s \text{ where } a \text{ qualified by the name of the original relation as prefix.})
\]

\[^2\]The notion of type compatibility is adopted from SQL where e.g. all numerical types are compatible among each other.
4 Integration Process

The core concepts of the FRAQL data model correspond 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.

By examining this data and performing appropriated queries, one 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 the extednted join and union operations. This procedure is shown in Fig. 1.

![Figure 1: Integration process](image)

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 [CMN99].

With knowledge about existing instance conflicts the definitions of import and integration relations are refined, i.e., transformation functions are introduced for attribute mapping, join and union predicates are modified and reconciliation functions are applied. These steps are repeated until no conflicts remain. The final definitions of import and integration relations 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 data are used for conflict identification as well as evaluating integration operations. Finally, success of the conflict resolution strategies is immediately visible in terms of these 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 [LC98].

5 Integration Conflicts: Detection and Resolution

Due to the 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., [KS91, SPD92]. In contrast, we will focus in this paper on instance-level conflicts. In the following, we introduce a simple classification and discuss detection and resolution 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 [LC98].

<table>
<thead>
<tr>
<th>vendor</th>
<th>year</th>
<th>orderNo</th>
<th>prodName</th>
<th>price ($)</th>
<th>stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpine Design</td>
<td>2000</td>
<td>AD-1234</td>
<td>XC-Shock</td>
<td>530</td>
<td>10</td>
</tr>
<tr>
<td>Bianchi</td>
<td>2000</td>
<td>B-6081</td>
<td>Grizzly</td>
<td>1750</td>
<td>4</td>
</tr>
<tr>
<td>Cannondale</td>
<td>1999</td>
<td>C-8193</td>
<td>Supler V 700</td>
<td>1908</td>
<td>1</td>
</tr>
</tbody>
</table>

(a) Relation for dealer A bikes

<table>
<thead>
<tr>
<th>prodNo</th>
<th>year</th>
<th>prodName</th>
<th>price (Euro)</th>
<th>stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>0431-860</td>
<td>1999</td>
<td>Bianchi Wannabe</td>
<td>825</td>
<td>12</td>
</tr>
<tr>
<td>0431-871</td>
<td>2000</td>
<td>Bianchi Grizzly</td>
<td>1805</td>
<td>3</td>
</tr>
<tr>
<td>0431-241</td>
<td>2000</td>
<td>Raleigh M 8000</td>
<td>1238</td>
<td>2</td>
</tr>
</tbody>
</table>

(b) Relation for dealer B bikes

Figure 2: Local dealer relations
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.

**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. 2.

Obviously, we can introduce a global type `bike_type` for both relations 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. 3.

<table>
<thead>
<tr>
<th>pid</th>
<th>vendor</th>
<th>orderNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0431-860</td>
<td>Bianchi</td>
<td>B-6081</td>
</tr>
<tr>
<td>0431-871</td>
<td>Bianchi</td>
<td>B-6070</td>
</tr>
<tr>
<td>0431-241</td>
<td>Raleigh</td>
<td>R-4010</td>
</tr>
</tbody>
</table>

Figure 3: Mapping relation `map_orderNo`

With the help of this table and a conversion function for the price attribute which converts euro to dollar, the import relations are defined as follows:

```sql
create table bikes_A of bike_type
  as import from dealerA.bikes;

create table 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)
  };
```

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 because the instance-level conflicts are related to these conflict classes, we discuss them in the following together.

**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 contains 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. However, we have to deal with key equivalence conflicts and attribute value conflicts, too.

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 \( r_1 \Join r_2 \) we can define the following assertions regarding the size \( \text{card}(r) \):

- \( r_1 \equiv r_2 \) (equivalence): \( \text{card}(r_i) = \text{card}(r_1) = \text{card}(r_2) \)
- \( r_1 \subseteq r_2 \) (inclusion): \( \text{card}(r_i) = \text{card}(r_1) \leq \text{card}(r_2) \)
- \( r_1 \cap r_2 \) (overlapping): \( 0 \leq \text{card}(r_i) \leq \min(\text{card}(r_1), \text{card}(r_2)) \)
- \( r_1 \neq r_2 \) (disjointness): \( \text{card}(r_i) = 0 \)

A second indicator is the number of NULL values in the join attributes obtained from the outer join of \( r_1, r_2 \).

After detecting key equivalence conflicts a resolution strategy has to be chosen. In FraQL there are two ways: first, the standard SQL facilities where the join expression can be refined, e.g. joining attributes are added or removed or an user-defined predicate is added. 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 relation as shown above for representation conflicts.

