Incremental Data Fusion based on Provenance Information

Carmem Satie Hara[1], Cristina Dutra de Aguiar Ciferri[2], and Ricardo Rodrigues Ciferri[3]

[1] Universidade Federal do Paraná – Curitiba, PR – Brazil, carmem@inf.ufpr.br
[2] Universidade de São Paulo – São Carlos, SP – Brazil, cdac@icmc.usp.br
[3] Universidade Federal de São Carlos – São Carlos, SP – Brazil, ricardo@dc.ufscar.br

Abstract. Data fusion is the process of combining multiple representations of the same object, extracted from several external sources, into a single and clean representation. It is usually the last step of an integration process, which is executed after the schema matching and the entity identification steps. More specifically, data fusion aims at solving attribute value conflicts based on user-defined rules. Although there exist several approaches in the literature for fusing data, few of them focus on optimizing the process when new versions of the sources become available. In this paper, we propose a model for incremental data fusion. Our approach is based on storing provenance information in the form of a sequence of operations. These operations reflect the last fusion rules applied on the imported data. By keeping both the original source value and the new fused data in the operations repository, we are able to reliably detect source value updates, and propagate them to the fusion process, which reapplies previously defined rules whenever it is possible. This approach reduces the number of data items affected by source updates and minimizes the amount of user manual intervention in future fusion processes.

1 Introduction

The huge amount of data available nowadays and the need to integrate data imported from several external sources continue to be a challenge to the database community. Although data integration has been investigated for several years, there is no single solution that suits all applications.

The integration process involves both schema and instance level integration. At the instance level, the integration process comprises two major problems [28]: entity identification ambiguity and attribute value conflict. Entity identification refers to the problem of identifying overlapping data in different sources. It has been the purpose of extensive research on the relational [23], entity-relationship [24], and XML [27] data models. Attribute value conflict refers to the problem of two or more sources containing information on the same entity or attribute, but with conflicting values. The process of combining several representations of one real world object into a single, consistent and clean representation is the goal of *data fusion* [7], which is the focus of this paper.

Data fusion is based on a set of strategies that determine how value conflicts are solved. A survey of existing approaches for data fusion can be found in [7]. As an example, when integrating several external sources, one can define that a conflict on a given numerical attribute should be solved by computing the average from the values provided by the sources. The data resulting from the fusion process can be stored in a local database for answering queries based on clean, mediated data. However, if no additional information is kept in the local database other than the fused value, when one of the data sources updates its value, all the other sources will have to be accessed again in order to apply the same fusion strategy. One approach for avoiding this is to keep the data provenance [11, 12]; that is, copies of the original values provided from the sources and the strategy applied to obtain the value stored in the database. Following this approach, the focus of this paper is on provenance-based data fusion.

In our system, provenance information is kept by mapping the application of fusion strategies to sequences of simple insert-remove-edit-copy operations. This resembles the works on manually curated data [8, 9], in which the system keeps a *log* of operations that the user undertakes in order to clean imported data. Although we consider a similar set of operations, in our system operations do not keep track of the user's actions, but keep the original source values and coordinate them with the local database. By keeping the original values, we are able to reliably detect source updates and propagate them to the local database. The problem of updating a database based on changes to an independently maintained source has recently been referred to as data coordination [22].

In this paper we propose a fusion system for XML which supports incremental updates based on provenance information. We assume that external sources have been previously processed for identifying entities, producing fully keyed documents. Our fusion approach is based on user-defined rules, which are stored in a *rule base*. Moreover, provenance information is kept in an *operations repository* that consists of simple operations that coordinate external sources with the local database. The operations repository along with the rule base allows incremental updates on the local database and minimizes the amount of user intervention in future fusion processes.

1.1 A Motivating Example

Consider two sources, s_1 and s_2, containing data on the same paper, but that disagree on the values reported for its attributes as depicted in Figures 1(a) and 1(b). In this example, we identify that `paper` information provided by s_1 and s_2 refers to the same publication because they coincide on their values for `title` and `year`. That is, `title` and `year` are the keys for `paper`. In order to generate a database with fused data, the user provides high-level *fusion rules* for deciding how value conflicts should be solved. As an example, the user can define that conflicts on paper's author `names` should be solved as follows: first, choose the value reported by the majority of the sources; if the conflict cannot be solved then consider the one reported from the most trustful source. Since in our example we only have two sources, the first strategy cannot be applied. Then, considering that we rely more on source s_1 than on s_2, the value stored in the database for the paper's first author is `John`, and the second author `Jack`, based on the values reported by s_1. Similarly, we can define that the database should contain the average number reported for `citationQty` based on all the sources. In this case, the resulting value stored in the database is `citationQty: 9`. The conflict on `city` is solved by *manually* choosing the value `Philadelphia`, reported by s_2. These results are stored in the mediated database, as shown in Figure 1(d).

When a new version of source s_1 or s_2 is uploaded, or new sources are integrated to the database, these fusion rules can be automatically reapplied, and only *new* conflicts are presented to the user in the fusion process. As an example, consider that a new version for s_1 is uploaded with a value 12 for `citationQty`. Given that the average number of citations, considering s_1 and s_2 is now 10, the database is updated with this value. Now consider that values from a third source s_3 is integrated to the database. If s_3 also contains values for the same paper, and reports `Jack` as its first author and `John` as the second, then the first strategy defined on the previous fusion process can be applied for updating the database with the values reported by the majority of the sources. That is, the `name` of the first `author` is updated to `Jack`, and the second to `John`, based on the values reported by s_2 and s_3.

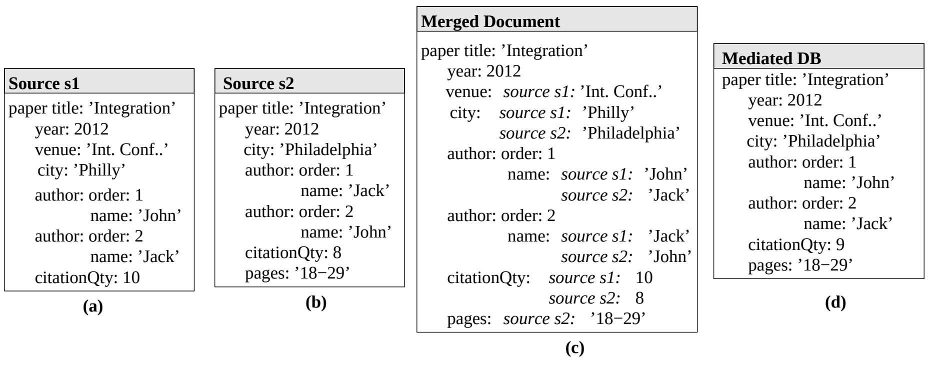

Fig. 1. Integration of two conflicting sources.

Our fusion system offers all the aforementioned functionality, and also allows the mediated database to be used as an integrated repository of curated data according to the users' decisions.

This paper builds on three previous works from the authors. The first proposes a system for XML data fusion, which allows the definition of data cleaning rules for solving value conflicts detected during the integration process [14]. The second presents a model for reapplying user's decisions in subsequent integration processes when data is manually curated by insert-remove-edit-copy operations that are stored in an operations repository [33]. At last, the third work introduces a data model for XML instance level integration that helps the resolution of value attribute conflicts by explicitly representing them in a merged document [26]. Here, we consider our previous works in the same setting. However, we focus on *mapping* fusion strategies to sequences of simple operations, and *reapplying* previous rules for incrementally updating the mediated database when new sources are uploaded or when sources are updated. The purpose of the system is to minimize the amount of user input in future fusion processes.

1.2 Organization

The paper is organized as follows. Section 2 describes preliminary definitions. Section 3 introduces the architecture of our fusion system, followed by the definition of our data model in Section 4. Section 5 details the modules the compose the system. Related work are presented in Section 6, and Section 7 concludes the paper.

2 Preliminary Definitions

Before describing the components of our data fusion system, we present definitions for XML keys (Section 2.1), strategies for data fusion (Section 2.2), and basic operations (Section 2.3). These notions have been previously proposed in the literature and we use them as building blocks in our system.

2.1 XML Keys

An XML document is typically modeled as a node-labeled tree T, which can be depicted in a directory style representation as illustrated in Figures 1(a) and 1(b). We assume that each XML tree has a distinct identifier, such as s_1 and s_2, which denotes its source. We refer to attribute and element nodes as *objects* throughout the article. Moreover, we say that an object is *simple* if it corresponds to a text element or an attribute, and *complex* otherwise.

