Conflict Resolution in Online Databases*

Yannis Katsis† ikatsis@cs.ucsd.edu
Alin Deutsch† deutsch@cs.ucsd.edu
Yannis Papakonstantinou† yannis@cs.ucsd.edu
Vasilis Vassalos‡ vassalos@aueb.gr

†CSE Department, UC San Diego
‡Athens University of Economics and Business

ABSTRACT

Shared online databases, such as Google Fusion Tables or Quickbase, allow non-programmer community members to collaboratively maintain and browse data. While community members may believe in conflicting facts (due to conflicting sources, measurements or opinions), current online databases do not yet offer support for the management of data conflicts.

Ricolla is a novel online database that treats data conflicts as first-class citizens. Unlike prior work in uncertain databases, which was made to provide a database back-end to application logic, Ricolla is tuned to the requirements of the online database paradigm, allowing intuitive visualization of conflicts and collaborative data editing/conflict resolution. The proposed end-to-end system makes the following contributions: a) an online database paradigm that captures conflicts, allowing data query and update, while enabling personalized, "as-you-go" conflict resolution, b) a data model and corresponding generic user interface for explaining the conflicts to the users in a simple and compact form that is aligned with the requirements of online databases; we formalize the "simplicity" and "compactness" requirements and show that Ricolla strikes a novel point in the tradeoff between simplicity, compactness and expressive power, c) personalized conflict resolution via *resolution actions* that allow each user to resolve individual data conflicts, and via a *resolution policy language* for performing a set of resolution actions based on high-level criteria (e.g., give priority to data based on their value, provenance or timestamps), and d) a set of algorithms for implementing the system on top of an RDBMS.

1. INTRODUCTION

Lately, *online databases* (Google Fusion Tables [2], QuickBase [3], Zoho Creator [5], Caspio Bridge [1], TrackVia [4] and many more) enable online communities to collaboratively maintain their data. Unlike conventional databases, which were designed to provide only a back-end to enterprise applications, online databases offer a package consisting of a generic web application interface ready to be used by non-programmers and an underlying database. If we temporarily neglect their sharing and collaboration aspect, we can think of the various online database offerings as ranging from spreadsheets to MS Access database-like interfaces for inserting data and visually formulating queries. Where they lie on the spreadsheet vs. database spectrum depends on their choice on the simplicity vs. expressive power tradeoff; we are more interested in those at the "database-like" end of the spectrum, though many of the lessons easily carry over to the entire spectrum. At any rate, the key novelty of online databases is that multiple users simultaneously enter and read data in the database via its interface.

Provision of a *generic* interface, *simplicity* of the interface and of the overall paradigm, and *pay-as-you-go support* are three features that distinguish online databases from conventional database systems.[1] The genericity and simplicity of the interface are required to make the system suitable for non-experts. Additionally, the pay-as-you-go approach ensures that the system remains operable even before the users fully manipulate (integrate, clean etc) the data.

Expanding online databases to accomodate data conflicts is a naturally motivated problem. The multiple users of the online community may hold conflicting beliefs, which they want to represent in the database. The reason for these conflicting beliefs could be different biases, different sources (whereas the one may be more up-to-date than the other), different interpretations of the same phenomena and many more. Instances of the latter often appear in the sciences, where, as explained in [28], researchers have contradicting opinions, about, for example, a genotype-phenotype map or a shadow on an X-ray. Each scientist or group often wants to record their opinion, even though conflicting beliefs, when entered into the database, will lead to conflicting data.[2]

An online database's first approach towards data conflict management is the "do nothing": The online database is very permissive (i.e., poses no constraints) and allows the writing users to enter any data. Then the reading users have to write queries that retrieve conflicting data, figure out how the data conflict and perhaps make their own "clean" copy representing their resolution to the conflicts.

An alternative approach has the online database allowing the enforcement of constraints and therefore potentially disallowing data conflicts altogether. A common constraint is uniqueness, which dictates that there may be only one value for each attribute of an entity. Unfortunately, this leads to a single, ad-hoc, and often undesirable data conflict resolution scheme: Last editor's opinion "wins".

Specialized community knowledge management systems, such as Wikipedia, had to grow beyond "last editor wins". So, a third approach emerged: They adopted a specialized workflow suitable

*This work has been supported by the NSF grant IIS-1117527.

[1]While all database systems provide a generic administrative interface, such interface is used only by the very few administrative users. Therefore it is not an essential ingredient of the database's success in a particular deployment.

[2]Yet another source of data conflicts, which is unrelated to conflicting beliefs, is erroneous data entry.

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

Figure 1: Schema of Movie Ac-Database

for the particular scenario. For instance, many systems assign a chief editor who eventually dictates which of the conflicting opinions is widely visible.

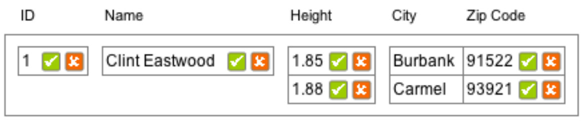
While each approach has its benefits in a particular ecosystem, they do not meet our online database requirements. First, many approaches fail the resolve-as-you-go assumption: The third approach that uses a more sophisticated workflow fails because it requires that conflicts are resolved before the data are usable. The first approach trivially fails because it does not even recognize data conflict and resolution. Even more importantly, all the approaches fail the genericity requirement: They force communities to agree on a single way to manage conflicts. Indeed, given the difficulty of agreement in a large-scale online community, the community's online database should ideally meet a fourth requirement: *Resolution personalization*, i.e., the ability of individual users (or groups of users) to resolve data conflicts in the way they see best.

To satisfy these requirements, we develop the **Ricolla** (**R**esolve **I**nconsistencies in a **COLLA**borative environment) online database. Community members are able to both enter conflicting data and easily inspect the conflicts and resolve them. Moreover, members or subgroups can resolve the conflicts as they see best for their purposes, while being allowed to maintain different opinions with other individuals or subgroups. Finally, the conflict resolution happens in an "as-you-go" fashion, allowing members to use the system and query the data even before all conflicts have been resolved.

In the following we present the functionality and architecture of Ricolla together with our contributions through a representative use-case: The creation of an online movie database, where cinephiles can collaboratively edit information about movies. The schema of the database, designed by the community initiator, is shown in Figure 1. For ease of exposition, it consists of 3 relations, holding information on actors, movies and their relationships.

Modeling and querying conflicting data. Let us introduce Lara; a Clint Eastwood fan. Lara has recently discovered that her favorite 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 listed Clint's height as 1.88m.[3] In a constraint-enforcing 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. Such a tuple is called an *alternative-capturing* tuple (in short *ac-tuple*). As we will formally explain in Section 2, an ac-tuple captures different *ac-alternatives* for the attribute values. For instance in Figure 2 it shows the two possible alternatives for the height and two possible alternatives for the fan mailing addresses (which were entered previously by other users). The 'right' and 'wrong' buttons next to each ac-alternative are not part of the ac-tuple but are shown on the

[3]Our running example employs real values found on the web 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.



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

Figure 2: Ac-tuple for Clint Eastwood

GUI to allow resolution of conflicts, as we will explain later.

Apart from introducing (conflicting) data, Lara can also query them, even when they contain conflicts. For example, utilizing Ricolla's visual query builder, Lara formulates a query asking for all actors with an address in Burbank and their movies. Assuming that the system contains the two movies for Clint Eastwood shown in Figure 4b, the system returns the query result shown in Figure 5. The latter is shown in the same way as the base data so that Lara can quickly grasp the conflicts that affect her query. For instance, the query result exhibits the information about the two possible values for Clint's height. By allowing users to query data before conflicts are resolved, Ricolla supports "as-you-go" conflict resolution.

Contribution: A new tradeoff between simplicity, compactness and expressive power. The data model (Section 2), called *ac-database*, and the corresponding generic report & update interface capture and represent data conflicts. The model's formal foundations draw from the database theory of possible worlds: an ac-database instance is a compact representation of a *set of possible worlds*. The literature contains several data models for representing sets of possible worlds (commonly referred to as data models for uncertain or incomplete data). However, Ricolla's data model differs from those in that it is tuned for the requirements of online database systems and is designed to give rise to an intuitive interface for such systems. In the interest of simplicity, it avoids the use of variables or complex provenance formulas in the data values representation, which, in contrast, are used in *c-tables*, *ULDBs* (used in the context of the Trio system) and *U-relations* (used in the context of the MayBMS system).

Moreover, we show that Ricolla's data representations are more compact than other data models, such as ULDBs. This, coupled with the direct visualization according to the data model, means lower cognitive load for users.

