DSToolkit: An architecture for flexible Dataspace Management*

Cornelia Hedeler[1], Khalid Belhajjame[1], Lu Mao[1], Chenjuan Guo[1], Ian Arundale[1], Bernadette Farias Lóscio[2], Norman W. Paton[1], Alvaro A.A. Fernandes[1], and Suzanne M. Embury[1]

[1] School of Computer Science, The University of Manchester
Oxford Road, Manchester M13 9PL, UK
chedeler, khalidb, maol, guoc, arundai7, norm, alvaro, embury@cs.manchester.ac.uk

[2] Universidade Federal de Pernambuco, Centro de Informtica
Cidade Universitria 50740-540 - Recife, PE - Brasil
bfl@cin.ufpe.br

Abstract. The vision of dataspaces is to provide various of the benefits of classical data integration, but with reduced up-front costs. Combining this with opportunities for incremental refinement enables a 'pay-as-you-go' approach to data integration, resulting in simplified integrated access to distributed data. It has been speculated that model management could provide the basis for Dataspace Management, however, this has not been investigated until now.
Here, we present DSToolkit, the first dataspace management system that is based on model management, and therefore, benefits from the flexibility provided by the approach for the management of schemas represented in heterogeneous models, supports the complete dataspace lifecycle, which includes automatic initialisation, maintenance and improvement of a dataspace, and allows the user to provide feedback by annotating result tuples returned as a result of queries the user has posed. The user feedback gathered is utilised for improvement by annotating, selecting and refining mappings. Without the need for additional feedback on a new data source, these techniques can also be applied to determine its perceived quality with respect to already gathered feedback and to identify the best mappings over all sources including the new one.

Keywords: Dataspace Management System, Dataspace lifecycle, Incremental improvement

1 Introduction

1.1 Motivation

Data integration in various forms has been the focus of ongoing research in the database community for over 20 years. The objective is to provide an integrated

* The work reported in this paper was supported by a grant from the EPSRC.

view and access to multiple heterogeneous data sources. Typically this involves developing a single global integration schema to which the schemas of the sources are related by some form of mapping. Using those mappings, queries posed over the central schema are then unfolded [24], optimised by a distributed query processor and evaluated over the sources. Various systems have been developed that are based on the approach of classical data integration, e.g., DB2 [23] or OpenII [45]. However, defining and maintaining mappings between a global integration schema and various source schemas has proven to be labour intensive [13], in particular in a world of ever more data sources that evolve over time to account for the changes in the data to be stored. This means that classical data integration is most effective when integrating small numbers of stable data sources, but less so for the integration of large numbers of evolving data sources, or for on-the-fly data integration.

The vision of *dataspaces* [19, 25] aims to realise various of the benefits of classical data integration but with much lower startup costs thereby supporting integration on demand but with lower quality of the resulting integration. The quality of the integration can then be improved over time in a 'pay-as-you-go' manner utilising feedback provided by the user or developer. Various dataspace management systems have been proposed in recent years, e.g., SEMEX [18], iMeMex [15], PayGo [35], and Q [48].

Bernstein *et al.* [8] speculated that model management could form the basis of dataspace management, but so far this has not been investigated as none of the dataspace management systems proposed so far are based on model management, and nor have any of the existing model management systems (e.g., MISM [1], Rondo [39], GeRoMe [31] or Automed [46]) been extended to support the incremental improvement that is integral to dataspace management systems.

Dataspace systems can vary in numerous dimensions [27, 26], but as illustrated in the surveys [27, 26] the dataspace systems proposed so far tend to be somewhat narrowly focussed, targeting specific applications and making assumptions that do not hold elsewhere, e.g., SEMEX [18], a personal information management system, requires domain knowledge to identify associations between schemas which may not always be available, iMeMex, also managing personal information, requires path-based queries called *trails* [15] that are provided manually and require the user to have a good understanding of the schemas and the relationships between them, a requirement that shuts out the casual user, and PayGo [35], integrating web sources, only supports a union schema as an integration schema.

In this paper we present DSToolkit, an architecture for flexible dataspace management that is based on model management, thereby benefiting from the flexibility of being able to manage multiple heterogeneous models, that supports the whole lifecycle of a dataspace [26] including automatic support for initialisation, also called bootstrapping, but also provides support for maintenance in the context of source changes and that provides a means for incremental improvement of the dataspace by the casual user.

1.2 Overview of the Approach

DSToolkit builds on the foundations provided by model management systems, in particular MISM [1] in the sense that we use a model-independent supermodel to capture schemas represented by heterogeneous models. Furthermore, *DSToolkit* contains implementations of various model management operators [7, 6, 8] to support the management of multiple schemas. Model management operators implemented include `match`, `merge`, `difference` and `compose`. Utilising the results of model management research enables us to provide a system that is flexible in its management of multiple heterogeneous schemas and the associations between constructs in those schemas and to support the maintenance of the dataspace. For example, constructs can be attributes or tables in relational schemas, or simple or complex elements in XML schemas.

However, various changes and extensions are necessary to turn a model management system into a dataspace management system:

1. We have generalised the supermodel even further to emphasise the commonalities in terms of the role the constructs play in the various models (e.g., whether they represent relationships between other constructs, can have values, or contain other constructs), rather than their differences. The more specific information on their differences is required for ModelGen [1], but only to a certain extent needed to be able to express relationships between constructs or to query the underlying data sources.
2. We have introduced various kinds of *morphisms*, which are not as integral to MISM as they are to other model management platforms, such as Rondo [39]. The morphisms we introduce represent associations between elements in different schemas at different levels of abstraction, and are *matches*, *schematic correspondences* and *mappings*. *Matches* give an indication that two constructs are similar to a certain extend and tend to be the relationships returned by matching algorithms. *Schematic correspondences* [36] are based on schematic heterogeneities introduced in [33, 32] and represent richer semantic relationships between schema elements, such as same or different names for the same construct, missing constructs (e.g., attributes), or horizontal- or vertical partitioning. *Mappings* are executable expressions that specify how the data that conforms to one schema needs to be restructured to conform to another schema.
3. We have altered various model management operators in that the majority of them are defined to operate on *schematic correspondences* rather than *matches* as they are semantically richer than *matches*, and we have introduced additional operators for the automatic inference of *schematic correspondences* (`inferCorrespondence`) in addition to the operators `match` and `merge` for full support of the automatic initialisation or bootstrapping of a dataspace starting with the identification of *matches* followed by the inference of *schematic correspondences* and the automatic generation of *mappings* [36]. To support the usage of the dataspace the operator `evaluateQuery` has been added, which enables queries that are posed over any schema repre-

sented in the supermodel to be evaluated over multiple data sources using query unfolding [24].

4. In order to enable the incremental improvement of a dataspace, we have added the functionality to gather user feedback on query results, and to utilise the feedback provided to annotate the mappings with their precision and recall with respect to the feedback, to select the best mappings for future query executions and to refine the mappings (operators `annotateMappings`, `selectMappings`, and `refineMappings`) [4].

By building on top of the results of model management research and extending the model management operators with those listed above, we have created a toolkit that provides support for flexible management of schemas represented using heterogeneous models, enables the reaction to changes in the schemas of the underlying sources or the addition of new sources, provides flexibility with respect to the data models queries can be posed over and the ability to handle multiple integration schemas rather than just a single one. Furthermore, *DSToolkit* can support the whole dataspace lifecycle including initialisation, maintenance and improvement and it enables the casual user to provide feedback.

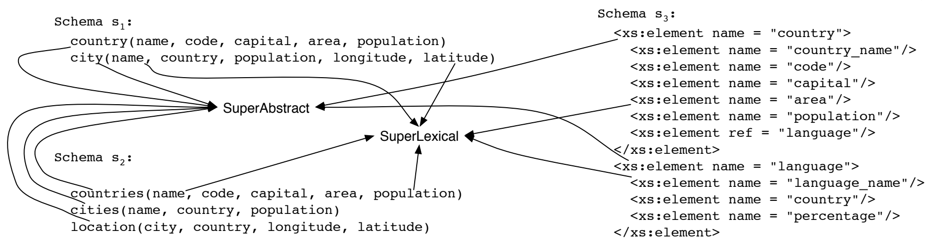
1.3 Contributions of the Paper

We present here *DSToolkit*, an architecture for flexible dataspace management that builds on the foundations provided by model management. The paper describes how the toolkit can be used to create dataspaces that exhibit various properties along the dimensions of dataspaces [26] rather than being limited to a specific point solution. The contributions of this paper are:

– Introduction of the *DSToolkit* and its comparison with prominent model management, data integration and dataspace management systems.
– Demonstration of its flexibility and ability to support the creation of dataspaces with different properties according to the dimensions of dataspaces using various examples.
– Illustration of the support that *DSToolkit* provides for the maintenance of a dataspace.
– Demonstration of the benefits that can be derived from user feedback, not just with respect to data sources already added to the dataspace, but also with respect to new data sources.

1.4 Organisation of the Paper

The paper is organised as follows: Section 2 presents a motivating example and gives a brief overview of the main functionality, capabilities and usage of *DSToolkit*. Section 3 places *DSToolkit* in the context of other prominent data integration and dataspace management systems, and discusses their commonalities and differences. Section 4 provides an overview of the architecture of the



```
country(name, code, capital, area, population)
city(name, country, population, longitude, latitude)
```

```
countries(name, code, capital, area, population)
cities(name, country, population)
location(city, country, longitude, latitude)
```

```
<xs:element name = "country">
  <xs:element name = "country_name"/>
  <xs:element name = "code"/>
  <xs:element name = "capital"/>
  <xs:element name = "area"/>
  <xs:element name = "population"/>
  <xs:element ref = "language"/>
</xs:element>
<xs:element name = "language">
  <xs:element name = "language_name"/>
  <xs:element name = "country"/>
  <xs:element name = "percentage"/>
</xs:element>
```

Fig. 1. Example source schemas.

toolkit, followed by Section 5 which describes how *DSToolkit* supports the initialisation and shaping of various kinds of dataspaces and how it can be queried. Section 6 explains how the toolkit can be utilised to improve the dataspace and to evaluate complementary data sources with respect to already provided user feedback. Section 7 concludes the paper.