Following the syntax proposed in [10], we define an XML key as *(context-path, (target-path, { key-paths}))*, where the values of the *key-paths* uniquely identify nodes reached following a *target-path* in the context of each subtree defined by the *context-path.*

Example 1. Given the XML trees depicted in Figures 1(a) and 1(b), the following key definitions allow us to uniquely identify a single node in each of the trees.

– k_1 : $(\epsilon, (paper, \{title, year\}))$: in the context of the entire document (ϵ denotes the root), a `paper` is identified by its `title` and `year` of publication;
– k_2 : $(paper, (author, \{order\})$: within the context of any subtree rooted at a `paper` node, an `author` is identified by its `order`;

– $k_3 : (paper, (citationQty, \{\}))$: within the context of any subtree rooted at a `paper` node, there exists at most one `citationQty` element; that is, it is identified by an empty set of values. Similarly, we can define uniqueness constraints for `venue`, `city`, `pages` and author `name` as follows: $k_4 : (paper, (venue, \{\}))$ and $k_5 : (paper, (city, \{\}))$ and $k_6 : (paper, (pages, \{\}))$ and $k_7 : (paper/author, (name, \{\}))$.

Observe that based on the key definitions, it is possible to generate a path expression for obtaining a node using key values as filters. As an example, based on k_1, we can obtain a (single) `paper` node from the trees in Figures 1(a) and 1(b) using the expression */paper[title='Integration' and year='2012']* and the first author with the expression */paper[title='Integration' and year='2012'] /author[order='1']*. Thus, these path expressions can be considered as the nodes' *keys* or *object identifiers*. We refer to nodes reached by key paths, such as *title* and *year* as *key nodes*.

2.2 Strategies for Data Fusion

There are a number of strategies proposed in the literature for solving value conflicts [6, 36, 16, 13, 17]. Here, we consider a set of strategies based on those proposed in [6]. We describe the ones that are used in this article below. However, the set of strategies can be much larger, with little impact on our fusion approach, as discussed in Section 6.

Trust Your Friends (TYF). This strategy is based on a reliability criterion. The user assigns a confidence rate for each source, and a value conflict is solved by choosing the one provided by the source with the highest confidence rate.

Meet In The Middle (MIM). This is a strategy to mediate the conflict by generating a new value that is a compromise among all conflicting values, e.g., an average of all conflicting numeric values.

Cry With The Wolves (CWW). This strategy is defined for choosing the value reported by the majority of sources.

Choose a Value (CAV). In this strategy the user manually chooses one value among those reported from the sources.

Pass It On (PIO). This is a non-resolving strategy. Although in most cases the user wants a single value for each data item, for some items she may want to postpone the decision for a future fusion process.

Observe that there are high-level strategies such as TYF, MIM, and CWW, and also value-based strategies such as CAV. In our approach, we reapply only high-level strategies on subsequent fusion processes without any user intervention. As an example, the conflict on `citationQty` described in Section 1.1 has been solved by computing the average value from all the ones reported from the sources (MIM strategy). This strategy can continue to be applied in future fusion processes, by taking into consideration the value updates and values uploaded from new sources.

However, this is not the case for value-based strategies. As an example, consider the conflict on `city` between sources s_1 and s_2 depicted in Figure 1(a) and (b). If the user manually chooses the value `‘Philadelphia’` reported by s_2 using the CAV strategy, we can assume that she will continue to do so as long as the sources keep providing the same values. Once one of them, say s_1, modifies its value to `‘Philadelphia, PA’`, it is not clear whether the decision of choosing the value reported from s_2 over s_1 is correct, and thus the strategy cannot be reapplied. Otherwise, inconsistencies would be introduced in the database without the user's consent.

2.3 Basic Operations

There are a number of definitions for basic operations on XML data, but here we adopt the ones proposed by [33]. Four operations are considered: *edit*, *copy*, *insert*, and *remove*. *Edit* is an unary operation that operates on simple objects and has the effect of modifying the object by assigning a value either provided by the user, or generated by the system as the result of an aggregate function. *Copy*, on the other hand, takes the value of a simple object provided from one source, for copying it to a second source. *Insert* and *remove* are operations on complex objects. *Insert* is a binary operation that creates a new object in one source, based on an object already stored in another source. In the newly created object, the identifiers are filled in with values obtained from the keys of the original object. Finally, *remove* is an unary operation that deletes an object from a source, based on its key.

Regarding the integration process, there are several methodologies for database integration. Here, we adopt a binary ladder strategy [2], in which we first analyze a first source, then analyze a second source by identifying its inconsistencies and managing them with regard to the first source, then analyze a third source by identifying its inconsistencies and managing them with regard to the first and second sources, and so on. Furthermore, as stated in Section 1, we assume that external sources have been previously processed for identifying entities, producing fully keyed documents. We adopt concepts that are similar to the notion of "insertion-friendly" set of keys defined in [10]. With insertion-friendly keys, one can unambiguously determine the position in the tree in which new elements should be inserted.

3 System Architecture

Our approach for tackling the problem of incrementally fusing XML data is based on keeping a repository of operations reflecting the user's decisions and data provenance, along with a rule base. That is, user-defined high-level fusion rules are stored in a *rule base.* The application of a strategy on a data item is mapped to a sequence of basic operations that are stored in the *operations repository.*

The architecture of the system is depicted in Figure 2. We consider the existence of several XML sources $s_1, \dots s_n$, that have been previously transformed to documents that follow the database schema. That is, we assume that any structural discrepancies among sources have been solved by a schema integrator prior to the fusion process. Moreover, we use key values as a means for *entity identification.* More specifically, whenever two elements from distinct sources are used to populate the same database element, based on their key values, they are considered to refer to the same entity in the real world. Thus, whenever their attribute values differ, we conclude that there is an *attribute value conflict* that should be solved.

The system is based on three modules: *fusion, validation,* and *update.* Data from each source is uploaded to the database separately by the *update* module. This module is responsible for checking whether imported elements already exist in the database, and if there are attribute value conflicts among them. If so, these attribute values are combined into a single representation in a *merged document.* In a *merged document*, data imported from several sources are combined whenever they are mapped to an object that coincide on their key values. Moreover, it explicit represents value conflicts among sources, along with the provenance for each value.

As an example, consider the source documents depicted in Figures 1(a) and 1(b) and the key definitions in Example 1. In the merged document, `paper` elements imported from sources s_1 and s_2 are combined because their `title` and `year` key elements coincide, revealing value conflicts on their `citationQty` and `city` values. Similarly, they disagree on who are the first and second authors of the paper. The resulting merged document, in which values of non-key simple objects are associated with their provenance, is depicted in Figure 1(c).

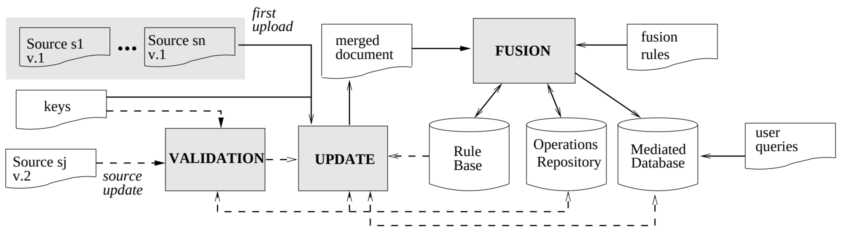

Fig. 2. Incremental fusion based on provenance.

Value conflicts are solved by the *fusion* module based on user-defined *fusion rules.* Fusion rules are stored in a *rule base.* They may be defined in the context of a single element or on a larger context involving multiple elements. Thus, if a newly detected conflict is within the context of an existing rule then the conflict can be automatically solved without any manual user intervention. As the result of applying a rule to a conflict, a value is written in the *mediated database*, which consists of fused data. That is, there exists at most one value associated with any element in the database. Since it contains no value conflicts, user queries are processed based on data stored in the mediated database.

In order to be able to reapply the same decisions in future fusion processes, applications of the rules are mapped to sequences of basic operations that are kept in the *operations repository*. Similar to database log files, these operations contain not only the new value of the data item, but also the original source values. As an example, consider the conflict on `citationQty` depicted in Figure 1(c). As a result of the application of the *Meet in the Middle* strategy, value 9 for the paper's `citationQty` is written in the mediated database, and the following sequence of basic operations is stored in the operations repository: (i) edit the mediated database (`db`) to 9; (ii) copy from `db` to s_1, modifying its value from 10 to 9; and (iii) copy from `db` to s_2, updating it from 8 to 9.