We present how Ricolla's data model achieves a new tradeoff between simplicity, compactness and expressive power. Then we formally show that this trade-off is effectively optimized in the sense described further down in this paragraph. In particular, we quantify the expressive power penalty (that we paid in the interest of compactness and simplicity) by characterizing the *class of queries* whose answers (which are in general arbitrary sets of possible worlds) can be still represented in Ricolla's data model (Section 4). In other words, we identify the class of queries under which the data 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). Furthermore, we show that this is the largest class of queries that guarantees closure; i.e., the result of more expressive queries is not representable in our data model. For such more expressive queries Ricolla computes an approximation of the query answer in our data model. At this point an important question arises about our choice of trade-off point: Could we make our data model slightly more expressive to guarantee closure for a larger class of queries, while at the same time keeping compactness and simplicity? We show that this is not possible, i.e., Ricolla's trade-off point is effectively optimized. If we want the model to support a sufficiently larger class of queries (which we will formally define later), the data model needs to be as expressive as existing data models for uncertain data (and thus correspondingly non-simple and non-compact). Note that we refer here to the data model used for the system's Frontend. Existing data models could conceptually still be used for the Backend (although, as we will explain later, in our current implementation we skip the intermediate layer and implement the system directly on top of an RDBMS).

Resolving conflicts. After inserting Clint's height and inspecting the data through a query, Lara decides to resolve some of the conflicts. 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 action, called a *resolution action* and carried out on the same GUI that shows the conflicting data (by clicking on the green 'right' button next to the corresponding ac-alternative), allows her to naturally reduce the number of conflicts. Note that a resolution action in Ricolla affects by default only the particular user's view of the community database. For example, Harry, another movie fan, would still see both addresses and could mark a different address as right. This allows users to record and maintain differing opinions and beliefs, which is a major requirement especially in the sciences, as discussed above. However, as we will see next, users have also the option to collaborate with each other by adopting the resolution actions carried out by their trusted collaborators or certain community authorities.

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 when reporting them to media, 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 for every actor the minimum height. 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, as well as other users' resolution actions. By allowing policies to operate on other users' resolution actions, Ricolla allows users to collaborate with their peers by borrowing their resolution actions. For example, Lara can write a policy specifying that she wants to use the opinion of her friend Harry to resolve the conflicts in the heights.

Contribution: Personalized conflict resolution per item or per policy. A set of **resolution actions** that allow community members to resolve individual conflicts (Section 3.2). 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. In the latter case, the system automatically translates the resolution actions on the query result to appropriate resolution actions on the base data that have the same effect. This allows community members to avoid resolving all conflicts and instead lazily resolve only those that affect queries in which they are interested.

A **resolution policy language** that allows community members to write rules that resolve *multiple* conflicts at once in their view of the community ac-database (Figure 3), based on certain criteria (Section 3.3). 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 (thus allowing collaboration between users). Employing a set of resolution policies, each individual user can independently decide which data she wants to see and which users' resolution actions (if any) she wants to take into account, thus creating her own *personalized* view of the community database.

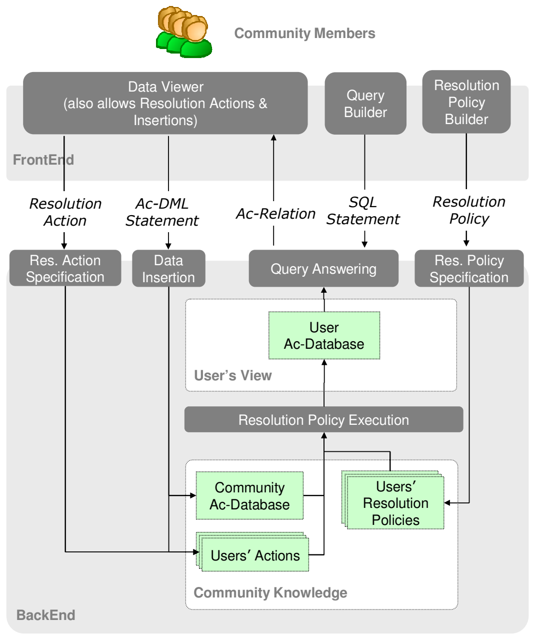


Figure 3: Ricolla Architecture

1.1 System Architecture

Figure 3 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.

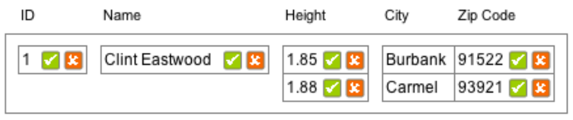
Dark boxes with rounded corners represent functions while rectangles correspond to internal data structures. Whenever users insert data into the system through the Frontend, these are appended to an append-only *community ac-database*. In parallel metadata about the insertion (such as the user's name 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 one or more resolution policies over these two storage areas to create her own view of the community ac-database (depicted in Figure 3 as *User Ac-Database*). This architecture enables simultaneously resolution personalization (by allowing every user to create her own personalized view) and collaboration (by allowing policies to also operate on other users' actions).

Contribution: An end-to-end system that allows the community members to do all of the following: a) represent conflicting data in a formally defined data model, b) resolve only those conflicts that affect the answer of a specific query (to avoid resolving conflicts of no interest to them) and c) maintain their own opinions on how certain conflicts have to be resolved.

Algorithms for implementing the system on top of a relational DBMS (Section 5). These include algorithms to store and retrieve an ac-database, to answer queries over an ac-database and to execute resolution policies.

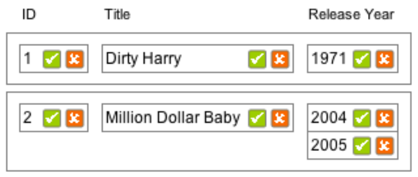
2. DATA MODEL

A conflict resolution system should allow users or applications to easily inspect the conflicts in the database. Therefore it should capture not only the possible data items but also the relationships



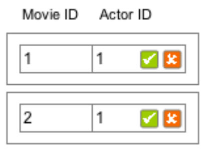
ID	Name	Height	City	Zip Code
1 ✓ ✗	Clint Eastwood ✓ ✗	1.85 ✓ ✗	Burbank	91522 ✓ ✗
		1.88 ✓ ✗	Carmel	93921 ✓ ✗

(a) Actor



ID	Title	Release Year
1 ✓ ✗	Dirty Harry ✓ ✗	1971 ✓ ✗
2 ✓ ✗	Million Dollar Baby ✓ ✗	2004 ✓ ✗
		2005 ✓ ✗

(b) Movie



Movie ID	Actor ID
1	1 ✓ ✗
2	1 ✓ ✗

(c) MovieActor

Figure 4: Ac-database of movie community

between them (e.g., that two items are mutually exclusive, or that they always have to co-exist) using a simple and compact structure.

To this end, Ricolla's Frontend employs a special data model, called *ac-database*, that exhibits those relationships in the data. Note that researchers have already proposed a multitude of models for representing the relationships between data items, commonly referred to as data models for uncertain data. However, as we will formally explain in Section 2.3, such models have been created as general-purpose representations utilized by backend database systems and are thus not suitable for an online database.

Next, we describe the ac-database, define its semantics and finally compare it to previously proposed models for uncertain data. This section discusses the data model that is exposed to the user at the Frontend and not to the one used at the Backend for storage. For the latter refer to the implementation section (Section 5).

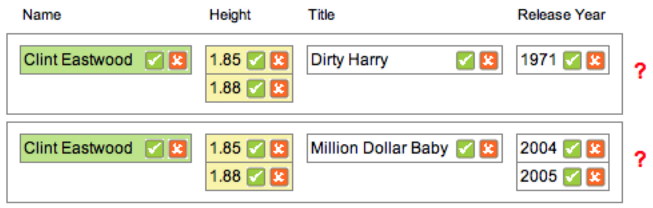
2.1 Definition

Our data model is structured around the notion of an *alternative-capturing tuple* (in short *ac-tuple*); a special form of tuple that captures mutually exclusive information about a single object. Before formally defining it, we first introduce it through an example.

Ac-Tuple Structure. Figure 4a shows an ac-tuple summarizing the conflicting information on Clint Eastwood, entered by Lara and other movie fans. It shows two possible heights (1.85m & 1.88m) and two possible addresses (Burbank, 91522 & Carmel, 93921).

As seen in the example, an ac-tuple can be vertically partitioned in a set of nested tables (4 in this 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 two possibilities for Clint's city and zip code pair: It is either (Burbank, 91522) or (Carmel, 93921).

