Uncertain Databases in Collaborative Data Management *

Reinhard Pichler[1], Vadim Savenkov[1], Sebastian Skritek[1], Hong-Linh Truong[2]

[1] {pichler, savenkov, skritek}@dbai.tuwien.ac.at
[2] truong@infosys.tuwien.ac.at
Vienna University of Technology

Abstract. We discuss an approach to collaborative data management based on uncertain databases. Note that, in a collaborative data management system, users may have contradicting opinions about the correct values of data items. In our approach, we propose to store all conflicting data versions in parallel and to resolve conflicts based on user ratings. We show that such a collaborative data management system can be nicely represented in an uncertain database using U-relations.

1 Introduction

With the Web 2.0 paradigm invading more and more areas of life, from entertainment to enterprise workflows and even e-government (take the Gov 2.0 initiative of the US government as an example, see http://www.gov2summit.com), decentralized community-oriented architectures become increasingly important. With a vast amount of curated datasets available to-date, we are confronted with scenarios where the data are exported, updated, shared and used by people through numerous online services. Consequently, several *approaches to collaborative data management* have emerged over the past years to support such scenarios. In case of unstructured data, solutions based on the Wiki idea [14] have been very successful. For sharing scientific data, portals such as BIRN [5] and GEON [9] have been created. Recently, the Orchestra system [12] considered "collaborative update exchange" for structured data, where multiple data peers connected by data dependencies can publish and receive updates to their data. Conflicting updates are reconciled based on the inter-peer trust relationships, established a priori. Propagation of data updates forward and backward along such data dependencies was studied in the Youtopia project [13]. Considering trust relations similar to [12], [8] introduced an approach for conflict resolution whose result is independent of the order in which updates arrive at the system, allowing for globally consistent states.

While useful for scenarios with independent data peers, maintaining a single consistent database version [12, 13] is not always satisfactory for scenarios where some consistent global state of the data in the network is required [8]. Moreover, a more fine-grained notion of trust would be desirable. That is, it should be possible to distinguish between the trustworthiness of a data source or a user in

* This work was supported by the Vienna Science and Technology Fund (WWTF), project ICT08-032. Vadim Savenkov is supported by the Erasmus Mundus External Co-operation Window Programme of the European Union

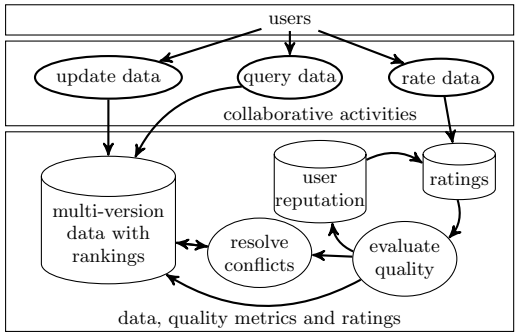

Fig. 1. A collaborative data management system

general and the acceptability of a concrete data item. Of course, in the long term, there must be a correlation between the two trust notions. Another approach for dealing with inconsistent data is a multi-versioned database concept, shared e.g. by uncertain databases [16] and the recent BeliefDB system [7], which allows different users to specify their own versions of each data tuple, called *beliefs*. There, also a belief-aware query language that allows e.g. to query for all users who share (or do not share) a particular belief is presented and analyzed from the complexity point of view.

In this paper, we present a new, *user-rating-based approach* to collaborative data management. Unlike Orchestra and Youtopia, we propose not to resolve such inconsistencies before storing the data, but to store all the different versions of the data in parallel, thus following the multi-versioned database model. We assume a scenario where all users work on the same data set, thereby updating and querying the data, but also rating its quality. In such a scenario, contradicting updates and inconsistencies (different beliefs of [7]) are unavoidable. However, unlike BeliefDB, we are not interested in tracking the beliefs of a particular user, but rather in combining the majority of common beliefs in a single consistent view on the data. The community feedback to various versions of each tuple (and, ultimately, various versions of the whole database) derived from beliefs of different users is being collected, in terms of ratings in the $[0, 1]$ scale. From this feedback the reputation of each user is derived, as a measure of alignment of her beliefs with the majority viewpoint. Figure 1 sketches our model of a collaborative data management system. It is similar to that of Wikipedia and P2P systems [10], but we focus on structured data, which have different query and update models.

Since various versions of data items have to be maintained in parallel, *uncertain databases* are a natural candidate for realizing such systems. Indeed, we show that *U-relations* (see [2]) with slight extensions are perfectly suited for this purpose. In U-relations, different versions of a data item are kept apart by assigning different values to some world set descriptor(s). A *world table* keeps track of all allowed value combinations for these descriptors. This allows one to store ratings for updates (in fact, for data versions created by updates) by annotating the various value assignments to world set descriptors. In addition

we also have to introduce a user table that stores the current reputation of each user and allows us to keep track of the initiator of each update.

Existing recommendation systems [1] show that user feedbacks are usually used for helping the user to select data suitable to the user's context. In this paper we assume that user ratings are made on the *data quality* of data tuples. The rationale behind this assumption is that data quality is critical in collaborative databases and currently there is a lack of techniques to evaluate data quality based on the user feedback and to combine such evaluated, community-based quality metrics into automatic, system-based quality metrics in traditional databases. We believe that if such a combination can be implemented in an integrated data management system, the quality of query answering can be improved. There exist a lot of data quality dimensions [4]. The approach suggested here is general enough to work with every quality dimension.

We believe that collaborative data management can become a perfect showcase for the emerging uncertain and probabilistic database technology, along with other application domains as scientific and sensoric data management [17] and information extraction [11] (see, e.g., [6] for a short survey, with a focus on probabilistic data processing).

Organization of the paper and summary of results.

