With Great Freedom for Inconsistent Data Comes Great Scalability Responsibility

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 community members to collaboratively maintain and browse data. While users may believe in conflicting facts (due to conflicting sources, measurements or opinions), current online databases do not offer support for the management of data conflicts.

Thus online databases could clearly benefit from technology for uncertain/incomplete databases. However, prior works on uncertain databases are of limited help when designing a conflict-aware online database, for two reasons: First, their performance degrades rapidly as the number of conflicting facts escalates, which can be the case in large user communities. Second, they were built as storage models, resulting in data models that are either non-simple or non-compact and thus may require additional, often non-trivial processing before they appear in the Frontend of an online database.

To overcome these problems, we describe Ricolla; a scalable online database with built-in support for data conflict management. Ricolla allows users to model conflicting data, inspect them in a compact form and resolve inconsistencies in an "as-you-go" personalized fashion, even in the presence of a large number of conflicts. It achieves this by coordinating a novel conflict-aware data model (shown to be both compact and simple) with respective efficient query answering algorithms, that allow the system to scale to a large number of data conflicts (as evidenced by a performance comparison against Trio and MayBMS; two recent research prototypes for uncertain data). In parallel, the data model's simplicity and compactness make it suitable for direct use by the Frontend.

1. INTRODUCTION

Lately, *online databases* (Google Fusion Tables [3], QuickBase [4], Zoho Creator [6], Caspio Bridge [1], TrackVia [5] and many more) enable online communities to collaboratively maintain their data. Multiple users simultaneously enter and read data in the online database via its web interface. We argue next that the freedom that characterizes the use of online databases fits perfectly the motivation of uncertain/incomplete databases [10, 11, 15, 17, 20, 21, 22] and, at the same time, highlights two yet-unaddressed challenges: (a) creating a model and query answering algorithm for uncertain data that scale in the number of contradicting facts present in the database and (b) ensuring that the data model is simple and compact to be directly used by the Frontend of an online database.

The users of an 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 (where 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 [32], researchers have contradicting opinions (e.g., about 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.

Online databases enable posting data in an unrestricted fashion, whereas a registered user can post data that may conflict with the data posted by other users. However, current online databases offer no or very limited support for inconsistency, falling in general in one of the following two categories: (a) they disallow (some or all) conflicts altogether by allowing the administrator to enforce a set of integrity constraints, or (b) they ignore inconsistencies, thus allowing conflicting data to be entered into the system but they do not provide the users with the tools to query conflicting data. Online databases can clearly benefit from technology for uncertain data [10, 11, 15, 17, 20, 21, 22, 29, 31, 25].

The presented system, called Ricolla (**R**esolve **I**nconsistencies in a **COLLA**borative environment), combines uncertain databases with the online database paradigm, where a central editor authority is not present: It allows each user to post data, potentially inconsistent with the data of other users, and overview inconsistencies (in either the source data or the query results) in a compact, easy-to-read representation. Furthermore, it allows the user to individually resolve inconsistencies, possibly disagreeing with other users' resolutions. This resolution happens in an "as-you-go" fashion, allowing users to browse and query the data even before all inconsistencies are resolved.

Combining the typical usage paradigm of online databases with uncertain data though raises a yet-unaddressed scalability challenge: The freedom to introduce multiple conflicting opinions may be actually exercised by the users, escalating the number of inconsistencies in the database. We show that with the current state-of-the-art in uncertain databases, as the number of conflicting opinions raises, this leads quickly to rapid query processing performance degradation. Furthermore, the prior literature has not suggested an effective Frontend for representing uncertain data.

Ricolla solves the scalability problem by tuning the data model, query answering algorithms and user interface. The first ingredient to the solution is a novel data model, called *ac-database*, and a corresponding generic report & update interface to capture and rep-

resent 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 (known 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 scalability, while retaining the simplicity needed by online databases. 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 other data models, such as *c-tables* [21], *ULDBs* [15] (used in the Trio system [9]) and *U-relations* [10] (used in the MayBMS system [12]).

We show that Ricolla's data representations are more compact than other data models, such as ULDBs. The compact representation enables a specially-tuned query answering algorithm, which maintains compactness and is therefore scalable for a large class of queries, named *join-consistent* $CQ_1^=$. We show both analytically and experimentally that Ricolla is superior in inconsistency-scalability to MayBMS and Trio, which incur an exponential blowup in the data size. This superiority stems from the fact that, as opposed to Trio and MayBMS, which are universal tools meant to accommodate any set of possible worlds, Ricolla exploits the fact that the ac-database captures only the restricted class of possible worlds which corresponds to our inconsistent online database setting.

Having shown that Ricolla's data model and query answering achieve scalability and simplicity for *join-consistent* $CQ_1^=$, we formally show that this trade-off between scalability, simplicity and query expressive power is effectively optimized in the sense that any attempt to expand beyond *join-consistent* $CQ_1^=$ will lead to query results where the inconsistency representation will either have to be non-compact or will be non-simple (as in c-tables) or will be an approximation of the actual result.

Contributions. In summary, Ricolla makes the following contributions: (a) an online database that captures conflicts, allowing data query and update, while enabling personalized, "as-you-go" conflict resolution, (b) a data model for capturing the conflicts in a compact form, (c) a simple user interface, in direct correspondence to the data model, for explaining data conflicts, aligned with the interaction paradigm of online databases, (d) a formalization of the "simplicity" and "compactness" properties, followed by proof that Ricolla's data model achieves a "sweet spot" in the tradeoff between simplicity, compactness and query expressive power for an identified wide class of queries, and a corresponding scalable query answering algorithm, (e) a proof that this sweet spot is optimized in the sense that queries beyond this identified class necessarily lead to a loss of compactness or simplicity, (f) an implementation of the system on top of an RDBMS and (g) an analytical and an experimental performance comparison between Trio, MayBMS and Ricolla, which show that Ricolla scales exponentially better than both Trio and MayBMS to large numbers of data conflicts.

Organization. The paper is organized as follows: In Section 2 we present Ricolla through a sample use case and explain its architecture. In Sections 3-5 we present its components; its data model (Section 3), the allowed user actions (Section 4) and the query answering mechanism (Section 5). Its implementation is described in Section 6 followed by a performance comparison against MayBMS and Trio in Section 7. Finally, in Sections 8 and 9 we discuss related work and conclude, respectively. All proofs can be found in the appendix.

2. SYSTEM OVERVIEW

In the following we provide an overview of Ricolla's functionality through a representative use case before describing its architecture.

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

Figure 1: Schema of Movie Ac-Database

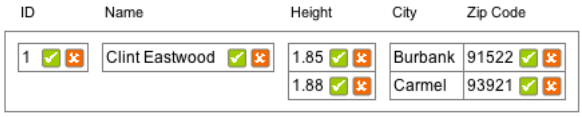


Figure 2: Ac-tuple for Clint Eastwood

2.1 User Interaction Example

We consider a community of cinephiles, who want to create an online database to collaboratively edit information about movies. The database's schema, 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. However, by inspecting the database, she finds out that some other user has listed Clint's height as 1.88m.[1] Instead of *replacing* the other user's value (which might be the correct one) by her own and thus creating a biased database, Ricolla allows her to simply *augment* the existing data 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 3, 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 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 query result is 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.

