What if Multiusers Wish to Reconcile Their Data?

Dayse Silveira de Almeida[1], Carmem Satie Hara[2] and Cristina Dutra de Aguiar Ciferri[1]
[1]*Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, Brazil*
[2]*Departamento de Informática, Universidade Federal do Paraná, Curitiba, PR, Brazil*

Keywords: Asynchronous Collaboration, Reconciliation, Data Integration, Data Sharing, Multiuser, Data Provenance.

Abstract: Reconciliation is the process of providing a consistent view of the data imported from different sources. Despite some efforts reported in the literature for providing data reconciliation solutions with asynchronous collaboration, the challenge of reconciling data when multiple users work asynchronously over *local* copies of the same imported data has received less attention. In this paper, we propose AcCORD, an asynchronous collaborative data reconciliation model based on data provenance. AcCORD is innovative because it supports applications in which all users are required to agree on the data integration in order to provide a single consistent view to all of them, as well as applications that allow users to disagree on the correct data value, but promote collaboration by sharing updates. We also introduce different policies based on provenance for solving conflicts among multiusers' updates. An experimental study investigates the main characteristics of the policies, showing the efficacy of AcCORD.

1 INTRODUCTION

Reconciliation is the process of providing a consistent view of the data imported from different sources. This problem has been widely investigated considering a single user, or multiple users working over a single copy of the data (Köpcke et al., 2010; Cao et al., 2013). However, the problem of reconciling data when multiple users work asynchronously over *distinct* copies of the data has received less attention. In this context, users may *collaborate* by sharing their decisions for solving data conflicts. However, they are allowed to *disagree* on which are the correct values. Thus, the goal of the reconciliation process can be to provide a either *single* consistent view for all users, or *distinct* views for each of them. That is, when data conflicts are detected, it may be required that all users agree on the correct data value in a *collaborative integration process*, or it may be allowed them to disagree, but help each other by sharing their decisions.

Conventional data integration involves schema and instance level integration. At the schema level the goal is to solve structural heterogeneity (Nguyen et al., 2011; Bhattacharjee and Jamil, 2012). Instance level integration faces two major problems: entity resolution and conflict resolution. The first refers to the problem of identifying data that refer to same object in real world, and the latter, to the problem of solving conflicts among values provided from different sources (Köpcke et al., 2010; Cao et al., 2013). In this paper, we focus on the conflict resolution problem.

Integration processes are time-consuming and usually involve manual intervention. Thus, when several users import data from the same sources, or subsets of these sources, share decisions and work collaboratively can be an interesting time-saving strategy. Given that the collaborative work should not interfere on each user integration process, a remote synchronization upon each update (Kermarrec et al., 2001) may not be convenient. Synchronous systems are highly interactive, and prevent users from working in a disconnected mode. Thus, for the sake of flexibility, an asynchronous reconciliation is desirable for several kinds of applications.

An asynchronous collaborative data reconciliation is characterized by a high degree of independence, in which users work autonomously, and are loosely connected to each other at any given time. An example of such a collaborative environment are e-health systems. These systems enable collaborative treatments among healthcare service providers, such as physicians, hospitals and laboratories, which can autonomously and independently curate, revise, and extend the shared data (Hossain et al., 2014). Other application scenarios include data sharing and curation in bioinformatics and bibliographical data.

Silveira de Almeida D., Satie Hara C. and Dutra de Aguiar Ciferri C..
What if Multiusers Wish to Reconcile Their Data?.
DOI: 10.5220/0005384401840195
In *Proceedings of the 17th International Conference on Enterprise Information Systems* (ICEIS-2015), pages 184-195
ISBN: 978-989-758-096-3

The independence and asynchronicity of individual integration processes pose two main challenges. First, how the conflict solving decisions are stored in order to be exchanged among collaborators. That is, in order to share their decisions, information on the updates made by each user must be stored. Such information are called *data provenance* and consists of a set of metadata that identify the data sources and transformations applied to them, from their inception to their current state (Cheney et al., 2009; Mahmood et al., 2013). Second, it is possible that not all collaborators agree on their updates. Thus, we need to solve conflicts among collaborators' updates, mainly when different collaborators take distinct decisions over the same data conflict.

In this paper, we propose AcCORD (*Asynchronous COllaborative data ReconcIliation moDel*). AcCORD is an asynchronous model for collaborative reconciliation in which each user's updates are kept in a repository of operations. Each user has its own repository for storing the provenance and his/her own copy of the sources. That is, whenever inconsistencies among imported sources are detected, the user may autonomously take decisions to solve them, and updates that are locally executed are registered in his/her own repository. Updates are shared among collaborators by importing each others repositories. Since users may have different points of view, repositories may also be inconsistent. Thus, in this paper we also propose policies for solving conflicts among repositories. Distinct policies can be applied by different users in order to reconcile the updates. Depending on the applied policy, the final view of the imported sources may either be the same for all users, that is, a single global integrated view, or result in distinct local views for each of them.

1.1 Motivating Example

Consider a set of sources in the domain of bibliographic data named after their owners, such as *Anne, Bruce and Eugene*. Users $U_1, \ldots, U_n$ are individually integrating these sources, by keeping local copies of the sources and updating them whenever conflicts on overlapping data are detected.

Example 1. Consider user U_1. After importing the data, he/she notices that they disagree on the values of attributes *Final page*, *Year* and *Initial page* of the paper entitled *"PrInt"*, and decides that *Eugene* is the most trustworthy source. Thus he/she copies (*cp*) its *Final page* value (*234*) to source *Bruce* overwriting the value *1365* and then from *Bruce* to *Anne* overwriting the value *1352*. These updates are stored in repository R_1 as depicted in Figure 1. In order to help presenting the examples, we add to the repositories an user identifier (column *user*) and an operation identifier (column *id*).

user	id	op	origin source	target source	key	attribute	origin value	target value	timestamp
U1	1	cp	Eugene	Bruce	Paper [Title=PrInt]	Final page	234	1365	08:28:43_ 03/18/2014
	2	cp	Bruce	Anne	Paper [Title=PrInt]	Final page	234	1352	08:29:01_ 03/18/2014

Figure 1: Repository R_1 with user U_1 decisions.

Here, we borrow the idea of data provenance based on operations from the PrInt model (Tomazela et al., 2013). However, our collaborative reconciliation model is no limited to this specific operation-based provenance model.