Note that by specifying the schema of the ac-fragments in an ac-tuple, one can correlate or de-correlate the values within the tuple. For instance, by creating a fragment that covers two attributes (e.g., city and zip code), a user can assign possible value pairs for both the city and zip code, thus correlating the two attributes. On the other hand, the separate fragment for height expresses the fact that the value for height is *independent* of the value for address. Note that one can decide how to correlate attribute values in a per-tuple basis. This means that two ac-tuples within the same relation may have ac-fragments of different schemas. Thus our data model differs from the standard nested relational data model [6]. As we will explain in Section 2.3, the ac-tuple's structure and its partitioning in fragments guarantees that the data model stays compact and simple.



Name	Height	Title	Release Year	
Clint Eastwood ✓ ✗	1.85 ✓ ✗	Dirty Harry ✓ ✗	1971 ✓ ✗	?
	1.88 ✓ ✗			
Clint Eastwood ✓ ✗	1.85 ✓ ✗	Million Dollar Baby ✓ ✗	2004 ✓ ✗	?
	1.88 ✓ ✗		2005 ✓ ✗	

Figure 5: Ac-tuples corresponding to movies of actors with a fan mailing address in Burbank

Correlating Ac-Tuples. Until now we looked only at a single ac-tuple. Let us now move to an ac-relation comprised of a *set* of ac-tuples. The main question when considering sets of tuples is whether they are correlated or not. Does each tuple represent independent conflicts or are the conflicts represented by two distinct tuples related? It turns out that to express even simple query answers, different ac-tuples within the same ac-relation have to be correlated. In our data model such correlations are depicted by marking ac-fragments of different ac-tuples with the same marker, as exhibited by the following example.

Consider the three ac-relations shown in Figure 4: the *Actor* relation with the Clint Eastwood ac-tuple, the *Movie* relation containing two movies and the *MovieActor* relation stating that Clint Eastwood has played in both. If we ask for actors in Burbank and their movies (which corresponds to joining these ac-relations, putting a selection *City*='Burbank' and projecting out some attributes), the system returns the ac-relation shown in Figure 5.

The query result contains two ac-tuples corresponding to the two Clint Eastwood movies. Note that the ac-fragment containing the height is the same across both tuples and has the same color (yellow or the lighter shade of gray in B/W). Each color visualizes what we call a *dependency marker* or simply *marker* of an ac-fragment. Although the current implementation represents markers as colors, one can envision other visualizations, such as for instance labels. Intuitively when two ac-fragments (across tuples) are identical and share the same marker, they correspond to the exact 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. For instance, in our example if actor Clint corresponding to the first tuple has height 1.85m, the same will hold for the actor corresponding to the second tuple, since they are the same person. Notice that marking fragments is more expressive than simply grouping tuples by a set of attributes. For instance, even though we can express the same information as in Figure 5 without markers by simply grouping the movies by the actor, this is not true in general. For example, if the database contained another actor that played in the movie "Million Dollar Baby", grouping the movies by actor would generate two tuples for the particular movie (each appearing in a different group), which would still have to be correlated through markers.

Optionality Flag. Finally, the data model can also express the fact that a tuple might not exist in the answer of a query. This is accomplished by assigning to an ac-tuple an *optionality flag* '?'. For instance, in Figure 5 the two tuples might not exist in the answer of the query asking for movies of Burbank actors (since Clint might live in Carmel instead). Thus they are both marked as optional. Similarly to ac-fragments, optionality flags might be marked to express correlations. In the current implementation markers are shown visually as colors. For instance, the flags of the two tuples in the previous example share the same marker, stating that if one does not exist (which will happen if Clint's address is in Burbank) the other will also not exist. Although not shown on the example, an ac-tuple might have more than one optionality flags.

Formal Definitions. We proceed in a top-down fashion, defining first the ac-database and then the components it is built from:

DEFINITION 2.1. AC-DATABASE & AC-RELATION: *An ac-database consists of a set of ac-relations, which in turn are sets of ac-tuples. Similarly to flat relations, each ac-relation has also an associated schema, which is a set of attributes.*

DEFINITION 2.2. AC-TUPLE: *An ac-tuple consists of a set of ac-fragments. Every ac-tuple belongs to an ac-relation and it has the schema of the relation. Finally, an ac-tuple might be assigned a set of optionality flags (each of which might have a marker).*

DEFINITION 2.3. AC-FRAGMENT & AC-ALTERNATIVE: *An ac-fragment is a relation whose schema is a subset of the schema of the ac-tuple in which it appears and whose rows are called ac-alternatives. The schemas of all ac-fragments of an ac-tuple form a partition of the ac-tuple's schema. Finally, each ac-fragment may be also assigned a marker. Two ac-fragments can share the same marker but only if they are identical (i.e., they have the same schema and contain the same set of alternatives).*

2.2 Possible World Semantics

Each ac-database (or ac-relation) represents a set of possible flat databases (or flat relations). Each such flat database (or relation) is referred to in the literature as a *possible world*. Intuitively a possible world of an ac-relation can be created by picking for every ac-fragment in the ac-tuples of the ac-relation exactly one of its ac-alternatives (while respecting the optionality flags and the markers). We refer to every such pick of ac-alternatives for a given ac-tuple as an *interpretation* of that ac-tuple, defined below:

DEFINITION 2.4. AC-TUPLE INTERPRETATIONS: *An interpretation of an ac-tuple is a flat tuple created by mapping each fragment of the tuple to a single alternative within that fragment.*

For example, the ac-tuple of Figure 4a has the four interpretations shown in Figure 6.

#1:	1	Clint Eastwood	1.85	Burbank	91522
#2:	1	Clint Eastwood	1.85	Carmel	93921
#3:	1	Clint Eastwood	1.88	Burbank	91522
#4:	1	Clint Eastwood	1.88	Carmel	93921

Figure 6: Interpretations of the ac-tuple in Figure 4a

If fragments were not marked, then we could consider each ac-tuple separately to create the possible worlds represented by the ac-relation. If they are however marked, then decisions taken for a fragment with a certain marker should be consistent across all ac-tuples in which a fragment with the same marker appears. To this end, we define the notion of *compatible ac-tuple interpretations*.

DEFINITION 2.5. COMPATIBLE AC-TUPLE INTERPRETATIONS: *Given two ac-tuples and one interpretation for each, we say that the two interpretations are compatible if they were derived from the respective ac-tuples by mapping each fragment of the ac-tuples with the same marker to the same alternative.*

Finally, we have to also take into account the optionality flags, which denote that an ac-tuple may be absent. To this end, we define a non-existence assignment, which specifies which optionality flags will lead to non-existing tuples. As is the case with ac-tuples, in this process we have to respect the markers of the optionality flags.

DEFINITION 2.6. NON-EXISTENCE ASSIGNMENT: *Given a set S of optionality flags, an non-existence assignment on S is a choice function that assigns to each flag in S a boolean, such that all flags with the same marker are assigned the same value.*

Given the above definitions, we can now define the set of possible worlds represented by an ac-relation and an ac-database:

DEFINITION 2.7. POSSIBLE WORLDS OF AN AC-DATABASE: *An ac-relation represents all possible flat relations that can be produced by following two steps: First, pick a non-existence assignment for the optionality flags in the ac-relation and second, for every tuple in the ac-relation that does not have at an optionality flag selected by the non-existence assignment, pick exactly one of its interpretations, such that all the interpretations chosen are pairwise compatible. Finally, an ac-database represents all flat databases that can be constructed by taking for each ac-relation in the ac-database one of the possible relational instances that it represents.*

From now on we will use $PWorlds(I)$ to denote the possible worlds represented by an ac-database or ac-relation I.

2.3 Comparison to other data models

Representing sets of possible worlds has been a long-studied problem. Researchers have proposed many data models for capturing sets of possible worlds, the most influential ones being Lipski's *v-tables* and *c-tables* [19]. The problem has gained traction again recently in uncertain and incomplete databases. This led to new data models, such as *Uncertainty-Lineage Databases (ULDBs)* [13] (proposed in the context of the Trio system [7]), *World-Set Decompositions (WSDs)* [9] and U-relations [8] (both designed for the MayBMS system [10]) and semiring-annotated relations [18]. The same problem has also been studied in the context of probabilistic databases [20, 15]. However, most models have been created as general-purpose representations and are not suitable for an online database, which places two unique requirements on the data model: *simplicity* and *compactness*. This led us to the design of the ac-database, which, as shown next, satisfies both requirements.