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 Burbank. She can reflect this knowledge in 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.

Personalized Resolution. 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

[1]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.

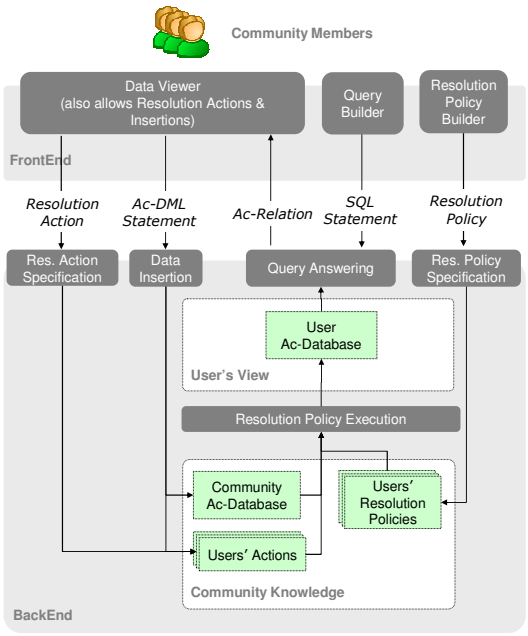


Figure 3: Ricolla Architecture

see both of Clint's addresses and could mark a different address as right. In general, Ricolla allows users to record and maintain differing opinions and beliefs, in effect creating their own *personalized* views of the community database. This is a major requirement especially in the sciences [32]. 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.

As-you-go Resolution. A notable feature of Ricolla's 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 eagerly and instead lazily resolve only those that affect queries of interest. In our running example, if Lara uses the query result in Figure 5 to set Clint's height to 1.85m, Ricolla will translate this choice to an appropriate resolution on the Actor table on Lara's behalf.

Bulk Resolution using Policies. 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 reported heights, she decides to choose for every actor the smallest among the heights listed. She could go to each actor tuple and manually select the minimum height, but Ricolla alternatively allows her to employ the *resolution policy builder* to write a policy that automatically selects for every actor the minimum height.

Collaborative Resolution. Ricolla allows a user to collaborate with her peers in conflict resolution, by specifying a resolution policy that takes into account other users' 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.

2.2 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). Ricolla is implemented on top of a relational DBMS. Its implementation, including the algorithms used for efficient query answering, are described in Section 6.

Although the current implementation supports a resolution policy language, designing resolution policies is an entire complementary area of research that includes among others the multitude of recommendation algorithms suggested in the literature [8]. Their investigation, as well as the description of the policy language, are beyond the scope of this paper. For the purpose of this paper, a resolution policy is any algorithm that operates over the community database and the users' actions and creates a personalized view of the community database. In the absence of any other policy, the system uses the default policy, which removes from each user's view all ac-alternatives that she has marked as wrong.

3. 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 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 3.3, such models have been created as general-purpose representations of sets of possible worlds and are thus not optimally tailored for performance in online databases.

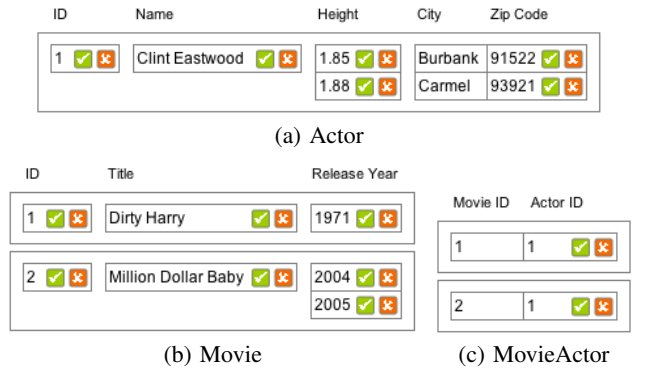
Next, we describe the ac-database, define its semantics and finally compare it to previously proposed models for uncertain data.

3.1 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 into a set of nested tables (4 in this case), called *ac-fragments*, which cover part of the ac-tuple's schema. Each ac-fragment row represents a possible assignment of values for the set of attributes in that ac-fragment and is called an *ac-alternative*. For instance,



(a) Actor

(b) Movie (c) MovieActor

Figure 4: Ac-database of movie community

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

the right-most ac-fragment, contains two ac-alternatives for Clint's city and zip code pair: 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 [7]. As we will explain in Section 3.3, the ac-tuple's structure and its partitioning in fragments guarantees that the data model stays compact and simple.

Correlating Ac-Tuples. An ac-relation is 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 even if an ac-relation starts out with uncorrelated ac-tuples, queries over it will yield correlated ac-tuples. 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). 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.

Marking fragments is more expressive than simply grouping tuples by a set of attributes. 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 can be marked to express correlations. In the current implementation markers of optionality flags are depicted 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 not in Burbank) the other will also not exist. Generally, 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 3.1. AC-DATABASE & AC-RELATION: *An ac-database consists of a set of ac-relations, which are sets of ac-tuples. Similarly to flat relations, each ac-relation has an associated schema, which is a set of attributes.*

DEFINITION 3.2. 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 3.3. 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).*

3.2 Possible World Semantics

The formal semantics build on the widely-followed definition of possible worlds. Each ac-database (or ac-relation) represents a set of possible flat databases (or flat relations). Each such flat database (or relation) is 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). Every such pick of ac-alternatives for a given ac-tuple is called an *interpretation* of that ac-tuple, defined below:

DEFINITION 3.4. 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 3.5. 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 3.6. NON-EXISTENCE ASSIGNMENT: *Given a set S of optionality flags, a 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 3.7. 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 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.

3.3 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* [21]. The problem has gained traction again recently in uncertain and incomplete databases. This led to new data models, such as *Uncertainty-Lineage Databases (ULDBs)* [15] (proposed in the context of the Trio system [9]), *World-Set Decompositions (WSDs)* [11] and U-relations [10] (both designed for the MayBMS system [12]) and semiring-annotated relations [20]. The same problem has also been studied in the context of probabilistic databases [17, 22, 31, 29, 25]. However, most models have been created as general-purpose representations and are not optimally tailored for online databases, whose scalability and interface needs place 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 (or some other equivalent mechanism that requires reasoning) to capture correlations in the data. As some studies [26, 28][2] 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, U-relations, as well as various data models introduced in the context of probabilistic database systems, such as PrDB [29] (where the correlations are captured through probabilistic graphical models), Orion [31] (which keeps the "history" of each tuple) and SPROUT [25] (which captures correlations through boolean formulas). 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 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 the 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 [10], it was noted that U-relations can also be exponentially more succinct than ULDBs for similar reasons.