• *System Model.* In Section 2, we describe the basic principles of our user-rating-based approach to collaborative data management. In particular, we describe how updates are incorporated into the database and how user ratings issued on an update can be aggregated to compute an overall rating of data items. We also explain how these ratings can be used to compute ratings for the entire database and to derive a reputation value for the user initiating an update.
• *Representation by U-relations.* We describe in Section 3 how uncertain databases can be used to implement a collaborative data management system. For this purpose, updates (or, equivalently, versions of data items) are annotated with reputation values. Care is taken that these reputation values can be computed incrementally and do not have to be recomputed every time a new rating arrives.
• *Extensions.* In Section 4, we discuss two important extensions of the basic model presented in Section 2, namely: We introduce the notion of "rigid updates" which allow the user to restrict the number of possible versions of tuples resulting from an update. Moreover, we describe how the deletion of tuples can be incorporated into our rating-based model. For both extensions, the required adaptations of the representation via U-relations are described. Directions for further extensions, which are left for future work, are discussed in Section 5.

2 Basic Collaboration Model

In this section, we describe the basic principles of our collaborative data management model. In the next section we will then show that this model can be very naturally implemented using uncertain databases. We note that there exists a huge body of rating and ranking systems in the literature. For the sake of presentation, we use a rather simple rating model, and show that it can be encoded in uncertain databases. We note that obviously also other methods could be used, without losing this nice property. The actual rating is not the main

contribution of this paper, but the observation that ratings in general fit nicely into the model of uncertain databases (this holds for several design decisions).

The fundamental idea of the approach is never to delete or overwrite any information once it has been added to the system, and to allow the insertion of conflicting and contradicting data. That is, rather than to reconcile updates and to maintain a single consistent version of the data, our database has multiple versions, each given by some non-conflicting combination of updates. A consequence of this approach is that the semantics of an update differs from the one usually assumed: In our case, *any update results in an insertion*. Throughout this paper, we assume every database relation to posses a key, and we define a *tuple* as an entry in the database identified by a key. Disagreement on the non-key values of a tuple leads to several *versions* of this tuple, which give rise (also in combination with the other tuples) to different *possible worlds*. That is, a version is a concrete value expression for some relation schema, while a tuple is a collection of versions for the same key. Every kind of update on a tuple is now mapped to the insertion of a new version.

Semantics Given the schema $\mathcal{R}$ of some relation, we consider a partitioning of the attributes appearing in $\mathcal{R}$ into sets of dependent attributes, which we call *blocks* (our notion of blocks should not be confused with the one in [6], where blocks refer to sets of tuples sharing the same key). We define the key attributes to always form a single block. In general, blocks are used whenever an object's property can be decomposed into several fields (like an address), but the values in these fields highly depend of each other. Therefore we allow as possible values for each block only those explicitly defined by updates, while different blocks, just as different tuples, are mutually independent, such that their values can be arbitrarily mixed. That is, for some *tuple* τ_i with the non-key attributes partitioned into blocks $C_1, \ldots, C_\ell$ and a set of possible values $c_j = \{c_j^1, \ldots, c_j^{k_j}\}$ for each C_j ($j \in \{1, \ldots, \ell\}$), we define the set of possible *versions* of τ_i as $c_1 \times \ldots \times c_\ell$. Furthermore, let $T = \{\tau_1, \ldots, \tau_n\}$ be a set of tuples where for every τ_i ($i \in \{1, \ldots, n\}$) there exist several possible versions $\tau_i^1, \ldots, \tau_i^{k_i}$, i.e. τ_i itself is a set $\tau_i = \{\tau_i^1, \ldots, \tau_i^{k_i}\}$. Then the set of possible worlds defined by T is $\tau_1 \times \ldots \times \tau_n$. Note that values assigned to different blocks can be arbitrarily recombined, even if the resulting tuple was never inserted explicitly. Hence attributes of different blocks have to be completely independent of each other (This rather strong assumption is relaxed in Section 4.1). We are thus able to consider updates affecting a single block only, as splitting every update that changes more than one block into several "unary" updates has the same effect. This allows us to identify every possible value for a block with one update that inserted exactly this value. As every tuple is built up from a unique set of blocks, every tuple version can be identified by a uniquely defined set of updates.

Updates First recall that in our data model, we never overwrite or delete any information stored in the database. Instead, every update gives rise to one (or several) new possible world(s), unless this world is already present. Formally, we define updates as value-assignments on block level, where for a block B of attributes $B_1, \ldots, B_k$, an update $u\colon (B_1, \ldots, B_k) \leftarrow (v_1, \ldots, v_k)[key]$ assigns to attribute B_i the value v_i for the tuple identified by *key*. One can distinguish

several types of updates: (1) Insertion of new tuples. (2) Deletion of tuples. (3) Update of non-key blocks. (4) Update of key blocks. Thereby, we defer the discussion of (2) to Section 4.2. An update affecting a key block corresponds either to the insertion of a new tuple, or, if the updated key is already present, to an update of the corresponding non-key blocks. The insertion of a tuple means just to insert a new tuple with exactly one version. For the update of a non-key block, assume an update on block C_j of tuple τ_i with versions $\tau_i^1, \ldots, \tau_i^{k_i}$. Let V be the set of versions of τ_i restricted to the blocks $C_1, \ldots, C_{j-1}, C_{j+1}, \ldots, C_\ell$. Then the result of the update is a new version for every $v \in V$, where C_j is defined according to the update.

Example 1. Consider, similar to [7], a database for monitoring an animal population, where people report their observations. As different people may have different opinions about what they saw, there will be probably disagreement about the data to store. Let this database contain a table with schema $obs(T, A, B, S)$, divided into blocks $K = \{T\}$, $B_1 = \{A, B\}$, and $B_2 = \{S\}$. Thereby K is the key, consisting of the time of the observation (for the sake of simplicity, we assume that time would give a unique identifier), B_1 describes the color (A) of the animal and its probable kind (B), while B_2 stores its estimated size (S) (note that while kind and size of an animal are not independent, the observations of these properties are). Assume for some observation (say made by Alice) the database to contain the tuple τ_i with the single version (t_1, a_1, b_1, s_1). Further assume that Bob made the same observation, but disagrees on the type (and color) of the animal seen. He issues an update $u_1\colon (A, B) \leftarrow (a_2, b_2)[t_1]$. This results in the two possible worlds $obs(t_1, \{(a_1, b_1)|(a_2, b_2)\}, s_1)$. If now John disagrees with the size of the seen animal, he performs an update $u_2\colon (S) \leftarrow (s_2)[t_1]$, resulting in the four possible versions $obs(t_1, \{(a_1, b_1)|(a_2, b_2)\}, \{s_1|s_2\})$ of τ_1. □