Simplicity. In online databases, it is important for a data model instance, and especially conflicts, to be visualized effectively and intuitively. An ac-database shows correlations between data at a glance (through markers) without the need for any further computation. In contrast, existing approaches fall in two categories w.r.t. simplicity. The first category consists of data models that employ variables and/or complex provenance formulas to capture correlations in the data. As some studies [23, 25][4] suggest, the use of variables makes them hard for users to understand. For instance, ULDBs annotate tuples with provenance formulas. Understanding whether two flat tuples can co-exist in a possible world requires parsing and reasoning on those formulas. Similarly for c-tables and U-relations. The second category consists of data models that were designed as storage models and not for a user interface. Representatives of this category include WSDs and U-relations which, having been built with efficient query evaluation in mind, decompose a single relation into multiple relations. While simplicity is in general subjective, it can be argued that avoiding variables and decomposition as done by ac-relations clearly improves intuitiveness.

Compactness. To prevent information overload, a data model should effectively summarize a set of possible worlds. Comparison

[4]References borrowed from [24]

of different data models in terms of compactness is not straightforward, as they generally employ different structures. To facilitate this comparison, we pick a metric, that is on one hand flexible enough to apply to different models and on the other hand provides an obvious quantitative indication of their compactness. Given an instance of any data model, we define its *size* to be the number of data values (i.e., cells) that it contains. For instance, the ac-tuple about Clint in Figure 4a contains 8 values and thus is of size 8. Given two instances, the one with smaller size is referred to as more *compact*. Note that in the interest of uniformity, our metric ignores variables and provenance formulas, which are present in other models such as ULDBs, c-tables and U-relations. However, in practice these constructs also increase the size of the representation and therefore reduce compactness.

It turns out that, according to our metric, an ac-database can be *exponentially* more compact than a ULDB in representing the same set of possible worlds. Intuitively, this happens because ULDBs do not employ fragments and thus store all possible interpretations of an ac-tuple (which, as we have explained above, corresponds to taking the cartesian product of the ac-fragments). For instance, a ULDB representing the set of possible worlds corresponding to the ac-tuple in Figure 4a will be similar to Figure 6 and will therefore have a size of 20, instead of 8 which is the case for the ac-database. In [8], it was noted that U-relations can also be exponentially more succinct than ULDBs for similar reasons. However, we can prove that even in the case of U-relations and WSDs, an ac-database offers *always* an at least as compact representation:

THEOREM 2.8. COMPACTNESS: *For any set of possible worlds S_{PW} that can be represented as an ac-database, there exists an ac-database representation that is at least as compact as any ULDB, WSD or U-relation representation of S_{PW}.*

It is interesting that c-tables offer a more compact representation than ac-databases. This is a result of using variables in the place of data values and then constraining the values of these variables through formulas. However, for this reason they also violate the simplicity requirement as explained above.

To conclude, previously proposed data models have placed a higher emphasis on expressive power and storage efficiency than suitability for visualization and user interaction. For instance, U-relations, WSDs and ULDBs are all *complete* (i.e., they can represent all finite sets of possible worlds). For the problem of user-guided collaborative inconsistency resolution the ac-database, while non-complete, is expressive enough while also being compact and intuitive. The need for non-complete but intuitive data models has also been recognized in [24]. Finally, note that the ac-database is used to drive Ricolla's Frontend, including conflict visualization and user interaction. Existing data models could still be used at the Backend for storage.

3. ACTIONS & POLICIES

Utilizing an interface driven by the ac-database model, Ricolla allows users to both model conflicting data and subsequently resolve the conflicts. Modeling conflicting data is supported through Ricolla's insertion actions, described below. Resolving conflicts on the other hand is enabled by a combination of two mechanisms: *resolution actions* and *resolution policies*. Resolution actions enable each user to express her opinion on how individual conflicts should be resolved. Resolution policies on the other hand allow her to use her own opinion or that of her peers to create her own personalized partially resolved version of the community database. (Insertion and resolution) actions and policies are described below.

3.1 Insertion Actions

A user can model conflicting data by carrying out insertion actions directly on the interface (denoted as "Data Viewer" in the system's architecture in Figure 3). Two types of insertions are allowed. If the user wants to simply express an additional opinion about an existing object, she can *insert an alternative* in an ac-fragment of an existing ac-tuple. On the other hand, if she wants to introduce information about a new object, she can *insert a new ac-tuple*. Since every ac-tuple might have a different schema, at the time of the ac-tuple's creation, the user has to specify how it is going to be partitioned into ac-fragments. This action does not have to be final. A user can still merge fragments after the creation of an ac-tuple. However, this can only be done if both fragments have a single alternative. Once a fragment contains at least two alternatives, it cannot be merged anymore.

3.2 Resolution Actions

Apart from modeling conflicts, users should be able to also resolve them (which conceptually corresponds to removing some of the conflicting opinions). In an ac-database a single conflicting opinion is modelled by a single ac-alternative. Thus, to offer the finest level of granularity in resolving conflicts, a conflict resolution system should allow the users to reason on individual ac-alternatives. In Ricolla this is accomplished through *resolution actions*, which are of the following two types:

- *Mark an ac-alternative as wrong.* This corresponds intuitively to removing (i.e., deleting) the ac-alternative. This action allows users to partially resolve a conflict even when they do not have the knowledge to fully resolve it. For instance, in our running example, if Clint was listed with 3 mailing addresses, in Carmel, Burbank and Paris, a user could restrict the conflicting values by removing Paris, even if she could not decide between Carmel and Burbank. On the other hand, when she can fully resolve a conflict, she can carry out the second type of resolution action:

- *Mark an ac-alternative as right.* This corresponds to marking all remaining ac-alternatives within the same ac-fragment as wrong. As such it can be simulated by a set of 'mark wrong' actions. However to facilitate faster resolution, Ricolla offers it conveniently as a separate action. Note that this action specifies that an ac-alternative is right only w.r.t. the other ac-alternatives *currently* in the same ac-fragment. It will not remain right if additional ac-alternatives are added to the fragment. However users can still ask the system to consider an ac-alternative as *always* right (even under updates to the ac-fragment) by formulating an appropriate resolution policy (described in the next section).

Both types of actions can be carried out directly on the interface corresponding to Ricolla's data model (by clicking on the 'right' or 'wrong' button next to an ac-alternative). This simplicity is not a coincidence but one of the requirements set when designing the data model. To allow easy resolution, we made sure that each conflict corresponds to a separate entity (i.e., the ac-alternative) with associated actionable items w.r.t. resolution. Other data models (such as those discussed in Section 2.3), do not satisfy this requirement, as conflicts are hidden within provenance formulas or variables and the possible ways to resolve them are far from obvious.

Since answers to queries can also be represented in Ricolla's data model [5], users can carry out resolution actions not only on the base

[5]We will present this result together with the exact class of queries for which it holds in Section 4.

```
<Policy>     ::- if <Condition> then <Outcome>
<Condition>  ::- [<Scope> and]
                 (<AttribCond> | <ActionCond>)
<Outcome>    ::- remove this alt | remove other alt

<Scope>      ::- <Relation> [and <Attribute>]
<Relation>   ::- relation = <String>
<Attribute>  ::- attribute = <String>

<AttribCond> ::- attribute-value =
                    <String> | MIN | MAX | MEDIAN

<ActionCond> ::- <ActionType> and
                [<ActionTime> and]
                [<ActionUser>]
<ActionType> ::- action-type =
                    inserted | marked wrong
<ActionTime> ::- action-time =
                    <DateTime> | MIN | MAX
<ActionUser> ::- action-user = <String>
```

Figure 7: Policy Language Syntax in EBNF

data but also on the query results. This accommodates all those users that access the community data through queries and want to focus only on the conflicts that directly affect their applications. Each resolution action carried out on the query result is then automatically translated to a set of resolution actions on the base data with the same effect. This is accomplished by adopting techniques developed for the classical view update problem [12].

Note that although resolution actions conceptually correspond to deletions, they are not implemented as such. Allowing all users to remove alternatives from the community ac-database would violate Ricolla's requirement of personalized conflict resolution: Each user should be able to resolve conflicts to her liking, without affecting the other users' view of the data.

To satisfy this requirement, resolution actions do not directly affect the community database. Instead they are recorded as annotations in a special *'User Actions'* table attached to each alternative. This table stores the users that have marked the alternative as wrong together with the timestamp of the actions.[6] Subsequently, each user can write resolution policies (explained next) over both the ac-database and the 'User Actions' tables to create her own view of the database. As we will see, the default policy is to remove from each user's view the alternatives that she has marked as wrong, thus implementing the expected semantics of resolution actions. However, she can express other policies to see the entire community database or adopt resolution actions of her friends. This architecture in which resolutions actions are stored as annotations that can be accessed by resolution policies, enables each user to decide independently which community data she wants to see.