2 Motivating Example

This section introduces a motivating example and uses it to illustrate key features of *DSToolkit*, such as the flexibility provided by model management operators for managing multiple schemas and the associations between them, and the ability to utilise user feedback gathered on query result tuples to improve mappings over previously integrated sources.

As an example, we assume that the user would like to create a dataspace containing information on countries, their cities, the languages spoken in the countries, and that s/he has identified three data sources d_1,...,d_3 with the schemas s_1,...,s_3 shown in Figure 1. We further assume that d_1 contains information on european countries, d_2 information on african countries, and d_3 contains information on the languages spoken in all countries.

DSToolkit provides the user with a number of options to create a dataspace that meets the requirements, some of which are introduced in the following. After the import of the data sources into the system, *DSToolkit* offers multiple options for choosing or generating the preferred view of the data:

– Provide a manually defined global schema;
– Use the schema of any of the imported data sources;
– Use model management operators to generate a schema that meets the user's requirements.

As none of the three data sources in the example contains information on countries, their cities and their spoken languages, neither of the schemas of the imported sources can be utilised as global schema, leaving the other two options. One option would be to specify a schema manually that is to be used as a global schema and import it. However, as *DSToolkit* is based on model management, there is also the option to use the flexibility provided by the model management operators to generate a schema that conforms to the desired view of the data.

This can be achieved as follows: (i) Matching schemas pairwise with each other to identify the similarities, called *matches*, between them. Examples of

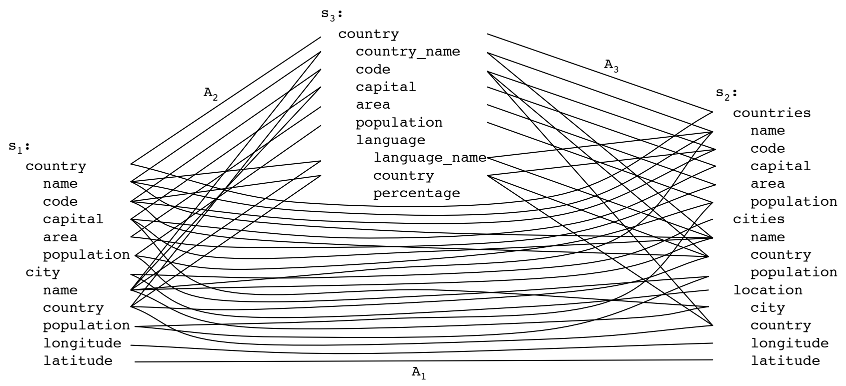


Fig. 2. Matches between source schemas.

matches between constructs in schemas s_1, s_2 and s_3 are shown in Figure 2. Some of the matches identified may be coincidental, e.g., the match between s_1.country. name and s_2.cities.name, which has been identified due to the same name of the two attributes. (ii) Using the information provided by the matches, semantically richer sets of *schematic correspondences* are inferred. Examples of schematic correspondences include `Same name, same construct (SNSC)`, which are represented in Figure 3 by lines between the corresponding constructs. Constructs can be attributes or tables in relational schemas or simple or complex elements in XML schemas. Other examples of schematic correspondences are `Different name same construct (DNSC)` (e.g., the attributes s_1.city.name and s_2.location.city represent the same constructs, i.e., the names of cities, but have different names), `Vertical partitioning (VP)` (e.g., the two tables `cities` and `location` in s_2 are a vertical partitioning of the table `city` in s_1), `Horizontal Partitioning (HP)`, and missing constructs, e.g., an attribute that is present in a table in one schema but not in the table that has been identified as corresponding in another schema. (iii) After the correspondences between schemas have been identified, they can be used to generate a merged integration schema.

The schema s_{m_2}, which is the schema generated by merging s_1, s_2 and s_3 is shown in Figure 4. The schematic correspondences between s_{m_2} and each of the source schemas are similar to those shown in Figure 3.

When the user is happy with the global schema created, the *mappings* between the global schema and each of the source schemas can be generated from the *schematic correspondences* between them [36]. The mappings for our example are shown in Figure 5.

After the mappings have been generated, the user can pose queries over the merged schema s_{m_2}. It is also possible to pose queries over any of the other schemas, e.g., source schemas or previously generated merged schemas, as long as the mappings between the schema and all the source schemas have been generated from schematic correspondences.

For example, let us assume that the user would like to get information on countries that are european and mediterranean, however, neither of the sources s/he has included so far in the dataspace contains information specific to mediterranean countries, though, as mentioned earlier d_1 contains information on euro-

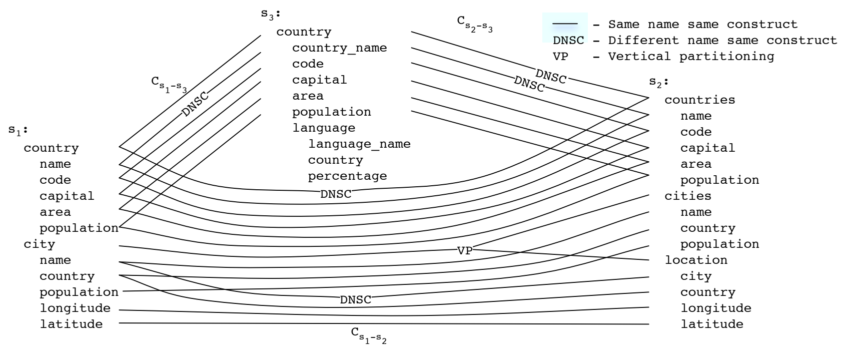


Fig. 3. Motivating example - schematic correspondences between source schemas.

```
country(name, code, capital, area, population)
city(name, country, population, longitude, latitude)
language(country, name, percentage)
```

Fig. 4. Merged schema s_{m_2}

pean countries. Let us also assume that the user would like to run the query q_1 `Select * from country o` posed over s_{m_2}. As all three source schemas s_1, s_2 and s_3 contain information on countries, the mappings map_1, map_3 and map_5 are used to expand the query over the sources using query unfolding [24], which will result in european (from d_1), african (from d_2) and all other countries (from d_3) to be returned. To improve the initial integration, *DSToolkit* allows the user to annotate the query results indicating whether a result tuple was expected, i.e., is a true positive, or unexpected, i.e., is a false positive. We also allow the user to provide tuples that s/he expected to be returned as result but were not returned, i.e., false negatives. In our example, the user has annotated a small number of result tuples as shown in Table 1. The table also shows which mappings produced which of the annotated result tuples. Using the feedback, the mappings that produced the results are annotated with their precision and recall with respect to the user feedback provided.

Given existing feedback, the user may pose restriction on the mappings to be used when evaluating the query, e.g., to ensure that the proportion of the results that are true positives is high compared with those that are false positives. To do so, the user specifies a query together with requirements in terms of precision and recall that should be met by the query results returned. To achieve this, $M' \subseteq M$ are selected that are to be used to expand the query [4].

map_1 = <s_{m_2}.country, select o.name, o.code, o.capital, o.area, o.population from s_1.country o>
map_2 = <s_{m_2}.city, select c.name, c.country, c.population, c.longitude, c.latitude from s_1.city c>
map_3 = <s_{m_2}.country, select o.name, o.code, o.capital, o.area, o.population from s_2.countries o>
map_4 = <s_{m_2}.city, select c.name, c.country, c.population, l.longitude, l.latitude
from s_2.cities c, s_2.location l
where c.name = l.city and c.country = l.country>
map_5 = <s_{m_2}.country, select o.country_name as name, o.code, o.capital, o.area, o.population
from s_3.country o>
map_6 = <s_{m_2}.language, select l.country, l.language_name as name, l.percentage from s_3.country.language l>

Fig. 5. Mappings between s_{m_2} and s_1, s_2, s_3

Table 1. Annotated result tuples of q_1.

name	code	capital	area	population	expected	not expected	mappings
France	F	Paris	547030	58317450	$\surd$		map_1, map_5
Turkey	TR	Ankara	780580	62484478	$\surd$		map_1, map_5
Italy	I	Rome	301230	57460274	$\surd$		map_1, map_5
Tunisia	TN	Tunis	163610	9019687		$\surd$	map_3, map_5
Morocco	MA	Rabat	446550	29779156		$\surd$	map_3, map_5
Algeria	DZ	Algier	2.38174e+06	29183032		$\surd$	map_3, map_5

3 Related Work

This section discusses related work and compares it with *DSToolkit*. The work presented comes from several related areas, namely, traditional data integration, model management, dataspace management, evaluating the data quality of complementary data sources, and utilising feedback to support data integration.

3.1 On Traditional Data Integration

As data integration has been a research focus of the database community for several decades now, many contributions have been made. Here, we focus on two prominent representative examples, namely the IBM Information Server (IIS)[3], a commercial information integration platform [22], and Open II[4], an open source information integration suite [45].

The IBM Information Server provides a suite of tools that provide support for the various stages of information integration for both data materialisation or federation approaches [22]. However, even though a large number of tools are provided, they still require significant manual effort at various stages of the integration process, e.g., up-front to understand the data to be integrated, define an integration schema and data quality rules, or during the integration process to define or adjust mappings or to specify how to reconcile duplicates [22].

OpenII [45] provides an extensible platform for information integration consisting of a repository which uses a model-generic metamodel to represent schemas and mappings and a number of importers/exporters. *DSToolkit* also uses a model-generic metamodel to capture information on heterogeneous schemas and morphisms (matches, schematic correspondences, mappings). Open II provides a number of tools to aid several information integration tasks, e.g., *Harmony* for schema matching, visualising and debugging of matches identified by multiple linguistic matchers, *Unity* to support the semi-automatic generation of mediated schemas, *RMap* and *XMap*, to generate the code needed for data exchange from matches identified by *Harmony* and confirmed or adjusted by the user.

As both *IIS* and *OpenII* provide support for traditional data integration they require a significant manual effort during the integration process and a good understanding of schemas and associations between them, thereby, making it almost impossible for the casual user to utilise either of them. In contrast, *DSToolkit* provides support for a fully automatic bootstrapping of the dataspace, i.e., the integration of various data sources, and only requires user interaction to provide

[3] www.ibm.com/software/data/infosphere/

[4] http://openii.sourceforge.net/

feedback on result tuples, which can be provided by a casual user. Furthermore, neither IIS or OpenII are based on model management, even though *OpenII* makes use of a model-generic metamodel to represent schemas and matches, and therefore, are unable to benefit from the flexibility model management operators provide for handling multiple heterogeneous schemas and associations between their elements.