Later, U_1 decides to manually edit (*ed*) the value of source *Eugene*, modifying the *Final page* to *235*. This causes a new record *[id: 3, op: ed, origin source: null, target source: Eugene, key: Paper[Title=PrInt], attribute: Final page, origin value: 235, target value: 234, timestamp: 14:45:03_03/18;2014]* to be appended to repository R_1. Note that we may consider the repository is now *inconsistent*, given that there are two distinct values being written on *Final page* of source *Eugene*. Since the repository reflects the user's decisions over time, its last operation is considered to contain the correct value, and previous decisions can be reapplied by propagating the update to both sources *Bruce* and *Anne*. These actions are reflected in the repository by rearranging and updating it, as depicted in Figure 2. □

user	id	op	origin source	target source	key	attribute	origin value	target value	timestamp
U1	3	ed	null	Eugene	Paper [Title=PrInt]	Final page	235	234	14:45:03_ 03/18/2014
	1	cp	Eugene	Bruce	Paper [Title=PrInt]	Final page	235	1365	08:28:43_ 03/18/2014
	2	cp	Bruce	Anne	Paper [Title=PrInt]	Final page	235	1352	08:29:01_ 03/18/2014

Figure 2: Consistent view of repository R_1 after source *Eugene* has been updated.

Now consider a set of users $U_1, \ldots, U_n$, each of them importing the same set of sources and solving data conflicts independently. Since they all agree to collaborate, they exchange their updates by exporting their repositories $R_1, \ldots, R_n$. Thus, each user U_i has now access not only to his/her own repository R_i, but can import into R_i the updates of all collaborators. They can all benefit by applying the imported updates on their own copies of the sources, and thus reducing their workload in the integration process. However, this can be done either if decisions made elsewhere do not conflict with their own, or among collaborators. If this is not the case, each of them has to decide how to solve the conflict.

Example 2. Consider again user U_1 and his/her repository after importing collaborators' repositories $U_2, \ldots, U_6$, depicted in Figure 3. It shows how each user decided on conflicts over attribute *Final page* of *Paper[Title=PrInt]*. Observe that a time-based conflict resolution among multiple users working independently may not be adequate in this context. In our running example, if the latest decision were considered to be the correct one, we would pick U_2's, since the records timestamps are the most recent. U_2 considered the value provided by *Bruce* to be the correct one, and thus this decision would also be propagated to U_1's copies of the sources. However, this may not be the intended action for user U_1, who had locally decided that source *Eugene* should prevail over *Bruce*. □

user	id	op	origin source	target source	key	attribute	origin value	target value	timestamp
U1	1	cp	Eugene	Bruce	Paper [Title=PrInt]	Final page	234	1365	08:28:43_ 03/18/2014
	2	cp	Bruce	Anne	Paper [Title=PrInt]	Final page	234	1352	08:29:01_ 03/18/2014
	3	cp	Bruce	Anne	Paper [Title=PrInt]	Year	2013	2012	08:29:17_ 03/18/2014
U2	1	cp	Bruce	Daniel	Paper [Title=PrInt]	Final page	1365	1358	13:41:47_ 03/18/2014
U3	1	ed	null	Daniel	Paper [Title=PrInt]	Final page	1357	1358	12:24:55_ 03/18/2014
	2	cp	Daniel	Carl	Paper [Title=PrInt]	Final page	1357	1352	12:32:47_ 03/18/2014
U4	1	cp	Daniel	Franklin	Paper [Title=PrInt]	Final page	1358	422	10:13:45_ 03/18/2014
	2	cp	Daniel	Gary	Paper [Title=PrInt]	Final page	1358	205	10:14:21_ 03/18/2014
	3	cp	Franklin	Hugo	Paper [Title=PrInt]	Final page	1358	335	10:15:42_ 03/18/2014
U5	1	cp	Bruce	Gary	Paper [Title=PrInt]	Final page	1365	205	09:45:57_ 03/18/2014
	2	cp	Gary	Ivan	Paper [Title=PrInt]	Final page	1365	722	09:50:02_ 03/18/2014
U6	1	cp	Bruce	Hugo	Paper [Title=PrInt]	Initial page	1353	323	07:25:56_ 03/18/2014

Figure 3: Repository R_1 with all users' decisions.

1.2 Contributions

In a multiuser reconciliation process, policies, such as based on voting or for giving priority to local decisions, have to be introduced. More important, it is necessary to have a generic model that can support different scenarios, according to the intended result among collaborative users. That is, they may want to collectivity generate of a single integrated view to all of them, or individually create distinct local views.

In this paper, we tackle these challenges, and make the following contributions:

- We introduce AcCORD, an asynchronous model of collaborative data reconciliation in which users' updates are stored in logs, called repositories. Repositories keep data provenance, that is, the operations applied to the data sources that led to the current state of his/her local database.
- We propose policies for solving possible conflicts resulting from an asynchronous collaborative mulituser reconciliation process. These policies can target applications that generate an integrated view or distinct local views as a result of the collaborative work.

The paper is organized as follows. Section 2 reviews related work, Section 3 introduces our proposed model for collaborative work, Section 4 describes the characteristics of the operations stored in the repository, Section 5 details the proposed policies for multiusers reconciliation, Section 6 addresses the experimental tests, and Section 7 concludes the paper.

2 RELATED WORK

Several techniques focusing on data sharing and data integration or asynchronous reconciliation, have been proposed in the literature (see for example the surveys in (Saito and Shapiro, 2005; Halevy et al., 2006)). In this section, we focus on the conflict resolution problem on multiuser level and the definition of a generic model that allows collaborative reconciliation in which users may be allowed to agree or disagree on their updates. We present the related work in two groups: (i) related work aimed at multiuser reconciliation in the database area; and (ii) related work aimed at asynchronous reconciliation in the distributed system area.

Orchestra (Taylor and Ives, 2006; Green et al., 2007; Ives et al., 2008) is a work in first group. It is a collaborative peer-to-peer system that uses provenance to share data. It is similar to AcCORD in that each user controls a local instance of the database and can work independently for a certain period. It also proposes a policy to reconcile conflicts among imported data. This policy is based on trust among data providers. It specifies that local conflicting operations have higher priority than imported ones, but does not specify how the providers' trust are computed. The main differences between Orchestra and AcCORD is that the former only allows collaborative users to share their updates. The single proposed policy does not necessarity make all database instances converge to a consistent state, and it does not create a global integrated view of the data. Thus, it does not support data integration, as AcCORD does. Further, Orchestra proposes only one policy to solve data conflicts. In this paper, we introduce different reconciliation policies, and AcCORD can support other policies as well. Thus, AcCORD is more flexible, and can be applied to different collaborative reconciliation scenarios.