THEOREM 3.8. COMPACTNESS: *For any set of possible worlds S_{PW} that can be represented as an ac-database, there exists an ac-*

[2]References borrowed from [27].

database representation that is at least as compact as any ULDB representation of S_{PW}.

c-tables and U-relations offer a more compact representation than ac-databases, if variables and formulas do not count on the metric. This is the result of using formulas to control the appearance of data values.

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 [27]. 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 (although, as our experiments in Section 7 show, these may introduce exponential overhead, which led us to the design of our own scalable Backend, described in Section 6).

4. ACTIONS

Utilizing an interface driven by the ac-database model, Ricolla allows users to both model conflicting data and subsequently resolve the conflicts. These tasks are supported by Ricolla's insertion actions and resolution actions, respectively, as described below.

4.1 Insertion Actions

A user models conflicting data by carrying out insertion actions directly on the interface (denoted as "Data Viewer" 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. If instead 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 or split 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 or split anymore.

4.2 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 below).

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 3.3), 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 [3], users can carry out resolution actions not only on the base 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 [14].

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.[4] Subsequently, each user can write a resolution policy over both the ac-database and the 'User Actions' tables to create her own view of the database, on which her queries are evaluated. For the purpose of this paper, a resolution policy is any algorithm that takes as input the ac-database and the 'User Actions' tables and returns a view of the ac-database. In the absence of an explicitly defined policy, Ricolla uses the default policy, which removes from each user's view the alternatives that she has marked as wrong, thus implementing the expected semantics of resolution actions. However, using an expressive policy language, 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 users to decide independently which community data they see.

5. QUERY ANSWERING

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

[4]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.

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 8), 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 [15, 11] 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 [15, 11]. In the case of Ricolla, they are formally defined as follows. In the following we assume bag semantics.

DEFINITION 5.1. 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 in general not possible for queries that fall outside this class, i.e. their possible answers may not be representable 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 5.2. 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 5.3. 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 for every ac-tuple in I all join attributes of the corresponding relation appear within a single fragment that has a single ac-alternative.*

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).

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 5.4. 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.*

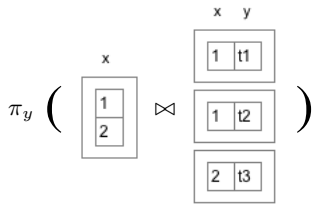


Figure 7: A (non-join-consistent) query Q_1 & instance I s.t. $PAnswers_{Q_1}(I)$ cannot be represented as an ac-relation

DEFINITION 5.5. $CQ_1^=$: *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 5.6. 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 7 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 3.3. 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 (a.k.a. as complete data models).

THEOREM 5.7. *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 (proof in the appendix).

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 I, we have in general two options: I either over-approximates or under-approximates the actual set P of worlds (i.e. I represents a superset or subset of P, respectively). For Ricolla we chose over-approximation, as it allows users to see all actual worlds (potentially with additional false positives). The effect on the Frontend is that it may not depict some correlations between ac-tuples, possibly suggesting more resolution actions than needed (resolving an ac-fragment actually resolves all fragments correlated with it). However, the Frontend is guaranteed to *never* miss a possible tuple (which would have happened for under-approximation).

DEFINITION 5.8. 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 5.9. 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 5.10. 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 (proof in the appendix). Note that in the 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, $CQ_1^=$ queries are still answered exactly by the same algorithm, which degrades gracefully when applied to queries outside the $CQ_1^=$ class. The query answering algorithm is described in Section 6.2.

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 most compact in the number of data values it contains (as defined in Section 3.3). We plan to investigate alternative notions of best approximation in our future work.

tid	*fid*	*aid*	*ID*	*Name*	*Height*	*City*	*Zip*
1	1	1	1	ϵ	ϵ	ϵ	ϵ
1	2	1	ϵ	Clint...	ϵ	ϵ	ϵ
1	3	1	ϵ	ϵ	1.85	ϵ	ϵ
1	3	2	ϵ	ϵ	1.88	ϵ	ϵ
1	4	1	ϵ	ϵ	ϵ	Burbank	91522
1	4	2	ϵ	ϵ	ϵ	Carmel	93921

(a) Actor$_d$

tid	*optid*

(b) Actor$_{opt}$

Figure 8: Flat representation of the ac-tuple in Figure 4a

6. 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 3 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*, and c) *answering queries*.

6.1 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 3.2 (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 8 shows the flat representation of the Clint Eastwood ac-tuple of Figure 4a. The data relation, shown in Figure 8a, 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

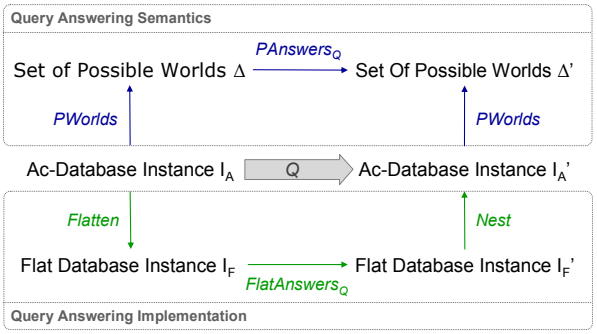


Figure 9: Query Answering Semantics and Implementation

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 8b) 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.

6.2 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 9 shows both the semantics of query answering (in terms of possible worlds as explained in Section 5) 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 10a and 10b 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 5, 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.[5]

A **projection** on an ac-relation translates to a projection on the

[5]The rewritings use the generalized projection operator of [30], which in addition to attributes can also output function results.

$\pi_{a_1,\ldots,a_n}(R)$	$\pi_{tid,fid,aid,a_1,\ldots,a_n}\sigma_{a_1\neq\epsilon\vee\ldots\vee a_n\neq\epsilon}(R_d)$
$\sigma_{a=c}(R)$	$\sigma_{a=c\vee a=\epsilon}(R_d)$
Join-consistent $\mathbf{CQ_1^=}$	
$R \bowtie_{R.a_i=S.a_j} S$	$\pi_{nt(tid_R,tid_S),pad}R_d \bowtie_{tid=tid_R} K \cup$ $\pi_{nt(tid_R,tid_S),pad}S_d \bowtie_{tid=tid_S} K$
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

$\pi_{a_1,\ldots,a_n}(R)$	R_{opt}
$\sigma_{a=c}(R)$	$\delta\pi_{tid,nopt(fid)}\sigma_{a\neq c\wedge a\neq\epsilon}(R_d) \cup R_{opt}$
Join-consistent $\mathbf{CQ_1^=}$	
$R \bowtie_{R.a_i=S.a_j} S$	$\pi_{nt(tid_R,tid_S),optid}(R_{opt} \bowtie_{tid=tid_R} K) \cup$ $\pi_{nt(tid_R,tid_S),optid}(S_{opt} \bowtie_{tid=tid_S} K)$
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 10: Rewriting of relational algebra operators for CQ queries

data relation and a removal of all flat tuples that correspond to alternatives that do not contain any of the attributes in the projection list. Moreover, a projection does not affect 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 10b, 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 10a implements these semantics as follows: The intermediate relation K (shown at the bottom of Figure 10) 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 10b employs relation K to copy the optionality flags of each input tuple to all output tuples that it helped produce.

