Inconsistency Resolution in Online Databases

Yannis Katsis[†], Alin Deutsch[†], Yannis Papakonstantinou[†], Vasilis Vassalos[‡]

[†]*CSE Department, UC San Diego*
{ikatsis, deutsch, yannis}@cs.ucsd.edu

[‡]*Athens University of Economics and Business*
vassalos@aueb.gr

***Abstract*— Shared online databases allow community members to collaboratively maintain knowledge. Collaborative editing though inevitably leads to inconsistencies as different members enter erroneous data or conflicting opinions. Ideally community members should be able to see and resolve these inconsistencies in a collaborative fashion. However most current online databases do not support inconsistency resolution. Instead they try to by-pass the problem by either ignoring inconsistencies and treating data as if they were not conflicting or by requiring inconsistencies to be resolved outside the system.**

To address this limitation, we propose Ricolla; an online database system that, by treating inconsistencies as first-class citizens, supports a natural workflow for the management of conflicting data. The system captures inconsistencies (so that community members can easily inspect them) and remains fully functional in their presence, thus enabling inconsistency resolution in an "as-you-go" fashion. Moreover it supports several schemes for the resolution of inconsistencies, allowing among others users to collaboratively resolve certain conflicts while disagreeing on others.

I. Introduction

The increasing need of communities to collaboratively maintain structured data led recently to the proliferation of web-applications that provide this functionality, referred to as *online databases*. However collaborative editing results in a major challenge: Since data originate from different users, they often *contradict* each other. These data conflicts usually arise either because of errors or due to differing opinions. Instances of the latter often appear in the sciences, where researchers have contradicting opinions, without being able to neither prove nor disprove them.

Current online databases typically avoid treating data conflicts by adopting one of two extremes:

On one hand, wiki-inspired online databases, such as Dabble DB, blist and Freebase *ignore the conflicts*. Being totally unconstrained, they allow users to enter any data even if they are contradicting. The advantage of this method is that it preserves all conflicting data for community members to see. However, it also severely restricts the utility of the data, as members are presented with conflicting information without an obvious way of finding inconsistencies and resolving them.

On the opposite end of the spectrum, traditional DBMS-driven online databases disallow data conflicts altogether. Enforcing a set of integrity constraints, they require a set of moderators to *resolve inconsistencies outside the system* and only insert data that satisfy the constraints. The obvious advantage is that community members see data that do not contradict each other. This approach however brings also two major drawbacks: First, it imposes on communities an artificial workflow in which all data have to be resolved before they can be used. This precludes them from employing a more natural workflow where they start using the (potentially inconsistent) data and only resolve conflicts in an incremental "as-you-go" fashion. Even more importantly however, it forces communities to agree on a single way to resolve the inconsistencies (that taken by the moderators), even when there is no such unanimously accepted way (as is the case in the presence of differing opinions discussed above).

It is thus obvious that current solutions simply overlook the needs of online communities for inconsistency management. Communities need a system that treats inconsistencies as first-class citizens and supports a natural workflow for their resolution. In this paper, we present **Ricolla** (**R**esolve **I**nconsistencies in **COLLA**borative environments); an online database with comprehensive support for inconsistency management. Employing Ricolla, community members can carry out the following tasks:

a) Enter conflicting data (in contrast to DBMS-driven online databases) and easily inspect the inconsistencies (in contrast to online database wikis). This departure from existing systems allows every community member to see all conflicting data and decide for herself which are to be believed.

b) Use the system and query the data even if they contain conflicts. This allows community members to postpone conflict resolution, enabling resolution in an "as-you-go" fashion.

c) Resolve inconsistencies in two different granularities: Either one at a time, or multiple inconsistencies at once based on higher-level criteria, such as provenance. Moreover inconsistency resolution can be done either collaboratively or individually, allowing users to adopt their friends' opinion for some conflicts, while maintaining their own opinion for others.

II. System Description

In the following we describe how Ricolla enables these 3 tasks and the research contributions associated to each of them. Due to the limited space, this cannot be a comprehensive and detailed review of the system. For that, please refer to [1].