We use the notion of *validation* for determining when the effects of the operations in the repository are identical to the ones already executed in previous fusion processes. Intuitively, we would like to ensure a strict reproduction of the user's decisions, guaranteeing that the same rules defined by the user to decide previous conflicts will be applied to solve conflicts on the same object in the future. In the example above, if in new versions of s_1 and s_2 their values for `citationQty` remain unchanged, the operations on both sources are considered *valid*, since they continue to update 10 to 9 in s_1 and 8 to 9 in s_2. However, if one of them updates its value, say s_1 updates it to 12, then the operation on s_1 that maps 10 to 9 is *invalid* since the original value recorded in the operation does not match the value reported by the new version. As a result, the *update* module includes `citationQty` and the values reported by s_1 and s_2 in the merged document generated as input to a *fusion* process.

In the new fusion process, for every conflict in the merged document, it is checked whether there exists a high-level fusion rule already defined for the object. If this is the case, the conflict is solved, and both the mediated database and the operations repository are updated. In our running example, the value for `citationQty` in the mediated database is updated to 10, and the sequence of operations in the repository is replaced with new ones that reflect the new value recorded in the database. On the other hand, if there exists no fusion rules, or the existing strategy is value-based then a new decision is requested from the user. Intuitively, if a sequence of operations is *valid* there is no need for re-executing them because their effects are already recorded in the system. *Invalid* operations indicate source updates, and they can be solved without user intervention if high-level fusion rules have been defined on the conflicting objects.

4 Data Model

In this section we present the structure of the data involved in our fusion system: merged document (Section 4.1), rule base (Section 4.2), and operations repository (Section 4.3). We also define how fusion rules should be mapped to the operations repository (Section 4.4).

4.1 Merged Document

There are three categories of XML documents in our system: data source, merged document, and mediated database. They all follow the same schema which satisfies the following constraint: every element in the schema is associated with a key that determines how the element is identified in the context of its parent, based solely on its simple components. As an example, the set of keys in Example 1 is insertion-friendly for all the XML documents in Figure 1 given that every object is either keyed by a set of simple objects, such as `paper` objects, or they are unique in the context of their parent, such as `citationQty` and `name`.

A merged document differs from the source and mediated database on the contents of its simple objects. Instead of having text values, the merged document contains a set of pairs *(sourceId, value)* for every non-key simple object. Intuitively, a merged document combines into a single node all the values extracted from a set of sources that are identified by the same key. Discrepancies among these values indicate a conflict that is solved by fusing them into a single value, which in turn is stored in the mediated database.

Definition 1. *Given a set of sources S and a set $\mathcal{K}$ of insertion-friendly XML keys, we define a* **merged document** T_m *as an XML tree with a set of nodes V, such that a leaf node $v \in V$ is either: (a) a key node, which contains a single text value; or (b) a non-key node containing a set of pairs (sourceId, value), where sourceId is the identifier of a source $s \in S$ and value is extracted from a node in s that has the same key that identifies v in T_m according to $\mathcal{K}$.*

An example of a merged document is illustrated in Figure 1(c). Given a merged document T_m and a key k, we define a function `value(`T_m`, `k`)` to return the set of pairs associated with the node v with key k in T_m. We define a similar function on sources and the mediated database to return the text value associated with a simple object.

4.2 Rule Base

Given that a merged document explicitly represents value conflicts, we need a means for defining how these conflicts are solved. In our system, this is accomplished by user-defined fusion rules, stored in the *rule base.*

Definition 2. *A* **fusion rule** *is a pair $\langle \sigma, \Sigma \rangle$, where*

(1) σ is a path expression representing the context covered by the strategy;

(2) Σ is a non empty list of strategies for handling value conflicts on nodes reached by the context path σ.

The context of a rule is defined by a path expression σ and therefore it may cover not only a single element or attribute node, but also a *set* of nodes reached by following σ. Furthermore, a rule may define a *list* of strategies for solving a conflict. Thus, if the first strategy is not able to single out a value for a given data item, the following strategies are considered one by one until either the end of the list is reached or the conflict is solved.

Example 2. Consider the value conflicts on paper's `citationQty` and author `names` depicted in Figure 1(c). The fusion rules described in Section 1.1 can be defined as follows.

- ⟨*/paper[title='Integration' and year='2012']/citationQty, [MIM]* ⟩
- ⟨*/paper[title='Integration' and year='2012']/author/name, [CWW, TYF]*⟩

The first rule determines that conflicts on `citationQty` for the paper identified by `'Integration'` as its `title` and `'2012'` as its `year`, is solved by the *Meet in the Middle (MIM)* strategy. That is, the average value is computed, considering the imported values from all sources. The second rule defines that for any `author` of the same `paper`, `name` conflicts are solved by first finding the value reported by the majority of the sources (*Cry With the Wolves - CWW* strategy). If the strategy does not single out a value then *Trust Your Friends (TYF)* strategy is applied. Assuming that the confidence rate of s_1 is higher than s_2, the value reported from s_1 is chosen over that from s_2.

Observe that rules are defined in a context defined by a path expression. Thus, if conflicts on `citationQty` of all papers are to be solved by the MIM strategy, we could define a rule with a larger context as follows: ⟨*/paper/citationQty, [MIM]*⟩. That is, conflicts on any node reached by the path */paper/citationQty* are solved using the same strategy. Besides the notion of rule context, in our previous work, we introduce the notion of a *valid* set of rules, based on the concept of rule overriding. That is, inspired by object-oriented concepts, when the context of a rule is contained in the context of another, we choose to apply the one most specific to the node that presents a value conflict. As an example, we can define a *general* rule for solving conflicts on author `names` as ⟨*/paper/author/name, [TYF]*⟩, which can be overrid by a rule that is specific for 2012 papers: ⟨*/paper[year='2012']/author/name, [CWW]*⟩.

4.3 Operations Repository

One of the main goals of our proposed system is the ability to reapply user's decisions in future fusion processes. Our approach to reach this goal is to map the application of fusion strategies to sequences of basic operations that are stored in the *operations repository.*

Definition 3. *An* **operations repository** *is a list of records, grouped into blocks, where each record refers to a basic operation with the following attributes:*

- `bId`: *sequential number that identifies a list of records; given two* `bId`*s* b_1 *and* b_2, $b_1 < b_2$ *if* b_1 *has been executed before* b_2*;*
- `objId`: *key value that uniquely identifies an object on which the operation is executed;*
- `op`: *the operation can be an object insertion (*`in`*), removal (*`rm`*), or a simple object value edition (*`ed`*) or copy (*`cp`*);*

– `origin`: *source from which the operation obtains an object (or value) to be inserted (or copied) to another source. It is set to* `null` *for removal and edit operations;*
– `target`: *source updated by the operation;*
– `prevVal`: *target object value overwritten by operations edit and copy;*
– `newVal`: *new target object value.*

In the sequence we present an example of a block of operations that results from the application of a fusion strategy.

Example 3. Consider the rule defined for attribute `citationQty` in our running example. Its value is set to 9 in the mediated database when sources s_1 and s_2 are uploaded based on the MIM strategy. In the operations repository we store one edit operation in order to modify the mediated database (`db`) value to 9, followed by two copy operations from `db` to s_1 and s_2, as illustrated in Figure 3.

bId	objId	op	orig	target	prevVal	newVal
14	paper[..]/citationQty	ed	`null`	db	10	9
14	paper[..]/citationQty	cp	db	s1	10	9
14	paper[..]/citationQty	cp	db	s2	8	9

Fig. 3. A block of operations resulting from the *Meet in the Middle* strategy

Observe that the operations keep the original value reported by the sources in the `prevVal` field. Thus, when new sources are uploaded or if one of the sources updates the value, the system can continue to compute the average. For instance, consider the situation described in Section 1.1, in which a new version for s_1 is uploaded with the value 12 for `citationQty`. If the value uploaded from s_2 were not kept in the operations repository, we would be unable to compute the new average value of 10.

Observe also that the three operations belong to the same block, indicated by the same block identifier (`bId: 14`). This is because in the validation process each operation involving the uploaded source is analyzed to check whether the value in the current version matches the value recorded in the operation. Consider again the update from 10 to 12 on s_1's `citationQty` and the operations in Figure 3. The first copy operation from `db` to s_1 is *invalid*, since the value of `prevVal` is 10, while the new version reports 12. However, not only this operation should be considered invalid, but the whole block, since it reflects the application of a fusion strategy. Moreover, the reapplication of the MIM strategy affects not only the value of s_1, but all the other sources and the `db`. Thus, a block is considered *invalid* if it contains an invalid operation. In other words, validation is an operation-based process, but once an operation is found invalid, the whole block in which it is contained is considered invalid.