3.2 On Model Management

Model management [7, 8] has been the focus of ongoing research for a number of years now and several systems have been proposed, e.g., Rondo [39], MISM [1], GeRoMe [31] and Automed [46]. All these systems use model-generic metamodels to abstract over the specifics of particular models and all provide importers at least for relational and XML schemas. The systems also provide a representation of morphisms between elements of the various schemas. The majority of the systems provide implementations for the model management operators `match`, which infers the morphisms between elements in different schemas, `merge`, which merges two schemas using the information provided in the morphisms between their elements, `modelGen`, which transforms a schema represented in one model into an equivalent schema represented in a different model, and some provide implementations of `compose`, which composes morphisms between schemas a, b and schemas b, c into morphisms between schemas a, c, and `difference` which returns the portion of a schema that does not participate in the morphisms between the schema and another schema.

Even though some of the systems have been around for a number of years and the purpose of being able to integrate multiple schemas is eventually to be able to query across their corresponding sources, only some of the model management platforms, namely GeRoMe [31] and Automed [37], have been extended for query answering. With the provision of model management operators, the majority of the platforms provide sufficient support for the maintenance in case of evolving schemas or additional sources to be integrated, however, even though it was speculated that model management platforms could provide the basis for dataspace management [8], the existing platforms tend not to have an analogue to `inferCorrespondence`, and have not been extended into a dataspace management platform that provides support for the 'pay-as-you-go' improvement that is characteristic of dataspace management.

3.3 On Dataspace Management

As dataspaces represent a fairly recent addition to the data integration landscape, the proposals have yet to reflect a shared understanding of best practice, and thus are diverse in their contributions across a variety of dimensions [26]. Proposals range from SEMEX [18], and iMeMex [15], both of which integrate personal information, over PayGo [35], which is targeted at the integration of web sources, to Q [48], the query system of ORCHESTRA [29], a collaborative data sharing system. None of the existing proposals for dataspaces are based on model management, making *DSToolkit* the first dataspace management system based on the solid foundations of model management and benefitting from the flexibility in managing diverse schemas provided by the approach. The majority of dataspace proposals tend to be point solutions addressing specific issues, but do not present a flexible approach that can be instantiated differently to create dataspaces with different properties along the dimensions identified in [26].

For example, SEMEX [18] requires an integration schema to be provided manually, whereas PayGo[35] forms a union schema, but neither approach provides support for generating a merged schema, as *DSToolkit* does in addition to accepting a manually specified integration schema or the option of choosing any of the source schemas as preferred view over the integrated data. SEMEX also provides no support for incremental improvement and even though PayGo[35] advocates incremental improvement, no details are provided on how this is achieved. In contrast, iMeMex starts with a union schema of all the integrated schema and provides support for manual improvement in the form of path-based queries called *trails*[15]. In contrast to our approach for incremental improvement which only requires users to indicate which result tuples meet their expectations, the need to provide path-based queries seems likely to exclude the casual user from improving the dataspace.

Similar to *DSToolkit*, UDI [14] can generate a merged schema automatically by matching source schemas and deriving a merged schema, but it makes simplifying assumptions in that the source schemas are limited to relational schemas with a single relation. UDI also provides no support for incremental improvement, which is the focus of Roomba [30]. Users are asked to provide feedback on matchings and mappings that have been determined by the system to provide the most benefit to the integration if annotated. Rather than requiring users to annotate matchings and mappings, Q [48], the query system of ORCHESTRA [29], asks users to provide feedback on the query results, similar to the feedback gathered by *DSToolkit*, and their rankings. This information is then propagated to the rankings of the matchings and mappings that produced the results.

3.4 On Evaluating the Data Quality of Complementary Data Sources

Data quality can be seen as how well the data meets the user's requirements, i.e., can be characterised as its "fitness for use"[49]. There are multiple measurable quality dimensions, e.g., accuracy, completeness, or currency [49].

A number of approaches have been proposed for quality-driven information integration (e.g., [42, 44]). In [42] the authors propose an approach for quality-aware query plan creation and selection of the plan with the best weighted aggregate quality score according to several specific quality criteria.

In the Data Quality Broker, which is part of the DaQuinCIS architecture [44], queries posed over a global schema are unfolded into queries over multiple sources, an approach also followed in *DSToolkit*. The values of results returned by each source are annotated by the source with estimates of their data quality, e.g., their accuracy or completeness. The data quality dimensions are defined such that they can be measured, e.g., the accuracy of a value is measured in terms of its edit distance to values in reference dictionaries, and the completeness of a result tuple is measured in terms of the number of its attributes that are not `null`. This information is used to reconcile result tuples that refer to the same entities, but also to propagate the best information as determined by the quality values attached back to the sources which have returned data of lower quality to improve the overall data quality in the cooperative information system.

In contrast to both approaches, where information on the quality of the data is provided by the source providers themselves, in *DSToolkit* the quality of the information provided by a source is inferred from the user feedback provided on results of queries evaluated over those sources. The annotation of mappings with quality criteria in [42] could be compared to the annotation of mappings with their precision and recall with respect to the user feedback provided on the query results in *DSToolkit*, with the difference of the origin of the quality annotation. We could see precision as an indication for the accuracy and recall as an indication of the completeness of the results returned by a given mapping.

3.5 On Using Feedback to Support Data Integration

User feedback is a growing theme in data integration literature [5]. It is seen by many researchers as the key ingredient to face the difficulties that lie in the specification of data integration components. For example, Chiticariu *et al.*[12] proposed a method for generating integration schema. To ensure the suitability of the schema generated to user requirements, feedback is solicited from end-users. McCann *et al.* [38] developed a community-based system that solicits feedback from end users with the view to informing the schema matching operation. In doing so, the feedback is used to assess the matches between attributes in two schemas. Feedback has also been proposed as a means for driving the specification of schema mappings. For example Jeffery, *et al.* [30] developed a decision-theoretic framework for specifying the order in which candidate mappings can be confirmed by soliciting feedback from users with the objective of providing the *most benefit* to a dataspace. To do so, they developed a utility function that estimates the benefit that can be drawn from knowing whether a given schema mapping is correct or not.

User feedback has also been used as a means for authoring integration queries, i.e., queries that involve multiple data sources. The *Q* system supports such a functionality [48]. Specifically, given a set of keywords specified by the user, the system suggests a list of candidate queries that may or may not meet user expectations. The user comments on the results returned by those queries. Based on these comments, the system ranks the list of candidate queries, the first query being the one that seems to meet user expectations the most.

While the above proposals show the key role user feedback can play to support data integration, they are confined to the use of feedback to support a single functionality. Differently, in our work, we try to make the most of the feedback supplied by end users, and use them to support three functionalities that are key to dataspaces improvement, viz., mapping annotation, selection and refinement.

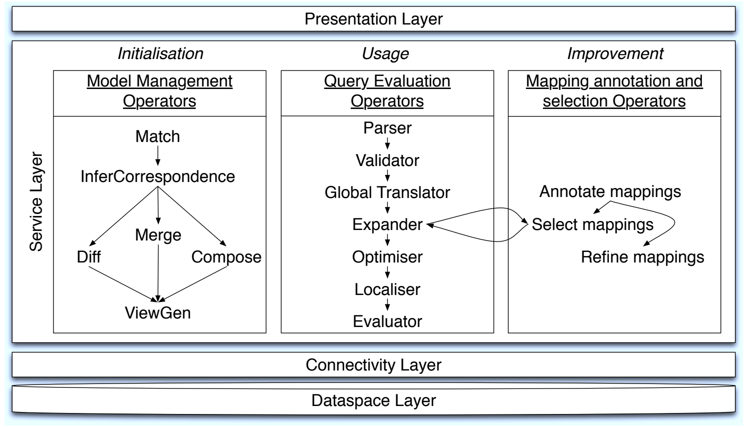


Fig. 6. Layered architecture of *DSToolkit*.

4 The DSToolkit Architecture

This section provides an overview of the architecture of *DSToolkit*. It is implemented following the layered architecture shown in Figure 6. The arrows between the operators indicate in which order they tend to be utilised in each phase of the life cycle of a dataspace, namely, initialisation, usage and improvement. The arrows also indicate the dependency of the operators on the output of a previous operator producing the input of another operator, e.g., during initialisation the operator `inferCorrespondence` uses the output of `match`, i.e., the matches produced, as input to infer the schematic correspondences which, in turn are used by `merge` as input along with the schemas to be merged.

DSToolkit consists of a *dataspace* layer that persistently stores the model-generic representation of schemas, the morphisms between them, i.e., the matches, schematic correspondences supported by the matches, and the mappings derived from the latter, queries posed, the corresponding results as well as user feedback gathered on the results. The UML diagram of the model-generic representation of schemas is shown in Figure 7 and introduced in Section 5.1. The UML diagram of the morphism model is shown in Figure 8 and discussed in Section 5.2.

The *connectivity* layer provides the means for storing the information represented in the *dataspace* layer persistently and for accessing that information. As the functionality provided by this layer is fairly straightforward, it is omitted from further discussion. The *connectivity* layer is used in turn by the *service* layer which contains the actual functionality of *DSToolkit*, namely the model management operators, the query processor, and the techniques for mapping annotation, selection and refinement based on user feedback provided on result tuples. The functionality provided for use during the initialisation and usage phase is discussed in more detail in Section 5, whereas the functionality provided for use during the improvement phase is introduced in Section 6. The *presentation* layer exposes a web-based user interface through which the user can access the functionality provided by *DSToolkit*, i.e., the operators provided by the *service* layer and introduced in the following.

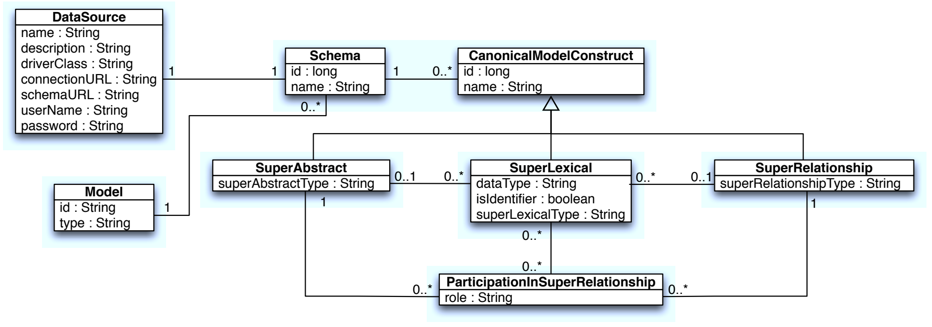


Fig. 7. UML Diagram of the canonical model.