Youtopia (Kot and Koch, 2009) is a system for collaborative integration of relational data. Unlike our work, Youtopia manages conflicts among updates from multiple users in a synchronous mode through serializability and concurrency control techniques. Also, Youtopia neither performs a provenance-based integration, nor introduces support to different reconciliation policies, as we propose in this work. Furthermore, both Orchestra and Youtopia, need the user's interference during the reconciliation process to decide data conflicts not solved by their policies, unlike AcCORD which applies the proposed policies to resolve all conflicts automatically.

PrInt (Tomazela et al., 2013) is a provenance model that supports data integration conflict resolution at instance level. Its main objective is the strict reproduction of user's decisions among distinct integration processes. These decisions are represented as operations, which contain provenance data. PrInt stores the operations in a repository and introduces the concept of repository consistency for managing conflicts in a single user level. Thus, it aims at data integration processes performed by a single user, as opposed to AcCORD, which focuses on several users performing a collaborative reconciliation process. Here, we borrow some ideas from PrInt, such as the notions of operations, repository and transitive and overlapping operations. However, we extend these principles to support multiple users working asynchronously. Also, we define a general model and different policies to manage inconsistencies in this new context.

We now move on to discuss related work in the second group, i. e. asynchronous collaboration in the distributed system area. We analyze existing techniques in a tandem, as they face the same limitations when compared to our work.

IceCube (Kermarrec et al., 2001) does operation-based reconciliation. It captures the static and dynamic reconciliation constraints between all pairs of operations, proposes schedules that satisfy the static constraints, and validates them against dynamic constraints. IceCube does not present algorithms to solve conflicts, but tries to avoid them by scheduling operations, in a particular order.

Bayou (Edwards et al., 1997) is another operation-based reconciler designed to support the construction of asynchronous collaborative applications. It provides a replicated, shared and weakly-consistent database. To this end, it supports conflict detection and resolution methods based on client-provided merge procedures, and can rollback the effects of previously executed write operations and reapply them according to a global serialization order. Bayou is not as flexible as IceCube, but it addresses distribution issues and provides a distributed infrastructure for collaboration.

Harmony (Pierce et al., 2004) is a state-based reconciler for structured data, which considers only the current version of the replicas, together with a version of the last state they had in common. Harmony deals with conflicts by allowing replicas to remain different after reconciliation. By contrast, operation-based reconcilers, as Bayou and IceCube, try to construct a common sequence of operations. Therefore, they deal with conflicts by omitting conflicting operations. This approach always achieves a common final state, but sometimes it backs out user changes.

Although IceCube, Bayou and Harmony focus on asynchronous reconciliation and use logs to support the process or allow copies to remain inconsistent for a given period, which are features similar to those adopted by AcCORD, the main objective of these techniques differs from ours in the following way: they present algorithms to find schedules that minimize the number of conflicts among updates, or let the conflict resolution up to the application. On the other hand, we show in this paper how to reach reconciliation when multiusers wish to share their updates and detail how conflict among operations can be solved.

3 AcCORD

We introduce AcCORD, an asynchronous collaborative data reconciliation model. The scenario considered by AcCORD is depicted in Figure 4. In this scenario, collaborative users $U_1, \ldots, U_n$ reconcile data imported from several data sources. Each user decides how to solve attribute value conflicts on corresponding objects according to his/her point of view (Figure 4(a)). To this end, each user may apply tools for helping the integration process and for storing updates in a operation-based provenance repository. As a result, each user produces a set of local views, which are integrated versions of the sources, and a repository, which contains provenance data composed of a sequence of operations that reflect his/her updates. As several collaborative users work on the same sources, they maintain their own local views, and the original sources are not updated.

After finishing the integration and provenance process, a given user, say user U_1, decides to reconcile his/her updates with the updates of the other collaborative users (Figure 4(b)), i.e. to perform a reconciliation process considering multiuser updates, which is the focus of our work. Note that we denote the process performed by a single user as an *integration* process, and the process performed by several users as a

reconciliation process, since the former always generate a single integrated view and the latter can generate the same integrated view to all users, or distinct local views for each of them. The proposed reconciliation process takes as input several sets of provenance data, each one representing decisions that a single user takes to solve inconsistencies among sources, and produces as output a set of provenance data representing the reconciliation of the updates of all users. We call each input as "provenance data on single user level", and the output as "provenance data on multiuser level".

The provenance data on multiuser level produced by the reconciliation process depends on the data reconciliation policy applied to solve conflicts among users' updates. It is used to update the repository of operations of the user who is executing the reconciliation, i.e. user U_1 in our running example. In this paper, we propose different policies to reconcile data in a multiuser level, which are detailed in Section 5. The collaborators may choose the same data reconciliation policy or different reconciliation policies, depending on intended final result. If the users are working collaboratively towards a unique integrated view of the data then they should all share the same updates, and thus apply the same data reconciliation policy. On the other hand, when users want to share their particular updates, but wish to maintain their own view of the reconciled data, each of them may apply distinct data reconciliation policies. Additionally, a given user may decide not to perform the multiuser reconciliation, working only in the single user level.

After updating the repository of operations, the user may apply an operation-based provenance model to update his/her local views according to the new provenance data stored in the repository (Figure 4(c)).

Before introducing our reconciliation policies, we detail in Section 4 some characteristics of the repository and restrictions that apply over its operations.

4 OPERATIONS REPOSITORY

AcCORD is based on a repository of operations storing provenance data. Although other models can be used, here we assume that repositories are sequences of records containing the following attributes:

- **op:** operation that reflects a user decision in the reconciliation process;
- **originSource:** source that provides the correct value of an entity's attribute.
- **targetSource:** source on which the attribute value was updated by *op*;
- **key:** key value that identifies an entity;
- **attribute:** attribute name on which *op* is performed;
- **originValue:** *originSource*'s attribute value;
- **targetValue:** *targetSource*'s attribute value before being overwritten by *originValue*;
- **timestamp:** *op* execution time.

Operations in a repository may contain dependencies in a single user level or in multiuser level. Definition 1 details dependencies among operations on a single user level. Here, we denote by $r(a)$ the value of attribute a in a record r.

Definition 1 (Dependencies among Operations)**.** *An operation b is dependent on an operation a when they are performed by the same user, a occurred before b, they involve the same attribute of the same object and the target of a is equal to the origin of b. That is,*

- *a(**targetSource**, key, attribute) = b(**originSource**, key, attribute) and a(timestamp) < b(timestamp).*

Also, an operation c is dependent on an operation a if there is an operation b that is dependent on a and c is dependent on b. That is,

- *a(**targetSource**, key, attribute) = b(**originSource**, key, attribute), a(timestamp) < b(timestamp), and*
- *b(**targetSource**, key, attribute) = c(**originSource**, key, attribute), and b(timestamp) < c(timestamp).*

Intuitively, an operation b depends on a if the value written by a is later used by b, propagating it to other sources.