4.4 Mapping Fusion Rules to Operations

In this section we present details on how the application of a fusion strategy is recorded as a block of operations in the repository. First, observe that as the result of a rule application either: (a) a rule successfully singles out a value for an object that presented a conflict or (b) the user decides to postpone the decision on how to solve it for future fusion processes (*Pass it on* - PIO strategy).

In case (a) the block of operations has the following structure. First, an operation for modifying the value in the mediated database (`db`) is generated followed by a sequence of operations to copy this value to each of the sources that provide values for the same object. Observe that there are basically two types of strategies for solving a conflict: choose one value among the conflicting ones, such as strategies TYF, CWW, and CAV, or generate a new value, such as strategy MIM. An example of a block generated from the application of the MIM strategy is presented in Figure 3, in which we modify the value of `db` using an edit operation. However, when the strategy chooses one of the values provided from a source s_i, instead of an edit operation, we generate a copy operation from s_i to `db`. As an example, if the conflict on `citationQty` described in Example 3 is solved by choosing the value provided by s_1 over s_2, we generate a block with two operations: a copy from s_1 to `db` followed by a copy from `db` to s_2.

In case (b), in which the user decides to postpone the fusion decision, we keep the values provided by the sources in the operations repository, by modifying them to a `null` value using an edit operation. Since the actual value remains unknown, no value is recorded in the mediated database. Consider again the conflict on `citationQty`. If the user applies the PIO strategy, two edit operations are recorded in the repository: from 10 (as `prevVal`) to `null` on s_1, and from 8 to `null` on s_2.

An algorithm for generating a block of operations is given in Figure 4. Procedure `insBlockOp` takes as input five parameters: the *strategy* that has been applied to solve the conflict; the key *objId* of the object with conflicting values; a set *allVal* of pairs *(sourceId, val)* with the values reported from the sources, which may include a pair *(*`db`*, val)* if the mediated database already contains a value for the node; the value *finalVal* that results from the application of the *strategy*; and the source identification *valSource* that provides *finalVal*. After obtaining a new block identification *bId* (Line 1), the procedure keeps in *dbPrevVal* the previous value stored in the mediated database (Lines 2 to 4). Lines 7 to 9 considers the case when the strategy is PIO, generating a block of operations to edit the value of each source to `null`. The case when a final value for solving the conflict has been determined is considered in Lines 11 to 17. A `valSource` with `null` value indicates that a new value, not extracted from the sources have been generated to solve the conflict. In this case, an edit operation is generated (Lines 11 and 12); otherwise, we generate a copy operation (Lines 14 and 15). In the sequence, copy operations from the database to all remaining sources are recorded in the same block (Lines 16 to 17).

Procedure `insBlockOp` can be executed in $O(|S|)$ time, where $|S|$ denotes the number of input sources. To see this, observe that the set *allVal* is of size

```
Procedure insBlockOp (strategy, objId, allVal, finalVal, valSource)
Input: strategy applied to solve the value conflict, objId of the node,
       allVal: a set of pairs (sourceId, val),
       finalVal: the value recorded in the mediated DB,
       valSource: the sourceId that provided finalVal
1.  bId:= new( block );                {generates a new bId}
2.  if there exists a pair (db, v') in allVal then
3.     dbPrevVal:= v';
4.     remove (db, v') from allVal;
5.  else
6.     dbPrevVal:= null;
7.  if strategy is 'PIO' then
8.     for each (sourceId, val) in allVal do
9.        insOpRep([bId, objId, 'ed', null, sourceId, val, null]);
10. else
11.    if valSource is null then  {the strategy created a new value}
12.       insOpRep([bId, objId, 'ed', null, 'db', dbPrevVal, finalVal]);
13.    else                       {the strategy chose a reported value}
14.       insOpRep([bId, objId, 'cp', valSource, 'db', dbPrevVal, finalVal]);
15.       remove (valSource, finalVal) from allVal;
16.    for all pairs (sourceId, val) in allVal do
17.       insOpRep([bId, objId, 'cp', 'db', sourceId, val, finalVal]);
```

Fig. 4. Algorithm for inserting a block of operations

$O(|S|)$ since it contains at most one element for each source. Thus, checking containment in the set (Line 2) and removal from the set (Lines 4 and 15) takes $O(|S|)$ time. Lines 8-9 and 16-17 also take $O(|S|)$ time since procedure `insOpRep` takes constant time for writing a record at the end of operations repository file.

One advantage of keeping the operations repository is the feedback the system can give back to the sources. That is, after the fusion process, we can easily generate a sequence of operations for making any source s_i consistent with the mediated database simply by selecting the operations in which the `target` is s_i. Considering again the contents of the operation repository in Figure 3, a feedback for s_1 consists of the first copy operation, while for s_2 it contains the second copy operation.

5 System Modules

Given the data model presented in the previous section, we are now ready to describe the functionality of the fusion, validate, and update modules that compose our system.

5.1 Fusion Module

The major goal of the fusion module is to generate a *mediated database* resulting from the fusion of data imported from several sources. The input to the fusion module is a *merged document* and a set of user-defined *fusion rules*. Besides generating the mediated database, rules are stored in the *rule base*, and the *operations repository* is updated in order to reflect to last fusion operations that produced the values stored in the database.

Figure 5 presents an algorithm for the fusion module. `Clean` is a recursive function that traverses a merged document in post-order. Observe that in a merged document, all value conflicts are in the leaves. Thus, when processing internal nodes, the algorithm only calls the `clean` function recursively in order to collect the set of source identifiers that populate its descendants. This is because the provenance of the uploaded values in the merged document are recorded only on the set of pairs *(sourceId, val)* associated with the leaves. Thus, given *obj*, an internal node in a merged document, there exists a correspondent node in a source if it contributes with at least a value for one of the *obj*'s descendants. For each of these sources, we generate an insert operation in the operations repository by invoking the procedure `insObjOpRep` (Lines 1 to 7). In this procedure, for each *sourceId* in the set, it checks whether an insertion operation for the source already exists, and if not a new one is recorded.

When processing a simple non-key object, the function first obtains its set of pairs *(sourceId, val)* and its set of value providers by calling `getValues` and `getSourceIds`, respectively (Lines 9 and 10). If all sources agree on the reported value, it is simply stored in the mediated database and a block of operations is generated in the operations repository (Lines 11 to 14). Otherwise, we first look if there exists already a fusion rule defined for the node (Line 16) and check whether the conflict can be solved calling procedure `applyRule`. Observe that `applyRule` only reapplies high-level strategies such as TYF, MIM, and TYF. If the existing strategy is value-based, such as CAV, then it is removed from the rule base, without solving the conflict. Thus, if the conflict persists, the definition of a new rule is requested from the user (Lines 21 to 26), which is stored in the rule base (Line 23). If after this process, the conflict still persists, we conclude that the user chooses not the solve it at the moment. Thus, we record a PIO strategy for the node in the rule base, and remove the node from the database if it exists. The fusion decision is also recorded in the operations repository by invoking the `insBlockOp` procedure (Line 33). It is worth noticing that by allowing the user to postpone the fusion decision, it is possible to upload several sources to the system before making any decision on how to solve the conflicts. That is, although we consider a binary ladder integration approach in which sources are uploaded to the system one-by-one, the cleansing decisions are not necessarily made considering one new source at a time. It is also worth noticing that human input are valuable and should be used whenever possible. Therefore, we designed our system so that we notify users when previous changes have been invalidated and allow them to give suggestions on how these changes might be managed.