5 Initialising, Shaping and Querying a Dataspace with *DSToolkit*

This section describes how a dataspace is set-up. More specifically, it describes the initialisation of a dataspace, how data sources can be added, integration schemas inferred that match the user's view of the data and how queries can be posed and are evaluated. The section introduces the models required to capture all the necessary information, and the operators, some of which are model management operators, some of which we have added for the dataspace management. The operators provided by *DSToolkit* (except `addDataSource` and `evaluateQuery`) with their signatures and descriptions are listed in Table 3.

5.1 Model-generic Canonical Model

This section presents the model-generic metamodel, called the *canonical model*, which is a generalisation of the MISM supermodel [1, 2] and is shown in Figure 7. MISM consists of two levels of schema descriptions, namely, the model-specific description and the source-model independent supermodel. The model-specific descriptions contain all the constructs that are required to represent schemas in a particular model, e.g., table, column and foreign key for relational models and root element, complex element, simple element and foreign key for XSD. The supermodel contains a small set of model-generic constructs that represent the model-specific constructs by aggregating over their similarities, e.g., whether they can have values as is the case for relational and object-relational columns, simple elements in XSD and object fields, all of which correspond to lexical in the MISM model, or whether a construct represents referential integrity constraints between other constructs, such as the foreign key in XSD, relational and object-relational, all of which correspond to foreign key in MISM.

We have generalised the constructs of the MISM supermodel further according to whether the constructs can have values, represented by *SuperLexical*, e.g., column of a relational table, or a simple element in XSD, whether they are collection objects, represented by *SuperAbstract*, e.g., a relational table, or a complex element in XSD, or whether they represent a relationship between constructs, represented by *SuperRelationship*, e.g., a foreign key relationship in

Table 2. Canonical model constructs, model-generic and model-specific constructs.

		Model-specific constructs per model			
Canonical model constructs	**MISM model-generic constructs**	**Relational**	**XSD**	**Object**	**Object Relational**
Super-Abstract	*Abstract*		Root element	Class	Typed table
	Aggregation	Table			Table
	StructOfAttributes		Complex element		Structured column
Super-Lexical	*Lexical*	Column	Simple element	Field	Column
Super-Relationship	*Foreign key*	Foreign key	Foreign key		Foreign key
	Abstract attribute			Reference field	Reference
	Generalisation			Generalisation	Generalisation

relational model or XSD, or the nesting of elements in XSD. Table 2 lists the constructs of our canonical model, the corresponding model-generic constructs of the MISM model and the corresponding model-specific constructs for relational, XSD, object and object-relational models. In addition to the example schemas shown in Figure 1, the figure also shows the corresponding construct in the canonical model, i.e., SuperAbstract for the relational tables in s_1 and s_2, as well as the root element (country) and the complex elements (language) in s_3. The relational columns in s_1 and s_2 and the simple elements in s_3 all correspond to SuperLexical, but for readability not all those correspondences are shown in Figure 1. Even though *DSToolkit* currently provides no importers for object and object-relational schemas (which can easily be added), as the *canonical model* is a generalisation of the MISM metamodel, providing support for object and object-relational schemas, our model supports both too. The exact type of each canonical model construct, e.g., whether a SuperAbstract represents a relational table or an XSD complex element or whether a SuperRelationship represents a relational or an XSD foreign key, is captured by the corresponding type (*superAbstractType*, *superLexicalType*, *superRelationshipType* in Fig. 7) as this information is later required for query rewriting (see Section 5.3). SuperAbstracts participating in a SuperRelationship can play different roles, e.g., the nested child element or its parent element, the referenced element in a foreign key relationship or the referencing element. Some relationships are also further specified by SuperLexicals, e.g., the attributes that form the (composite) foreign key. This information is captured in *ParticipationInSuperRelationship*.

When a *data source* d_i with its *schema* s_i is added using the operator (d_i,s_i) $\leftarrow$`addDataSource`($dataSourceName$, *description*, $driverClass$, *url*, $userName$, *password*) for data sources where the schema information can be obtained from the same url as the data or $(d_i,s_i)\leftarrow$`addDataSource`($dataSourceName$, *description*,$driverClass$,*url*,*schemaUrl*,$userName$,*password*) for data sources where the schema information can be found in a different location to the data source itself, the schema information is imported according to the correspondence between the model-specific constructs and the canonical model constructs. All the remaining operators operate over the representation of schemas in the canonical model, thereby abstracting over the differences of heterogeneous models.

Table 3. Operators provided by *DSToolkit*

Operation	Signature	Description
`match`	$A \leftarrow$ `match`$(s_i, s_j, [d_i, d_j])$	Match schemas s_i and s_j with each other using schema-based and if instances from data sources d_i and d_j with schemas s_i and s_j are provided, instance-based matchers to obtain set A of matches between them.
`infer Correspondences`	$C \leftarrow$ `inferCorrespondences`(A)	Generate set of schematic correspondences C from set of matches A.
`compose`	$C \leftarrow$ `compose`(C_1, C_2)	Compose sets of schematic correspondences C_1, C_2 that hold between s_i, s_j and s_j, s_k into correspondences C that hold between s_i, s_k.
`merge`	$(s_m, C_1, C_2) \leftarrow$ `merge`(s_i, s_j, C)	Merge two schemas s_i and s_j using the schematic correspondences C that hold between them and return merged schema s_m with correspondences C_1 and C_2 that hold between s_m, s_i and s_m, s_j.
`difference`	$C' \leftarrow$ `difference`(s_i, s_j, C)	Return schematic correspondences C' that represent differences between schemas s_i and s_j.
`viewGen`	$M \leftarrow$ `viewGen`(C)	Derive set of mappings M from set of schematic correspondences C.
`selectMappings`	$M' \leftarrow$ `selectMappings`(M, precisionTarget, recallTarget)	Given a set of mappings M and a precision- or recall target λ select mappings $M' \subseteq M$ to be used to answer a query such that union of results returned by selected mappings M' achieve λ.
`refineMappings`	$M' \leftarrow$ `refineMappings`(M)	Given set of mappings M produce mappings M' that meet user's requirements with respect to feedback provided on results better than M.

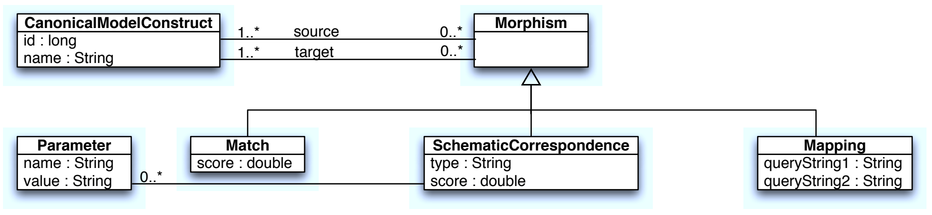


Fig. 8. UML Diagram of the morphism model.

5.2 The Bootstrapping Process and the Morphisms Generated

Here we present the bootstrapping process with the operators to generate the various morphisms expressing relationships between heterogeneous schemas and the models to capture them. The steps required to determine the morphisms between schemas, merge the schemas and generate mappings that specify how the data needs to be transformed are explained in more detail in the following.

We also introduce multiple specialised kinds of morphisms, which were presented in Rondo [39]. A Morphism in general represents a binary relationship between two sets of instances of constructs. We distinguish between three kinds of morphisms of varying semantic richness, which we introduce in more detail in the following. A UML diagram of the different kinds of morphisms with their specific properties, their generalisation and their associations with *CanonicalModelConstructs* is shown in Figure 8.

Match and Infer Schematic Correspondences between Schemas. To be able to evaluate queries across multiple data sources, the relationships between the constructs in their schemas need to be identified and expressed in a way that can be utilised for unfolding queries posed over a schema of choice. To identify those relationships, firstly, the schemas need to be matched. A *match a* is a bidirectional morphism between sets of constructs which indicates that the constructs are similar to a certain extent; the confidence in which is indicated by the (similarity) *score* property of the *match a*, whereby a higher score indicates more confidence. A large corpus of literature is available on various matching approaches and algorithms (e.g., see [43, 17]). To identify the set A of matches between two schemas s_i and s_j the operator $A \leftarrow \texttt{match}(s_i, s_j, [d_i, d_j])$ can be used, which uses existing schema-, and if optional instance data is available from data sources d_i and d_j, instance-based matchers [16, 41, 3]. Examples of matchers include string based matchers using, e.g., edit distance or n-grams to determine how similar two strings are, data type matchers comparing the data types of constructs, or structure-based matchers comparing the structure of constructs within a schema such as the nesting of elements in XSD. The computational complexity of schema matchers is generally $O(n^2)$ [9].

However, these matches do not provide sufficient information for deriving mappings that express how data is to be transformed. To bridge the semantic gap between semantically poor matches and semantically rich mappings and to enable the automatic generation of mappings from matches, we use *schematic correspondences*, which are based on schematic heterogeneities introduced in [33, 32] and which, as shown in [36], provide enough information to infer the mappings automatically. The operation $C \leftarrow \texttt{inferCorrespondences}(A)$ is used to infer a set of semantically rich schematic correspondences C from the set of matches A. A schematic correspondence has a *score*, which as before for matches indicates the confidence associated with the correspondence. Only a subset of the matches A provided as input may result in schematic correspondences, i.e., only those matches that provide the most support for schematic correspondences of the following types: *different name for same construct (DNSC), same name for same construct (SNSC), missing constructs (MC), horizontal - (HP) and vertical partitioning (VP).* For example, in Figure 2 matches between `city.population` in s_1 and `countries.population` in s_2 and vice versa have been identified. However, as can be seen in Figure 3 these have not resulted in a schematic correspondence between these elements, as other matches have provided more evidence for correspondences between `city` in s_1 and `cities` in s_2 as well as `country` in s_1 and `countries` in s_2 and their corresponding attributes rather than between the `population` attribute in `city` and `country`.