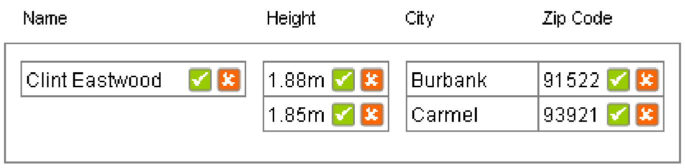
A. Model conflicting data

To support inconsistency management, Ricolla allows individual community members to explicitly model inconsistencies. Let us showcase this functionality through an example.

Our running example employs a movie database with the schema shown in Figure 1, where cinephiles can collaboratively edit information about movies. Consider Lara; a Clint Eastwood fan. Lara has recently discovered that her favorite

```
Actor(Name, Height, City, ZipCode)
Movie(Title, ReleaseYear)
MovieActor(Title, Name)
```

Fig. 1. Schema of Movie Ac-Database



Name	Height	City	Zip Code
Clint Eastwood ✓ ✗	1.88m ✓ ✗	Burbank	91522 ✓ ✗
	1.85m ✓ ✗	Carmel	93921 ✓ ✗

Fig. 2. Ac-tuple for Clint Eastwood

movie actor is 1.85m tall and she wants to update the movie database to reflect that. By inspecting the database, she finds out that some other user has already listed Clint's height as 1.88m.[1] In a DBMS-driven online database, she would be forced to replace 1.88m by 1.85m, since integrity constraints would allow only a single height value for each actor. However that would result in a system whose values are biased by Lara, who after all might be wrong about Clint's height. In Ricolla on the other hand, instead of *replacing* existing data, she can simply *augment* them with her own (conflicting) opinions. Utilizing the system's GUI, she can add as another possible height for Clint Eastwood, next to 1.88m, the value 1.85m. Figure 2 depicts the tuple summarizing the information for Clint Eastwood after Lara's insertion as shown on Ricolla's GUI. This tuple shows two possible values for the height and two possible values for the fan mailing addresses (which were entered previously by other users).

Contributions. To enable this functionality, Ricolla incorporates a **data model** for representing conflicting data, called *alternative capturing database* (in short *ac-database*). An ac-database is structured around the notion of an *ac-tuple*; a special type of tuple that captures inconsistent data about a single object. For instance, Figure 2 shows an ac-tuple summarizing the conflicting information on Clint Eastwood.

An ac-tuple can be vertically partitioned in a set of nested tables (3 in our case), which we call *ac-fragments* and cover part of the ac-tuple's schema. Each row in an ac-fragment represents a possible assignment of values for the set of attributes in that ac-fragment and is therefore called an *ac-alternative*. For instance, the right-most ac-fragment, contains two ac-alternatives, capturing the fact that Clint Eastwood's fan mailing address is either in Burbank with zip code 91522 or in Carmel with zip code 93921. Note that our data model is flexible enough to allow us to either correlate or not correlate the attribute values in a tuple by creating ac-fragments with appropriate schemas: To correlate the values of two attributes (e.g. city and zip code) and allow only alternatives that provide values for both attributes, we can simply create a fragment that covers both attributes (as is the case in our example). To allow this correlation decision to be made locally, different ac-tuples within an ac-relation can have ac-fragments of different schemas. Apart from ac-fragments and ac-alternatives, the ac-database also includes provisions for correlating ac-fragments *across* ac-tuples through coloring as well as representing the fact that a tuple might not exist at all in the database (through so-called empty alternatives). We will see these features in more detail in our discussion of Ricolla's query capabilities.

At this point it is important to note that an ac-database has formal foundations, as its semantics are formally defined in terms of *sets of possible worlds*. In particular, an ac-database instance summarizes a set of possible databases (known in the literature as possible worlds). For instance an ac-database containing only the ac-tuple of Figure 2 represents 4 possible databases; each of them constructed by selecting exactly one alternative for each ac-fragment.

Even more importantly, the ac-database is to the best of our knowledge the first formally-defined inconsistency-capturing data model that has been designed with inconsistency resolution in mind. Recent work in incomplete/uncertain databases led to a multitude of data models for representing sets of possible worlds. Notable examples include the recent *Uncertainty-Lineage Databases (ULDBs)* [2] (proposed in the context of the Trio system) and *World-Set Decompositions (WSDs)* [3] (designed in the context of the MayBMS system).