Algorithm *clean* can be executed in $O(|T|^3|R||S|)$ time, where $|T|$ is the size of the merged document, $|R|$ is the size of the rule base and $|S|$ is the number of sources. Observe that each node in the tree is processed once. For internal nodes, procedure `insObjOpRep` is invoked to determine whether the collected

```
Function clean (obj)
Input: obj: an object with key objId in a merged document mergedDoc
Output: setsIds: set of sourceIds that populate an obj's descendant in mergedDoc
1.  if obj is an internal node then
2.     setsIds:= {};
3.     for all obj's children c do
4.        if c is not a key object then
5.           setsIds:= setsIds ∪ clean( c );
6.     insObjOpRep( objId, setsIds );
7.     return setsIds;
8.  else
9.     allVal:= getValues( objId ); {return set of pairs (sourceId, val)}
10.    setsIds:= getSourceIds( allVal );
11.    if all sources provide the same value v then
12.       updateDB( objId, v );
13.       sourceFinalVal:= smallest sourceId in setsIds;
14.       insBlockOp( null, objId, allVal, v, sourceFinalVal );
15.    else
16.       rule:= getRule(objId); {obtain list of strategies [r1, ..., rn] from rule base}
17.       solved:= false;
18.       while not solved and rule not empty do
19.          r:= extractFirst( rule );
20.          solved:= applyRule( r, allVal, finalVal, sourceFinalVal );
21.       if not solved then                {request new rule from the user}
22.          newRule:= getNewRule( objId ) from user input;
23.          storeRuleBase( newRule );
24.          while not solved and newRule not empty do
25.             r:= extractFirst( newRule );
26.             solved:= applyRule( r, allVal, finalVal, sourceFinalVal );
27.       if not solved then
28.          r:= 'PIO';
29.          finalVal:= null;
30.          remDB( objId );
31.       else
32.          updateDB( objId, finalVal );
33.       insBlockOp( r, objId, allVal, finalVal, sourceFinalVal );
34.    return setsIds;
```

Fig. 5. Algorithm clean

sourceIds have already been inserted in the operations repository. This takes a single traversal of the operations repository, which is of size $O(|T| * |S|)$, since each node in the tree may have at most one record for each source in the set S. For leaves, on the other hand, the execution of `getValues`, `getSourceIds`, and also for checking whether all sources agree on their values take $O(|S|)$ time, given that each leaf may have at most one value for each source. If all sources agree, procedures `updateDB` and `insBlockOp` procedures are invoked, which takes

$O(|T|)$ and $O(|S|)$ time, respectively. The existence of value conflicts among sources requires the application of a rule. Function `getRule` can be executed in $O(|R| * |T|^2)$ time. Each rule is considered once, and the path expression σ in the rule is evaluated on T to check whether it contains the cleaning node. This is executed in $(|T| * |\sigma|)$ time [18] which is $O(|T|^2)$. Once the rule to be applied is singled out, each of its strategies are applied by calling `applyRule`, which can be executed in time $O(|S|)$ since the rules require scanning through the values provided by each source. The execution time for getting a new rule require the same time as the application of a new rule, in addition to storing it in the rule base, which takes $O(1)$ given that the rule is written at the end of the file. Updates on the database in lines 30 and 32 takes $O(|T|)$, while the insertion of a new block of operations in line 33 is $O(|S|)$. Thus, the entire algorithm is $O(|T| * ((|T| * |S|) + |S| + (|T| + |S|) + (|R| * |T|^2 + |S| + |T| + |S|)))$, which is $O(|T|^3|S||R|)$.

Example 4. Consider again our running example. Suppose that first, source s_1 is uploaded to the mediated database. Observe that for the first uploaded source s_1, the merged document is almost identical to the source, but with the non-key leaves annotated with the provenance of the single element in the set *{(s_1, val)}*. Since there are no conflicts, algorithm `clean` generates a document identical to s_1 in the mediated database and records these operations in the operations repository. The contents of the operations repository at this point is illustrated in Figure 6. Observe that the insertion operations (blocks 4, 6, and 8) have been generated by procedure `insObjOpRep`, while the remaining blocks are recorded by `insBlockOp`.

bId	objId	op	orig	target	prevVal	newVal
1	paper[..]/venue	cp	s1	db	`null`	‘Int. Conf..’
2	paper[..]/city	cp	s1	db	`null`	‘Philly’
3	paper[..]/author[order=‘1’]/name	cp	s1	db	`null`	‘John’
4	paper[..]/author[order=‘1’]	in	s1	db		
5	paper[..]/author[order=‘2’]/name	cp	s1	db	`null`	‘Jack’
6	paper[..]/author[order=‘2’]	in	s1	db		
7	paper[..]/citationQty	cp	s1	db	`null`	‘10’
8	paper[title=‘Int..]	in	s1	db		

Fig. 6. Operations repository after upload of source s_1

If in the sequence s_2 is uploaded, the contents of the operations repository are modified as presented by Figure 7. New operations are generated for s_2's internal nodes (blocks 11, 13, and 16) but not for s_1, because they have already been generated during s_1's upload. Since value conflicts have been detected on `city`, author `names`, and `citationQty`, new blocks reflecting the user's decisions are recorded (blocks 9, 10, 12, and 14). Observe also that previously existing

bId	objId	op	orig	target	prevVal	newVal
1	paper[..]/venue	cp	s1	db	null	‘Int. Conf..’
~~2~~	~~paper[..]/city~~	~~cp~~	~~s1~~	~~db~~	~~null~~	~~‘Philly’~~
~~3~~	~~paper[..]/author[order=‘1’]/name~~	~~cp~~	~~s1~~	~~db~~	~~null~~	~~‘John’~~
4	paper[..]/author[order=‘1’]	in	s1	db		
~~5~~	~~paper[..]/author[order=‘2’]/name~~	~~cp~~	~~s1~~	~~db~~	~~null~~	~~‘Jack’~~
6	paper[..]/author[order=‘2’]	in	s1	db		
~~7~~	~~paper[..]/citationQty~~	~~cp~~	~~s1~~	~~db~~	~~null~~	~~‘10’~~
8	paper[title=‘Int..]	in	s1	db		
9	paper[..]/city	cp	s2	db	‘Philly’	‘Philadelphia’
9	paper[..]/city	cp	db	s1	‘Philly’	‘Philadelphia’
10	paper[..]/author[order=‘1’]/name	cp	s1	db	‘John’	‘John’
10	paper[..]/author[order=‘1’]/name	cp	db	s2	‘Jack’	‘John’
11	paper[..]/author[order=‘1’]	in	s2	db		
12	paper[..]/author[order=‘2’]/name	cp	s1	db	‘Jack’	‘Jack’
12	paper[..]/author[order=‘2’]/name	cp	db	s2	‘John’	‘Jack’
13	paper[..]/author[order=‘2’]	in	s2	db		
14	paper[..]/citationQty	ed	null	db	10	9
14	paper[..]/citationQty	cp	db	s1	10	9
14	paper[..]/citationQty	cp	db	s2	8	9
15	paper[..]/pages	cp	s2	db	null	‘18-29’
16	paper[title=‘Int..]	in	s2	db		

Fig. 7. Operations repository after upload of source s_2

blocks involving these objects are removed from the repository (blocks 2, 3, 5, and 7). These removals are executed by the update module, which is based on the validation process, described in the next section.

5.2 Validation Module

The main goal of the validation module is to determine whether the execution of the operations in the repository have the same effect if executed on new versions of the sources. That is, if a new version presents no updates then all operations on the source are *valid* and thus there is no need for reexecuting them. On the other hand, an *invalid* operation indicates a source update which requires the object to go through a new fusion process. By selecting objects involved in invalid operations, the validation process can substantially reduce the volume of data considered in subsequent fusion processes. The validation module is also responsible for detecting removals and insertions in the source.

An algorithm for the validation module is presented in Figure 8. It takes as input an XML tree provided by a source identified by *sId*, and produces as output three sets, all of them containing *objIds*: *invalidUpdate* for invalid operations due to source value updates, *invalidRem* for invalid operations due to removals in the source, and *newObjIds* for elements inserted in the new version. First, we initialize the set *newObjIds* with the keys of all nodes in the document *s*,

```
Function validate (s)
Input: s: an XML tree with a new version for data source sId
Output: invalidUpdate: objIds for invalid operations due to updates,
        invalidRem: objIds for invalid operations due to removals
        newObjIds: objIds of elements inserted in the new version
1.  newObjIds:= set of all objIds in s;
2.  invalidUpdate:= {};
3.  invalidRem:= {};
4.  for each block b in the operations repository do
5.    for each record r in b do
6.      if r.origin == sId or r.target == sId then
7.        if r.objId is not in newObjIds then
8.          insert r.objId in invalidRem;
9.        else
10.         remove r.objId from newObjIds;
11.         if (r.op == 'cp' or r.op == 'ed') and
              ((r.origin == sId and value(s, r.objId) <> r.newVal) or
              (r.target == sId and value(s, r.objId) <> r.prevVal)) then
12.           insert r.objId in invalidUpdate;
13. return (invalidUpdate, invalidRem, newObjIds);
```

Fig. 8. Algorithm for validating operations

and the remaining sets as empty (Lines 1 to 3). Then each block in the operations repository is examined. Observe that blocks consist of operations on the same object and there exists at most one operation involving each source, but multiple operations involving the mediated database. Recall that a block is considered invalid if it contains at least one invalid operation. Thus, when validating a block, we first check whether there exists an operation involving sId. If so, either the object continues to be provided by the source or it has been removed. In the latter case, we consider the operation invalid and insert the *objId* in the *invalidRem* set (Lines 7 and 8). In the former case, since the object has already been provided in a previous version, we remove it from *newObjIds* (Line 10). Moreover, the algorithm checks if the value provided in the new version remains unchanged. Recall that in an operation record, `newValue` refers to the value provided by the `origin` source to update the `target` source, while `prevValue` refers to the value previously stored in `target`. Thus, we consider an operation invalid either if it contains sId as the `origin` and the value provided by the new version disagrees with the operation's `newValue` attribute or if it contains sId as the `target` and the provided value disagrees with the `prevValue` attribute (Lines 11 and 12). Recall that function `value` *(s, objId)* is responsible for extracting the value associated with the node with key *objId* in the document s.