The type of a schematic correspondence is captured in its property *type*. Some correspondences require additional parameters, e.g., for VP the join predicates, which are captured in the *Parameters* that can be associated with a correspondence (Fig. 8). A genetic algorithm is used to search the space of all possible schematic correspondences supported by the input matches and find an optimal solution, the runtime of which can be controlled by various parameters, such as population size and number of generations [40].

Model Management Operations for Manipulating Schemas and Correspondences. Once schematic correspondences between constructs in schemas

```
Schema s5:

continent(name, area)
encompasses(country, continent, percentage)
```

Fig. 9. Schema s_5

have been identified, those and the schemas can be manipulated using model management operators [7, 8]. Model management operators have been shown to be useful for various scenarios that are of importance in data integration, such as schema integration and evolving schemas [6]. In addition to `match` *DSToolkit* provides implementations of $(s_m, C_1, C_2) \leftarrow \texttt{merge}(s_i, s_j, C)$, which merges two schemas s_i and s_j utilising the schematic correspondences C that hold between s_i and s_j and returns the merged schema s_m with sets of schematic correspondences C_1 and C_2 that hold between s_m, s_i and s_m, s_j, respectively, $C \leftarrow \texttt{compose}(C_1, C_2)$ which composes sets of schematic correspondences C_1, C_2 that hold between s_i, s_j and s_j, s_k, respectively, into schematic correspondences C that hold between s_i, s_k and $C' \leftarrow \texttt{difference}(s_i, s_j, C)$, which returns schematic correspondences C' that represent the differences between the two schemas s_i and s_j. Differences can include missing constructs or different names for the same constructs. The computational complexity of the operators is $O(n^2)$.

Model management operators provide flexibility for creating merged schemas that meet the user's requirements by merging multiple schemas and composing schematic correspondences, or to choose any of the source schemas and generate schematic correspondences between the selected schema(s) and all the other source schemas. For example, assume that the user who created the merged schema s_{m_2} shown in Figure 4 by matching all the source schemas with each other, inferring schematic correspondences between them and creating the merged schema s_{m_2} would also like to get information on the continents the countries are located in. The user has found another relational data source d_5 with the schema s_5 shown in Figure 9. As the user is aware that matching s_5 against the integration schema s_{m_2} might miss the association between s_{m_2}`.country.code` and s_5`.encompasses.country` due to the different names of the attributes and the lack of instances for s_{m_2} he decides to match s_5 with s_1 making use of instance data and infers the schematic correspondences between them, but he could have chosen any of the other source schemas. The results of $A_5 \leftarrow \texttt{match}(s_1, s_5, d_1, d_5)$ and $C_{s_1-s_5} \leftarrow \texttt{inferCorrespondences}(A_5)$ are shown in Figure 10.

To be able to merge s_{m_2} with s_5 the correspondences between the two schemas are needed, which can be obtained by composing the correspondences between s_{m_2}, s_1 and s_1, s_5: $C_{s_{m_2}-s_5} \leftarrow \texttt{compose}(C_{s_{m_2}-s_1}, C_{s_1-s_5})$. These correspondences are then used for merging s_{m_2} with s_5 to create s_{m_3}: $(s_{m_3}, C_{m_3-s_{m_2}}, C_{s_{m_3}-s_5}) \leftarrow \texttt{merge}(s_{m_2}, s_5, C_{s_{m_2}-s_5})$. `Compose` is then used to generate the schematic correspondences between the newly merged schema s_5 and all the other source schemas by composing the correspondences between s_{m_3}, s_{m_2} and those between s_{m_2} and each of the source schemas s_1, s_2 and s_3, respectively. If the user decides that one of the source schemas is actually the preferred schema to pose queries over, s_5 could be matched with the remaining schemas s_2 and

Matches between s_1 and s_5:

<{s_1.country.name}, {s_5.continent.name}, 0.5>
<{s_1.country.area}, {s_5.continent.area}, 0.8>
<{s_1.country}, {s_5.continent}, 0.4>
<{s_1.city.name}, {s_5.continent.name}, 0.5>
<{s_1.country.code}, {s_5.encompasses.country}, 0.55>
<{s_1.city.country}, {s_5.encompasses.country}, 0.67>

Schematic correspondences between s_1 and s_5:

<{s_1.country}, {s_5.continent}, different name same construct, 0.3>
<{s_1.country.name}, {s_5.continent.name}, same name same construct, 0.5>
<{s_1.country.area}, {s_5.continent.area}, same name same construct, 0.8>
<{s_1.country.code}, {s_5.continent}, missing attribute, 0.9>
...
<{s_1.country.population}, {s_5.continent}, missing attribute, 0.9>
<{s_1.country.code}, {s_5.encompasses.country}, different name same construct, 0.55>
<{s_1.city.country}, {s_5.encompasses.country}, same name same construct, 0.67>

Fig. 10. Matches and schematic correspondences between s_1 and s_5.

s_3 and the correspondences inferred from the matches returned. The resulting correspondences would be similar to those shown in Figure 10. Using the model management operators as illustrated in Section 2 and here, the user has generated a number of schemas and a number of sets of schematic correspondences.

ViewGen and the Resulting Mappings. As shown in [36] the schematic correspondences provide enough information to generate mappings automatically using the operator $M \leftarrow \texttt{viewGen}(C)$, which generates mappings M that correspond to the schematic correspondences C. The mappings are executable expressions that specify in form of two query strings, the specific properties of a mapping, how data that conforms to one schema has to be transformed to conform to another schema. The operator can be applied to any schematic correspondences between any two schemas (source- or merged integration schemas), allowing users to choose their favourite schema to pose queries over.

Iterating over the schematic correspondences the corresponding view between the participating source- and target-SuperAbstracts is generated. Depending on the cardinality of the participating SuperAbstracts, and more specifically on the kind of schematic correspondence, different approaches are used to generate the executable mapping. For example, in the case of one-to-one schematic correspondences, such as same name for same construct or different name for same construct, the view for populating the single target SuperAbstract from the single source SuperAbstract is generated with additional renaming applied in the case of the latter schematic correspondence. In the case of a one-to-many schematic correspondence, e.g., horizontal partitioning or vertical partitioning, the executable mapping for populating the single target SuperAbstract from the multiple source SuperAbstracts is generated by applying the union in the former case or by applying the join on the key attributes that are present in all vertically

Table 4. CMql algebra.

Operator
SCAN(SuperAbstract, Predicate) → Collection
REDUCE(Collection, {SuperLexical}) → Collection
JOIN(Left_Collection, Right_Collection, Predicate) → Collection
UNION(Left_Collection, Right_Collection) → Collection
EvaluateSQL(dataSource , SQLqueryString, Predicate, {resultTuple})→ {resultTuple}
EvaluateXQuery(dataSource , XQueryString, Predicate, {resultTuple})→ {resultTuple}

partitioned SuperAbstracts in the latter case. The computational complexity of the algorithm presented in [36] is $O(n)$.

5.3 Using the Dataspace: Evaluate Query

This section provides a brief overview of the expansion of queries posed over a schema represented in the canonical model into queries over potentially multiple sources, the translation of the source-specific sub-queries into the source-model-specific query languages and their evaluation [28].

We have defined *CMql*, a declarative query language inspired by SQL but defined over the constructs of the canonical model introduced in Section 5.1. A *CMql* query has the following form: `SELECT` $sl_1, ..., sl_n$ `FROM` $sa_1, ..., sa_m$ `WHERE` p, where sl_1,...,sl_n is a project list of *SuperLexicals*, sa_1,...,sa_m is a list of *SuperAbstracts*, and p is a conjunctive predicate. Queries can be posed over any schema, be it schemas of imported sources, global schemas generated using merge, or a manually specified global schema.

CMql queries are parsed, validated, translated into the algebra shown in Table 4 following standard translation schemes [20] during which selection predicates are pushed down into the SCAN operator, expanded, optimised, source-specific subqueries rewritten into the source-specific query languages and the query is evaluated with subqueries being sent to the query evaluator of the corresponding sources to be evaluated locally. The query processor consisting of components for each of those tasks (shown in Figure 6 - the components under Query Evaluation) is an extended version of the OGSA-DQP distributed query processor [34] which has been adapted for the models used in *DSToolkit*. The UNION operator is used in the context of query unfolding whereas EvaluateSQL and EvaluateXQuery are used to evaluate the source-specific subqueries that have been rewritten into the source-model-specific query languages. Queries are expanded using query unfolding [24] with the mappings between the schema over which the query is posed and any of the schemas of the sources over which the query is to be evaluated. For expansion, either all the appropriate mappings generated by `viewGen` can be used or a subset of the mappings can be selected using the operator `selectMappings` which is explained in more detail in Section 6. The expanded queries are optimised, and subqueries of the optimised query plan that are associated with specific sources are translated into the source-specific query languages. The translated subqueries are passed to EvaluateSQL and EvaluateXQuery with information on the source over which the subquery is to be evaluated, i.e., which local query evaluator is to evaluate the subquery.

Both operators can be parameterised with a predicate and result tuples, e.g., in the case of joins between two different sources. The query evaluator follows the iterator model [21], whereby each operator returns one result tuple at a time which can then be processed by the subsequent query operators, thereby removing the need to wait for a query operator to finish. A result tuple consists of multiple result values which in turn consist of a name of the corresponding superLexical and the actual value.

As mentioned earlier, *DSToolkit* not only allows multiple global schemas to co-exist, but also enables the user to pose queries over any schema of their choice, be it a merged schema, any of the source schemas or a manually provided schema that represents the user's preferred view of the data. For example, assume a user of data source d_1 would like to pose the following query q_3 over the dataspace:

```
SELECT c.name, c.population, c.longitude, c.latitude, o.name, o.population
FROM city c, country o
WHERE c.country = o.code
   AND o.population > 100000000
   AND c.population > 5000000.
```

However, s/he would like to pose the query over the schema s_1 of d_1 rather than any of the merged schemas, but would like the query to be evaluated not just over d_1, but also over d_2, which are the two data sources containing information on countries and their cities. The query is translated into the *CMql* algebra, expanded using query unfolding and the mappings between `city` and `country` in s_1 and s_2, which are similar to mappings map_1 and map_2 in Figure 5, optimised and the source-specific subqueries are rewritten. The operator tree of the expanded query is shown in Figure 11 and the resulting query with the source-specific queries rewritten is shown in Figure 12.