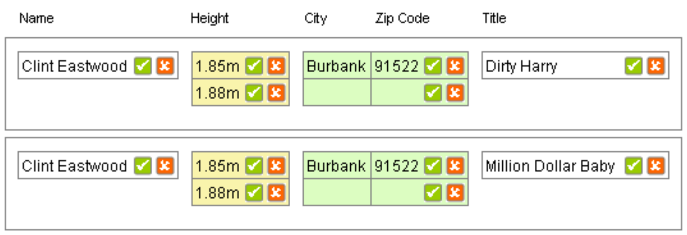
However none of these data models made the cut for Ricolla's GUI, as they do not offer neither an intuitive nor a compact representation of a set of possible worlds, suitable for the GUI of an inconsistency resolution tool. For instance, figuring out whether two alternatives across tuples are correlated requires in ULDBs parsing and reasoning on complex provenance formulas. On the other hand, as we will see next, such information is directly visible on an ac-database through the use of colors and requires no computation, making thus an ac-database more intuitive than ULDBs. The reason for this discrepancy lies in the goal of ULDBs and WSDs which is completeness (i.e. the ability to represent any set of possible worlds), in contrast to intuitiveness and compactness, which is the goal of the ac-database (which on the other hand is incomplete). The need for incomplete but more intuitive data models has also recently been recognized in [4]. A formal definition of the ac-database and its semantics together with a thorough comparison to other data models is included in [1].

B. *Query conflicting data*

Apart from introducing (conflicting) data, Ricolla allows users to also query the data, even when they contain conflicts. In this way, community members do not have to resolve the conflicts before they can use the corresponding data. Instead they can employ a natural "as-you-go" resolution workflow, where querying is interleaved with resolution steps.

For instance in our running example, utilizing Ricolla's visual query builder, Lara can formulate a query asking for all actors with an address in Burbank and their movies. Assuming that the system contains two movies for Clint Eastwood ("Dirty Harry" and "Million Dollar Baby"), the system returns the result shown in Figure 3. The query result is shown in the

[1]All values used in our running example are real values that can be found on different web-sites at the time of writing. For instance, `www.celebheights.com` lists Clint Eastwood's height as 1.85m, while `www.imdb.com` lists it as 1.88m.



Name	Height	City	Zip Code	Title
Clint Eastwood ✓ ✗	1.85m ✓ ✗	Burbank	91522 ✓ ✗	Dirty Harry ✓ ✗
	1.88m ✓ ✗		✓ ✗	
Clint Eastwood ✓ ✗	1.85m ✓ ✗	Burbank	91522 ✓ ✗	Million Dollar Baby ✓ ✗
	1.88m ✓ ✗		✓ ✗	

Fig. 3. Ac-tuples corresponding to movies of actors with a fan mailing address in Burbank

same way as the base data and so Lara can quickly grasp the inconsistencies that affect it. For instance, she can easily see that there are two possible values for Clint's height.

Before explaining the technical contributions that enable query answering, let us first use this example to explain two features of the ac-database that we briefly mentioned before; colored fragments and empty alternatives. In Figure 3 the fragment for the address has a special empty row, which we call *empty ac-alternative* and signifies that the corresponding tuple may not exist. For example, the empty alternatives in the tuples of Figure 3 show that these two tuples might not exist in the query result. This will happen if Clint's address is in Carmel, since the query is asking for Burbank actors. In addition to empty alternatives, an ac-database may also contain colored ac-fragments to represent correlation across ac-tuples. Intuitively, two ac-fragments (across ac-tuples) that are identical and share the same color, correspond to the same object. Therefore if a certain ac-alternative turns out to be true in one, the *same* ac-alternative will be true in the other (e.g. if Clint Eastwood's height is 1.85m in the first tuple, it also have to be the same in the second). Thus coloring allows users to easily distinguish when choices across ac-tuples are correlated.