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 both 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:

\[
\sigma_{\text{r}_1.A \neq \text{r}_2.A} (r_1 \Join r_2)
\]

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

For resolving this kind of conflicts several alternatives are possible. The simplest way is to define a projection for the preferred attribute. However, this is a static solution because this applies to all tuples. A more advanced way is provided by the FraQL reconciliation functions extending the standard join operators. Here, the value of the resulting attribute can be computed 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. 4). Therefore, if

<table>
<thead>
<tr>
<th>orderNo</th>
<th>price ($)</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AD-1234</td>
<td>530</td>
<td>Head shock, rear shock</td>
</tr>
<tr>
<td>B-6081</td>
<td>900</td>
<td>Head shock, disc brake</td>
</tr>
<tr>
<td>R-4010</td>
<td>1150</td>
<td>Head shock, aluminum frame</td>
</tr>
</tbody>
</table>

Figure 4: Vendor description relation bike\(_{\text{models}}\)
the entry in the dealer relation refers to the same model year, the more recently price value from the
vendor relation should be used, otherwise the dealer price appears in the integrated result (perhaps it is a
phase-out model). First, a reconciliation function is implemented in Java:

```java
Tuple resolve_prices (Tuple i1, Tuple i2) {
    Tuple res = new Tuple ();
    res.setString ("vendor", i1.getString ("vendor")); // copy vendor
    ... // copy remaining attributes from i1
    // resolve price attributes
    if (i1.getFloat ("year") == i2.getFloat ("year"))
        res.setFloat ("price", i2.getFloat ("price"));
    else
        res.setFloat ("price", i1.getFloat ("price"));
    return res;
}
```

After registering this function in the FRaQL system, it can be used in queries for resolving the conflict:

```sql
select *
from bikes_A b join bike_models bm on b.orderNo = bm.orderNo
reconciled by resolve_prices;
```

Please note that a similar solution can be implemented by using the CASE clause of SQL.

**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 structural equivalent,
which is achieved by resolving structural 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:

- $r_1 \equiv r_2$: $\text{card}(r_1) = \text{card}(r_2)$
- $r_1 \subseteq r_2$: $\text{card}(r_1) = \text{card}(r_2)$
- $r_1 \cap r_2$: $\max(\text{card}(r_1), \text{card}(r_2)) \leq \text{card}(r_1) \leq \text{card}(r_1) + \text{card}(r_2)$
- $r_1 \neq r_2$: $\text{card}(r_1) = \text{card}(r_1) + \text{card}(r_2)$

Again, this kind of conflict is resolvable in two ways: either by transforming the keys of one relation
with the help of conversion functions or mapping tables or by refining the union condition. For this
purpose FRaQL uses the modified union operator, which enables the specification of attributes relevant
for deciding equivalences (cf. section 3).

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

```sql
create table bikes of bike_type as
    bikes_A union bikes_B on vendor, orderNo;
```
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. These conflicts are resolved by specifying a user-defined reconciliation function for the union operation. This function is called for each pair of tuples fulfilling the comparison conditions. The affected tuples are passed to the function and the resulting tuple is inserted into the global relation.

Back to our integrated bike relation, we could sum up the stock, if both dealers offer the same model. This is realized by defining a reconciliation function implemented in Java.

```java
Tuple sum_stock (Tuple i1, Tuple i2) {
    Tuple res = new Tuple ();
    res.setString ("vendor", i1.getString ("vendor"); // copy vendor
    ... // copy remaining attributes
    // sum stock from "both" tuples
    res.setFloat ("stock", i1.getFloat ("stock") + i2.getFloat ("stock");
    return res;
}
```

This function is used in the integrating query for conflict resolution:

```sql
create table bikes of bike_type as
    bikes_A a union bikes_B b on vendor, orderNo
    reconciled by sum_stock;
```

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 possible for obvious cases only, 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` and `union` operation).

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 instance-level integration. Based on these facilities a tool for example-driven integration was realized, which we discuss in the next section.