For arbitrary joins, 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 do, 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 by 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, where Q_d computes the ac-tuples and Q_{opt} their optionality flags. The queries are guaranteed to compute the ac-database that represents the possible answers to Q_{ac} (in the case of join-consistent $CQ_1^=$ queries) or an (over-)approximation of them (in the case of arbitrary CQ queries).

Resolution policies and query answering. Recall that queries do not operate directly on the ac-database but on the result of a user-defined resolution policy. Thus to answer a query, Ricolla first applies the policy (which can be any algorithm, that given the flat representation of the ac-database and the 'User Actions' tables returns a view over it) and then runs the query answering algorithm on its result. However, this requires first materializing the entire policy result. To avoid this overhead, Ricolla also supports policies that are relational views. In this case, the user query is composed with the SQL view corresponding to the policy, to create on-the-fly only the part of the policy output relevant to the user query.

7. EXPERIMENTAL EVALUATION

As explained in Section 3.3, Ricolla employs the ac-database as its Frontend data model to achieve a simple and compact representation of conflicting data suitable for an online database. At its Backend it could conceptually employ any system for representing sets of possible worlds. However, we show next experimentally that it is still preferable to use a custom Backend (described above) specifically tailored to the storage of an ac-database, as existing systems do not scale with the number of conflicts in the online database. In particular, we show that Ricolla scales exponentially better w.r.t. uncertainty than two recent systems for uncertain data.

The compared systems. We compared Ricolla's Backend against two recent research prototypes for uncertain data: MayBMS [12] and Trio [9]. Trio is based on the ULDB data model [15], which (see Section 3.3) stores for each ac-tuple all its possible interpretations (i.e., the cartesian product of the ac-tuple's fragments). This leads to a blowup of the size of the base data which is exponential in the number of attributes in the base relations. MayBMS' data model, called U-relations [10], avoids this issue by allowing the administrator to manually vertically partition an ac-relation into multiple flat relations. Each flat relation U_i has intuitively the same effect as an ac-fragment; it still stores the possible interpretations of a tuple t but in this case only for the attributes of t that appear in U_i's schema. Thus U-relations with vertical decomposition avoid the exponential blowup in the size of the base data when the data are stored in the system. However, the exponential blowup still occurs at query processing time, since the query answering algorithm gradually merges vertical decompositions to eventually return a single non-decomposed U-relation [10].

```
Actor(ID, Name, Sex, BirthDate, Height, Weight)
Film(ID, Name, InitialReleaseDate)
Performance(ID, ActorID, FilmID, Character)
```

(a) Data Set Schema

Relation	Size (in tuples)
Actor	10,000 tuples
Performance	30,635 tuples
Film	25,223 tuples

(b) Data Set Size

Figure 11: Data Set Properties

Q_1: Return all actors
```
SELECT Name, BirthDate, Sex, Height, Weight
FROM Actor
```

Q_2: Return female actors (Selectivity 17%)
```
SELECT Name, Sex, BirthDate, Height, Weight
FROM Actor
WHERE Sex = 'Female'
```

Q_3: Return info on Charlize Theron (Selectivity 0.01%)
```
SELECT Name, Sex, BirthDate, Height, Weight
FROM Actor
WHERE Name = 'Charlize Theron'
```

Q_4: Return actors & characters played
```
SELECT Name, Character
FROM Actor A, Performance P
WHERE A.ID = P.ActorID
```

Q_5: Return actors & their movies
```
SELECT Film, Name, Sex, BirthDate
FROM Actor A, Performance P, Film F
WHERE A.ID = P.ActorID AND F.ID = P.FilmID
```

Figure 12: Queries

Data set and queries. To compare the systems, we used a modified version of our movie-community example based on real data from Freebase [2]. From the latter we extracted a subset of actor and movie data and converted them to relational data conforming to a schema with relations *Actor*, *Film* and *Performance* (*Performance* models the relationship between the other two relations). The schema and size of the resulting dataset is shown in Figure 11.

On this dataset, we compared the execution times of five queries Q_1-Q_5 shown in Figure 12. These queries were designed to (a) be representative examples of queries encountered in an online database and (b) cover the entire spectrum of constructs supported by our query language (i.e., projections, selections and joins). Q_1 returns the entire *Actor* relation, while Q_2 and Q_3 return subsets of it with increasing selectivity. Finally Q_4 and Q_5 join the *Actor* relation with one and two other relations, respectively.

The experiments were conducted on a Virtual Machine running Windows 7 with a 2.4GHz Intel Core i5 CPU and 1GB RAM with the latest publicly available packaged versions of the systems available at the time of writing.[6] Trio runs on Postgres 8.2.23 and MayBMS is a modified version of Postgres 8.3.3. To avoid discrepancies caused by different DBMSs, we ran Ricolla on the same

[6]There is also a source version of MayBMS available, which is claimed to be more efficient than the packaged version that we tested. However, we do not expect it to solve the scalability problems, since these are inherent in the system as we explained above.

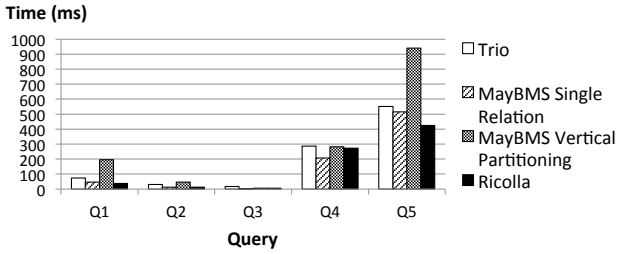


Figure 13: Query execution times in the absence of uncertainty

DBMS used by MayBMS. We report the time needed to execute the SQL queries issued by each system and fetch the results, as returned by Postgres. Note that this might be more than one SQL query (e.g., in the case of Ricolla it is two SQL queries returning the data and the optionality relations). Trio is the only system in which we report the time to execute a modified version of the SQL queries issued by the system. The reason is that Trio asks the DBMS to order the results, so that they can be easily parsed by the Frontend and grouped into Trio's equivalent of ac-tuples. To avoid penalizing Trio for Frontend processing (which has to happen also in the other systems), we measured the time of executing the queries without the ORDER BY clause. All times are averages over four executions with a warm cache of one execution.

Experiment 1: Performance in the absence of uncertainty. As a baseline, we evaluated the performance of all three systems in the absence of uncertainty. To this end, we imported the data set in each system creating an uncertain database representing a single possible world. In the case of Ricolla the import yielded ac-tuples with a single ac-fragment containing a single ac-alternative. Similarly for Trio. For MayBMS, since one can specify different vertical decompositions, we used two data representations, corresponding to the two extremes in the spectrum of decomposition. The first implementation, called "MayBMS Single Relation" stored the *Actor* relation as a single U-relation (similarly to Trio's ULDB), while the second, named "MayBMS Vertical Decomposition" decomposed it into 6 U-relations (one per attribute). The *Film* and *Performance* relations were stored in both cases as single U-relations.