Ratings & User Reputation Next we describe how the quality of a tuple version is estimated. Users are allowed to rate either updates or tuple versions, where rating a tuple version is the same as giving this rating to all the (uniquely defined) updates that build up this version. The votes given on one update by different users are aggregated, where the influence of a rating is weighted according to the reputation of the user giving the vote. The reputation of a user is derived from the ratings given to the updates performed by the user, and as stated in the introduction, is interpreted as a score value normalized to $[0, 1]$. Every update is automatically rated with the reputation of the user who performed it. We fix the following notation. Let $U = \{u_1, \ldots, u_n\}$ be a set of n updates, and let $R_i = \{r_1, \ldots, r_{m_i}\}$ be a set of m_i ratings for every $u_i \in U$. With $user(u_i)$ and $user(r_j)$, we denote the user who performed the update u_i, or gave the rating r_j, respectively. Finally, let $rep(u_i)$ and $rep(r_j)$ be the reputation of $user(u_i)$ and $user(r_j)$, respectively.

We define the aggregated rating for u_i by

$$rating(u_i) = \frac{\sum_{\alpha=1}^{m_i} (r_\alpha \cdot rep(r_\alpha))}{\sum_{\alpha=1}^{m_i} (rep(r_\alpha))} \tag{1}$$

and the rating of a tuple version $\tau_i^j = (b_1, \ldots, b_k)$ (where $b_1, \ldots, b_k$ are grouped to blocks $c_1, \ldots, c_\ell$) as $rating(\tau_i^j) = \sum_{\alpha=1}^{\ell} (w_\alpha \cdot rating(c_\alpha))$. Thereby $rating(c_\alpha) =$

$rating(u_i)$, with u_i being the update that sets the value for the block C_α in τ_i to c_α, and w_α is some weighting factor ($\sum_{\beta=1}^{\ell} w_\beta = 1$) that expresses the influence of each block. Denoting the number of attributes in a block C_α with $|C_\alpha|$, a reasonable value for w_α could be $(1/\sum_{\beta=1}^{\ell} |C_\beta|) \cdot |C_\alpha|$, which can be adopted accordingly if not all attributes are considered equally important. Obviously $rating(u_i)$ can be incrementally computed. For m_i votes, instead of storing them together with the user reputations, it suffices to store $\overline{rat}(u_i) = \sum_{\alpha=1}^{m_i}(r_\alpha \cdot rep(r_\alpha))$ and $\overline{rep}(u_i) = \sum_{\alpha=1}^{m_i}(rep(r_\alpha))$, from which the new rating after a new vote can be easily derived.

Concerning user reputation, we argue that it is advantageous not to consider all contributions of a user from the beginning, but only her more recent work. Thereby "recent" is defined in terms of a *time window* that defines which contributions to take into account. Leaving its concrete definition as a parameter to the system, beside its actual size it can be also defined either in terms of time (e.g. all the contributions from the last year), or in terms of contributions (e.g. the last 100 updates performed by this user). For the computation of a user's reputation (say $user_\mu$), consider $U = \{u_1, \ldots, u_n\}$ as the set of updates done by $user_\mu$ that fall into the time window. We define her reputation as

$$reputation(user_\mu) = \frac{\sum_{\alpha=1}^{n}(rating(u_\alpha) \cdot \overline{rep}(u_\alpha))}{\sum_{\alpha=1}^{n} \overline{rep}(u_\alpha)} = \frac{\sum_{\alpha=1}^{n} \overline{rat}(u_\alpha)}{\sum_{\alpha=1}^{n} \overline{rep}(u_\alpha)} \qquad (2)$$

The reputation of a user can change because of two reasons. Either a new rating enters the time window, or some ratings (i.e. some updates) fall out of it. In both cases the reputation can be easily computed incrementally by storing $\overline{rat}' = \sum_{i=1}^{n} \overline{rat}(u_i)$ and $\overline{rep}' = \sum_{i=1}^{n} \overline{rep}(u_i)$. If a new rating arrives, the following steps are required for updating the user reputation: (1) Identify the user who performed the update just rated. (2) Check if the update is currently in the time window. (3) Update the user's reputation and the values stored for the user. For performance reasons, the current time window for each user is explicitly stored as an index for the contained updates. From the description, it goes without saying that also several alternatives for aggregating the user reputation could be used, like the moving average to "fade out" older updates.

The initial reputation of a new user is 0 until she makes some contribution that is rated by other users. The system, however, can be easily adapted to support any other initial reputation. If a user is invited by some other user, she could get the reputation of the inviting user. In such cases, for aggregation, the initial reputation can be modeled as "rating" for some dummy update u_0.

Example 2. Recall the scenario in Example 1, and consider the situation after u_1. We split the initial update done by Alice into three smaller ones. One that inserted the key, one that set $B_1 = (A, B)$ to (a_1, b_1) and one update $B_2 = (S) \leftarrow (s_1)[t_1]$. We denote the updates with u_K, u_0 and u_0', respectively. Also, let $\tau^1 = (t_1, a_1, b_1, s_1)$, and $\tau^2 = (t_1, a_2, b_2, s_1)$. Assume $\overline{rat}(u_0) = 8.4$, $\overline{rep}(u_0) = 12$, $\overline{rat}(u_0') = 2.4$, $\overline{rep}(u_0') = 8$, $\overline{rat}(u_1) = 25.2$, $\overline{rep}(u_1) = 28$, hence, by (1), $rating(u_0) = \frac{8.4}{12} = 0.7$, $rating(u_0') = 0.3$, and $rating(u_1) = 0.9$. We omit the influence of the key and assume a uniform influence of the non-key attributes

A, B, S, that is $w_K = 0$, $w_{B_1} = \frac{2}{3}$, and $w_{B_2} = \frac{1}{3}$. Then we get $\mathit{rating}(\tau^1) = \frac{2}{3} \cdot 0.7 + \frac{1}{3} \cdot 0.9 \approx 0.77$ and $\mathit{rating}(\tau^2) = \frac{2}{3} \cdot 0.3 + \frac{1}{3} \cdot 0.9 = 0.5$.