3.3 Resolution Policies

To create her own personalized (partially) resolved version of the community database, a user writes a set of *resolution policies* over both the ac-database and the 'User Actions' tables. Intuitively, a resolution policy is a rule specifying which ac-alternatives should be removed from the user's view, according to some high-level criteria. As is the case with the data model, the resolution policy language has to strike the right balance between expressive power and simplicity. Therefore, the design of the resolution policy language was guided by the following desiderata:

[6]The 'User Actions' table of an ac-alternative also contains a tuple for the user that inserted the alternative. This information might later be used by resolution policies as we will explain next.

1. *Allow schema-independent policies:* Users often want to use similar resolution decisions for many attributes in the database. For instance, if a user wants to only see the latest version of the database, she should be able to express a policy that selects the latest inserted alternative *for all attributes* of the ac-database. To enable such scenarios, the resolution policy language should allow for policies that do not necessarily depend on the schema of the database.

2. *Enable collaboration:* Users should be able to resolve the conflicts not only based on the data, but also based on their friends' opinions. For instance, a user should be able to borrow her friends' resolution actions.

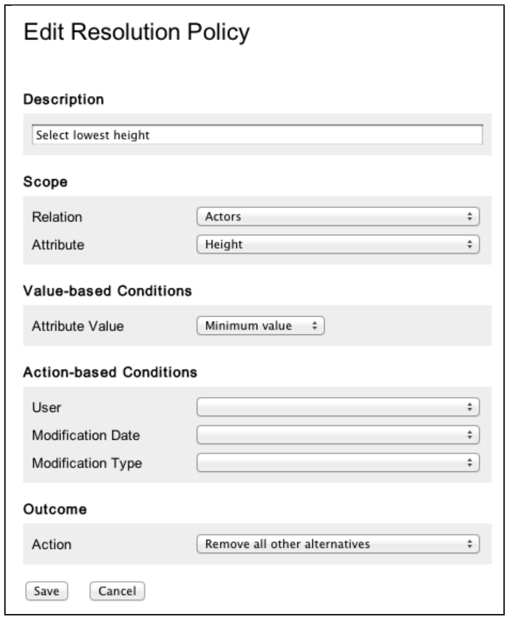
3. *Detect and prevent recursion:* Collaborative resolution policies, if not designed carefully, can easily lead to recursive policies (e.g., two policies specifying that user A trusts B and vice versa). Recursive policy definitions are an interesting area of research and have recently received attention [17]. However, they can lead to counter-intuitive results and thus in this work we opted for a resolution policy language that disallows recursive definitions.

Based on these desiderata, we designed a policy to have the syntax shown in Figure 7. The syntax is depicted in the Extended Backus-Naur Form (EBNF). A policy consists of two parts: a) a condition which is checked against all ac-alternatives appearing in the ac-database to create a set of alternatives of interest and b) an outcome of one of the two types 'remove this alt' or 'remove other alt', signifying that for each alternative satisfying the condition, the alternative (or the other alternatives in the same fragment, respectively) will be removed from the user's view.

To enable collaboration (*desideratum #2* above), the policy's condition can be expressed over both the data in the ac-database and the actions in the 'User Actions' tables. By placing a condition on the value of the data, she can select ac-alternatives that either have a specific value or have the minimum/maximum/median value among all ac-alternatives of the corresponding ac-fragment. For instance, this allows her to express Lara's resolution policy, described in the introduction, which selects the minimum height for each actor. By placing a condition on the user actions on the other hand, she can select alternatives that were manipulated in a certain way by her and/or other users in the system (e.g., select all ac-alternatives inserted by Lara). Finally, users can restrict the scope of the policy by specifying the relation and/or attribute within a relation in which the policy applies. Specifying the scope however is optional, thus allowing users to write policies that apply to all relations or all attributes within a relation. This satisfies *desideratum #1* on being able to express schema-independent policies.

Note that the syntax displayed in Figure 7 is used for internal representation purposes only and not for the Frontend of the system. Users can easily express policies through a visual resolution policy builder. Figure 8 shows a screenshot of the resolution policy builder expressing the minimum height policy described above.

Semantics. Given an ac-database I, a 'User Actions' store U (comprised of all 'User Actions' tables) and a resolution policy P, executing P on (I, U) returns a new User Ac-Database created from I by removing for each alternative a satisfying the policy's condition either the alternative (if the policy's outcome is 'remove this alt') or all other alternatives in the same fragment (if the policy is of type 'remove other alt'). This can be straightforwardly extended to a set of resolution policies. In this case each resolution policy P_i in the set is evaluated against the initial ac-database and 'User Actions' store leading to a set of ac-alternative deletions.


Edit Resolution Policy

Description

Select lowest height

Scope

Relation: Actors

Attribute: Height

Value-based Conditions

Attribute Value: Minimum value

Action-based Conditions

User

Modification Date

Modification Type

Outcome

Action: Remove all other alternatives

Figure 8: Screenshot of the Resolution Policy Builder

```
if action-type = marked wrong and
   action-user = Lara
then remove this alt
```

Figure 9: Default Resolution Policy for Lara

The result of running this set of policies against (I, U) is a new User Ac-Database constructed from I by carrying out the union of deletions described by the policies in the set. The order in which resolution policies are defined does not affect the result, as conceptually all of them are evaluated against the initial ac-database.

In Ricolla's implementation, the ac-database is stored as a relational database and the resolution policies are implemented as SQL queries evaluated over this relational representation. The implementation of resolution policies is described in detail in Section 5.3.

Expressiveness. The resolution policy language is expressive enough to capture many common use cases, including among others all examples mentioned in the paper together with several resolution schemes hardcoded into existing systems. Examples of the latter include the last-edit policy mentioned in the introduction or schemes where the community trusts a designated 'data curator'.

As mentioned above, Ricolla contains for each user a default policy in which a user's 'mark as wrong' resolution actions are interpreted as deletions. This default policy, depicted in Figure 9 for Lara, implements the expected semantics of resolution actions. However, a user can at any time discard this default policy or add new policies. All policies specified by the user will be executed in parallel to create the user's view of the community ac-database.

Finally, the resolution policy language allows for collaboration, while avoiding recursion (*desiderata #2* and *#3*). This is the result of its two-tiered architecture, which allows policies to operate on the 'User Actions' tables (to enable collaboration, as discussed) but not on other policies (thus avoiding recursion). Note that although a user's policy cannot operate on another user's policy, a user can still at any time make the effect of her resolution policies available to other users by having Ricolla translate it to a set of resolution actions. The translation is straightforward, as policies of type 'remove this/other alt' are translated to a set of 'mark as wrong/right' resolution actions, respectively.

4. QUERY ANSWERING

The resolve-as-you-go requirement of Ricolla dictates that users can get the answers to their queries even when the system contains conflicting data. Prior work, referred to as consistent query answering (see Section 6), suggests to return only the answers that are consistent w.r.t. the set of constraints expressed over the database schema. In contrast, Ricolla follows the approach taken by works on uncertain data [13, 9] and returns to the user all non-yet-resolved inconsistent data, pertaining to the query. The user may then resolve some of the inconsistencies in the query answer.

In this spirit, the query result is the set of *all possible query answers*. The possible query answering semantics are typical in works on uncertain data [13, 9]. In the case of Ricolla, they are formally defined as follows. In the following we assume bag semantics.

DEFINITION 4.1. POSSIBLE ANSWERS: *Given a query Q and an ac-database I, the possible answers to Q over I is the following set of possible worlds (over a single relation):*

$$PAnswers_Q(I) = \{Q(DB) | DB \in PWorlds(I)\}$$

Currently Ricolla supports queries out of the class CQ of conjunctive queries (i.e., select-project-join queries). If the query does not contain self joins and there is no uncertainty in the ac-database on the join attributes (typically the primary keys and foreign keys), then we call this a join-consistent $CQ_1^=$ query. For join-consistent $CQ_1^=$ queries, Ricolla returns an ac-relation that precisely encodes their possible answers. This is though not possible for CQ queries that fall outside this class, as it turns out that their possible answers cannot be represented as an ac-relation. Therefore, Ricolla answers such queries by returning an ac-relation that *approximates* their possible answers. Towards formalizing these statements, we start by defining the join attributes of a schema, which will in turn help us define the class of join-consistent $CQ_1^=$ queries.

DEFINITION 4.2. JOIN ATTRIBUTES: *Given an ac-database schema S, the set of* join attributes *is the set of attributes which are allowed to participate in query joins.*

As is well known, join attributes are typically primary keys and foreign keys. For our purposes it does not matter which exact pairs of attributes are going to be joined together but rather whether an attribute can appear in a join on not. Given a set of join attributes over a schema, an ac-database over the same schema is *join-consistent* if it does not have uncertainty in the value of the join attributes.