Figure 13 shows the query execution times on each of the four resulting databases: Trio, MayBMS Single Relation, MayBMS Vertical Partitioning and Ricolla. All systems apart from MayBMS Vertical Partitioning perform similarly, which shows that Ricolla behaves like the state of the art in the absence of uncertainty. The slowdown of MayBMS Vertical Partitioning is due to the overhead of joining multiple partitions. For non-selective queries this overhead becomes significant even when 4-5 columns are projected.

Experiment 2: Scaling with uncertainty. Since in an online database, users are expected to exercise their freedom in introducing conflicting data, leading to a big amount of uncertainty, we next evaluated how the systems scale w.r.t the number of conflicts in the data. To this end, we augmented the *Actor* relation of the original dataset D_1 with random alternatives, thus creating five new datasets D_2, D_4, D_8, D_{16}, D_{32} with increasing amounts of uncertainty. To create a dataset D_i, we added to each ac-tuple of *Actor* in D_1 an average of i ac-alternatives *per attribute* (apart from the join attributes, shown underlined in Figure 11, which can contain only one alternative). For instance, in D_8, each *Actor* ac-tuple contains 6 fragments (one for each attribute), each with an average of 8 alternatives (apart from the fragment of the join attribute ID which contains a single alternative). Thus each *Actor* ac-tuple in D_i has on average i^5 possible interpretations. Relations *Film* and *Performance* were left unmodified.

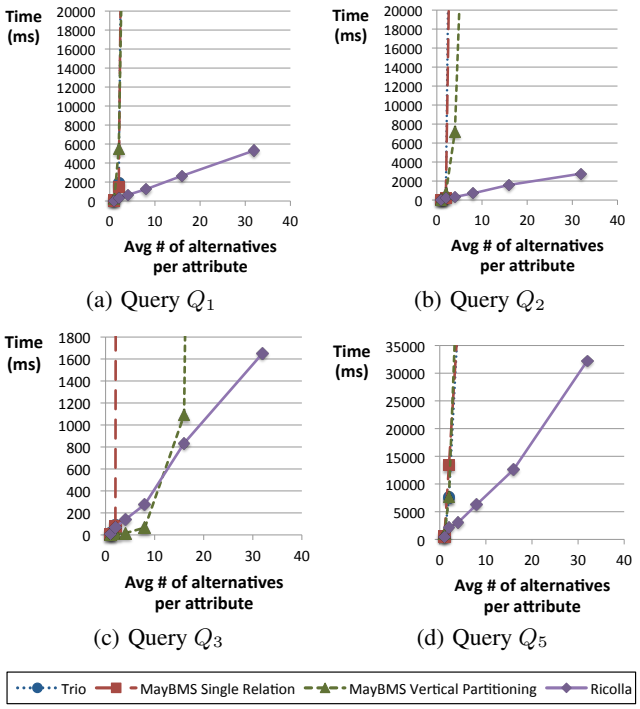


Figure 14: Query execution times under increasing uncertainty

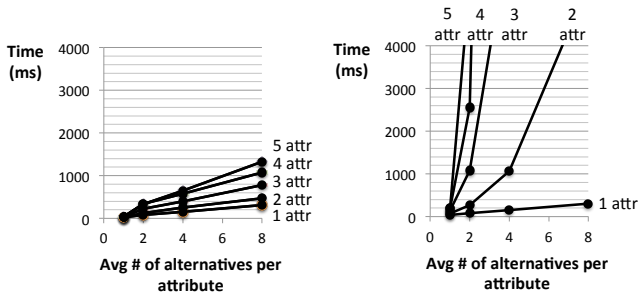
Figure 14 shows how each system scales w.r.t. uncertainty for four out of the five queries (due to lack space, we omitted Q_4, which behaves similarly to the other queries). Lines crossing the top of each graph correspond to times exceeding the vertical scale. Note that, even though we ran experiments up to 32 alternatives per attribute to stress-test Ricolla, in most cases the difference in scalability among systems becomes obvious even for small amounts of uncertainty, such as two to four alternatives per attribute.

Trio and MayBMS Single Relation become impractical even for a small number of conflicts, failing to scale beyond two alternatives per attribute. This is the result of the exponential blowup in the size of the base data explained above. In particular, in D_i, for each *Actor* ac-tuple, Trio and MayBMS Single Relation require i^5 tuples (as many as its interpretations), while Ricolla uses only $5 * i + 1$ (as many as the total number of alternatives in the ac-tuple). More importantly, since the exponential blowup happens in the base data, this exponential overhead exists for *any query* that is executed in those systems (however selective this might be).

MayBMS Vertical Partitioning also exhibits an exponential behavior w.r.t. uncertainty but for selective queries (e.g., Q_3 which returns only a single ac-tuple) the rate of increase of the execution time with uncertainty is much lower than Trio and MayBMS Single Relation. The reason is that, in contrast to these approaches, MayBMS Vertical Partitioning exhibits the exponential blowup in the size of the query output and not in the size of the base data.

Experiment 3: Scaling with number of attributes in the query result. In particular, the query output's size for MayBMS Vertical Partitioning is exponential in the number of attributes in the query's projection list. To verify this, we compared MayBMS Vertical Partitioning to Ricolla for modified versions of query Q_1 in which we varied the projection list from one to five attributes. Figure 15 show the execution times of these queries on datasets D_i, $i \in \{2, 4, 8\}$. While MayBMS Vertical Partitioning behaves similarly to Ricolla for one attribute, it does not scale to more than two attributes.

In summary, among all compared systems, only Ricolla scales



(a) Ricolla (b) MayBMS Vertical Part.

Figure 15: Execution times for variants of Q_1 with different projection lengths under increasing uncertainty

linearly with uncertainty in all cases; the other systems do not scale even for four alternatives per attribute in most cases. This is based on Ricolla's data model and query answering algorithm, which in tandem allow for a compact representation of both input and output data. Recall though that Ricolla is tailored to only those sets of possible worlds that can be easily explained to the user of an online database, while Trio and MayBMS were built as general systems allowing the representation of *any* finite set of possible worlds.

8. RELATED WORK

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

Querying inconsistent data. Prior work on querying inconsistent data [13, 18] (summarized in [16]) introduced the *Consistent Query Answering* semantics. According to them, a query returns only those tuples that exist in the query answer against *all* minimal ways of resolving inconsistencies in the original database (known as minimal repairs). Thus consistent query answering removes inconsistencies from the answer, in contrast to Ricolla whose goal is to show the conflicts so that users can inspect and resolve them. Note that the consistent query answers can be inferred from Ricolla's possible answers through a linear scan: They correspond to ac-tuples with a *single* alternative in each fragment.