Now suppose that John performs update u_2. Let $\overline{\mathit{rat}}'(\mathit{John}) = 7.5$ and $\overline{\mathit{rep}}'(\mathit{John}) = 15$, hence $\mathit{reputation}(\mathit{John}) = 0.5$. His update is automatically rated by 0.5, which results in $\mathit{rating}(u_2) = \frac{0.5 \cdot 0.5}{0.5} = 0.5$. (It is easy to check that his reputation remains unchanged.) Now let some user user_3 with a high reputation of 0.9 disagree with u_2, which she expresses by giving a rating of 0.3 on u_2. This results in $\overline{\mathit{rat}}(u_2) = 0.52$ and $\overline{\mathit{rep}}(u_2) = 1.4$, hence $\mathit{rating}(u_2) = \frac{0.25+0.27}{1.4} \approx 0.372$. Because of this rating, the reputation of John changes to $\mathit{reputation}(\mathit{John}) = \frac{7.5+0.25+0.27}{15+0.5+0.9} \approx 0.489$. Yet another user, with reputation 0.2 agrees with u_2 and rates it 0.7. Then $\overline{\mathit{rat}}(u_2) = 0.66$ and $\overline{\mathit{rep}}(u_2) = 2.1$, hence $\mathit{rating}(u_2) = \frac{0.66}{1.6} = 0.4125$. For the reputation of John, this has the effect of $\overline{\mathit{rat}}'(u_2) = 8.16$ and $\overline{\mathit{rep}}'(u_2) = 16.6$, hence $\mathit{reputation}(\mathit{John}) \approx 0.492$.

That is, the resulting rating for $\tau^3 = (t_1, a_1, b_1, w_2)$, and $\tau^4 = (t_1, a_2, b_2, w_2)$ is $\mathit{rating}(\tau^3) = \frac{2}{3} \cdot 0.7 + \frac{1}{3} \cdot 0.4125 \approx 0.60417$ and $\mathit{rating}(\tau^4) = \frac{2}{3} \cdot 0.3 + \frac{1}{3} \cdot 0.4125 = 0.3375$. □

Foreign Keys The semantics described in this section allows also for foreign keys. Consider a relation R_1 with a block A of key attributes, and another relation R_2, that contains a block C_j with the same number and type of attributes as in A. Then C_j can be defined as a foreign key to A. To ensure referential integrity between C_j and A, it suffices to check for every update on C_j whether for the newly added values c_j there already exists a tuple τ of R_1 where the key A has the value c_j. If this is the case then referential integrity is ensured in all possible worlds. This is due to our assumption that different versions of τ are due to different values of the non-key attributes; however, the value c_j of the key attributes of τ is the same in all possible worlds.

Query Answering We define query answering in terms of the best rated world. That is, given a query from a user, in a first step, the best rated possible world is selected. Then the query is answered on this world. This approach has the advantage to provide the user a consistent view of the worlds, independent of the issued query. We therefore have to define the rating of a possible world. As each world is a set of tuples, we define the rating of a world as the average of the ratings over all tuple versions appearing in this world. That is, for a possible world W_i containing tuples $W_i = \{\tau_1, \ldots, \tau_n\}$, $\mathit{rating}(W_i) = \frac{1}{n} \cdot \sum_{\alpha=1}^{n} \mathit{rating}(\tau_\alpha)$. We note that, in our model, every possible world has exactly the same number of tuples, as for every primary key, exactly one tuple version must be selected.

Obviously, the best rated world does not need to be unique. If this is the case, we prefer more recent updates: Going from the newest update to the oldest, if some of the best rated worlds contains the update (i.e. that have the corresponding block set to the value defined by the update), remove all worlds that do not contain this update, until only a single world is left. In the model described above, the most recent, best rated world can be found easily. Just pick for every block the best rated update. If there are several best rated updates for one block, then pick the youngest one. Of course, our approach could be easily adapted to other preference criteria: e.g., the most often rated update or the update with highest $\overline{\mathit{rep}}$.

3 Representation in U-relations

In this section we show that the data model described above can be encoded in U-relations. We use the database schema for U-relations as proposed in [2], consisting of the vertical decomposition tables (VDTs) for every attribute and the world table (W). For every tuple τ_i and every block C_j, we use a unique world set descriptor (WSD) $x_{i,j}$, and define in W one distinct assignment to $x_{i,j}$ for every possible value for C_j in any version of τ_i. That is, let $\tau_i = \{\tau_i^1, \ldots, \tau_i^k\}$ be a tuple with k versions, $C_j = B_1, \ldots, B_{s_j}$ a block, and $V = \bigcup_{\alpha=1}^{k} \pi_{C_j}(\tau_i^\alpha)$, where $\pi_{C_j}(\tau_i^\alpha)$ denotes the projection of τ_i^α onto the values for the attributes in C_j. In the vertical decomposition table (VDT) of every attribute $B_\beta \in C_j$, there exists exactly one distinct assignment to $x_{i,j}$ for every $v \in V$. For the vertical decompositions of the key attributes, there exists exactly one variable $x_{i,A}$ for every tuple τ_i, with exactly one assignment for $x_{i,A}$.