Contributions. Although seemingly straightforward, query answering brings complications with it, due to the incompleteness of the data model. The problem arises from the fact that in general, query results can be arbitrary sets of possible worlds. Since the ac-database is an incomplete model, the query result might thus not be representable as an ac-database.

Although this is the case in general, we were able to find a class of queries, whose result can be represented as an ac-database. This is known in the literature as **the class of queries under which the model is closed**. By identifying it, we provide guarantees on which query answers can be represented in our data model without information loss (i.e. without losing any correlation between the tuples that might exist in the query answer). We were able to prove that the ac-database is closed under a sufficiently large class of queries, which correspond to select-join relational algebra queries with some additional restrictions on the join attributes. This result guarantees that for every query in this class, Ricolla will be able to present the result as an ac-database as shown above. A formal definition of this query class can be found in [1].

C. Resolve conflicts

The ultimate goal of community members is to resolve the inconsistencies present in an ac-database. Ricolla facilitates inconsistency resolution through two means. One way is by resolving one inconsistency at a time through resolution actions, as described below.

Going back to our running example, assume that Lara decides to resolve some of the inconsistencies. She knows that out of the two mailing addresses listed for Clint in Figure 2, the correct one is the one in Burbank. She can enter this information into the system by simply marking this ac-alternative as right. This is done by clicking on the green button next to the Burbank alternative on Figure 2 or 3. This action, called a *resolution action* and carried out on the same GUI that shows the conflicting data, allows her to naturally reduce the number of conflicts. Note that a resolution action affects only the particular user's view of the community database. For example, Peter, another movie fan, would still see both addresses and could mark a different one as right. This allows users to maintain differing opinions, which is a major requirement especially in the sciences, as discussed above.

However resolving one inconsistency at a time can be cumbersome, especially when higher-level criteria could be used to resolve multiple inconsistencies at once. To this end Ricolla provides so-called resolution policies, explained below.

After resolving the conflict on Clint's address, Lara decides to resolve the conflicts in the height values of the actors. Given actors' reputation of inflating their heights, she decides to choose for every actor in the database the smallest among the heights listed. Instead however of going to each actor tuple and manually selecting the minimum height, she employs the resolution policy builder to write a *resolution* policy that automatically selects the minimum height for each actor. To capture most common use-cases, resolution policies can resolve conflicts based on conditions that might involve not only the actual data (such as the actor's height above) but also annotations on them, such as their provenance or the timestamp of their insertion. For example, Lara can write a policy specifying that she trusts the heights provided by her friend Peter more than those provided by other users.

Contributions. To enable this conflict resolution functionality we designed two resolution mechanisms: First, a set of **resolution actions** that allow community members to resolve individual conflicts. Different members can maintain different opinions by resolving conflicts in different ways. Another notable feature of the resolution actions is that they can be carried out not only on the base data but also on query answers. It can be proven that a resolution action on the query result can be translated to a set of resolution actions on the base data that have the same effect on the ac-database. This allows members to avoid resolving all conflicts and instead lazily resolve only those that affect queries of interest.

Second, a **resolution policy language** that allows community members to write rules that resolve *multiple* conflicts at once based on certain criteria. These criteria may be based not only on the values appearing in the database but also on annotations of those values (e.g. timestamp) or other users' opinions. Moreover, the language is general enough to express both policies that are schema dependent (e.g. resolve the height

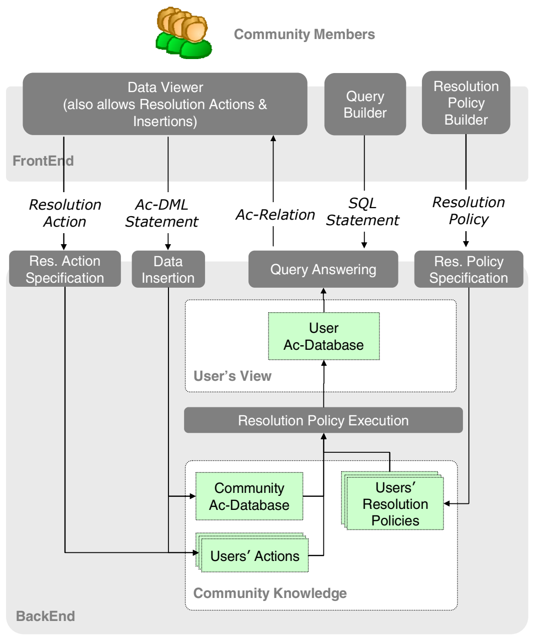


Fig. 4. Ricolla Architecture

values in a certain way) or schema independent (e.g. resolve values in all attributes of the database in a certain way).