Modeling inconsistent data. Numerous works proposed data models for uncertain data. However, as explained in Section 3.3, 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 [32] allows users to reconcile data in a P2P system while allowing disagreement. However, disagreement is only temporary, since after each reconciliation all non-resolved conflicts are discarded. In contrast, Ricolla keeps and displays to the users all inconsistencies, until they can resolve them. Other systems, such as HumMer [24] and Fusionplex [23] 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.

Algorithms for resolving conflicts automatically. Several algorithms have been proposed in complementary work, either in the particular context of automatic conflict resolution (e.g., using a trust network between users [19]) or in the broader context of recommendation systems (see survey in [8]). As explained in Section 4.2, these can be applied in Ricolla as resolution policies.

9. CONCLUSION

We have proposed Ricolla; an online database with built-in support for the management of data conflicts, whose viability is supported by analytical and experimental evaluation. While being formally grounded on the existing theory on uncertain data, Ricolla goes a step further than existing systems by incorporating a unique combination of user interface, data model and query answering algorithms specially tuned to the needs of an online database, which leads to exponentially better scalability w.r.t. uncertainty.

10. REFERENCES

[1] Caspio Bridge. http://www.caspio.com/bridge/.
[2] Freebase. http://www.freebase.com/.
[3] Google Fusion Tables. http://www.google.com/fusiontables/.
[4] QuickBase. http://quickbase.intuit.com/.
[5] TrackVia. http://www.trackvia.com/.
[6] Zoho Creator. http://creator.zoho.com/.
[7] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
[8] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *IEEE TKDE*, 17(6):734 – 749, 2005.
[9] 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.
[10] L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *ICDE*, 2008.
[11] L. Antova, C. Koch, and D. Olteanu. 10^(10^6) worlds and beyond: Efficient representation of incomplete information. In *ICDE*, 2007.
[12] L. Antova, C. Koch, D. Olteanu. MayBMS: Managing incomplete information with probabilistic world-set decompositions. *ICDE* 2007.
[13] M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS*, 1999.
[14] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Trans. Database Syst.*, 6(4):557–575, 1981.
[15] O. Benjelloun, A. Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 17(2), 2008.
[16] J. Chomicki. Consistent query answering. In *ICDT*, 2007.
[17] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDB J.*, 16(4):523–544, 2007.
[18] A. Fuxman, E. Fazli, and R. J. Miller. Conquer: efficient management of inconsistent databases. In *SIGMOD*, 2005.
[19] W. Gatterbauer and D. Suciu. Data conflict resolution using trust mappings. In *SIGMOD*, pages 219–230, 2010.
[20] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
[21] T. Imieliński and W. Lipski, Jr. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.
[22] V. Lakshmanan, N. Leone, R. Ross, and V. Subrahmanian. Probview: a flexible probabilistic database system. *TODS*, 22(3), 1997.
[23] A. Motro and P. Anokhin. Fusionplex: resolution of data inconsistencies in the integration of heterogeneous information sources. *Inf. Fusion*, 7(2):176–196, 2006.
[24] 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.
[25] D. Olteanu, J. Huang, and C. Koch. SPROUT: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, 2009.
[26] W. C. Purdy. A logic for natural language. *Notre Dame Journal of Formal Logic*, 32(3):409–425, 1991.
[27] A. D. Sarma, O. Benjelloun, A. Halevy, and J. Widom. Working models for uncertain data. In *ICDE*, 2006.
[28] R. A. Schmidt. Relational grammars for knowledge representation. In *Variable-Free Semantics*, pages 162–180. 2000.
[29] P. Sen, A. Deshpande, L. Getoor. PrDB: Managing and exploiting rich correlations in probabilistic databases. *VLDB J.*, 18(5), 2009.
[30] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, fourth edition.

[31] S. Singh, C. Mayfield, R. Shah, S. Prabhakar, S. Hambrusch, J. Neville, and R. Cheng. Database support for probabilistic attributes and tuples. In *ICDE*, 2008.
[32] N. E. Taylor and Z. G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, 2006.

APPENDIX

A. PROOFS

We next present the proofs of our theoretical results. To establish the necessary terminology, we first review the definitions and semantics of U-relations and ULDBs in Section A.1, before presenting the proofs that utilize these definitions in Section A.2.

A.1 Existing data models

A.1.1 U-Relation

For the purposes of our proofs, we will use a simplified version of the U-relation introduced in [10], which however still is a complete model for uncertain data.

Definition. According to this simplified definition, a U-relation is a pair (W, U) consisting of a *data-table* U and a *world-table* W. Intuitively, the data-table stores the possible alternatives together with formulas expressing their correlations, while the world-table holds the possible values that the variables involved in these formulas may take. An example of a U-relation can be seen below:

W	V	Rng
	c_1	l_1
	c_1	l_2
	c_2	l_3

U	D	A_1	A_2
	$c_1 \mapsto l_1$	Movie1	2004
	$c_1 \mapsto l_2, c_2 \mapsto l_3$	Movie1	2005
	$c_2 \mapsto l_3$	Movie2	1971

Formally, the world-table W has schema $W[V, Rng]$ and stores tuples of the form (c_i, l_j), where c_i is a variable and l_j a value in the domain of c_i. The data-table U has schema $U[D, \bar{A}]$, where D defines ws-descriptors (which intuitively are formulas guarding the appearance of a tuple) and $\bar{A} = A_1, \ldots, A_n$ define the rest of the attributes of the relation. A ws-descriptor is a formula of the form $c_1 \mapsto l_1, \ldots, c_k \mapsto l_k$, where c_i is a variable and l_i is a value in the domain of c_i according to W (i.e., $(c_i, l_i) \in W$).

Semantics. A U-relation $(W, U[D, T, \bar{A}])$ represents the set of possible worlds over a single flat relation U' with schema $U'[\bar{A}]$, each of whom can be created as follows: Select a total valuation over all variables in W (recall that the domain of each variable is described in W) and then select only those tuples of U whose ws-descriptors satisfy the valuation and project them on their $\bar{A}$ attributes. A ws-descriptor is said to satisfy a valuation v when for each assignment of the form $c_i \mapsto l_i$ in v, the ws-descriptor either contains the term, or it does not contain any term involving c_i.

It is easy to see that the U-relation, as described above, is a complete model for a schema consisting of a single relation (i.e., it can represent any finite set of possible worlds over a single relation). In particular, given a set of possible worlds P over a single relation, one can create a U-relation (W, U) that represents P as follows: Assign to each possible world in P an ID. Then create a world-table W that holds a single variable c_1, whose domain is the set of possible world IDs. Finally, create a data-table that holds the bag-union of all tuples appearing in a world in P, with each tuple augmented by a ws-description of the form $c_i \mapsto l_j$, where l_j the ID of the possible world in which this tuple appears. Based on the semantics of the U-relation, (W, U) represents P. Thus it is obvious that even a U-relation with singleton ws-descriptors (i.e., ws-descriptors of the form $c_i \mapsto l_i$) constitutes a complete data model over a single relation.