Example 3. τ_1 from Example 1 could be represented in U-relations as follows:

tid	WSD	T	WSD	A	WSD	B	WSD	S
τ_1	$x_{1,K} \to 1$	τ_1	$x_{1,1} \to 1$	a_1	$x_{1,1} \to 1$	b_1	$x_{1,2} \to 1$	s_1
			$x_{1,1} \to 2$	a_2	$x_{1,1} \to 2$	b_2	$x_{1,2} \to 2$	s_2

(Except for T, we discard the column holding the tuple id in the relations, as we consider only a single tuple. The corresponding world table is depicted in Example 4.) Note that every update is identified by an assignment to a WSD, e.g. $x_{1,1} \to 2$ corresponds to u_1, or $x_{1,2} \to 2$ to u_2. As discussed in Example 2, the initial update is split into the three updates u_K, u_0, and u_0'. For u_K, a WSD $x_{1,K}$ with the single assignment 1 is created, and added together with t_1 into the corresponding table. Similar for u_0 and u_0', where the WSDs $x_{1,1}$ and $x_{1,2}$ are used. For u_1, a new assignment for $x_{1,1}$ is created, and corresponding entries are added to the VDTs for A and B. Similarly for u_2 and $x_{1,2}$. □

As sketched in the example, every update operation identified above can be easily mapped to this representation: For the update of a non-key block C_j of some existing tuple τ_i, as already stated above, we only need to consider updates of a single block of a single tuple. The update process consists of creating a new assignment for the unique variable $x_{i,j}$, and adding a new entry for τ_i, the new assignment for $x_{i,j}$, and the new value to the VDT for every $B_\beta \in C_j$. If the value is already present for this block, then the update is ignored. The insertion of a new tuple is similar to the above operation. To insert a new tuple $(a_1, \ldots, a_k, b_1, \ldots, b_n)$, first insert the key $(a_1, \ldots, a_k)$ with a new WSD, and then perform ℓ updates on this tuple, by setting the values for the blocks $c_1, \ldots, c_\ell$.

To keep track of the ratings and user reputations, the information described above can be stored as follows. The information about ratings on updates can be stored in the world table W, as every update corresponds to the combination of a variable and an assignment, i.e. to one row in W. We extend $W(\mathit{WSD}, \mathit{WSDvalue})$ to $W(\mathit{WSD}, \mathit{WSDvalue}, \overline{\mathit{rat}}, \overline{\mathit{rep}}, \mathit{rating}, \mathit{timestamp}\ [, \mathit{user}])$. Thereby *user* stores a reference to the user who initiated the update. If the time window is stored explicitly for every user, this reference can be omitted. The information $\overline{\mathit{rat}}$, $\overline{\mathit{rep}}$, and *rating* need to be present for every quality dimension

tracked. The current time window for each user can be maintained using a relation $tw(user,\ tid,\ WSD,\ WSDvalue,\ (count|timestamp))$. We can further log which user gave a rating to which update. Finally, in an additional table we store for every user her current reputation as well as $\overline{rat}'$ and $\overline{rep}'$.

Example 4. The left table shows the world table to Example 3, while the right one stores the user information. We omit the representation of the time windows.

WSD	ass.	$\overline{rat}$	$\overline{rep}$	*rating*
$x_{1,K}$	1	1	1	1
$x_{1,1}$	1	8.4	12	0.7
$x_{1,1}$	2	2.4	8	0.3
$x_{1,2}$	1	25.2	28	0.9
$x_{1,2}$	2	0.66	2.1	0.4125

user	$\overline{rat}'$	$\overline{rep}'$	*reputation*
John	8.16	16.6	0.492
$user_3$	20.34	22.6	0.9
$user_4$	2.46	12.3	0.2

□

Query Answering We defined the semantics for query answering as answering the query on the most recently updated, top rated world. On the representation level, every possible world is identified by a (total) assignment to the WSDs [2]. As described above, in the world table W, we store every possible assignment to a WSD along with its rating (which corresponds to a rating on some update). Choosing one assignment to every WSD defines one set of tuples. For each of these tuples, we already defined their ratings, hence the rating of the world can be easily computed.

4 Extensions

In this section, we consider two important extensions of the basic scenario, namely, tuple updates with inter-block dependencies and tuple deletions. First we discuss the required adaptations of our basic model described in Section 2. We then also discuss the impact on the representation by U-relations.

4.1 Inter-block dependencies

In the basic scenario, updates of the attribute blocks were independent of each other. Assume now that the user wants to update the values of several blocks at the same time, e.g., by inserting new values for all three non-key attributes in our running example. Moreover, one may want to exclude from any possible world the combination of the new size with any of the prior values for kind and color of the animal. We call such updates spanning several blocks *rigid*.

In principle, such updates could be handled by simply merging the two blocks into one. However, this naive approach would lead to an explosion of the number of tuples needed to represent the possible worlds stored in the previously independent blocks. We therefore propose a more expressive representation scheme here.

Example 5. Consider the following shorthand notation of the schema of our running example: $obs = \overline{KB_1B_2}$ where K is the key block and B_1 and B_2 are the blocks of attributes as defined in Example 1. Let *obs* contain the tuple τ_1 with the uncertain value $\tau_1 = R(k, \{b_1|b_1'\}, \{b_2|b_2'\})$, giving rise to 4 possible worlds and the following decomposition into three partitions:

$\mathbf{obs_K}$ T	WSD K
τ_1	x_K k

$\mathbf{obs_{B_1}}$ T	WSD	B_1
τ_1	$x_1 \to 1$	b_1
τ_1	$x_1 \to 2$	b'_1

$\mathbf{obs_{B_2}}$ T	WSD	B_2
τ_1	$x_2 \to 1$	b_2
τ_1	$x_2 \to 2$	b'_2

Suppose that some observer now wants to ensure that the size added only appears in conjunction with the animal type she also added. That is, she needs to insert a pair of block values b''_1, b''_2 as a rigid update of the tuple τ_1. If one now chooses to merge respective blocks, it will be necessary to first explicitly specify all possible worlds compactly represented in the partitions obs_{B_1} and obs_{B_2}:

$\mathbf{obs_K}$ T	WSD	A
τ_1	$x_K \to 1$	k

$\mathbf{obs_{B_1}}$ T	WSD	B_1
τ_1	$x_1 \to 1$	b_1
τ_1	$x_1 \to 2$	b_1
τ_1	$x_1 \to 3$	b'_1
τ_1	$x_1 \to 4$	b'_1
τ_1	$x_1 \to 5$	b''_1

$\mathbf{obs_{B_2}}$ T	WSD	B_2
τ_1	$x_1 \to 1$	b_2
τ_1	$x_1 \to 2$	b'_2
τ_1	$x_1 \to 3$	b_2
τ_1	$x_1 \to 4$	b'_2
τ_1	$x_1 \to 5$	b''_2

Clearly, the total number of tuples needed for such "decompression" of the succinct representation with independent partitions is exponential in the number of blocks that have to be merged. Therefore, the succinctness of representation due to partitioning in U-relations is deteriorated by rigid updates. □

Note that in [3] it is also observed that updates might make the decompression of the succinct representation by U-relations necessary. However, in our case, the blow-up of the U-relations is even more problematic since, in contrast to [3], we want to be able to increase the number of possible worlds as a consequence of an update (as we never overwrite any data). We therefore propose to use *compound* (that is, non-normalized [3]) WSDs for blocks. To express arbitrary dependencies between N blocks in a tuple, it is sufficient that the WSD of each block contain N variables.