III. System Architecture

Figure 4 depicts the architecture of the resulting system. The Frontend, shown on the top, allows users to visually formulate queries and resolution policies, inspect the data, insert new data and carry out resolution actions. These actions are supported by the Backend, shown on the bottom. To make the system as flexible as possible, we have opted for an architecture that allows each individual user to independently decide which data she wants to see and which users' resolution actions (if any) she wants to take into account. Dark boxes with rounded corners represent functions while rectangles correspond to internal data structures. Whenever users insert data into the system through the Frontend, these are appended to an append-only *community ac-database*. In parallel metadata about the insertion (such as the user's name and the timestamp of the insertion) are stored in a separate storage area, containing the *user actions*. Resolution actions carried out by the users are also recorded in the same area. Essentially the community ac-database and the user actions storage contain a description of all the relevant edit history of the system. Each user can subsequently write a resolution policy over these two storage areas to create her own view of the community ac-database (depicted in Figure 4 as *User Ac-Database*). By allowing resolution policies to operate not only on the data but also on the user actions, each community member can individually decide whether she wants to maintain her own opinion or reuse resolution actions carried out by her peers. For instance, she can choose to trust her own resolution actions when it comes to resolving heights, but trust her friend for fan mailing addresses. She can even disregard her own resolution actions if she wishes so, as the default behavior of the system in which a user's resolution actions affect her view of the database, is implemented simply as a resolution policy.

IV. Demonstration Scenario

In Ricolla's demonstration we will employ the movie scenario outlined above. The ac-database will be populated by real movie data found on the web. Subsequently audience members will be able to use Ricolla to query the movie data, add their own opinions or resolve existing inconsistencies. Moreover, to provide them with a realistic experience of the collaborative aspect of the tool, we will recreate the feel of a community by setting up multiple machines and allowing different audience members to access the system in parallel. This will allow them to explore how data added by others affect their own view of the database as well as how they can use resolution actions taken by a different user to their advantage.

Moreover, to give them an idea of the scalability of the system, we will implement a user simulator; a program that will simulate a user's interaction with the system. By increasing the number of user simulators running, audience members interacting with the system will be able to watch its behavior for large-scale communities: its response time, the size of the ac-database and the rate in which it increases.

V. Conclusion

Online communities need an online database system that allows them to collaboratively resolve inconsistencies in an 'as-you-go' fashion, while tolerating inconsistencies. To address this need, we designed Ricolla. While formally grounded, Ricolla enables a natural workflow for the processing of conflicting data. Users can model conflicts (through the ac-database model), query the data and resolve data either collaboratively or individually in two different granularities (through resolution actions and policies).

Please note that although our discussion was guided by the particular use-case of a movie community, Ricolla's applicability is much more general. Inconsistencies are a natural consequence of collaborative editing and as such appear in most online databases that allow sharing of structured data. Notable examples of communities that will benefit from the system include internet user communities (such as the movie community presented above or Freebase, which is the structured equivalent of Wikipedia) and scientific communities (where the ability to maintain different opinions is particularly imperative).

References

[1] Y. Katsis, A. Deutsch, Y. Papakonstantinou, and V. Vassalos, "Inconsistency Resolution in Online Databases," CSE, UC San Diego," Technical Report, CS2009-0945.

[2] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom, "Databases with uncertainty and lineage," *VLDB J.*, vol. 17, no. 2, pp. 243–264, 2008.

[3] L. Antova, C. Koch, and D. Olteanu, "10^{10^6} worlds and beyond: Efficient representation and processing of incomplete information," in *ICDE*, 2007, pp. 606–615.

[4] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom, "Working models for uncertain data," in *ICDE*, 2006, p. 7.