DEFINITION 4.3. JOIN-CONSISTENT AC-DATABASE: *An ac-database I over schema S is said to be* Join-Consistent *w.r.t. a set of join attributes over S, if it satisfies the following properties:*

- *For every ac-tuple in I all join attributes of the corresponding ac-relation appear within the same fragment of the ac-tuple and*
- *all alternatives of that fragment are identical when projected on the join attributes.*

For instance, in our running example we expect the users of the system to formulate queries joining on movie IDs and actor IDs. Therefore we consider as the set of join attributes the set J = $\{$*Movie.ID*, *Actor.ID*, *MovieActor.MovieID*, *MovieActor.ActorID*$\}$. The ac-database of Figure 4 is join-consistent w.r.t. J as it does not contain multiple values for any of the join attributes. For example, the ac-tuples in relation Movie contain only a single possible value for the movie ID (which is a join attribute).

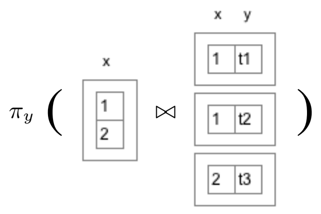


Figure 10: A (non-join-consistent) query Q_1 & instance I s.t. $PAnswers_{Q_1}(I)$ cannot be represented as an ac-relation

Next we define the class of join-consistent $CQ_1^=$ queries. These are CQ queries that (a) contain joins only on the join attributes and (b) do not contain self-joins.

DEFINITION 4.4. JOIN-CONSISTENT QUERY: *A query Q expressed over an ac-database schema S is called* Join-Consistent *w.r.t. a set of join attributes over the same schema, if Q contains joins only on the join attributes.*

DEFINITION 4.5. $CQ_1^=$: *The class $CQ_1^=$ contains all select-project-join relational algebra queries containing at most one instance of each relation. Selections involve equalities between attributes and constants.*

Having defined the set of ac-databases and queries of interest, we can now formulate the closure result, which proves that the result of a join-consistent $CQ_1^=$ query is an ac-relation:

THEOREM 4.6. CLOSURE: *Let J be a set of join attributes, Q a join-consistent (w.r.t. J) $CQ_1^=$ query and I a join-consistent (w.r.t. J) ac-database. Then the possible answers of Q over I can be represented by an ac-relation, which we denote by $Q(I)$.*

This result raises the following question: Is there a subset of CQ queries that is larger than join-consistent $CQ_1^=$ and whose results can still be described by ac-relations? It turns out that this is not the case for any straightforward extension of CQ. In particular, if we relax the restriction on either join-consistency or the absence of self-joins, we can find an ac-database I and a query Q s.t. the possible answers to Q over I cannot be represented as an ac-relation.

Figure 10 shows such a query Q_1 and ac-database I. It is easy to see that the set of the possible answers to Q_1 over I consists of the two possible worlds $p_1 = \{\langle t1\rangle, \langle t2\rangle\}$ and $p_2 = \{\langle t3\rangle\}$. Assume that we can represent $PAnswers_{Q_1}(I)$ as an ac-relation R. Since p_1 consists of two tuples, the ac-relation R has to contain at least two ac-tuples (as an ac-tuple can give rise to at most one tuple in a possible world). The first ac-tuple will contain $t1$ as an alternative and the second will contain $t2$. Moreover, R cannot contain additional ac-tuples as that would lead to a possible world with more than two tuples. Finally, since p_2 contains a single tuple, one of the ac-tuples in R has to have an optionality flag. There are two possibilities: If only a single tuple (let's say the one that has $t1$ as an alternative) has an optionality flag, then R will also model the possible world $\{\langle t2\rangle\}$. If on the other hand, both ac-tuples have an optionality flag, then R will also model the empty possible world. Both possibilities correspond to contradictions, since we have assumed that R precisely models $PAnswers_{Q_1}(I)$. Hence we have proven that the possible answers to a non-join-consistent query cannot in general be represented as an ac-relation.

This result however raises another important question: Since ac-relations cannot precisely represent the results of arbitrary CQ queries, should researchers instead look for another data model, which precisely captures the possible answers of CQ queries? The following theorem shows that such an effort would be futile, since the possible answers to CQ queries over an ac-database do not fall within a restricted class of possible worlds. Instead they can be an arbitrary finite set of possible worlds (over a single relation), requiring thus powerful and complex data models as the ones discussed in Section 2.3. In that sense, the combination of the ac-database data model with join-consistent $CQ_1^=$ queries is an optimized tradeoff between expressiveness and data model simplicity. The next more expressive trade-off point is data models that can represent every finite set of possible worlds (aka as complete data models).

THEOREM 4.7. *For every finite set P of possible worlds over a single relation, there exists an ac-database instance I and a CQ query Q such that $PAnswers_Q(I) = P$.*

This can be shown by proving that every U-relation (which is a complete model) can be modelled as the result of a CQ query over some ac-database instance. Given the futility of attempting to enlarge the ac-database model without sacrificing simplicity, we enabled Ricolla to answer CQ queries that fall outside the class of join-consistent $CQ_1^=$ by approximating their possible answers.

Approximating the query answer. In approximating the answer of an arbitrary CQ query as an ac-relation, we have in general two options: We can compute an ac-relation that either over-approximates or under-approximates the original set of possible worlds. For the purpose of Ricolla we took the route of an over-approximation, as this allows the end-users to see all possible worlds in the original set (with potentially some additional false positives).

DEFINITION 4.8. APPROXIMATION: *Given a finite set of possible worlds P over a single relation, an approximation of P is an ac-relation I such that $P \subseteq PWorlds(I)$.*

However approximations can be arbitrarily large. To this end, we define the notion of a *best approximation*, which is an approximation with the minimum number of ac-tuples. For the purposes of the following theorem we define the cardinality of an ac-relation I (denoted by $|I|$) to be the number of ac-tuples in I.

DEFINITION 4.9. BEST APPROXIMATION: *An approximation I of a finite set of possible worlds P over a single relation is a* best approximation *of P iff $|I| \leq |I'|$ for every approximation I' of P.*

Unfortunately, the discovery of the best approximation is impractical, since computing it is NP-hard:

THEOREM 4.10. BEST APPROXIMATION HARDNESS: *Computing the best approximation of a finite set of possible worlds represented as a U-relation is NP-hard.*

This can be shown through a reduction from MAX-2-SAT. Note that in the above theorem the set of possible worlds that we want to approximate is given as a U-relation. This is made to ensure that the input to the 'best approximation' problem is a compact representation of a set of possible worlds and not some arbitrarily large representation (e.g., an enumeration of the set of possible worlds) which would have led to an artificially low complexity.

Given the intractability of approximating a query answer, Ricolla utilizes a heuristic to compute some (non-best) approximation of the answer for all CQ queries that fall outside the class of join-consistent $CQ_1^=$. Of course, queries within that class are answered exactly. The query answering algorithm is described in Section 5.2.

tid	*fid*	*aid*	*ID*	*Name*	*Height*	*City*	*Zip*
1	1	1	1	ϵ	ϵ	ϵ	ϵ
1	1	1	ϵ	Clint...	ϵ	ϵ	ϵ
1	2	1	ϵ	ϵ	1.85	ϵ	ϵ
1	2	2	ϵ	ϵ	1.88	ϵ	ϵ
1	3	1	ϵ	ϵ	ϵ	Burbank	91522
1	3	2	ϵ	ϵ	ϵ	Carmel	93921

(a) Actor$_d$

tid	*optid*

(b) Actor$_{opt}$

Figure 11: Flat representation of the ac-tuple in Figure 4a

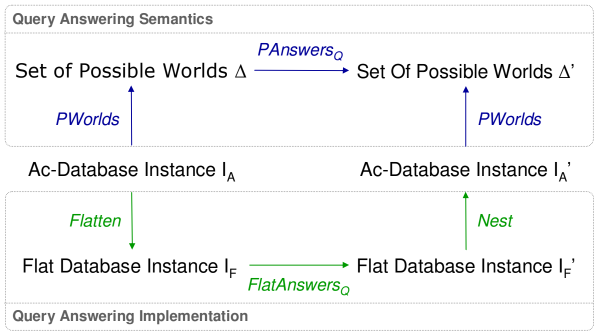

Query Answering Semantics
Set of Possible Worlds Δ
PAnswers$_Q$
Set Of Possible Worlds Δ'
PWorlds
PWorlds
Ac-Database Instance I$_A$
Ac-Database Instance I$_A$'
Flatten
Nest
Flat Database Instance I$_F$
Flat Database Instance I$_F$'
FlatAnswers$_Q$
Query Answering Implementation

Figure 12: Query Answering Semantics and Implementation

Finally, note that one can also envision other notions of best approximation, such as an approximation that is minimal in the set of possible worlds it represents or one that is the most compact in the number of data values it contains (as defined in Section 2.3). We plan to investigate these alternative definitions of best approximation as part of our future work.