Example 6. The final state of Example 5 has the following representation:

$\mathbf{obs_K}$ T	W_K	K
τ_1	$x_k \to 1$	k

$\mathbf{obs_{B_1}}$ T	W_{B_1}	W_{B_2}	B_1
τ_1	$x_1 \to 1$		b_1
τ_1	$x_1 \to 2$		b'_1
τ_1	$x_1 \to 3$	$x_2 \to 3$	b''_1

$\mathbf{obs_{B_2}}$ T	W_{B_1}	W_{B_2}	B_2
τ_1		$x_2 \to 1$	b_2
τ_1		$x_2 \to 2$	b'_2
τ_1	$x_1 \to 3$	$x_2 \to 3$	b''_2

Since only the value combinations connected to consistent variable assignments are admitted, one can check that the U-relations above define exactly the desired five possible worlds: E.g., the tuple (b_1, b''_2) is not part of the (uncertain) projection $\pi_{B_1,B_2}(obs)$, as the variable assignment $x_1 \to 1$, $x_1 \to 3$ is inconsistent, whereas (b_1, b'_2), corresponding to the assignment $x_1 \to 1$, $x_2 \to 2$, is a possible answer. □

The following procedure can then be used to accommodate new inserts into the database: Let C be an attribute block, and assume that each attribute belongs to a separate U-relational partition. Thus, each attribute in C has a compound WSD of at most $|C|$ variables. Consider a rigid update u for a tuple τ introducing a new value assignment for attributes $C' \subseteq C$; it can be accommodated in the following two steps:

1. *Build a WS descriptor $\bar{w}_u$ for u:* For each attribute $A \in C'$ with a corresponding variable x_A in the WS descriptor of $\tau.C$, check if the value $u.A$ is already present in $\tau.A$. If yes, re-use the found assignment for x_A in $\bar{w}_u$; otherwise, take a fresh domain value not occurring in the WS descriptor of $\tau.A$ as an assignment for x_A in $\bar{w}_u$.
2. *Perform the insert:* for each attribute $A \in C'$, insert the tuple $(id(\tau), \bar{w}_u, u.A)$ in the partition P_A of the U-relation.

For example, a rigid update b_1, b_2^* for the blocks B_1, B_2 of the relation *obs* in its final state as shown in the Example 5 will be assigned a WSD $x_1 \to 1, x_2 \to 4$.

Note that new updates can be composed of the values already occurring in other updates: the update $\tau_1.B_1 = b_1''$ can be admitted (and assigned a WSD $x \to 3$), despite the rigid update $B_1B_2 \leftarrow (b_1'', b_2'')$ being already present in the table. A possible meaning of such new insert is "the value b_1'' can be combined not only with b_2'', but with any other value of the block B_2".

Ratings of rigid updates In the basic scenario, each update is identified by an assignment to its WSD variable. The aggregated ratings associated with each update are summarized in the world table. If rigid updates are allowed, the world table needs to be extended to accommodate compound WSDs: the number of variable/value column pairs equals the maximal number of blocks in any relation described by the world table; representation of ratings of updates remains the same as in the Example 3. Deriving ratings of tuples from the update ratings must be redefined, however. We address this issue in Section 4.3.

4.2 Deleting tuples

So far we have only considered disagreements on the correct values for the tuples stored in the database. We now extend the system to also allow to express that some tuple should not be present at all. This is modelled by introducing a special tuple version, namely $\emptyset$, to express that there exist possible worlds that do not contain any version of this tuple at all. Under this semantics, deletion of a tuple corresponds to adding $\emptyset$ to the set of possible versions for this tuple. Concerning our definition of updates, we model a deletion as $u\colon (A_1, \ldots, A_k) \leftarrow ()[key]$, where $A_1, \ldots, A_k$ are the key attributes. We do, however, not allow $\emptyset$ to be the only version of a tuple, but require at least one other version to exist.

Representing $\emptyset$ in U-relations can be done easily. Given a relation R with the key-block K, for every tuple $\tau_i \in R$, so far the WSD consists of exactly one variable $x_{\tau_i,K}$ with a unique assignment (say $x_{\tau_i,K} \to 1$) that encodes the value of the key for τ_i. Inserting the version $\emptyset$ for τ_i can be done by adding another assignment for $x_{\tau_i,K}$ (say $x_{\tau_i,K} \to 0$) to the world table, without inserting an entry for this assignment to the VDTs of the key attributes. With non-normalized U-relations, it can be further ensured that this (empty) "selection" for the key values cannot be combined with any other non-null values of the non-key attributes: It suffices to add for every tuple τ_i the assignment $x_{\tau_i,K} \to 1$ to the WSDs of every non-key attribute block. That is, the VDT of each non-key block is extended by a column WSD_{key} that contains $x_{\tau_i,K} \to 1$ (resp. by columns $WSD_{key}, WSD_{key}val$ with $(x_{\tau_i,K}, 1)$. Hence selecting $x_{\tau_i,K} \to 0$ will return no value for any attribute of τ_i.

Example 7. Consider the representation in Example 6. To support deletion, the WSD of the key block is added to each partition.