We now assume that a different user of the same dataspace would like to pose the query q_4 shown in Figure 13 over the merged schema s_{m_3} and would like the query to be evaluated over all the sources that contain the relevant information on cities in countries in which spanish is spoken and the continent the country is located in. The query tree corresponding to q_3 is shown in Figure 14. In the figure the mappings that can be used to expand the query are also listed where mappings $map1_{m_3}$, ..., $map6_{m_3}$ are the mappings between s_{m_3} and each of the source schemas s_1, s_2, and s_3 and are comparable to the mappings with the corresponding names map_1, ..., map_6 shown in Figure 5. The remaining mappings between s_5 and s_{m_3} are shown in Figure 15.

Without any indication of which mappings should be used in cases where there are multiple mappings suitable to populate a particular construct, the query is expanded by creating the union of the query in which the constructs over which the query is posed are populated by each combination of the suitable mappings. As there are six different ways of combining available mappings the expanded query is rather large. For this reason, we only show the part of the query that has been expanded using mapping $map6_{m_3}$ for `language`, $map1_{m_3}$ for `country`, $map10_{m_3}$ for `encompasses` and $map4_{m_3}$ for `city`. The resulting part of the expanded and rewritten query is shown in Figure 16.

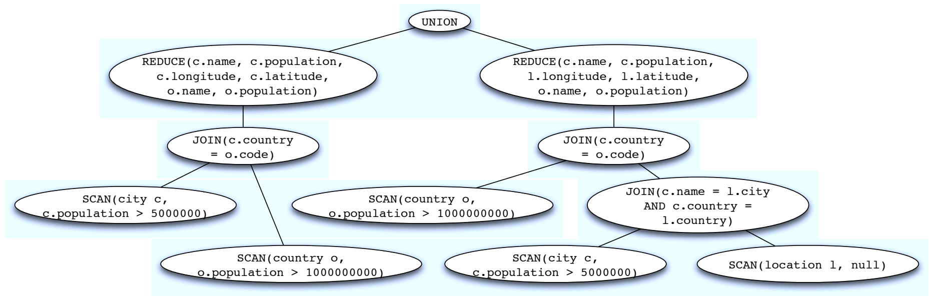

UNION
REDUCE(c.name, c.population, c.longitude, c.latitude, o.name, o.population)
REDUCE(c.name, c.population, l.longitude, l.latitude, o.name, o.population)
JOIN(c.country = o.code)
JOIN(c.country = o.code)
SCAN(city c, c.population > 5000000)
SCAN(country o, o.population > 1000000000)
JOIN(c.name = l.city AND c.country = l.country)
SCAN(country o, o.population > 1000000000)
SCAN(city c, c.population > 5000000)
SCAN(location l, null)

Fig. 11. Expanded query algebra tree of query q_3.

```
UNION(
    EvaluateSQL(d_1,
        SELECT c.name, c.population, c.longitude, c.latitude, o.name, o.population
        FROM city c, country o
        WHERE c.country = o.code
            AND o.population > 100000000 AND c.population > 5000000, null, null),
    EvaluateSQL(d_2,
        SELECT c.name, c.population, l.longitude, l.latitude, o.name, o.population
        FROM city c, location l, country o
        WHERE c.name = l.city AND c.country = l.country
            AND c.country = o.code
            AND o.population > 100000000 AND c.population > 5000000, null, null))
```

Fig. 12. Expanded and rewritten query q_3

```
SELECT c.name, c.population, c.longitude, c.latitude, o.name, e.continent, l.name
FROM city c, country o, encompasses e, language l
WHERE c.country = o.code
  AND o.code = e.country
  AND o.code = l.country
  AND l.name = "spanish"
```

Fig. 13. Query q_4.

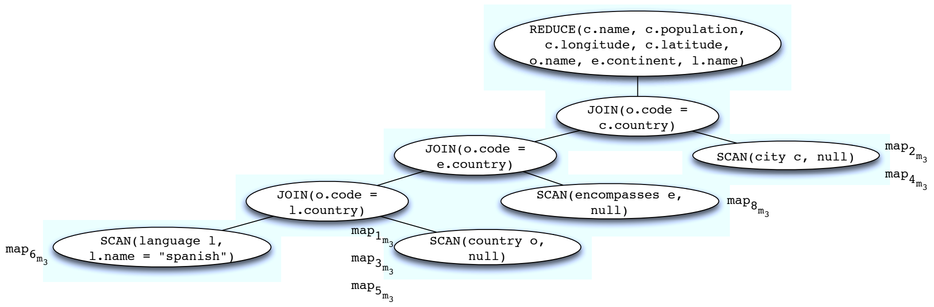


Fig. 14. Query algebra tree of query q_4.

$map9_{m_3}$ = <s_{m_3}.continent, select c.name, c.area from s_5.continent c>
$map10_{m_3}$ = <s_{m_3}.encompasses, select e.country, e.continent, e.percentage from s_5.encompasses e

Fig. 15. Mappings between s_{m_3} and s_5

```
...
UNION(...,
    REDUCE(
        JOIN(
            JOIN(
                JOIN(
                    EvaluateXQuery(d3,
                        <result>
                            let $s3 := doc("...")
                            for $l in $s3/country/language
                            where $l/name = "spanish"
                            return
                                <tuple>
                                    <l.country>{fn:data($l/country)}</l.country>
                                    <l.name>{fn:data($l/language_name)}</l.name>
                                    <l.percentage>{fn:data($l/percentage)}</l.percentage>
                                </tuple>
                        </result>, null, null),
                    EvaluateSQL(d1,
                        SELECT o.name, o.code, o.capital, o.area, o.population
                        FROM country o, null, null),
                    o.code = l.country),
                EvaluateSQL(d5,
                    SELECT e.country, e.continent, e.percentage
                    FROM encompasses e, null, null),
                o.code = e.country),
            EvaluateSQL(d2,
                SELECT c.name, c.country, c.population, l.longitude, l.latitude
                FROM cities c, location l
                WHERE c.name = l.city AND c.country = l.country
                    AND l.name = "spanish", null, null),
            o.code = c.country),
        {c.name, c.population, c.longitude, c.latitude, o.name, e.continent, l.name}))
```

Fig. 16. Expanded and rewritten partial query q_4

6 Improving Dataspace Using User Feedback on Results

Without the knowledge of which mappings meet the user's requirements or intentions the closest, queries have to be expanded using the combination of all potential mappings. As illustrated in the previous section this can lead to rather complex queries that are evaluated over numerous sources, some of which may not contain any of the data the user is actually interested in. Also, not all combinations of mappings may return results that the user would like to see. However, rather than ask the user to provide feedback on the mappings themselves as in [11, 10] or alter them to meet their needs, both of which requires a good understanding of schemas and mappings, thereby excluding the casual user, we ask the user to provide feedback on result tuples returned by the queries s/he has posed. This section describes the user feedback gathered and its application for annotating mappings, selecting and refining them to improve the integration over time [4]. This section also illustrates how previously gathered feedback can be utilised to evaluate the perceived quality of new complementary data sources with respect to the feedback previously provided, i.e., to evaluate how well a new data source matches the expectations of the user without requiring additional feedback from the user. The feedback can also be utilised along with new mappings over new sources for selection and refinement of mappings over already integrated sources. In contrast, rather than reusing previously provided

Table 5. Precision and recall of map_1, map_3 and map_5 based on user feedback in Table 1.

Mapping	Precision	Recall
map_1	1	1
map_3	0	0
map_5	0.5	1

feedback to gain information on new data sources, in [47] users need to provide feedback on the results returned by queries evaluated over the new sources.

Users can provide feedback on query result tuples of their own choosing indicating whether they expected a particular tuple that was returned to be present, i.e., a true positive (TP), or whether the tuple was not expected to be returned, i.e., a false positive (FP). Furthermore, users can also provide result tuples that were not returned, but that they expected to be part of the query result, i.e., false negatives (FN). It is worth mentioning that a true positive tuple of a given mapping may be a false negative tuple for another mapping. To illustrate this, consider that the user specifies that the tuple t is expected. The tuple t is a true positive for a given mapping m if m returns t, and is a false negative, otherwise. The feedback provides partial information on the extent of the construct in the schema over which the query was posed. This (partial) information can be used to calculate the precision and recall of the mappings that were used to produce those annotated results. Precision and recall are calculated as follows:

$$Precision(m, UF) = \frac{|TP(m, UF)|}{|TP(m, UF)| + |FP(m, UF)|} \quad (1)$$

$$Recall(m, UF) = \frac{|TP(m, UF)|}{|TP(m, UF)| + |FN(m, UF)|} \quad (2)$$

where $|TP(m,UF)|$, $|FP(m,UF)$, $|FN(m,UF)|$ denote, respectively, the number of TPs, FPs and FNs returned by a mapping m according to the user feedback UF on query results involving the mapping m.

For example, returning to the user feedback provided on result tuples to query q_1 shown earlier in Table 1, this user feedback is utilised to annotate the mappings map_1, map_3 and map_5 with their respective precision and recall computed by Equations 1 and 2, respectively. The results are shown in Table 5. Another option for annotating the mappings would be the use of gold-standard ground truth data, if available, rather than asking the user for feedback. This option, however, is not discussed here.

6.1 Improving Mappings and their Selection for Query Re-runs

The user feedback is not only used to annotate the mappings with their respective precision and recall based on the user feedback gathered. It can also be used to select the mappings that meet the user's requirements for future re-runs of queries and to generate better mappings with respect to the user's expectations using refinement. Both are explained in more detail in the following.