Example 3. In Figure 3, operation *2* performed by U_5 depends on operation *1* performed by the same user. If U_5 performs another operation *3* with the values: *[id: 3, op: cp, origin source: Ivan, target source: John, key: Paper[Title=PrInt], attribute: Final page, origin value: 1365, target value: 1366, timestamp: 09:53:26_03/18/2014]*, this new operation also depends (by transitivity) on operation *1*. □

Definition 2 details conflicting operations on multiuser level.

Definition 2 (Conflicting Operations)**.** *Two operations a and b conflict when they are performed on the same attribute of the same object, and if: (i) the target source of a is equal to the target source of b, or (ii) the target source of a is equal to the origin source of b. That is,*

1. *a(**targetSource**, key, attribute) = b(**targetSource**, key, attribute), or*
2. *a(**targetSource**, key, attribute) = b(**originSource**, key, attribute).*

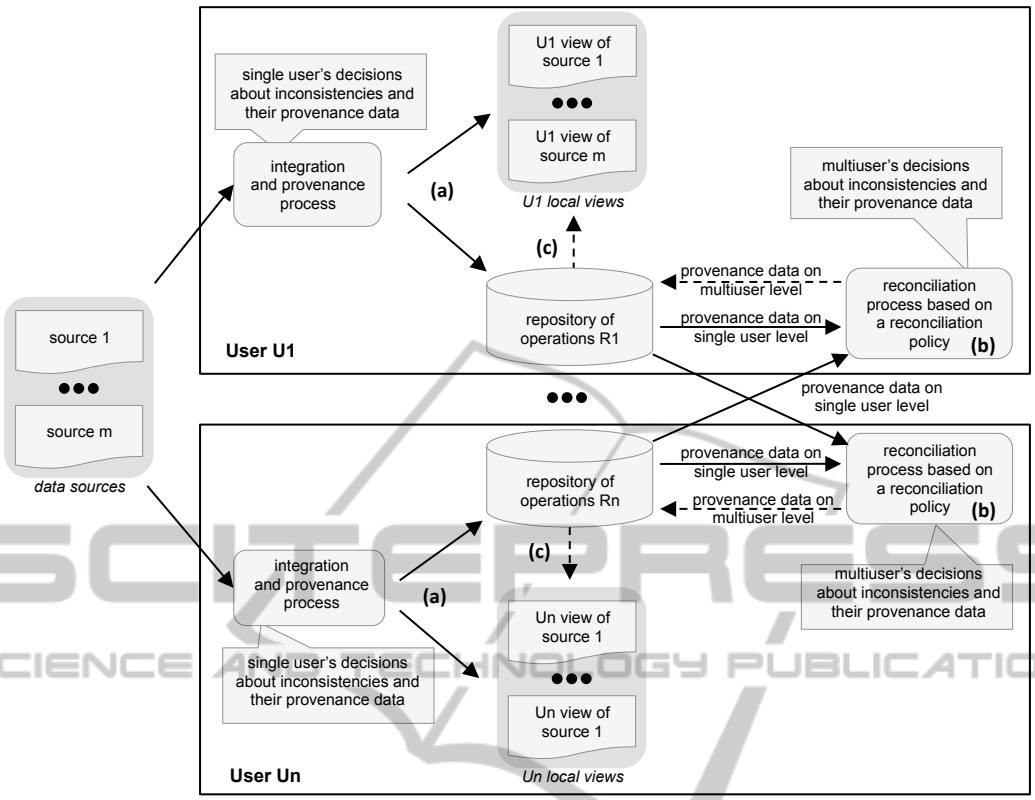

Figure 4: The proposed AcCORD model.

Example 4. Consider again the example in Figure 3. Operation *1* of U_2 conflicts with operation *1* of U_1 because *origin source* (*Bruce*) in the operation of U_2 is equal to *target source* of U_1's operation. Intuitively, while U_2 considers *Bruce* to have the correct value to write on *Daniel*'s source, U_1 overwrites *Bruce*'s value with *Eugene*'s. Also, operation *1* of U_2 conflicts with operation *1* of U_3 because the *target source* in both operations are the same. Here, the conflict results from the fact that U_2 and U_3 are writing distinct values on the same data item. □

5 PROPOSED POLICIES

We propose four policies to reconcile data on a multiuser level. As described in Section 3, each policy takes as input several sets of "provenance data on single user level" generated by users $U_1, ..., U_n$ and produces as output a set of "provenance data on multiuser level", which is used to update the repository of operations of the user that required the multiuser reconciliation process.

The general procedure for sharing updates is as follows. First, a user imports a set of "provenance data on single user level" and cluster their operations according to *key* and *attribute*. Then, clusters are analyzed in order to detect inconsistencies. If the operations in a cluster conflict (Definition 2), these conflicting operations and their dependent operations (Definition 1) may be maintained in the cluster, removed from the cluster or updated, according to the chosen policy. Otherwise, if the operations in a cluster do not conflict, they remain in the cluster. After this process, the operations that remained in the clusters represent the "provenance data on multiuser level", and now composes the new version of the user's repository of operations.

The policies differ from each other with respect to the way they manage conflicting operations and their dependent operations. We introduce these policies and discuss their features and applicability in the following sections.

5.1 The LocalView Policy

The *LocalView Policy* is characterized by giving priority to updates made locally over those made elsewhere. Thus, whenever imported updates conflict with his/her own, they are simply ignored.

Consider, for example, that U_1 is reconciling

his/her repository with those imported from users $U_2, \ldots U_6$. Conflicts in R_1 after importing repositories $R_2, \ldots, R_6$ are managed as follows. For each cluster, if a given operation p performed by $U_2, ..., U_n$ conflicts with an operation o performed by U_1, the policy removes p from the cluster. Also, the policy removes all dependent operations on p from the cluster.

Example 5. Consider repository R_1 depicted in Figure 3. Figure 5 shows R_1 after U_1 applies the multiuser reconciliation process using the *LocalView Policy*. R_1 does not contain operations from users U_2, U_3, U_4 and U_5, as they conflict and thus are removed from the cluster. Observe that U_6's operation is kept in the repository since it is applied to a different attribute, *Initial page*. □

user	id	op	origin source	target source	key	attribute	origin value	target value	timestamp
U1	1	cp	Eugene	Bruce	Paper [Title=PrInt]	Final page	234	1365	08:28:43_ 03/18/2014
	2	cp	Bruce	Anne	Paper [Title=PrInt]	Final page	234	1352	08:29:01_ 03/18/2014
	3	cp	Bruce	Anne	Paper [Title=PrInt]	Year	2013	2012	08:29:17_ 03/18/2014
U6	1	cp	Bruce	Hugo	Paper [Title=PrInt]	Initial page	1353	323	07:25:56_ 03/18/2014

Figure 5: Repository R_1 after the reconciliation process using the *LocalView Policy*.