$\mathbf{obs_K}$		
T	W_K	K
τ_1	$x_K \to 1$	k

$\mathbf{obs_{B_1}}$				
T	W_K	W_{B_1}	W_{B_2}	B_1
τ_1	$x_K \to 1$	$x_1 \to 1$		b_1
τ_1	$x_K \to 1$	$x_1 \to 2$		b_1'
τ_1	$x_K \to 1$	$x_1 \to 3$	$x_2 \to 3$	b_1''

$\mathbf{obs_{B_2}}$				
T	W_K	W_{B_1}	W_{B_2}	B_2
τ_1	$x_K \to 1$		$x_2 \to 1$	b_2
τ_1	$x_K \to 1$		$x_2 \to 2$	b_2'
τ_1	$x_K \to 1$	$x \to 3$	$x_2 \to 3$	b_2''

Deletion of τ_1 is allowed by leaving the above tables unchanged, but adding the assignment $x_k \to 0$ to the world table. □

Rating tuple deletions The deletion of a tuple can be rated like every other update on the database. As $\emptyset$ is expressed by an entry in the world table, it is also no problem to store the rating there, hence also just like the rating on every other update. The definition of the rating of a tuple version τ_i^k requires a slight modification: If $\tau_i^k \neq \emptyset$, then it is as defined in Section 2. Otherwise, if $\tau_i^k = \emptyset$, then $\mathit{rating}(\tau_i^K) = \mathit{rating}(u_\alpha)$, where u_α is the update that inserted $\emptyset$.

4.3 Adapting the top database semantics

So far in this section we have concentrated solely on the representation of updates and possible worlds defined by them. We now shift our focus to user ratings and, consequently, to the reconciliation of conflicting updates, which is an essential task from the user point of view. In the following, we assume each relation to contain at least one non-key attribute block. Note that this is no real restriction, as otherwise there could not be multiple versions for this relation anyway.

Conceptually, the conflict-resolution approach remains exactly the same as in the basic scenario: We first look for WSD variable assignments maximizing the overall world rating, and then we use the corresponding top rated world for query answering. However, the extensions discussed earlier in this section have several important implications:

Maximal feasible variable assignments Because of the introduction of rigid updates, variable assignments for different variables are no longer independent of each other, as it was the case in the basic scenario. Due to the fact that the WSD of each rigid update is now a set of variable assignments, the dependencies between these assignments have to be enforced. We must thus speak of *maximal feasible variable assignments* defined by consistent sets of WSDs.

Example 8. Consider the setting from Example 7. There, $x_k \to 1, x_1 \to 3, x_2 \to 3$ is an example of a maximal feasible variable assignment. The assignment $x_k \to 1, x_1 \to 1$ is feasible but not maximal, whereas $x_1 \to 3, x_2 \to 2$ and $x_k \to 0, x_1 \to 1, x_2 \to 1$ are not feasible: No valid combination of updates issued to the database gives such a variable assignment on the WSD. □

Definition 1. *A set* $\mathbf{U}$ *of WSDs (resp. the set* $\mathbf{u}$ *of updates identified with* $\mathbf{U}$*) is* proper *if it satisfies the following conditions:*
— Consistency*: No two WSDs in* $\mathbf{U}$ *contain different assignments for the same variable (resp. no two updates in* $\mathbf{u}$ *have different values for the same block).*

— Maximal coverage. *Let domain* $dom(\mathbf{U})$ *denote the set of all variables in* $\mathbf{U}$. *We request that the domain of* $\mathbf{U}$ *is maximized in the following sense: there is no update in the database identified with the WSD* U*, such that* $\mathbf{U} \cup U$ *is consistent and* $dom(\mathbf{U}) \subset dom(\mathbf{U} \cup \{U\})$ *(resp. every tuple is either deleted by* $\mathbf{u}$ *or the values for all blocks of the tuple are specified by* $\mathbf{u}$*).*
— Irreducibility. *Let* $nkdom(\mathbf{U})$ *denote* $dom(\mathbf{U})$ *restricted to non-key variables: that is, variables not occurring in the WSD of any key block. We request that for each WSD* $U \in \mathbf{U}$*,* $nkdom(\mathbf{U} \setminus \{U\}) \subset nkdom(\mathbf{U})$ *(resp. no update in* $\mathbf{u}$ *spans only the blocks which are also spanned by other updates).*

It can be easily shown that any proper set of WSDs (resp. proper set of updates) defines a maximal feasible variable assignment.

Example 9. Let a relation R have a schema $\overline{KABC}$ where each letter defines an attribute block, the key one denoted by K. Suppose that a tuple identified by a key value k is comprised by the following four rigid updates: $\{u_1^{r=0.3}\colon AB \leftarrow (a,b), u_2^{r=0.45}\colon C \leftarrow c, u_3^{r=0.5}\colon AC \leftarrow (a,c), u_4^{r=0.2}\colon B \leftarrow b\}$ (where superscripts denote update ratings) which give rise to a single possible tuple value (k,a,b,c), and one update u_5, where the tuple was deleted.

According to the procedure of U-relational updates from Section 4.1, a value of a non-key block determines the corresponding WS variable assignment: Hence, there are three assignments to non-key variables $x_a \to v_a$, $x_b \to v_b$, and $x_c \to v_c$, and two assignments to the key variable, namely $x_k \to 1$ and $x_k \to 0$. The variable assignments of the updates are $U_1 = x_k \to 1 \wedge x_a \to v_a \wedge x_b \to v_b$, $U_2 = x_k \to 1 \wedge x_c \to v_c$, $U_3 = x_k \to 1 \wedge x_a \to v_a \wedge x_c \to v_c$, and $U_4 = x_k \to 1 \wedge x_b \to v_b$.

One can check that there are three proper update sets assigning a non-empty value to the tuple k: namely, $\mathbf{u}_1 = \{u_1, u_2\}$, $\mathbf{u}_2 = \{u_3, u_4\}$ and $\mathbf{u}_3 = \{u_1, u_3\}$. We say that all of them *define* a maximal variable assignment $\mathcal{A} = \{x_k \to 1, x_a \to v_a, x_b \to v_b, x_c \to v_c\}$. By definition, $\mathcal{A}$ is feasible (as all three update sets are consistent). Yet another maximal feasible variable assignment is $x_k \to 0$ implying that tuple k must not be part of the respective worlds. □

Finding a best rated world then amounts to selecting a proper update set that maximizes the world rating. It turns out, however, that the computation of the rating of some world defined by a given set of updates also requires adaptation if rigidity is allowed.