Function `validate` can be executed in $O(|T|^2|S|)$ time. To see this, observe that the number of blocks in the operations repository is the number of nodes in the database T, and that each block contains at most $|S|$ records, one for each source. When processing a record in the repository, we have to check whether the object is in the set *newObjIds*, which is of size $O(|T|)$ and possibly obtain its value in the database, which takes $O(|T|)$ time. Thus, the time complexity of the algorithm is $O(|T| * |S| * |T|)$.

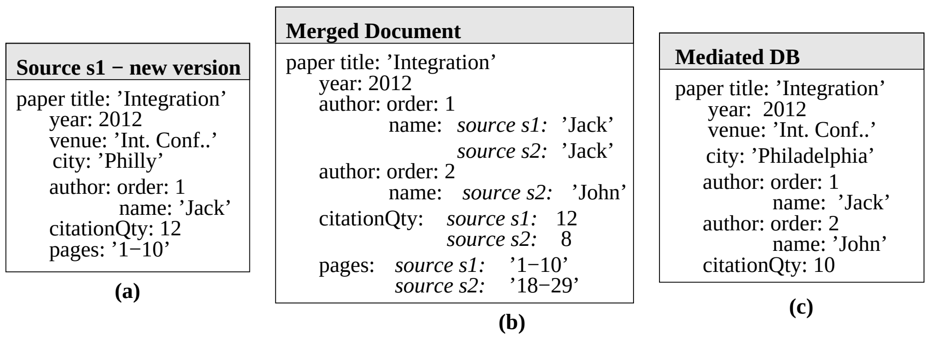

Fig. 9. Upload of s_1’s new version.

Example 5. As an example, consider the new version for source s_1 (referred to as $s_1.v_2$) presented in Figure 9(a) and the operations repository in Figure 7. The operation in the block with `bId:1` is valid because s_1 is the `origin` and the value in `newVal` coincides with the value in the new version $s_1.v_2$. The insertion operation in block 4 is also valid because there exists a node with the same key for the author with `order:1` in $s_1.v_2$. However, the insertion in block 6 is invalid because there exists no author with `order:2` in $s_1.v_2$ and thus this object is inserted in the *invalidRem* set. The next block containing an invalid operation is the one with `bId:10`. The first operation in this block has s_1 as the `origin` but the value in `newVal` (*‘John’*) disagrees with the value in $s_1.v_2$. Thus, the object is inserted in the *invalidUpdate* set. After processing all the operations in the repository, the only remaining object in the *newObjIds* is */paper[..]/pages* given that it is the only new object in the new version. The final contents of the other two sets are: *{/paper[..]/author[order=‘1’]/name, paper[..]/citationQty}* for *invalidUpdate*, and *{/paper[..]/author[order=‘2’], /paper[..]/author[order=‘2’]/name}* for the set *invalidRem.*

The sets resulting from the validate module are then given as input to the update module, which is responsible for generating a merged document with conflicts involving the updated objects.

5.3 Update Module

The goal of the update module is twofold. First, it generates a merged document which explicitly represents conflicts involving elements in a new source or in elements updated in a new version of a source. This document is the input to the fusion process described in Section 5.1. Second, it removes from the operations repository blocks containing invalid operations.

An algorithm for the update module is given in Figure 10. Function `update` takes as input a new version of a source *sId*, represented by an XML tree, and the three sets generated by the `validate` function and processes each of them as follows. First, observe that *objIds* in the *invalidUpdate* set are always simple elements since they are the ones that contain associated values. Thus, we simply remove the block of operations involving the object from the operations repository, and an element in the merged document is generated substituting the value associated with *sId* by the one provided by its new version (Lines 2 to 6). Observe that function `extractValues` receives a list of operation records, and returns a set of pairs *(sourceId, value)*, where *value* consists of the original value provided by *sourceId*. That is, if in the operation record *sourceId* is the `origin` then *value* is extracted from the `newValue` attribute; otherwise, it is extracted from `prevValue`. Moreover, procedure `insMerged`*(mergedDoc, objId, allVal)* inserts an element n identified by *objId* in *mergedDoc* and all the elements in the path from the root to n if they do not exist. The values for n are obtained from the set of pairs in *allVal*, except the pair associated with the source '`db`'. The sets *invalidRem* and *newObjIds* are processed similarly (Lines 7 to 18, and 19 to 24, respectively). Observe, however, that when an object is removed from the new version, we must check if *sId* was the only source that provided it. If this is the case, we also have to remove it from the mediated database (Lines 11 and 16).

The update module may be invoked both after a validation process or for the first upload of a new source. For new sources, function `update` is called with both *invalidUpdate* and *invalidRem* as empty sets, and *newObjIds* containing the set of all *objId*s in the new document.

Function `update` can be executed in $O(|T|^2|S|)$ time. Observe that each object in the source is either in the set *invalidUpdate*, *invalidRem*, or *newObjIds*. For each of them, the operations repository, of size $O(|T| * |S|)$ is traversed once in order to be updated, and either the database or the merged document, both of size $|T|$, has to be updated. These operations require a single traversal on the tree. Thus the entire function is $O(|T| * (|T||S| + |T|)$, which is $O(|T|^2|S|)$.

Example 6. Consider again the contents of the sets of updated *objIds* in Example 5 and the operations repository in Figure 7. Based on the contents of *invalidUpdate*, block 10 (on `paper[..]/author[order='1']/name` and block 14 (on `paper[..]/citationQty` are removed from the repository and elements in the merged document are inserted with the original values provided by source s_2 and the values in the new version of s_1, as depicted in Figure 9(b). Similarly, based on the contents of *invalidRem*, block 6 (on `paper[..]/author[order='2']`) is removed from the repository. Observe that in this case, the corresponding object is not removed from the database because the repository still contains an insertion operation based on the `author` list provided by s_2. Moreover, block 12 is removed, with operations on `paper[..]/author[order='2']/name`, and an element is generated in the merged document, containing only the value provided

```
Function update (s, invalidUpdate, invalidRem, newObjIds)
Input: s: an XML tree provided by source sId
       invalidUpdate, invalidRem, newObjIds: sets of objIds of updates on s
Output: mergedDoc: updated values in s combined with values from other sources
1.  mergedDoc:= $\epsilon$;
2.  for each objId in invalidUpdate do
3.    newVal:= value( s, objId );
4.    extract block b involving objId from the operations repository;
5.    allVal:= extractValues(b) $-$ {(sId, _)} $\cup$ {(sId, newVal)};
6.    insMerged( mergedDoc, objId, allVal );
7.  for each objId in invalidRem do
8.    if objId refers to an internal node then
9.      remove [_, objId, 'in', sId, 'db', null, null] from the operations repository;
10.     if there exists no other operation on objId in the operations repository then
11.       remDB( objId );
12.   else
13.     extract block b involving objId from the operations repository;
14.     allVal:= extractValues(b) $-$ {(sId, _)};
15.     if allVal is empty then
16.       remDB( objId );
17.     else
18.       insMerged( mergedDoc, objId, allVal );
19. for each objId in newObjIds do
20.   if object with key objId is a simple object then
21.     newVal:= value( s, objId );
22.     extract block b involving objId from the operations repository;
23.     allVal:= extractValues( b ) $\cup$ {(sId, newVal)};
24.     insMerged( mergedDoc, objId, allVal );
25. return mergedDoc;
```

Fig. 10. Algorithm for the update module

by s_2. Given that the set *newObjIds* contains a single element `paper[..]/pages`, block 15 is removed from the repository, and the value provided by s_2 extracted from the operation is combined with the new element inserted in s_1. The resulting merged document is presented in Figure 9(b). Observe that elements `venue` and `city`, which remained unchanged in the new version are not inserted in the merged document since their operations remain valid.