5. IMPLEMENTATION

In a first iteration we implemented Ricolla on top of an RDBMS. The benefits from this approach are twofold: First, it leverages the query answering and optimization capabilities offered by RDBMSs. Second, it allows enterprises hosting Ricolla to reuse their existing infrastructure. As part of our future work, we plan to investigate alternative techniques of implementing the system.

The system consists of 4 main components described next: a) storing an ac-database as a flat database, called *flattening*, b) retrieving an ac-database from its flat representation, called *nesting*, c) *answering queries* and d) *executing resolution policies*.

5.1 Flattening & Nesting

Storing an ac-database in an RDBMS requires a procedure for converting each ac-relation to one or more flat relations. This procedure, called *flattening*, should be invertible to ensure that an ac-relation can be *nested* back from its flat representation.

A straightforward approach of flattening an ac-relation is to store for each ac-tuple its interpretations, as defined in Section 2.2 (augmented with information about the schema of each ac-tuple). However, creating the interpretations of an ac-tuple involves taking the cartesian product of its ac-fragments. Hence this approach leads to an exponential blowup in the space requirements of the flat relation, compared to the ac-relation it represents. This not only wastes memory resources but it also increases the time required to retrieve an ac-relation from its flat representation, since the nesting algorithm has to at least scan all tuples stored in the RDBMS.

To avoid this problem, we designed a flat representation that is linear in the size of the original ac-database. Each ac-relation R is converted to two flat relations: a *data relation* R_d storing the ac-alternatives of R's ac-tuples and an *optionality relation* R_{opt} holding their optionality flags.

Due to lack of space, we demonstrate the flattening procedure through an example. Figure 11 shows the flat representation of the Clint Eastwood ac-tuple of Figure 4a. The data relation, shown in Figure 11a, stores one flat tuple per ac-alternative. The special ϵ values are used to pad the ac-alternatives to fit the schema of an ac-relation (since in general an alternative covers only a subset of this schema). For each alternative Ricolla keeps three identifiers: a tuple identifier (*tid*), a fragment identifier (*fid*) and an identifier of the alternative within the fragment (*aid*). These serve two purposes: First, they capture the structure of each ac-tuple, thus allowing the system to reconstruct the original ac-relation from the stored ac-alternatives. Second, they (the fragment ids in particular) are used to capture the markers of ac-fragments; i.e., two fragments with the same marker share the same *fid* value. The only information not stored in the data relation are the ac-tuple optionality flags. These are kept in a separate optionality relation (shown in Figure 11b) storing for each ac-tuple (represented by its *tid*) the identifiers of all its optionality flags. Similarly to ac-fragments, the marker of an optionality flag is represented by its id. In other words, two flags with the same marker share the same id.

5.2 Answering Queries

In a two-layered system like ours where we have two data models (one for the Frontend and another for the Backend), query answering can be generally accomplished in two ways: By operating either on the Frontend data model (i.e., on the ac-database) or on the Backend model (i.e., on its flat representation). However running the query answering algorithm on the ac-database involves first recreating the *entire* ac-database from its flat representation. Therefore we opted for the second approach: A query Q_{ac} over the ac-database is rewritten to a set of queries that can be executed inside the RDBMS over the flat representation of the ac-database. In this way, we avoid nesting the entire database and in parallel we leverage the query answering and optimization capabilities of RDBMSs. Subsequently the flat query results (which correspond to the flat representation of the possible answers to Q_{ac}) are passed as input to the nesting algorithm to construct an ac-database representing the possible answers to Q_{ac}. Figure 12 shows both the semantics of query answering (in terms of possible worlds as explained in Section 4) and its actual implementation.

We describe next the rewriting procedure that translates a CQ query over an ac-database to two queries over its flat representation, returning the data and optionality relation. To create these flat queries, Ricolla appropriately rewrites each relational algebra operator of the original query. Figures 13a and 13b show the operator-level rewritings used to create the queries returning the data relation and the optionality relation, respectively. The join operator is special in that it is treated differently for the special type of joins that appear in join-consistent $CQ_1^=$ queries and for arbitrary joins. Based on the results of Section 4, showing that we cannot represent the query results in the second case as an ac-relation, the rewriting yields an ac-relation that represents in the first case the precise answers, while in the second only an approximation of them.[7]

[7]The rewritings use the generalized projection operator of [26], which in addition to attributes can also output function results.

$\sigma_{a=c}(R)$	$\sigma_{a=c \vee a=\epsilon}(R_d)$
$\pi_{a_1,\ldots,a_n}(R)$	$\pi_{tid,fid,aid,a_1,\ldots,a_n}(R_d)$
Join-consistent CQ$_1^=$	
$R \bowtie_{R.a_i=S.a_j} S$	$\pi_{nt(tid_R,tid_S),pad} R_d \bowtie_{tid=tid_R} K \cup$ $\pi_{nt(tid_R,tid_S),pad} S_d \bowtie_{tid=tid_S} K$
Arbitrary CQ	
$R \bowtie_{R.a_i=S.a_j} S$	$\pi_{nt(tid_R,tid_S),pad} \sigma_{a_i=\epsilon} R_d \bowtie_{tid=tid_R} K \cup$ $\pi_{nt(tid_R,tid_S),pad} \sigma_{a_j=\epsilon} S_d \bowtie_{tid=tid_S} K \cup$ $\pi_{nt(tid_R,tid_S),nf,pad} R_d \bowtie_{R_d.a_i=S_d.a_j} \sigma_{a_j \neq \epsilon}(S_d)$

(a) Rewriting for the data relation

$\sigma_{a=c}(R)$	$\delta\pi_{tid,nopt(fid)} \sigma_{a \neq c \wedge a \neq \epsilon}(R_d) \cup R_{opt}$
$\pi_{a_1,\ldots,a_n}(R)$	R_{opt}
Join-consistent CQ$_1^=$	
$R \bowtie_{R.a_i=S.a_j} S$	$\pi_{nt(tid_R,tid_S),optid}(R_{opt} \bowtie_{tid=tid_R} K) \cup$ $\pi_{nt(tid_R,tid_S),optid}(S_{opt} \bowtie_{tid=tid_S} K)$
Arbitrary CQ	
$R \bowtie_{R.a_i=S.a_j} S$	$\pi_{nt(tid_R,tid_S),nopt'}(K)$

(b) Rewriting for the optionality relation

where $K : \delta\pi_{R_d.tid \text{ as } tid_R, S_d.tid \text{ as } tid_S}($
$R_d \bowtie_{R_d.a_i=S_d.a_j} \sigma_{S_d.a_j \neq \epsilon}(S_d))$

Figure 13: Rewriting of relational algebra operators for CQ queries

A **projection** on an ac-relation simply translates to a projection on the data relation and it does not cause any modification to the optionality relation (since projecting out columns of an ac-relation cannot create new optionality flags or remove existing ones).

Selection is slightly more involved. Applying a selection on an attribute a with a value c keeps only those ac-alternatives with a value c for a. To implement these semantics, the rewriting of the selection operator selects from the data relation all flat tuples that have a value c or ϵ for a. The latter correspond to ac-alternatives that do not have a in their schema. Moreover, a selection may also introduce new optionality flags. In particular, whenever an input ac-tuple contains an alternative b not satisfying the selection condition, the corresponding output ac-tuple has to be marked as optional. The reason is that in one possible answer (the one produced by executing the query against the possible world in which b exists) this tuple will not exist. Therefore, as shown in Figure 13b, the rewriting of the selection operator for the optionality relation creates new optionality flags for ac-tuples that contain alternatives not satisfying the selection condition. The identifier of these flags is created by a function *nopt* (standing for *new optid*) that generates fresh optionality ids based on the *fid* of the fragment that contains such an alternative. This happens because if in the query input two fragments with the same marker contain an alternative that does not satisfy the selection condition, in every possible answer one tuple will exist iff the other exists. Thus they have to be assigned optionality flags with the same marker (and hence with the same id).

Finally, for the **join**, we distinguish two cases: Joins on attributes of different relations that do not contain uncertainty (i.e., joins that appear in join-consistent $CQ_1^=$ queries) and arbitrary joins.