### 6 Example-driven Data Integration: A Prototype Implementation

The interactive and iterative nature of the data integration process requires tool support enabling the definition and evaluation of integration operations as well as direct monitoring of the integration results. Based on the techniques described in the previous sections we have developed a prototype of such a tool. The VIbE system combines features known from Query-by-Example (QbE) with facilities for data definition, conflict detection and conflict resolution. It uses the query language FrAQL for accessing different data sources in a homogeneous way, for defining and retrieving schema elements as well as for performing queries. Integration with VIbE works as follows. As starting point a first coarse application model represented for example as type hierarchy should exists. This could result from a schema integration process supported through tools like SIGMABench [SST+99]. Next, the database integrator selects
the required sources, browses the available relations and imports the appropriated relations by defining FRAQL import relations. For this purpose an existing object type can be selected or a new one has to be defined. In addition, an initial mapping between the structure of the imported relation and the defined type is specified. The VIBE system supports this step by providing default mappings. After the definition of the import relation a table skeleton is displayed and the integrator can perform QbE-like queries in order to check the data.

Based on the import relations the integration relations can be defined in a graphical editor, where relation nodes are connected with operator nodes. Here, a stepwise approach with intermediated conflict detection and resolution according to the integration method discussed in section 4 is recommended. The integrator has to choose the source relations, the appropriated integration operation and optionally a reconciliation function. The resulting integration relation can be queried directly – either by a user-defined QbE query or via predefined conflict detection queries. A first impression about existing conflicts is given by the “conflict map”, a special view which visualizes data discrepancies in a colored map (Fig. 5). This view is computed by an outer join on the key attributes. Therefore, key equivalence conflicts and representation conflicts are highlighted with different colors in the integrated relation. The integrator is able to identify these parts and to zoom in for further examination.

![Figure 5: Conflict map](image)

Based on the evaluation of the query result, i.e. if instance-level conflicts were detected, the definition of the integration relation can be refined by adding or modifying a reconciliation function. These functions are defined in the VIBE system as well. The current prototype implementation requires coding by hand, but we intent to add a more sophisticated solution which allows to derive simple functions from giving example data.

Currently, VIBE provides only some simple detection techniques as described in the previous section. However, in conjunction with FRAQL it forms a framework for more advanced and domain-specific detection approaches, which are examined now and will be added in the future.

The result of the integration process provided by VIBE is the integrated schema for FRAQL. This schema definition contains all required mapping information for schema translation and conflict resolution which are evaluated by the FRAQL query processor. At this stage, an application can query the integrated and (hopefully) conflict-free data.
7 Related Work

The problem of schema integration is addressed by several approaches [BLN86, PBE95]. For describing conflicts arising in the integration phase various classifications were developed, e.g. in [KS91, SCG93, SPD92].

Structural conflicts and resolution strategies are discussed in detail in [KCGS95]. Techniques for managing schematic heterogeneity (meta conflicts) based on SchemaSQL features are presented in [Mil98]. Resolving description conflicts by using a rule-based data conversion language is described in [CDSS98], [MZ98] presents a schema-based data translation solution. In [Ken91] solving domain and schema mismatch problems with an object-oriented database language is discussed.

For instance integration problems several solutions have been proposed. [LP93] examines the entity identification problem, formulates it as a matching problem and defines important properties. An approach for resolving attribute value conflicts based on Dempster-Shafer theory, which assigns probabilities to attribute values is described in [LSS94]. [LC98] introduces a object-oriented data model where global attributes consist 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, which determines tolerable differences, and a resolution function for an automatic conflict resolution can be defined. In [LCC99] an approach is 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 [SSR94], 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.

Query languages supporting the integration of heterogeneous sources are particularly multidatabase languages like MSQL [GLRS93], SQL/M [KGK95] and SchemaSQL [LSS96]. 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, which support the specification of relations with data dependent output schemata. Our language FRAQ is extends these by additional resolution techniques for description and structural conflicts.

8 Conclusions

Instance integration is an important aspect of integrating heterogeneous data sources. Here, mapping between objects from different sources has to be defined and discrepancies and differences in data representation have to be eliminated. Because this has to be addressed during the definition of global views a tight interaction between schema integration and instance integration tasks is necessary.

In this paper we have presented first results from our work on an example-driven integration approach. The main issue is the combination of a query language providing advanced conflict resolution mechanisms and an interactive query and definition tool with extensible support for conflict detection. The approach is currently evaluated in a digital library scenario, where bibliographical data from heterogeneous libraries have to be integrated [SES00].

As a further improvement we aim at incorporating the following functionality into the FRAQ system. If we expect or know that there will be a conflict for which we need a reconciliation function, we might give the system a few examples for conflicting values such that the automated conflict resolution can analyze these conflicting values and, if possible, propose a reconciliation function for resolving this
conflict. In this way we do not always need to program an own reconciliation function for each possible conflict. Instead we could ‘ask’ the FRAQL system whether there is already a solution.

References