Considering only the fusion rules presented in Example 2, value conflicts on elements `citationQty` and author `name` can be solved without any manual intervention, but not on `pages`. Thus, the fusion module requests a new rule from the user. If she decides to postpone the decision then the system sets the strategy to be PIO. The resulting mediated database is presented in Figure 9(c) and the final contents of the operations repository is given in Figure 11. Here we do not show the removed blocks, but only the ones that remained from the previous snapshot and the new blocks, which are above and below the dashed line, respectively.

bId	objId	op	orig	target	prevVal	newVal
1	paper[..]/venue	cp	s1	db	`null`	‘Int. Conf..’
4	paper[..]/author[order=‘1’]	in	s1	db		
8	paper[title=‘Int..]	in	s1	db		
9	paper[..]/city	cp	s2	db	‘Philly’	‘Philadelphia’
9	paper[..]/city	cp	db	s1	‘Philly’	‘Philadelphia’
11	paper[..]/author[order=‘1’]	in	s2	db		
13	paper[..]/author[order=‘2’]	in	s2	db		
16	paper[title=‘Int..]	in	s2	db		
17	paper[..]/author[order=‘1’]/name	cp	s1	db	‘John’	‘Jack’
17	paper[..]/author[order=‘1’]/name	cp	db	s2	‘Jack’	‘Jack’
18	paper[..]/author[order=‘2’]/name	cp	s2	db	‘Jack’	‘John’
19	paper[..]/citationQty	ed	`null`	db	9	10
19	paper[..]/citationQty	cp	db	s1	12	10
19	paper[..]/citationQty	cp	db	s2	8	10
20	paper[..]/pages	ed	`null`	s1	‘1-10’	`null`
20	paper[..]/pages	ed	`null`	s2	‘18-29’	`null`

Fig. 11. Operations Repository after upload of s_1’s new version

Although the complexity of the algorithms in the paper had been presented in terms of the input size, the complexity of an incremental algorithm can also be measured in terms of the size of changes in the input and output, which represents the updating costs that are inherent to the incremental problem itself. With this respect, an incremental algorithm is said to be bounded if its cost can be expressed as a function of the size of changes. Intuitively, the algorithm is bounded if it processes only the subset of data input and output that change [29]. Recall that the goal of function `validate` is to determine which objects have been changed. That is, $|changed| = |invalidUpdate| + |invalidRem| + |newObjIds|$. Moreover, these changes affect the value of these objects in the mediated database and the corresponding records in the operations repository. Given that the merged document T is built based on these records, $|affected| = |T|$. The extraction of these records from the operations repository and the update of the mediated database can be done in time defined as a function of $|affected|$ if there exists an appropriate index structure on the *objId* both on the operations repository and the mediated database. Similarly, in order to bound the time complexity of the *clear* function to $|affected|$ we need an auxiliary structure to get the fusion rule defined on each object affected by the changes in the source new version.

6 Related Work

Data integration and cleaning have been studied extensively by the database community [4, 7]. Most of previous works consider data on relational format, but recently it has been stressed the need for investigating the problem of solving conflicts on semi-structured data. XClean [34] is a system that allows declarative and modular specification of a cleaning process. It consists of a declarative language with operators that cover not only the fusion process, but also entity identification and combination of values that refer to the same object. The main goal is to provide a modular system that can be easily extended with new operators. Potter's Wheel [30] follow a cleaning strategy based on a set of operations to transform data, such as *format, drop, copy, merge, split, divide, fold* and *select*. However, instead of storing the result of a data transformation, the sources are stored along with the definition of the transformation. The transformation is applied on-the-fly whenever a consistent and clean information is required. Hummer [5] and Fusionplex [25] are systems that focus on the fusion process. Hummer proposes an extension for SQL with fusion functions that can be applied to attributes in the query result. Fusionplex is also a strategy-based system in which conflicts are solved based both on metadata such as timestamp, cost, accuracy, and availability, and value-based strategies. However, none of these systems focus on incremental updates on fused data when sources are updated, which is the goal of our work.

There are a number of strategies for data fusion proposed in the literature [6, 36, 16, 13, 17], and a survey can be found in [7]. The strategies we described in Section 2.2 and used throughout the paper were introduced in [6]. However, extending our system with new strategies have little impact on our incremental update approach. First, observe that as a result of the application of a strategy, one of the following sequence of operations is recorded: *(case 1)* a copy operation from a source to the mediated database and several copy operations from the mediated database to each remaining source; and *(case 2)* an edit operation in the mediated database and several copy operations from the mediated database to each source. In the work described in [36], given a large number of facts that correspond to conflicting information obtained from several websites, it applies an iterative method to infer the trustworthiness of websites and to determine the confidence of facts based on the inferred trustworthiness. Solomon [16] is a system that can detect copying between sources and measure the quality of sources based on the intuition that copying may change the sources' quality. It applies the results to solve data conflicts and to decide true values of entities. In [13], it is proposed a model for determining the relative accuracy of attributes. Based on accuracy rules and an inference system, this work determines whenever possible a unique entity whose attributes are composed of the most accurate values from all conflicting attributes from the same real world object. If there is not enough information to generate a complete entity, the work computes the top-k candidate entities based on a preference model. Another work that focus on determining a unique entity whose attributes are consistent and store the most current value is described in [17]. The conflict resolution is solved by specifying data currency in terms of a partial currency order and currency constraints, and by enforcing data consistence with conditional functional dependencies. Based on the results produced by the aforementioned strategies, the user can solve a value conflict by choosing the most trustworthy fact, the most appropriate true value, the most appropriate top-k candidate entity, or by simply agreeing with the result returned by the strategy. As the value that is chosen to solve a value conflict is always obtained from a given source, our system can be extended to consider these strategies recording their results in the operations repository following *case 1*.

Regarding provenance-based integration systems that have been proposed in the literature, the ELIT (Exploration and LIneage Tracing) system [32] focuses on the lineage tracing problem in mediator-based integration systems. It collects information related to provenance during query processing in order to use this information to identify the data in the heterogeneous sources that contributed to a query answer. In Trio [3, 35], data provenance is used to estimate the quality of imported data. Similar to our approach, the system stores values of the same piece of data imported from external sources. However, these value conflicts are solved by attaching confidence rates to values. But ELIT and Trio differ from our work on how provenance is applied in the integration process. Differently from our work, neither ELIT nor Trio store provenance on data transformations, which in our system are based on fusion rules and, therefore, they cannot be used to reapply previous fusion decisions.

Two systems that follow the operation-based approach, i.e., keep track of provenance related to data and transformations based on operations, are CPDB (Copy-Paste DataBase) [8, 9] and CHIME (Capturing Human Intension Metadata with Entities) [1]. The main goal of CPDB is to manage provenance for manually curated databases, as defined by its authors as follows. "Given a definition of the complete and correct history of a database as it evolves over time, the goal of CPDB is to store sufficient provenance information to be able to answer queries about the history given only the provenance information and the final database state(s)". Also, Buneman et al. [8] investigate four techniques for storing provenance information, named naive provenance, transactional provenance, hierarchical provenance e transactional-hierarchical provenance. These techniques are aimed to reduce the provenance storage size, by defining different levels of details, from a higher level of detail (i.e., naive provenance) to a lower level of detail (i.e., transactional-hierarchical provenance). On the other hand, in CHIME the user first integrates heterogeneous sources into a single relation using as a basis a set of operations. These operations are collected automatically and store the data used in the integration, as well as which data is correct. Then, the user may query this integrated relation to extract information about the data collected in order to perform data audit. Our model differs from CPDB and CHIME on its purpose. We aim at reapplying fusion decisions in subsequent source uploads, while this feature is not supported neither by CPDB nor CHIME.

Orchestra [21] is a system for sharing structured data that is collaboratively authored by a large community of users. It models the exchange of data among sites as update exchange among peers, which is subject to transformations through schema mappings. Also, it employs data provenance for enforcing trust policies that are used to solve conflicts and for performing update exchange incrementally. Panda [19, 20] is a generic framework for selectively update the output of a data-oriented workflow. That is, the user selects the data items she wants to update, and the system traces back their origin in the workflow in order to recompute their current values. The application of a fusion rule can be considered a data transformation in the Panda setting. However, Orchestra and Panda are based on specific characteristics that differs them from our work. Orchestra requires that each source provides its updates (delta) since the last integration process, while our approach does not require delta files, and has a much richer set of conflict solving rules. Panda is generic for any data transformation, and although one of its goals is similar to the idea of reapplying previous decisions (defined as workflows) in subsequent ones, it does not provide details on how the reapplication can be applied for data fusion processes, which is the focus of this paper.