The operations represent decisions that the user takes to solve inconsistencies, the *LocalView Policy* should be used whenever he/she believes that local decisions are more accurate than decisions made by others. Also, this policy guarantees that the user decisions will never be overwritten by other users decisions. Thus, when collaborators decide to apply the *LocalView Policy*, the local views for each them will probably be different, as they might have taken different decisions to solve the same conflict. This policy is adequate for data sharing collaborative works.

5.2 The RemoveConflicts Policy

The *RemoveConflicts Policy* is characterized by not prioritizing conflicting operations, independently of the user that required the reconciliation process.

It manages conflicting operations and their dependent operations as follows. For each cluster, the policy removes all the conflicting operations and their dependent operations.

Example 6. Consider repository R_1 depicted in Figure 3. Figure 6 shows repository R_1 after U_1 required the multiuser reconciliation process using the *RemoveConflicts Policy*. R_1 only contains operations that do not conflict. □

The *RemoveConflicts Policy* removes all conflicting operations and their dependent operations when-

user	id	op	origin source	target source	key	attribute	origin value	target value	timestamp
U1	3	cp	Bruce	Anne	Paper [Title=PrInt]	Year	2013	2012	08:29:17_ 03/18/2014
U6	1	cp	Bruce	Hugo	Paper [Title=PrInt]	Initial page	1353	323	07:25:56_ 03/18/2014

Figure 6: Repository R_1 after the reconciliation process using the *RemoveConflicts Policy*.

ever it is not able to decide which conflicting operation is correct. These operations should be returned to the collaborative users so that they can discuss about the potential inconsistencies and agree on the integrated values for the imported data. Provenance data related to the integrated values should be appended to U_1's repository of operations. Thus, this policy should be used to postpone the decision on which is the value to be kept in the local database, aiming at generating a single global integrated view to all users. Also, when the collaborative users decide to apply the *RemoveConflicts Policy*, the local views of each user will eventually be the same.

5.3 The Timestamp Policy

The *Timestamp Policy* is characterized by prioritizing the temporal order of conflicting operations.

It manages conflicting operations and their dependent operations as follows. For each cluster, the policy chooses to keep in the cluster the most recent conflicting operation, i.e. the conflicting operation o that has the largest timestamp. The policy also keeps in the cluster all dependent operations on o. Each remaining conflicting operations p in the cluster are managed according to its conflict type. If the conflict between o and p occurs because the *origin source* of p is equal to the *target source* of o, the value of *origin value* of p and its dependent operations are changed to the same value of *origin value* of o. These updated operations are also moved to the end of all clusters. Otherwise, if the conflict between o and p occurs because the *origin source* of o is equal to the *target source* of p, or, the *target source* in both operations are equal, then operation p is removed from the cluster, since it cannot be redone. As for the dependent operations on p, they are updated such that their *originValue* are set to the same value of *origin source* of o, and they are moved to the end of all clusters.

Example 7. Consider repository R_1 depicted in Figure 3. Figure 7 shows repository R_1 after U_1 required the multiuser reconciliation process using the *Timestamp Policy*. The operation with the largest timestamp is that performed by U_2; therefore, operation *1* of U_2 is kept in the repository. As for the other users, operation *1* of U_1 conflicts with operation *1* of U_2 because *target source* in the operation of U_1 is equal to *ori-*

gin source in U_2's operation. Thus, operation *1* of U_1 is removed from the repository and its dependent operation *2* is updated and moved to the end of the repository. Operation *3* of U_1 does not conflict with any other, and remains in the repository. Also, operation *1* of U_3 conflicts with operation *1* of U_2 because the values of *target source* in both operations are the same. Thus, operation *1* of U_3 is removed and its dependent operation *2* is updated and moved to the end of the repository. Further, operations *1* and *2* of U_4 conflicts with operation *1* of U_2 because *origin source* in the U_4's operations are equal to *target source* in the operation of U_2. Thus, operations *1*, *2* and the dependent operation *3* of U_4 are updated and moved to the end of the repository. Finally, operations of U_5 do not conflict with operation *1* of U_2 and operation of U_6 does not conflict with any other. Operations of both users compose the repository R_1. □

user	id	op	origin source	target source	key	attribute	origin value	target value	timestamp
U1	3	cp	Bruce	Anne	Paper [Title=PrInt]	Year	2013	2012	08:29:17_03/18/2014
U2	1	cp	Bruce	Daniel	Paper [Title=PrInt]	Final page	1365	1358	13:41:47_03/18/2014
U5	1	cp	Bruce	Gary	Paper [Title=PrInt]	Final page	1365	205	09:45:57_03/18/2014
	2	cp	Gary	Ivan	Paper [Title=PrInt]	Final page	1365	722	09:50:02_03/18/2014
U6	1	cp	Bruce	Hugo	Paper [Title=PrInt]	Initial page	1353	323	07:25:56_03/18/2014
U1	2	cp	Bruce	Anne	Paper [Title=PrInt]	Final page	1365	1352	08:29:01_03/18/2014
U3	2	cp	Daniel	Carl	Paper [Title=PrInt]	Final page	1365	1352	12:32:47_03/18/2014
U4	1	cp	Daniel	Franklin	Paper [Title=PrInt]	Final page	1365	422	10:13:45_03/18/2014
	2	cp	Daniel	Gary	Paper [Title=PrInt]	Final page	1365	205	10:14:21_03/18/2014
	3	cp	Franklin	Hugo	Paper [Title=PrInt]	Final page	1365	335	10:15:42_03/18/2014

Figure 7: Repository R_1 after the reconciliation process using the *Timestamp Policy*.

The *Timestamp Policy* maintains the most recent update made by any user, i.e. the conflict operation that has the most recent timestamp. It is motivated by the same principle adopted by several commercial and non-commercial systems, such as Google Docs[1] and Wikipedia[2]. This policy should be used to generate a single global integrated view. Also, when the collaborative users decide to apply the *Timestamp Policy*, the local views of each user will be eventually the same.

5.4 The Voting Policy

The *Voting Policy* is characterized by maintaining the conflicting operation that reflects the majority of decisions.