Mapping Selection. Using all candidate mappings to answer a user query may have undesirable consequences. In particular, there is a risk that the query will take a long time to be evaluated, if the number of candidate mappings is large, and that the resulting tuples are largely false positives, if the mappings are of poor quality. To overcome this issue, we implemented an operation that selects the mappings to be used to evaluate a user query. Not all users have the same requirements as to the completeness and soundness of the results. Because of this, the selection operation we developed provides the user with a means to specify his requirements with respect to results soundness, by specifying a threshold for precision, or to the completeness, by specifying a recall threshold. Specifically, given a set of candidate mappings M and a precision- or recall target λ set by the user, the $M' \leftarrow$ `selectMappings`(M, precisionTarget, recallTarget) operator selects the mappings $M' \subseteq M$ that are to be used to answer the query such that the union of query results returned by the selected mappings M' achieve λ. The problem is formulated as a constrained optimisation problem in which the selected target λ (say precision) is the constraint and the other value (say recall) is maximised and is solved using tabu search [40].

For an example, we revisit the user feedback provided on results of query q_1 in Table 1 and the resulting precision and recall of the corresponding mappings shown in Table 5. If the user chooses a desired precision target $0.5 < \lambda \leq 1$, only mapping map_1 would be used to answer the query, whereas for a precision target $0 < \lambda \leq 0.5$ both mappings map_1 and map_5 would be used. The recall of the mappings is either 1 or 0, so any recall target of $0 < \lambda \leq 1$ would exclude mapping map_3 from being utilised.

Mapping selection cannot only be used to exclude mappings that are correct in the sense that they associate the correct constructs with each other but do not meet the user's requirements with respect to the results they return, e.g., all three mappings map_1, map_3 and map_3 associate the construct representing countries in s_{m_2} with the constructs representing countries in the corresponding source schemas, however, not all the results returned meet the (subjective) expectation of the user. It can also be used to exclude incorrect mappings, which could be produced by the automatic bootstrapping process. For example, lets assume we have an additional data source d_6 with the following schema s_6:

```
province(name, country, capital, area, population)
```

Using the automatic bootstrapping process, the following incorrect mapping map_{11} could have been produced, associating incorrectly the construct representing countries in s_{m_3} with the construct representing province in s_6:

$map_{11} = <$ s_{m_3}.country, select p.name, p.country as code, p.capital, p.area, p.population from s_6.province p>

which would return information on provinces rather than countries. After the user has annotated some results returned by this mapping as false positives and chosen a precision target of $0 < \lambda$ this mapping is excluded from further query

```
$map_{12}$ = <$s_{m_2}$.country,
    select name, code, capital, area, population
    from (
        (select o.name, o.code, o.capital, o.area, o.population
        from $s_1$.country o)
    union
        (select o.name, o.code, o.capital, o.area, o.population
        from $s_2$.countries o)
    )>
```

Fig. 17. Mapping map_{12} created by refinement.

expansions. Gathering feedback only on result tuples which contain the data the user is familiar with rather than mappings enables casual users to improve the dataspace themselves rather than having to rely on developers who have a good understanding of both schemas and mappings.

Mapping Refinement. Mapping selection can only be successful in returning exactly the query results the user expects if the mappings meet the user's requirements. However, this may not be the case, e.g., as mentioned earlier d_1 contains information on european and d_2 information on african cities and countries, i.e., they form a horizontal partitioning. This, however, was not detected during the bootstrapping process as can be seen in the lack of a mapping that creates the union of the information from both sources (Fig. 5). If a user is interested in information from both european and african cities and countries, neither of the mappings between the merged schema and each of the source schemas will fully meet the requirements, only a union would achieve this.

The operator $M' \leftarrow$ `refineMappings`(M) aims to produce mappings M' that meet the user's requirements with respect to the feedback provided on result tuples better than the existing mappings M by trying to increase the number of true positives and/or reduce the number of false positives [4]. False positives can be reduced by filtering the results using the operators of the relational algebra that allow filtering, namely, join, intersection and difference. To increase the number of true positives union can be used. The space of mappings that can be generated by creating the join, intersection, difference or union of existing mappings M is very large. To explore the space, an evolutionary algorithm [40] is used which creates new mappings from existing mappings by mutating mappings, i.e., applying join, or combining mappings by applying intersection, difference or union (for more detail on the approach, see [4]).

Consider for example a user who is interested in information about european and african countries, and consider the following two candidate mappings:

```
$map_1$ = <$s_{m_2}$.country, select o.name, o.code, o.capital, o.area, o.population from $s_1$.country o>
$map_3$ = <$s_{m_2}$.country, select o.name, o.code, o.capital, o.area, o.population from $s_2$.countries o>
```

Both these mappings return tuples that are of relevance to the user. The two mappings, however, do not return the same set of expected tuples. This suggests an opportunity for increasing the recall by unioning the source queries of the two mappings. Using our refinement algorithm, we were able to create a new mapping map_{12} shown in Figure 17 to increase the recall of the results.

6.2 Assessing the Quality of New Data Sources and Improving the Mappings over Existing Data Sources

As well as applying mapping annotation, selection and refinement to mappings over already integrated data sources that have been used to expand queries the user has posed and the results of which the user has annotated, these techniques can also be used for mappings over new complementary data sources without requiring additional feedback. When a new data source is integrated such that mappings are generated that populate constructs in an existing global schema from the new source, previously executed queries can be re-run automatically and the tuples returned by the new mappings compared with tuples previously annotated by the user. When an annotated tuple with the same attribute values as a result tuple returned from the new source is found, this annotation is transferred to the new result tuple, thereby annotating tuples that have been produced using new mappings over the new source without requiring additional feedback. Once the result tuples returned by the new mappings are annotated, this information can be used to annotate those new mappings using the process described earlier in this section. This gives an indication of how well the new source meets the user's requirements with respect to the results its corresponding mappings return and with respect to the user feedback gathered previously.

For an example, we revisit the query q_1 of the user who is only interested in countries that are both european and mediterranean and has provided the feedback on some of the result tuples as shown in Table 1 indicating his expectations. Lets further assume that the user added the new data source d_4 containing information on mediterranean countries with the schema s_4:

```
mediterraneanCountry(name, code, capital, area, population)
```

and that the mapping map_7 was generated:

map_7 = <s_{m_2}.country,
  select m.name, m.code, m.capital, m.area, m.population
  from s_4.mediterraneanCountry m>

A rerun of query q_1 over d_4 using map_7 to expand the query will return amongst other tuples all the tuples shown in Table 1 as all the countries listed in the table and annotated by the user are mediterranean countries. The previously provided feedback shown in the table is used to annotate map_7 with its precision of 0.5 and its recall of 1.

As soon as mappings are annotated with their respective precision and recall this information can be utilised for mapping selection and refinement. This can be done independent of whether the annotation is based on user feedback provided on result tuples or whether the annotation has been inferred automatically by comparing tuples produced by new mappings with those that the user annotated previously.

As an example for selection of new mappings based on their automatically inferred annotation, we consider the mapping map_7 with its inferred precision

Table 6. Annotated result tuples of q_5.

name	code	capital	area	population	expected	not expected	mappings
Iceland	IS	Reykjavik	103000	270292	√		map_1
United Kingdom	GB	London	244820	58489975	√		map_1
Belarus	BY	Minsk	207600	10415973		√	map_1
Liechtenstein	FL	Vaduz	160	31122		√	map_1
Germany	D	Berlin	356910	83536115		√	map_1

and recall that are better than those of map_3. This new mapping will be selected along with map_1 and map_5 for future evaluations of q_1 as long as a precision target λ of $0 < \lambda \leq 0.5$ or a recall target of $0 < \lambda \leq 1$ is specified.

As an example for the refinement of existing mappings using information from new sources, let us assume that a user is interested in european countries that are located on an island. The user has previously executed the query q_5 `SELECT * FROM country o` posed over s_{m_2} but stipulated that the query should only be evaluated over d_1, as this source contains only information on european countries, which results in a large number of unexpected results as the majority of european countries are not located on an island. The user previously provided the feedback on a small number of result tuples shown in Table 6.

He has managed to find a data source d_7 containing information on countries that are located on islands and information on that island with the schemas s_7:

```
locatedOnIsland(country, island_name, area, longitude, latitude)
```

During the bootstrapping process the new information has been added to s_{m_2}, resulting in this schema:

```
country(name, code, capital, area, population)
city(name, country, population, longitude, latitude)
language(country, name, percentage)
locatedOnIsland(country, island_name, area, longitude, latitude)
```

with the following mapping to populate `locatedOnIsland` in the new merged schema with the information in d_7:

map_{13} = <s_{m_2}.locatedOnIsland,
  select l.country, l.island_name, l.area, l.longitude, l.latitude
  from s_7.locatedOnIsland l>

Adding the information from s_7 has not resulted in new mappings for information that was already part of the schema, e.g., `country`, but during the bootstrapping process it has been identified by `match` that `locatedOnIsland.country` represents the same information as `country.code`, namely the abbreviated names of countries. Using this information and the previously gathered feedback shown in Table 6 the refinement algorithm creates the following mapping, which joins the information on countries with the information on which countries are located on islands, thereby reducing the number of false positives, i.e., the european countries that are not located on islands.

```
$map_{14}$ = <$s_{m_2}$.country,
    select o.name, o.code, o.capital, o.area, o.population
    from $s_1$.country o, $s_7$.locatedOnIsland l
    where o.code = l.country>
```

Rerunning query q_5 using the new mapping map_{14} to expand the query returns information on all the countries that are european and are located on an island, i.e., all the tuples that are expected by the user but none of the tuples that are not expected as indicated by the feedback in Table 6. This means that map_{14} is annotated with its precision of 1 and its recall of 1. The mapping map_{13}, which provides completely new information on islands, cannot be annotated, though, as the tuples it produces have not been returned by any previously run query and, therefore, have not been annotated with feedback that can be reused.