In the first case, the lack of uncertainty on the join attributes means that for any two ac-tuples $t1$ and $t2$, all interpretations of $t1$ will join with all interpretations of $t2$. Therefore to create the result of the join between $t1$ and $t2$ it suffices to create a new ac-tuple that contains the concatenation of the fragments of the original ac-tuples. The new ac-tuple also inherits the optionality flags of the two input tuples. The rewriting of the join operator for the data relation shown in Figure 13a implements these semantics as follows: The intermediate relation K (shown at the bottom of Figure 13) computes pairs of identifiers of tuples that agree on the join attributes. Subsequently, for each such pair the rewriting retrieves the alternatives of the corresponding tuples and pads them with ϵ values to make them conform to the schema of the join result. The function *nt* creates a fresh tuple id for each pair of joined input tuples. Finally, the rewriting for the optionality relation shown in Figure 13b employs relation K to copy the optionality flags of each input tuple to all output tuples that it helped produce.

In this case of arbitrary joins on the other hand, the existence of uncertainty on the join attributes means that only some of the interpretations of $t1$ and $t2$ will join with each other. To find the ones that join, Ricolla computes the join between the fragment of $t1$ that contains the join attributes and the corresponding fragment of $t2$. The remaining fragments of both tuples (i.e., the ones that do not contain join attributes) carry over unmodified to the output ac-tuple as in the case of restricted joins. Moreover, since only some of the interpretations of the input tuples join with each other, each ac-tuple in the join output is marked as optional with a fresh optionality flag (produced through the function *nopt'*).

Using the above operator-level rewritings, Ricolla translates any CQ query Q_{ac} over an ac-database schema to two queries Q_d and Q_{opt} over the corresponding flat schema. These queries are then used to compute the ac-database that represents the possible answers to Q_{ac} over an ac-database instance (in the case of join-consistent $CQ_1^=$ queries) or an approximation of them (in the case of arbitrary CQ queries).

5.3 Executing Resolution Policies

Similarly to queries, resolution policies are applied directly on the flat representation of the ac-database. Ricolla translates resolution policies to appropriate flat queries. Executing these queries over the flat representation of an ac-database I yields the flat representation of the result of applying the original resolution policies on I. The generated queries (omitted due to lack of space) operate on the data relation and the optionality relation and find all ac-alternatives that satisfy the conditions set by the resolution policy. Then they either remove them from the attribute tables (if the policy specifies a 'remove this alt' outcome) or they remove all other ac-alternatives (for policies of type 'remove other alt'). The main difference between the relational queries generated for user queries and those for resolution policies, is that the latter can also operate on the *UserActions* relation, which stores the contents of the 'User Actions' tables of all alternatives.

Once Ricolla translates a policy down to relational queries, it can generate the User View. However the latter is not materialized. Instead, when a user asks a query, her query (translated to queries over the store) is composed with the queries corresponding to the resolution policy. By not materializing the User View, Ricolla can on-the-fly create only the part of the view of interest to the queries.

6. RELATED WORK

Researchers have looked at several aspects of the problem of managing conflicting data:

Querying inconsistent data. A significant amount of work [11, 16] (nicely summarized in [14]) focuses on querying a relational database that contains inconsistent data. To this end, they propose the *Consistent Query Answering* semantics. According to them, a query returns the set of tuples that appear on the answers to the query against *all* minimal repairs of the original database. A minimal repair of a database is a way to convert it to a consistent database with the least amount of changes. Intuitively the consistent query answers are those tuples that are *guaranteed* to exist, regardless of how the inconsistency is resolved.

Ricolla on the other hand, provides more informative answers, containing also inconsistent data. This allows users to inspect conflicts and resolve them. Note that the consistent query answers can be inferred from the possible answers through a linear scan: They correspond to ac-tuples with a *single* alternative in each fragment.

Recently, these frameworks were extended to take only *preferred repairs* into account [27]. Preferences are specified through rules resembling Ricolla's resolution policies.

Modeling inconsistent data. A lot of works proposed data models for uncertain data. However, as explained in Section 2.3, these models are not suitable for the Frontend of an online database as they trade simplicity and compactness for expressive power.

End-to-end systems for managing inconsistent data. Several systems were proposed as attempts to solve the problem of inconsistent data management. However they cannot be used effectively in our setting. ORCHESTRA [28] allows users to reconcile data while allowing disagreement. However, in contrast to Ricolla, it applies to P2P scenarios, where community users have their own local databases. Moreover, it takes inconsistencies into account and tries to resolve them *only* when a user decides to reconcile the data published by their peers. After each reconciliation period, local databases are consistent (and conflicts that could not be resolved discarded). In contrast, Ricolla's goal is to always keep and display to the user all inconsistencies, until she can resolve them. Other systems, such as HumMer [22] and Fusionplex [21] allow inconsistency resolution in the context of data fusion. They provide resolution policy languages but they lack a *formally defined* model for displaying inconsistent data. Moreover they are designed for a *single* user and are thus not applicable in collaborative scenarios.

Resolving conflicts based on trust. Recently, [17] looked at the problem of resolving data conflicts given a trust network defined between users. The difficulty of the problem arises from the fact that the trust network can be any graph that might contain long paths (a sequence $u_1, \dots u_n$ of users such that u_i trusts u_{i+1}) and even cycles (i.e., two users that trust each other). In contrast, in this work we only consider for each user a graph with paths of length one: Each user explicitly states through her resolution policies the set of users that she trusts. Thus resolving the conflicts for her does not conceptually involve traversing a complex trust network and thus we do not run into the problem of defining and computing the stable semantics encountered in [17]. This would change if we were to allow recursive policies but, as explained in Section 3.3 this is out of the scope of this work.

7. CONCLUSION

In this paper, we present Ricolla; an end-to-end online database tuned for conflict resolution in a pay-as-you-go fashion. While being formally grounded, Ricolla enables a natural way for the management of conflicting data: Users can model and inspect conflicts (through the ac-database), query the data (based on the closure properties of the ac-database) and resolve data either collaboratively or individually (through resolution actions and policies). As part of our future work, we plan to increase Ricolla's expressivity (e.g., by considering more expressive query languages and other types of approximations) as well as investigate alternative implementations (native or on top of existing systems).

8. REFERENCES

[1] Caspio Bridge. http://www.caspio.com/bridge/.

[2] Google Fusion Tables. http://www.google.com/fusiontables/.

[3] QuickBase. http://quickbase.intuit.com/.

[4] TrackVia. http://www.trackvia.com/.

[5] Zoho Creator. http://creator.zoho.com/.

[6] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[7] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. U. Nabar, T. Sugihara, and J. Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[8] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, pages 983–992, 2008.

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

[10] L. Antova, C. Koch, and D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions. In *ICDE*, pages 1479–1480, 2007.

[11] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.

[12] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.

[13] O. Benjelloun, A. D. Sarma, A. Y. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2):243–264, 2008.

[14] J. Chomicki. Consistent query answering: Five easy pieces. In *ICDT*, 2007.

[15] N. N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.

[16] A. Fuxman, E. Fazli, and R. J. Miller. Conquer: efficient management of inconsistent databases. In *SIGMOD*, 2005.

[17] W. Gatterbauer and D. Suciu. Data conflict resolution using trust mappings. In *SIGMOD*, pages 219–230, 2010.

[18] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[19] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[20] L. V. S. Lakshmanan, N. Leone, R. Ross, and V. S. Subrahmanian. Probview: a flexible probabilistic database system. *ACM Trans. Database Syst.*, 22(3):419–469, 1997.

[21] A. Motro and P. Anokhin. Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. *Inf. Fusion*, 7(2):176–196, 2006.

[22] F. Naumann, A. Bilke, J. Bleiholder, and M. Weis. Data fusion in three steps: Resolving schema, tuple, and value inconsistencies. *IEEE Data Eng. Bull.*, 29(2):21–31, 2006.

[23] W. C. Purdy. A logic for natural language. *Notre Dame Journal of Formal Logic*, 32(3):409–425, 1991.

[24] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, page 7, 2006.

[25] R. A. Schmidt. Relational grammars for knowledge representation. In *Variable-Free Semantics*, volume 3 of *Artikulation und Sprache*, pages 162–180. secolo Verlag, Osnabrück, Germany, 2000.

[26] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, fourth edition.

[27] S. Staworko, J. Chomicki, and J. Marcinkowski. Preference-driven querying of inconsistent relational databases. In *EDBT Workshops*, pages 318–335, 2006.

[28] N. E. Taylor and Z. G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, pages 13–24, 2006.