It manages conflicting operations and their dependent operations as follows. For each cluster, the policy counts how many users have chosen the same *origin value* for a conflicting attribute. The operations whose *origin value* was chosen by the majority represent the winning operations and are kept in the cluster. All operations dependent on them are also kept in the cluster. Similar to the *Timestamp Policy*, each remaining conflicting operation p in the cluster is managed according to its conflict type. If the conflict between p and any operation belonging to the winner group, say operation o, is because the *origin source* of p is equal to the *target source* of o, the value of *origin value* of p and its dependent operations are changed to the same value of *origin value* of o. These updated operations are also moved to the end of all clusters. Otherwise, if the conflict between p and any operation belonging to the winner group occurs because the *origin source* of o is equal to the *target source* of p, or, the *target source* in both operations are equal, operation p is removed from the cluster, since it is not possible to redo the operation. As for the dependent operations on p, they are updated such that their *origin value* are set to the same value of *origin source* of any operation from the winner group, and moved to the end of all clusters. If there is no winner, all conflicting operations and their dependent operations are removed from the cluster.

Example 8. Consider repository R_1 depicted in Figure 3. Figure 8 shows repository R_1 after U_1 required the multiuser reconciliation process using the *Voting Policy*. The winning operations are operation *1* of U_2 and operation *1* of U_5. These two operations, together with operation *2* of U_5, which is dependent of operation *1* of U_5, are kept in the repository. R_1 does not contain operation *1* of U_1 as it was removed from the repository because it conflicts with operation *1* of U_2 and with operation *1* of U_5 (its *target source* is equal the *origin source* in both operations). On the other hand, the dependent operation *2* of U_1 is updated and moved to the end of the repository. Operation *3* of U_1 is maintained in the repository because it is not a conflicting operation. Also, operation *1* of U_3 was removed from the repository because it conflicts with operation *1* of U_2 (its *target source* are equal). Its dependent operation is updated and moved to the end of the repository. Further, operation *1* of U_4 conflicts with operation *1* of U_2. So, this operation and its dependent operation *3* are updated and moved to the end of the repository. Operation *2* of U_4 conflicts with operation *1* of U_2 (its *origin source* is equal the *target source* in operation *1* of U_2), thus its *origin value* is set to the same value of *origin value* of U_2's operation *1*, and it is moved to the end of the repository. □

The *Voting Policy* is motivated by the fact that if the majority of users trust on a determined attribute

[1]Google Inc., https://www.google.com

[2]Wikimedia, http://www.wikipedia.org/

user	id	op	origin source	target source	key	attribute	origin value	target value	timestamp
U1	3	cp	Bruce	Anne	Paper [Title=PrInt]	Year	2013	2012	08:29:17_ 03/18/2014
U2	1	cp	Bruce	Daniel	Paper [Title=PrInt]	Final page	1365	1358	13:41:47_ 03/18/2014
U5	1	cp	Bruce	Gary	Paper [Title=PrInt]	Final page	1365	205	09:45:57_ 03/18/2014
	2	cp	Gary	Ivan	Paper [Title=PrInt]	Final page	1365	722	09:50:02_ 03/18/2014
U6	1	cp	Bruce	Hugo	Paper [Title=PrInt]	Initial page	1353	323	07:25:56_ 03/18/2014
U1	2	cp	Bruce	Anne	Paper [Title=PrInt]	Final page	1365	1352	08:29:01_ 03/18/2014
U3	2	cp	Daniel	Carl	Paper [Title=PrInt]	Final page	1365	1352	12:32:47_ 03/18/2014
U4	1	cp	Daniel	Franklin	Paper [Title=PrInt]	Final page	1365	422	10:13:45_ 03/18/2014
	3	cp	Franklin	Hugo	Paper [Title=PrInt]	Final page	1365	335	10:15:42_ 03/18/2014
	2	cp	Daniel	Gary	Paper [Title=PrInt]	Final page	1365	205	10:14:21_ 03/18/2014

Figure 8: Repository R_1 after the reconciliation process using the *Voting Policy*.

value, then this value seems to be the correct one, and should be adopted as such. When it is not possible to decide which is the correct value of an attribute, then the conflicting operations removed from the clusters should be returned to the collaborative users, similarly to the discussion in Section 5.3. Finally, the use of the *Voting Policy* by all collaborative users represents an integration scenario, in which all local views will eventually be the same.

6 PERFORMANCE EVALUATION

AcCORD and its reconciliation policies were validated through performance tests using real data extracted from 4 curricula of researchers who work for the Department of Computer Science of the University of São Paulo at São Carlos. These curricula were chosen because they had a large number of publications in common.

The size of the data sources amounts to 3.11 MB. The largest curriculum had 1.20 MB and contained 25 objects, while the smallest had 238 KB and contained 3 objects. Each object had the following attributes: *title*, *year*, *venue*, *local of publication*, *language*, *media*, *type of publication*, *pages*, *ISSN*, *volume*, *number*, 6 attributes for *keywords*, and the *authors* of the publication, which were composed of the attributes *name*, *citation name* and *citation order*.

To aid the user in the integration and provenance process, we used the original implementation of PrInt described in Section 2. Each user interacted with PrInt to integrate imported data from the curricula and to generate his/her own repository with *copy*, *edit*, *remove* and *insert* operations, which reflected the user's decisions. The attributes of provenance stored in each repository followed those defined in Section 4. The largest repository had 390 KB and contained 749 operations, while the smallest repository had 40 KB and contained 79 operations.

The prototype of AcCORD was implemented in C++ using Qt version 4.6.2 and compiled with GNU g++ version 4.4.3. The experiments were conducted on a computer with an Intel Core 2 Duo 2.93 GHz processor and 4 GB of main memory.

The experiments consider a collaborative reconciliation work performed by several users, and that user U_1 imports the sets of "provenance data on single user level" from other users, and applies a reconciliation policy. We report the effects of managing conflicting operations to the repository R_1, which is the repository of U_1. The goal of the experiments is to investigate the effects of adopting of each policy regarding: (i) the total number of operations removed, which may cause loss of user decisions; (ii) the effect of increasing the number of collaborators (iii) the effect of each policy in removing operations performed locally, reflecting losses of the user's own decisions.

In Section 6.1 we report the results when U_1 applies the reconciliation process for the first time. Section 6.2presents the results when U_1 applies the reconciliation process several times consecutively.

6.1 First Reconciliation Process

In this experiment, we varied the number of collaborative users from 2 to 16. For each scenario, we considered that U_1 required the reconciliation process for the first time. Thus, repository R_1 always contains only the operations of U_1's first integration process, i.e. R_1 does not contain operations related to any previous reconciliation process.