7 Conclusions

In this paper, we presented *DSToolkit*, an architecture for flexible dataspace management and the first dataspace management system that:

1. Supports the whole lifecycle of a dataspace, namely initialisation, maintenance, usage and improvement. For usage, i.e., querying across multiple data sources, *DSToolkit* provides the means for structured *CMql* queries posed over any schema to be expanded, optimised and for source-specific sub-queries to be translated into the source-model-specific query language for evaluation over the data sources, thereby meeting one of the requirements for a dataspace management systems [19] and sub-challenge 2.2 in [25]. *DSToolkit* enables the user to specify the precision- or recall-target that the query results should meet, which is part of the requirement identified as sub-challenge 4.1 in [25]. However, *DSToolkit* does not provide support for keyword queries yet nor does it deal with uncertain or inconsistent data.
2. Is based on model management, thereby benefitting from the flexibility provided by the model management operators for managing heterogeneous models and associations between them. In the dataspace vision, it was highlighted that a dataspace management system needs to be able to support multiple data models and cope with integrated data sources over which it has no full control [19, 25]. Building on the basis of model management means that *DSToolkit* provides support for both, dealing with different data models and changes in the schemas of the integrated data sources. However, *DSToolkit* only provides support for structured data sources, but not unstructured.
3. Enables casual users to improve the integration by providing feedback on result tuples which is then utilised to annotate, select and refine the mappings used to expand the queries. *DSToolkit* benefits from the interaction with the user, i.e., the feedback provided by the user, by utilising it to determine which mappings meet the users' requirements better than others and generating new mappings with the aim to improve the integration of the data sources. The need for the analysis of the users' interaction with the dataspace and

the creation of additional relationships between sources or other forms of improvements of the dataspace was identified as challenge 5 in [25]. *DSToolkit* also estimates the quality of the query result in terms of its precision and recall based on previously gathered user feedback, a point identified as part of sub-challenge 2.2 in [25].

4. Can utilise the feedback gathered previously to determine the perceived quality of new data sources with respect to how well the data in the new source matches the user's expectations, to select and refine both mappings over previously integrated source and those over the new sources without requiring additional feedback from the user. In [25] the authors argue that it is important to reuse the information provided by users as much as possible (sub-challenge 5.3). *DSToolkit* reuses previously provided feedback for the integration of new data sources, therefore, reducing the amount of feedback the user has to provide.

The presented approach where the data remains in the sources, however, is not the only option for a dataspace system. The integrated data sets could also be curated, annotated and/or cleaned, which requires a (modified) copy of the data sources.

References

1. Atzeni, P., Bellomarini, L., Bugiotti, F., Gianforme, G.: Mism: A platform for model-independent solutions to model management problems. J. Data Semantics 14, 133–161 (2009)
2. Atzeni, P., Gianforme, G., Cappellari, P.: A universal metamodel and its dictionary. T. Large-Scale Data- and Knowledge-Centered Systems 1, 38–62 (2009)
3. Aumueller, D., Do, H.H., Massmann, S., Rahm, E.: Schema and ontology matching with coma++. In: SIGMOD Conference. pp. 906–908 (2005)
4. Belhajjame, K., Paton, N.W., Embury, S.M., Fernandes, A.A.A., Hedeler, C.: Feedback-based annotation, selection and refinement of schema mappings for dataspaces. In: EDBT. pp. 573–584 (2010)
5. Belhajjame, K., Paton, N.W., Fernandes, A.A.A., Hedeler, C., Embury, S.M.: User feedback as a first class citizen in information integration systems. In: CIDR. pp. 175–183 (2011)
6. Bernstein, P.A.: Applying model management to classical meta data problems. In: CIDR. pp. 209–220 (2003)
7. Bernstein, P.A., Halevy, A.Y., Pottinger, R.A.: A vision for management of complex models. SIGMOD Record 29(4), 55–63 (2000)
8. Bernstein, P.A., Melnik, S.: Model management 2.0: manipulating richer mappings. In: SIGMOD Conference. pp. 1–12 (2007)
9. Bernstein, P.A., Melnik, S., Petropoulos, M., Quix, C.: Industrial-strength schema matching. SIGMOD Record 33(4), 38–43 (2004)
10. Cao, H., Qi, Y., Candan, K.S., Sapino, M.L.: Feedback-driven result ranking and query refinement for exploring semi-structured data collections. In: EDBT. pp. 3–14 (2010)
11. Chai, X., Vuong, B.Q., Doan, A., Naughton, J.F.: Efficiently incorporating user feedback into information extraction and integration programs. In: SIGMOD Conference. pp. 87–100 (2009)

12. Chiticariu, L., Kolaitis, P.G., Popa, L.: Interactive generation of integrated schemas. In: SIGMOD Conference. pp. 833–846 (2008)
13. Chiticariu, L., Tan, W.C.: Debugging schema mappings with routes. In: VLDB. pp. 79–90 (2006)
14. Das Sarma, A., Dong, X., Halevy, A.: Bootstrapping pay-as-you-go data integration systems. In: SIGMOD. pp. 861–874 (2008)
15. Dittrich, J., Salles, M.A.V., Blunschi, L.: imemex: From search to information integration and back. IEEE Data Eng. Bull. 32(2), 28–35 (2009)
16. Do, H.H., Rahm, E.: Coma: a system for flexible combination of schema matching approaches. In: VLDB. pp. 610–621 (2002)
17. Do, H.H., Rahm, E.: Matching large schemas: Approaches and evaluation. Inf. Syst. 32(6), 857–885 (2007)
18. Dong, X., Halevy, A.Y.: A platform for personal information management and integration. In: CIDR. pp. 119–130 (2005)
19. Franklin, M.J., Halevy, A.Y., Maier, D.: From databases to dataspaces: a new abstraction for information management. SIGMOD Record 34(4), 27–33 (2005)
20. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database Systems The Complete Book. Pearson International Edition, 2nd edn. (2009)
21. Graefe, G.: Encapsulation of parallelism in the volcano query processing system. In: SIGMOD Conference. pp. 102–111 (1990)
22. Haas, L.M.: Beauty and the beast: The theory and practice of information integration. In: ICDT. pp. 28–43 (2007)
23. Haas, L., Lin, E., Roth, M.: Data integration through database federation. IBM SYSTEMS JOURNAL 41(4), 578–596 (2002)
24. Halevy, A.Y.: Answering queries using views: A survey. The VLDB Journal 10(4), 270–294 (2001)
25. Halevy, A.Y., Franklin, M.J., Maier, D.: Principles of dataspace systems. In: PODS. pp. 1–9 (2006)
26. Hedeler, C., Belhajjame, K., Fernandes, A.A.A., Embury, S.M., Paton, N.W.: Dimensions of dataspaces. In: BNCOD. pp. 55–66 (2009)
27. Hedeler, C., Belhajjame, K., Paton, N.W., Campi, A., Fernandes, A.A.A., Embury, S.M.: Dataspaces. In: SeCO Workshop. pp. 114–134 (2009)
28. Hedeler, C., Paton, N.W.: Utilising the mism model independent schema management platform for query evaluation. In: BNCOD (2011)
29. Ives, Z.G., Green, T.J., Karvounarakis, G., Taylor, N.E., Tannen, V., Talukdar, P.P., Jacob, M., Pereira, F.: The orchestra collaborative data sharing system. SIGMOD Record 37(3), 26–32 (2008)
30. Jeffery, S.R., Franklin, M.J., Halevy, A.Y.: Pay-as-you-go user feedback for dataspace systems. In: SIGMOD Conference. pp. 847–860 (2008)
31. Kensche, D., Quix, C., Li, X., Li, Y., Jarke, M.: Generic schema mappings for composition and query answering. Data & Knowledge Engineering (DKE) 68(7), 599–621 (2009)
32. Kim, W., Choi, I., Gala, S.K., Scheevel, M.: On resolving schematic heterogeneity in multidatabase systems. Distributed and Parallel Databases 1(3), 251–279 (1993)
33. Kim, W., Seo, J.: Classifying schematic and data heterogeneity in multidatabase systems. IEEE Computer 24(12), 12–18 (1991)
34. Lynden, S., Mukherjee, A., Hume, A.C., Fernandes, A.A.A., Paton, N.W., Sakellariou, R., Watson, P.: The design and implementation of OGSA-DQP: A service-based distributed query processor. Future Generation Comp. Syst. 25(3), 224–236 (2009)

35. Madhavan, J., Cohen, S., Dong, X.L., Halevy, A.Y., Jeffery, S.R., Ko, D., Yu, C.: Web-scale data integration: You can afford to pay as you go. In: CIDR. pp. 342–350 (2007)
36. Mao, L., Belhajjame, K., Paton, N.W., Fernandes, A.A.A.: Defining and using schematic correspondences for automatically generating schema mappings. In: CAiSE. pp. 79–93 (2009)
37. McBrien, P., Poulovassilis, A.: P2p query reformulation over both-as-view data transformation rules. In: International Conference on Databases, Information Systems, and Peer-to-Peer computing (DBISP2P). pp. 310–322 (2006)
38. McCann, R., Kramnik, A., Shen, W., Varadarajan, V., Sobulo, O., Doan, A.: Integrating data from disparate sources: A mass collaboration approach. In: ICDE. pp. 487–488 (2005)
39. Melnik, S., Rahm, E., Bernstein, P.A.: Rondo: a programming platform for generic model management. In: SIGMOD. pp. 193–204 (2003)
40. Michalewicz, Z., Fogel, D.: How to solve it: modern heuristics. Springer (2000)
41. Mork, P., Seligman, L., Rosenthal, A., Korb, J., Wolf, C.: The harmony integration workbench. J. Data Semantics 11, 65–93 (2008)
42. Naumann, F., Leser, U., Freytag, J.C.: Quality-driven integration of heterogenous information systems. In: VLDB. pp. 447–458 (1999)
43. Rahm, E., Bernstein, P.A.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
44. Scannapieco, M., Virgillito, A., Marchetti, C., Mecella, M., Baldoni, R.: The architecture: a platform for exchanging and improving data quality in cooperative information systems. Inf. Syst. 29(7), 551–582 (2004)
45. Seligman, L., Mork, P., Halevy, A.Y., Smith, K.P., Carey, M.J., Chen, K., Wolf, C., Madhavan, J., Kannan, A., Burdick, D.: Openii: an open source information integration toolkit. In: SIGMOD Conference. pp. 1057–1060 (2010)
46. Smith, A., Rizopoulos, N., McBrien, P.: Automed model management. In: ER. pp. 542–543 (2008)
47. Talukdar, P.P., Ives, Z.G., Pereira, F.: Automatically incorporating new sources in keyword search-based data integration. In: SIGMOD Conference. pp. 387–398 (2010)
48. Talukdar, P.P., Jacob, M., Mehmood, M.S., Crammer, K., Ives, Z.G., Pereira, F., Guha, S.: Learning to create data-integrating queries. PVLDB 1(1), 785–796 (2008)
49. Wang, R.Y.: A product perspective on total data quality management. Commun. ACM 41(2), 58–65 (1998)