Data provenance has also been used in the literature to support Extract-Transform-Load (ETL) processes in data warehousing environments (e.g., [15, 31]). However, the use of provenance is typically to store metadata that allows one to trace the data origin and transformations, and not at incremental application of the transformations.

Finally, incremental updates on a database based on source updates has been recently referred to as data coordination [22]. However, the approach proposed in [22] differs from ours on how updates on the sources are detected. While we rely on the operations validation process, [22] proposes a materialization of the source data followed by an algorithm for detecting the differences with a new version. To the best of our knowledge, our approach is the first to apply an operation-based provenance model in the context of data fusion processes.

7 Conclusion

In this paper we presented a system that tackles the problem of incrementally updating a database populated with fused data provided by external sources. The approach is based on storing the data provenance in an operations repository that consists of records that contain both the original and new values. Since the operations coordinate the database with the sources, they can be used to provide feedback to the sources with the results of the fusion process. We proposed a validation process, which reliably determines whether a source updated an object provided in previous processes. When an update is identified, the value is combined with the values provided from other sources, extracted from the operations repository, in order to go through a new fusion process. By filtering out the objects that remained unchanged in new versions, and reapplying previously defined fusion rules, we can substantially minimize the need for manual user intervention in future fusion processes.

We intend to extend the XFusion [14] tool with the functionality presented in this paper and run some experiments in order to determine the efficacy of the proposed approach. Efficient storage and index structures to support the operations repository is also a topic for future investigation. In this paper, we adopt the where-provenance model [12], by keeping the origin of each data item that contributed to a value stored in the database, and the fusion rule that originated the final value. We can extend the proposed model by keeping all the source updates, so that it would be possible to obtain historical data by tracing back what have been the updates since their first upload to the system. Another line of investigation consists of extending the proposed framework for allowing update operations directly on the mediated database with operations logged in the operations repository. That is, the mediated database would combine imported data with local generated data. We intend to investigate how these direct operations impact those resulting from the application of fusion rules, possibly extending previous results [33] with characterizations of transitive and overlapping operations. We also plan to extend our system with the data fusion strategies surveyed in Section 6.

References

1. Archer, D.W., Delcambre, L.M.L., Maier, D.: A framework for fine-grained data integration and curation, with provenance, in a dataspace. In: Proceedings of the 1st Workshop on the Theory and Practice of Provenance. pp. 1–10 (2009)
2. Batini, C., Lenzerini, M., Navathe, S.B.: Comparative analysis of methodologies for database schema integration. ACM Computing Surveys 18(4) (Dec 1986)
3. Benjelloun, O., Sarma, A.D., Hayworth, C., Widom, J.: An introduction to ULDBs and the Trio system. IEEE Data Engineering Bulletin 29(1), 5–16 (2006)
4. Bhattacharya, I., Getoor, L.: Collective entity resolution in relational data. IEEE Data Engineering Bulletin 29(2), 4–12 (2006)
5. Bilke, A., Bleiholder, J., Naumann, F., Böhm, C., Weis, M.: Automatic data fusion with hummer. In: Proceedings of the 31st VLDB Conference. pp. 1251–1254 (2005)
6. Bleiholder, J., Naumann, F.: Conflict handling strategies in an integrated information system. In: Proceedings of the International Workshop on Information Integration on the Web (IIWeb) (2006)
7. Bleiholder, J., Naumann, F.: Data fusion. ACM Computing Survey 41(1), 1–41 (2008)
8. Buneman, P., Chapman, A., Cheney, J.: Provenance management in curated databases. In: SIGMOD'06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data. pp. 539–550 (2006)
9. Buneman, P., Chapman, A., Cheney, J., Vansummeren, S.: A provenance model for manually curated data. In: IPAW'06: Proceedings of the International Provenance and Annotation Workshop. pp. 162–170 (2006)
10. Buneman, P., Davidson, S., Fan, W., Hara, C., Tan, W.C.: Keys for XML. Computer Networks 39(5), 473–487 (Aug 2002)
11. Buneman, P., Khanna, S., Tan, W.C.: Data provenance: Some basic issues. In: FST TCS 2000: Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 87–93. Springer-Verlag, London, UK (2000)
12. Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. In: ICDT'01: Proceedings of 8th International Conference on Database Theory. pp. 316–330 (2001)

13. Cao, Y., Fan, W., Yu, W.: Determining the relative accuracy of attributes. In: SIGMOD'13: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 565–576 (2013)
14. Cecchin, F., Ciferri, C.D.A., Hara, C.S.: Xml data fusion. In: Proc. of the International Conference on Data Warehousing and Knowledge Discovery (DaWaK) (2010)
15. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. The VLDB Journal 12(1), 41–58 (2003)
16. Dong, X., Berti-Equille, L., Hu, Y., Srivastava, D.: SOLOMON: Seeking the truth via copying detection. PVLDB 3(2), 1617–1620 (2010)
17. Fan, W., Geerts, F., Tang, N., Yu, W.: Inferring data currency and consistency for conflict resolution. In: ICDE'13: Proceedings of the IEEE International Conference on Data Engineering. pp. 470–481 (2013)
18. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing xpath queries. In: VLDB'2002: Proceedings of the 28th International Conference on Very Large Data Bases. pp. 95–106 (2002)
19. Ikeda, R., Widom, J.: Panda: A system for provenance and data. IEEE Data Engineering Bulletin 33(3), 42–49 (2010)
20. Ikeda, R., Salihoglu, S., Widom, J.: Provenance-based refresh in data-oriented workflows. In: Proceedings of the 20th ACM international conference on Information and knowledge management. pp. 1659–1668. CIKM '11, ACM, New York, NY, USA (2011), `http://doi.acm.org/10.1145/2063576.2063816`
21. 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)
22. Lawrence, M., Pottinger, R., Staub-French, S.: Data coordination: Supporting contingent updates. Proceedings of the VLDB Endowment 4(11), 831–842 (2011)
23. Lim, E.P., Srivastava, J., Prabhakar, S., Richardson, J.: Entity identification in database integration. Information Sciences 89(1) (1996)
24. Menestrina, D., Benjelloun, O., Garcia-Molina, H.: Generic entity resolution with data confidences. In: Proceedings of the International VLDB Workshop on Clean Databases. Seoul, Korea (2006)
25. Motro, A., Anokhin, P.: Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. Information Fusion 7(2), 176–196 (2006)
26. do Nascimento, A.M., Hara, C.S.: A model for XML instance level integration. In: SBBD'08: Proceedings of the 23rd Brazilian Symposium on Databases. pp. 46–60 (2008)
27. Poggi, A., Abiteboul, S.: XML data integration with identification. In: International Workshop on Database Programming Languages (DBPL) (2005)
28. Prabhakar, S., Richardson, J., Srivastava, J., Lim, E.P.: Instance-level integration in federated autonomous databases. In: Hawaiian Conference for System Science (1993)
29. Ramalingam, G., Reps, T.W.: An incremental algorithm for a generalization of the shortest-path problem. Journal of Algorithms 21(2), 267–305 (1996)
30. Raman, V., Hellerstein, J.M.: Potter's wheel: An interactive data cleaning system. In: VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases. pp. 381–390 (2001)
31. Sellis, T.K., Skoutas, D., Simitsis, A., Vassiliadis, P.: Data provenance in ETL scenarios. In: Proceedings of the 1st Workshop on Principles of Provenance. pp. 1–3 (2007)

32. Shiri, N., Taghizadeh-Azari, A.: Lineage tracing in mediator-based information integration systems. In: Proceedings of the 5th International School and Symposium on Advanced Distributed Systems. pp. 267–282 (2005)
33. Tomazela, B., Hara, C.S., Ciferri, R.R., Ciferri, C.D.A.: Empowering integration processes with data provenance. Data & Knowledge Engineering 86, 102–123 (2013)
34. Weis, M., Manolescu, I.: Declarative XML data cleaning with XClean. In: International Conf. on Advanced Information Systems Engineering (CaiSE). pp. 96–110 (2007)
35. Widom, J.: Trio: A system for data, uncertainty, and lineage. In: C. Aggarwal, editor, Managing and Mining Uncertain Data, chap. 5. Springer (2009)
36. Yin, X., Han, J., Yu, P.S.: Truth discovery with multiple conflicting information providers on the web. IEEE Transactions on Knowledge and Data Engineering 20(6), 796–808 (2008)