Note that the original definition of the U-relation, introduced in [10], is slightly more involved allowing among others: (a) the representation of possible worlds over an arbitrary relational schema (that may contain more than one relation) and (b) a more compact representation (through the use of vertical decompositions). Although in the main body of the paper we refer to the original definition of U-relations, for the purposes of our proofs the simplified definition described above is sufficient, as we only need a data model that is provably complete for schemas with a single relation (and which might be non-compact as we will not reason about its compactness).

A.1.2 ULDB

As described in [15], a relation in the ULDB data model, called *x-relation* is a set of x-tuples, each of each is a multiset of one or more tuples called alternatives. An x-tuple may be marked as optional and an alternative may be marked with a provenance formula. Intuitively, x-tuples in ULDBs correspond to ac-tuples in ac-databases and alternatives in the former correspond to ac-alternatives in the latter. For the purposes of the proofs, we do not need to define the exact semantics of ULDBs (for these the reader is referred to [15]). For our subsequent discussion it suffices to know that: (a) each x-relation over some schema represents a set of possible relational instances over the same schema and (b) each such instance will contain a subset of the alternatives appearing in the x-relation.

A.2 Proofs

A.2.1 Compactness

PROOF OF THEOREM 3.8 [COMPACTNESS]. Let I_{ac} be an arbitrary ac-relational instance representing some set of possible worlds P. We will show that any ULDB that represents the same set P of possible worlds is not more compact than I_{ac}. Let I_{ULDB} be an arbitrary x-relation that represents the same set of possible worlds as I_{ac}.

It follows from the semantics of an ac-relation, that for every interpretation of an ac-tuple in I_{ac}, there exists a possible world in P that contains that interpretation. On the other hand it follows from the semantics of an x-relation, that in order for an x-relation to represent a possible world containing some tuple t, this tuple should appear as the alternative of some x-tuple in the x-relation. Combining these two statements, we infer that, since I_{ULDB} represents the same set of possible worlds as I_{ac}, it contains as alternatives all interpretations of all ac-tuples in I_{ac}.

Moreover, I_{ULDB} contains such an interpretation t as an alternative as many times as the number of ac-tuples in I_{ac} that have t as their interpretation. To see why this is the case, assume that n ac-tuples of I_{ac} have t as one of their interpretations. Then, one possible world represented by I_{ac} would contain n occurrences of t. Since every alternative in an x-relation can give rise to at most one tuple in a possible world, I_{ULDB} should also contain n alternatives that are identical to t.

Thus, if I_{ac} has n ac-tuples with $k_1, \ldots, k_n$ interpretations, then I_{ULDB} will contain at least $k_1 + \ldots + k_n$ alternatives. Let w be the width (i.e., the number of columns) of relations I_{ac} and I_{ULDB}. Based on the above, I_{ULDB} contains at least $w(k_1 + \ldots + k_n)$ data values (i.e., cells). On the other hand, by definition each ac-tuple of width w, having k_i interpretations contains at most $w \times k_i$ values. Thus, I_{ac} uses at most $w(k_1 + \ldots + k_n)$ data values and therefore it is not less compact than I_{ULDB}. $\square$

A.2.2 Closure

PROOF OF THEOREM 5.6 [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. We need to prove that the possible answers of Q over I can be represented by an ac-relation. We will prove it in a per-operator basis, i.e., we will prove it first for $CQ_1^=$ queries involving only a projection, then for $CQ_1^=$ queries involving only a selection and finally for $CQ_1^=$ queries involving only a join. Since a $CQ_1^=$ query can be computed by composing multiple such subqueries, the result follows.

Projection. Let Q be $\pi_{\bar{x}}(R)$, where R is a relation in the schema of I. We will prove that the possible answers of Q over I can be represented by an ac-relation. Let I_R be the relational instance of R in I. For every possible world DB represented by I_R, the corresponding possible world $Q(DB)$ in the possible answers of Q over I_R will have the tuples in DB projected on their $\bar{x}$ attributes. To represent the same behavior in an ac-relation, it suffices to take the original ac-relation and project away all but the $\bar{x}$ attributes. Let us denote this new ac-relation by I'_R. In this way, whenever a tuple appeared in the set of possible worlds represented by I_R (because of a certain choice of ac-alternatives and optionality flags), the same tuple projected on $\bar{x}$ will appear in the set of possible worlds represented by I'_R. Therefore this new ac-relation represents the possible answers of Q over I.

Selection. Let Q be $\sigma_{x=v}(R)$, where R is a relation in the schema of I and let I_R be the relational instance of R in I. We can use similar reasoning to show that the possible answers of Q over I can be represented by an ac-relational instance I'_R that is derived from I_R as follows: For each ac-alternative that violates the selection predicate, remove this alternative and introduce in the corresponding ac-tuple a fresh optionality flag. If you introduce two optionality flags into two different tuples due to the same alternative (i.e., due to alternatives in fragments that share the same dependency marker), then use the same marker for both optionality flags.

Join. Let Q be $R \bowtie S$, where $R \neq S$ and R, S are relations in the schema of I and let I_R, I_S be the relational instances of R, S, respectively in I. We can use similar reasoning to show that the possible answers of Q over I can be represented by an ac-relational instance I_Q that can be derived from I_R and I_S as follows: For each pair of ac-tuples from I_R and I_S that share the same values for the join attributes, create a new ac-tuple that contains the concatenation of fragments of the original tuples, as well as the union of their optionality flags. Moreover, to capture correlations, whenever two ac-tuples in the new relation I_Q are introduced due join with the same I_R (or I_S) tuple, the corresponding fragments from I_R (or I_S, respectively) should share the same marker in both tuples. □

A.2.3 Tightness

PROOF OF THEOREM 5.7 [TIGHTNESS]. We want to prove that for every set P of possible worlds over a single relation, there exists an ac-database instance I and a CQ query Q s.t. $PAnswers_Q(I) = P$. We will prove this by showing that for every U-relation (which is a complete data model) we can construct an ac-database instance I and a CQ query Q such that the possible answers to Q over I represent the same set of possible worlds as the U-relation.

As input, we consider a U-relation with singleton ws-descriptors as defined in Section A.1.1. Given such a U-relation $(W, U[D, T, \bar{A}])$, we create two ac-relations $W'[D]$ and $U'[D, \bar{A}]$, s.t. the possible answers to the query $Q : \pi_{\bar{A}}(W' \bowtie U')$ represent the same set of possible worlds as the U-relation (W, U).

Intuitively, the goal of this construction is to create ac-relations W' and U' that are the equivalents of the original relations W and U, respectively. The former will state the possible values that every variable can take and the latter will state the tuples together with the variable assignments that control their appearance.