Figure 9 depicts: (i) the *total number of operations* stored in R_1 after U_1 imported all sets of provenance data, but before applying a reconciliation policy; and (ii) the number of operations in R_1 after executing the reconciliation process, whith each of the proposed policies. The results demonstrated that the *RemoveConflicts Policy* and the *Voting Policy* removed more operations than the other policies. Also, they had the same behavior. The result obtained for the *RemoveConflicts Policy* was expected, as it removes all conflicting operations, but the result obtained for the *Voting Policy* was unexpected. However, by analyzing the sources, we identified that there were no winners for any set of conflicting operations. In such a situation, the *Voting Policy* has the same behavior as the *RemoveConflicts Policy*.

Figure 10 depicts: (i) the *initial number of operations* in R_1 after U_1 imported all sets of provenance data, but before U_1 applied a reconciliation policy; and (ii) the number of operations from U_1 removed

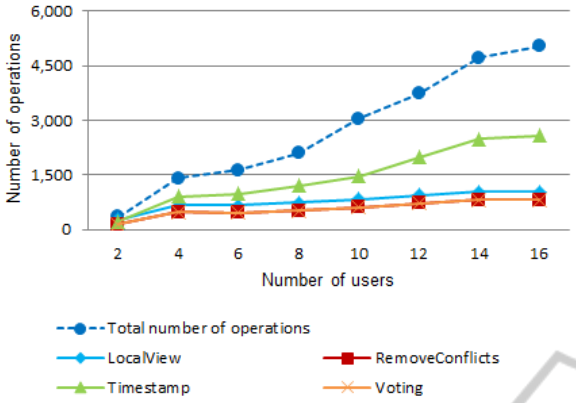

Figure 9: Number of operations of all users considering each proposed policy.

from R_1 after the reconciliation process. The goal of this is to show the loss of changes made by the local user U_1. As expected, the *LocalView Policy* did not remove any operation. The *Timestamp Policy* removed an average number of U_1's operations, as this policy is able to update and keep some conflicting operations and the ones dependent on them. For the some reason reported in previous experiments, the *RemoveConflicts Policy* and the *Voting Policy* had the same behavior, removing a larger number of U_1's operations.

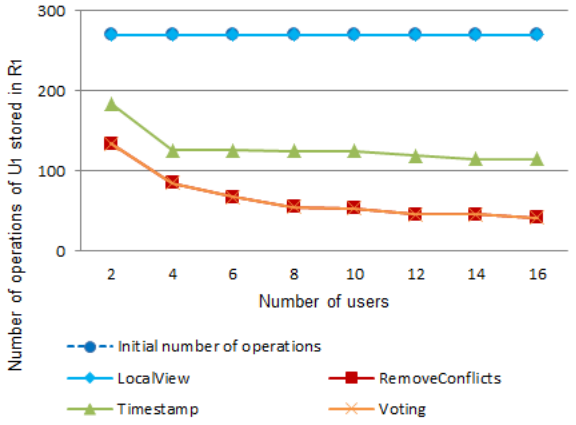

Figure 10: Number of operations of user U_1 applying each proposed policy.

6.2 Several Consecutive Reconciliation Processes

For all tests described in this section, we fixed the number of collaborative users to 4. We considered that U_1 executed the reconciliation process 5 times consecutively. Thus, in reconciliation process 1, at time t_1, R_1 contains operations of repositories $R_1, \ldots, R_4$; in reconciliation process 2 at time t_2, R_1 contains operations of repositories $R_1, \ldots, R_4$, such that these repositories also contain the new operations generated from time t_1 to t_2; and so on.

Figure 11 depicts the number of operations stored in R_1 before and after applying each policy. The results show that the *Timestamp Policy* provides the best results, as it removes the smallest number of conflicting operations after each reconciliation process. The *LocalView Policy* removed an average number of operations, while the *RemoveConflicts Policy* and the *Voting Policy* have the same behaviour and remove more operations that the other policies. The results also show that for all policies, even for the *RemoveConflicts Policy* and the *Voting Policy*, the size of the repository grows after the execution of each new reconciliation process. For example, consider the *Timestamp Policy*. Before the first reconciliation process there were 1417 operations, and at the end of the fifth process, the repository contained 2445 operations. The difference represents the contribution other users made to U_1's work in the reconciliation process.

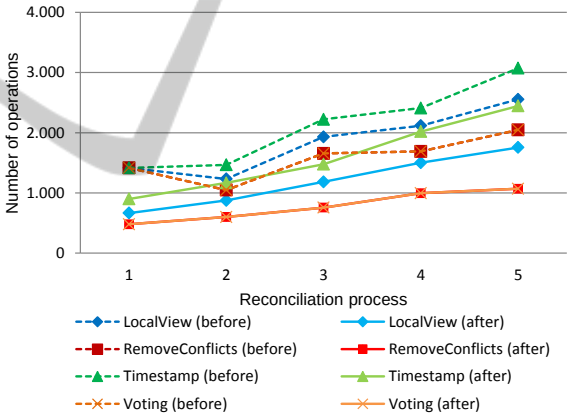

Figure 11: Number of operations stored in R_1 before and after each reconciliation process by each proposed policy.

We next report the effect of each policy, considering only the local operations of U_1. The results, depicted in Figure 12, are similar to those discussed for Figure 10: the *LocalView Policy* did not remove any operation, the *Timestamp Policy* removed an average number of operations, and the *RemoveConflicts Policy* and the *Voting Policy* removed the largest number of operations.

As in all previous tests, the *RemoveConflicts Policy* and the *Voting Policy* showed the same behavior. Thus, in order to determine how they differ, we considered a new scenario with only 3 collaborative users. Figure 13 depicts the number of operations of U_1 stored in R_1 before and after applying each policy after 5 consecutive reconciliation processes. The re-

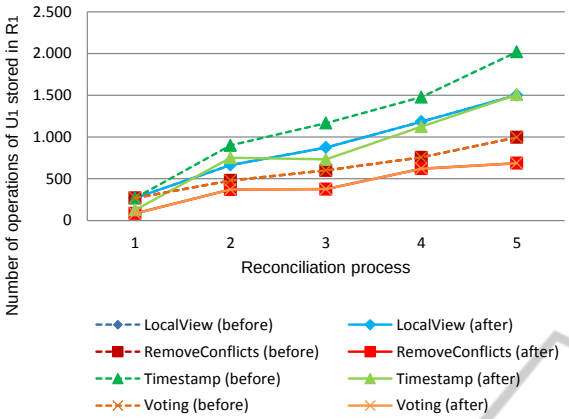

Figure 12: Number of operations of U_1 applying each proposed policy.

sults show fewer number of removals for the *Voting Policy*. The first reconciliation process starts with operations 1338 for both policies, while the fifth strats with 1788 for the *RemoveConflicts Policy* and 2286 for the *Voting Policy*. At the end of the fifth process, R_1 contained 1084 operations applying the *RemoveConflicts Policy* and 1909 applying the *Voting Policy*. This result was achieved because there were winning operations among conflicting operations considering the *Voting Policy*.