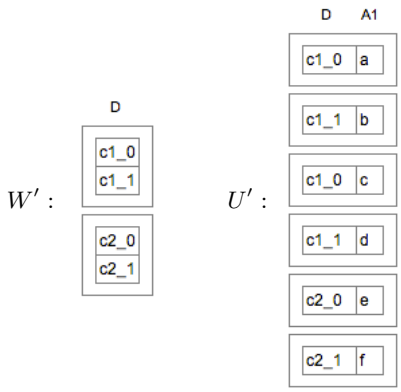
To construct W' and U' we proceed as follows: First, for each value l_j in the domain of a variable c_i, we create a fresh value $c_i\_l_j$ used to represent the assignment $c_i \mapsto l_j$. Then, we populate ac-relation W' by creating for every variable in the world-table W a single ac-tuple, whose ac-alternatives are all values $c_i\_l_j$, such that l_j is in the domain of c_i. Intuitively, each ac-tuple in W' states the possible values that each variable can take. Finally, we populate U' by creating for every tuple $(d, \bar{a})$ in the data-table U a ac-single tuple with a single ac-fragment and a single alternative $(d', \bar{a})$, such that d' is the value corresponding to the ws-descriptor d (i.e., if $d = \{c_i \mapsto l_j\}$ then $d' = c_i\_l_j$). Intuitively, U' holds the contents of U with the ws-descriptors converted to values.

For example, given the following U-relation:

W	V	Rng
	c_1	0
	c_1	1
	c_2	0
	c_2	1

U	D	A_1
	$c_1 \mapsto 0$	a
	$c_1 \mapsto 1$	b
	$c_1 \mapsto 0$	c
	$c_1 \mapsto 1$	d
	$c_2 \mapsto 0$	e
	$c_2 \mapsto 1$	f

the construction outlined above yields the following ac-relations W' and U':



Based on the semantics of the ac-database described in Section 3.2, each possible world represented by the ac-database instance (W', U') contains one assignment for each original variable in W (i.e, a total valuation over the variables in W) and it also contains all tuples in U. Thus by joining W' and U' on D and projecting on the $\bar{A}$ attributes, we get in each possible world in the query answer all tuples in U (projected on their $\bar{A}$ component) whose ws-descriptors satisfy the same total valuation. In other words, the possible answers to $Q : \pi_{\bar{A}}(W' \bowtie U')$ represent the same set of possible worlds as the initial U-relation (W, U). □

A.2.4 Best Approximation Hardness

PROOF OF THEOREM 5.10 [BEST APPROXIMATION HARDNESS]. To prove the theorem, we first prove the following lemma about the number of ac-tuples in a best approximation:

LEMMA A.1. *Let $|I|$ be the number of tuples in a relational instance I. Given a finite set of possible worlds P over a single relation, any best approximation of P contains exactly $max_{I \in P}|I|$ ac-tuples (or in words, as many ac-tuples as the number of tuples in the largest possible world within P).*

PROOF. It follows from the semantics of the ac-relation that each ac-tuple can give rise to at most one flat tuple in a possible world. Thus, any approximation of P (i.e., any ac-relation that represents a superset of P) should contain *at least* as many ac-tuples as the number of tuples in the largest possible world in P (otherwise it would not be able to represent the largest possible world).

We next show that the best approximation will contain exactly that many tuples by constructing an approximation of P that has as many ac-tuples as this lower bound. Let R be the bag-union of the sets of tuples in the possible worlds in P. Consider an ac-relation with $max_{I \in P}|I|$ identical ac-tuples, s.t. each ac-tuple is composed of a single ac-fragment and has as its ac-alternatives all tuples in R. Additionally each ac-tuple is marked as optional with its own independent optional flag. The constructed ac-relation represents all possible worlds that can be constructed by taking a subset of cardinality at most $max_{I \in P}|I|$ of the set of all tuples in the union of the possible worlds in P. In turn, this means that the constructed ac-relation represents a superset of P and therefore it is an approximation of P.

Having shown that (a) any approximation of P has at least $max_{I \in P}|I|$ ac-tuples and (b) there exists an approximation of P that has exactly $max_{I \in P}|I|$ ac-tuples, we have shown that any *best* approximation of P has exactly $max_{I \in P}|I|$ ac-tuples. □

To prove the NP-hardness of computing the best approximation we prove NP-hardness of the corresponding decision problem concerning the arity of a best approximation:

THEOREM A.2. *Given a finite set of possible worlds P over a single relation, deciding whether a best approximation of it contains at least k ac-tuples for any natural number k is NP-hard.*

PROOF. We show this by reduction from the MAX-2-SAT problem. The MAX-2-SAT problem (which has been shown to be NP-hard) is defined as follows: Given a boolean formula ϕ in conjunctive normal form with two literals per clause and a natural number k, decide whether at least k clauses of ϕ can be simultaneously satisfied by an assignment.

Given an input ϕ, k to the MAX-2-SAT problem our goal is to construct a U-relation (W, U) s.t. at least k clauses of ϕ can be simultaneously satisfied iff a best approximation of (W, U) has at least k ac-tuples.

To this end, we create a relation U that for each clause $(x_i \wedge x_j)$ in ϕ has two tuples; one with a ws-descriptor $x_i \mapsto 1, x_{new} \mapsto 0$ and another with a ws-descriptor $x_j \mapsto 1, x_{new} \mapsto 1$ (if x_i is negated, then $x_i \mapsto 1$ is replaced by $x_i \mapsto 0$ and similarly for x_j). In this case, x_{new} is a fresh variable that is created for every clause in ϕ. Moreover we create a relation W stating that the domain of all variables (those appearing in ϕ as well as those fresh variables introduced during the construction phase) is $\{0, 1\}$. Note that the values of the $\bar{A}$ attributes in the tuples in the U-relation are irrelevant; therefore we use a single A attribute and assign to it arbitrary values.

For example, the formula

$$(x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$$

leads to the following U-relation (W, U):

W	V	Rng
	x_1	0
	x_1	1
	x_2	0
	x_2	1
	x_{new1}	0
	x_{new1}	1
	x_{new2}	0
	x_{new2}	1

U	D	A_1
	$x_1 \mapsto 1, x_{new1} \mapsto 0$	a
	$x_2 \mapsto 0, x_{new1} \mapsto 1$	b
	$x_1 \mapsto 1, x_{new2} \mapsto 0$	c
	$x_2 \mapsto 1, x_{new2} \mapsto 1$	d

It is easy to see that at least k clauses of ϕ can be simultaneously satisfied iff there exists a possible world represented by (W, U) with at least k tuples (since a total valuation over the variables in ϕ satisfying at least k clauses can be extended to a total valuation over the variables in (W, U) that produces a possible world with at least k tuples and vice versa). Thus at least k clauses of ϕ can be simultaneously satisfied iff the largest possible world represented by (W, U) has at least k tuples. But according to Lemma A.1 any best approximation of the set of possible worlds P represented by (W, U) has as many ac-tuples as the number of tuples in the largest possible world in P. Thus at least k clauses of ϕ can be simultaneously satisfied iff any best approximation of the set of possible worlds represented by (W, U) has at least k ac-tuples (which completes the reduction). □