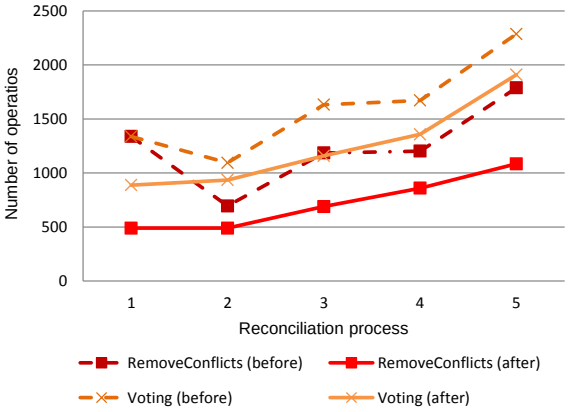

Figure 13: Number of operations stored in R_1 before and after each reconciliation process by RemoveConflicts and Voting policies.

7 CONCLUSIONS

In this paper, we focus on the challenge of reconciling data when multiple users work asynchronously over local copies of the same imported data. Our contributions are twofold. First, we propose AcCORD, an asynchronous collaborative data reconciliation model, and second, we introduce reconciliation policies for solving conflicts resulting from multiusers' updates. We base our proposals on the following major concepts: (i) data provenance, which represents decisions that the users take to solve conflicts on imported data; (ii) repository, which works as a log and stores data provenance as operations; and (iii) flexibility, by supporting applications in which all users are required to agree on the integration process for providing a single consistent view to all of them, as well as applications that allow users to disagree on the correct data value, but promote collaboration by sharing updates.

AcCORD can be applied to several kinds of applications characterized by a high degree of independence, in which users work autonomously, and are loosely connected to each other at any given time. Examples of such applications include e-health systems, curation in bioinformatics and data sharing in bibliographical data. Another advantage of AcCORD refers to its extensibility, since it does not consider any particular integration or operation based provenance model. Thus, users are free to choose tools at their convenience, and still be able to apply AcCORD to collaboration among them.

Our work highlights the need for different reconciliation policies to be available when users perform asynchronous collaborative data reconciliation. The proposed policies target either applications that allow an integrated view or distinct local views of the collaborative work. Experiments have been conducted to analyze the proposed policies considering both a single reconciliation process and several consecutive processes. The results obtained showed the main characteristics of the policies in terms of managing conflicting users' decisions.

The *RemoveConflicts Policy* removed the largest number of operations, as expected. Because of the tests input data, the *Voting Policy* presented the same worst case behavior as the *RemoveConflicts Policy* in all experiments, except the last. In the last test, considering a scenario with 3 users, the *Voting Policy* managed to have winning operations, and showed better results in terms of number of removals, than the *RemoveConflicts Policy*. The *Voting Policy* removed from 30.7% to 58.1% less operations than the *RemoveConflicts Policy*. Also, The *Timestamp Policy* removed the smallest number of operations. The *LocalView Policy* showed average results, by removing operations in conflict with local operations instead of managing and reusing them.

The results presented in this paper show the efficiency of AcCORD, as they corroborate the hypothesis that collaborative data reconciliation can provide

gains for the local user, saving him/her from part of the work already done by other users. These gains exist even when the number of concurrent users on the same sources grow increasing the number of conflicts.

We are currently extending AcCORD with new reconciliation policies, such as a policy based on the confidence of sources and one based on the history of user's updates. We also plan to investigate how to improve the feedback given to the collaborative users when the reconciliation policies identify conflicting decisions.

ACKNOWLEDGEMENTS

This work has been supported by the following Brazilian research agencies: CAPES, CNPq, FAPESP, Fundao Araucria and FINEP.

REFERENCES

Bhattacharjee, A. and Jamil, H. (2012). A schema matching system for on-the-fly autonomous data integration. *International Journal of Information and Decision Sciences*, 4(2-3):167–181.

Cao, Y., Fan, W., and Yu, W. (2013). Determining the relative accuracy of attributes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 565–576.

Cheney, J., Chiticariu, L., and Tan, W.-C. (2009). Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474.

Edwards, W. K., Mynatt, E. D., Petersen, K., Spreitzer, M. J., Terry, D. B., and Theimer, M. M. (1997). Designing and implementing asynchronous collaborative applications with Bayou. In *Proceedings of the 10th Annual ACM Symposium on User Interface Software and Technology*, pages 119–128.

Green, T. J., Karvounarakis, G., Ives, Z. G., and Tannen, V. (2007). Update exchange with mappings and provenance. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 675–686.

Halevy, A. Y., Rajaraman, A., and Ordille, J. J. (2006). Data integration: The teenage years. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 9–16.

Hossain, M. S., Masud, M., Muhammad, G., Rawashdeh, M., and Hassan, M. M. (2014). Automated and user involved data synchronization in collaborative e-health environments. *Computer in Human Behavior*, 30:485–490.

Ives, Z. G., Green, T. J., Karvounarakis, G., Taylor, N. E., Tannen, V., Talukdar, P. P., Jacob, M., and Pereira, F. (2008). The orchestra collaborative data sharing system. *SIGMOD Record*, 37(3):26–32.

Kermarrec, A.-M., Rowstron, A., Shapiro, M., and Druschel, P. (2001). The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*, pages 210–218.

Köpcke, H., Thor, A., and Rahm, E. (2010). Evaluation of entity resolution approaches on real-world match problems. *PVLDB*, 3(1):484–493.

Kot, L. and Koch, C. (2009). Cooperative update exchange in the Youtopia system. *PVLDB*, 2(1):193–204.

Mahmood, T., Jami, S. I., Shaikh, Z. A., and Mughal, M. H. (2013). Toward the modeling of data provenance in scientific publications. *Computer Standards & Interfaces*, 35(1):6–29.

Nguyen, H.-Q., Taniar, D., Rahayu, J., and Nguyen, K. (2011). Double-layered schema integration of heterogeneous XML sources. *Journal of Systems and Software*, 84(1):63–76.

Pierce, B. C., Schmitt, A., and Greenwald, M. B. (2004). Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania.

Saito, Y. and Shapiro, M. (2005). Optimistic replication. *ACM Computing Surveys*, 37(1):42–81.

Taylor, N. E. and Ives, Z. G. (2006). Reconciling while tolerating disagreement in collaborative data sharing. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 13–24.

Tomazela, B., Hara, C. S., Ciferri, R. R., and Ciferri, C. D. A. (2013). Empowering integration processes with data provenance. *Data & Knowledge Engineering*, 86:102–123.