Composing update ratings In the basic scenario, the rating of a tuple value was given by: $rating(\tau_i^j) = \sum_{\alpha=1}^{\ell}(w_\alpha \cdot rating(c_\alpha))$ where every c_α is a block value corresponding to some update (recall that in absence of rigid updates each block value is given by exactly one update) and weight w_α depends on the attributes comprising the respective block C_α. We call this *the composition formula*.

In presence of rigid updates, we no longer have ratings for each block straight away, but rather the rating for each rigid update. We re-define the rating of a block value c_α as the maximal rating of an update in which c_α is contained.

Example 10. As an example, consider the update set $\mathbf{u}_3 = \{u_1, u_3\}$. We take $r(A \leftarrow a) = max(r(u_1), r(u_3)) = 0.5$, $r(B \leftarrow b) = r(u_1) = 0.3$, and $r(C \leftarrow c) = r(u_3) = 0.5$ (we shorten $rating(\cdot)$ as $r(\cdot)$ for the sake of readability). According

to the composition formula, $rating(\mathbf{u}_3) = \frac{1}{3}(0.5 + 0.3 + 0.5) \approx 0.43$ (the weight w_α of each block is taken equal to $\frac{1}{3}$, so that the tuple ratings fall into $[0, 1]$).

As shown in Example 6, $\mathbf{u}_3$ is not the only proper update set defining the variable assignment $\mathcal{A}$. Its rating — and the rating of the corresponding tuple value (k, a, b, c) — is thus not uniquely determined: other possible candidates are $rating(\mathbf{u}_1) = \frac{1}{3}(0.3+0.3+0.45) = 0.35$ and $rating(\mathbf{u}_2) = \frac{1}{3}(0.5+0.5+0.2) = 0.4$, obtained by the composition formula. Similarly to selection of block ratings, we choose the maximum (i.e. $rating(\mathbf{u}_3) \approx 0.43$) as the rating of the tuple value. □

Let $\mathcal{A}$ be a maximal feasible variable assignment, corresponding to a world $\mathcal{W}_\mathcal{A}$. Its rating can be found in two steps:

1. For each tuple value τ determined by $\mathcal{A}$, take $rating(t)$ to be the maximum rating computed according to the composition formula from any proper update set defining $\mathcal{A}$.
2. Take the maximum of the tuple ratings as the world rating (Other aggregates like sum or average can be used instead).

The top world is then chosen as the most recently updated world from the worlds with the highest rating, as described in Section 2.

5 Conclusion and Future Work

In this paper we presented collaborative data management as a relevant application area for uncertain databases. We proposed an update storage model in which user contributions are never overwritten or deleted, but persist in the database in their original form, leading to different possible worlds. We described a framework in which community feedback is used to reconcile contradicting updates and to compute reputation values for each user according to the feedback on her updates. One of the main points of the paper was to describe a relevant use case for uncertain databases, in particular based on U-relations. Due to the lack of space, we have left out some important questions that have to be addressed in a real-world system: e.g. how to prevent creation of virtual users to rate dummy updates for the sake of earning high reputation [15].

Directions for future work are thus manyfold. A particularly important question is the choice of the semantics for query answering. Unlike the approach chosen here, in probabilistic databases queries are conceptually evaluated over all possible worlds, and then the "best" possible answer is returned. It would be thus interesting to extend our framework to other semantics for query answering and to design efficient methods for query evaluation under these semantics. Another open issue is a precise complexity analysis of query answering.

Besides determining the best choices for all these problems, there are also several open questions concerning techniques of the underlying uncertain databases. A potential shortcoming of the semantics described in Section 2 is the arbitrary combination of all available, compatible updates. Roughly speaking, every update is applied to all possible worlds in parallel. The user, however, might just want to introduce exactly one new possible world, or to be able to specify the set of possible worlds to which the update is applied. Hence we do no longer

make any assumptions about dependent and independent attributes, but the set of possible worlds is just defined by the updates performed. To the best of our knowledge, no algorithm for these kinds of inserts in U-relations are known. So far, only updates that really overwrite the old data have been considered [3], but the insertion of new possible worlds has not been addressed.

Another problem arises from the assumption that data is never lost. In practice, this will most probably not be possible or maybe not even desirable. While defining some agreement which data to delete can be easily done, to the best of our knowledge, also the problem of efficiently deleting possible worlds from U-relations has not yet been completely solved.

Finally we plan an implementation and experimental evaluation of the proposed framework.

References

1. G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734 – 749, june 2005.
2. L. Antova, T. Jansen, C. Koch, and D. Olteanu. Fast and simple relational processing of uncertain data. In *Proc. ICDE 2008*, pages 983–992. IEEE, 2008.
3. L. Antova and C. Koch. On APIs for probabilistic databases. In *Proc. of QDB/MUD '08*, pages 41–56, 2008.
4. C. Batini, C. Cappiello, C. Francalanci, and A. Maurino. Methodologies for data quality assessment and improvement. *ACM Comput. Surv.*, 41(3), 2009.
5. BIRN. `http://nbirn.net/cyberinfrastructure/portal.shtm`.
6. N. Dalvi, C. Ré, and D. Suciu. Probabilistic databases: diamonds in the dirt. *Commun. ACM*, 52(7):86–94, 2009.
7. W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu. Believe it or not: adding belief annotations to databases. *Proc. VLDB Endow.*, 2(1):1–12, 2009.
8. W. Gatterbauer and D. Suciu. Data conflict resolution using trust mappings. In *Proc. of SIGMOD 2010*, pages 219–230. ACM, 2010.
9. GEON. `http://www.geongrid.org/`.
10. T. Gruber. Collective knowledge systems: Where the social web meets the semantic web. *J. Web Sem.*, 6(1):4–13, 2008.
11. R. Gupta and S. Sarawagi. Creating probabilistic databases from information extraction models. In *VLDB*, pages 965–976, 2006.
12. Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. The ORCHESTRA collaborative data sharing system. *SIGMOD Rec.*, 37(3):26–32, 2008.
13. L. Kot and C. Koch. Cooperative update exchange in the Youtopia system. *PVLDB*, 2(1):193–204, 2009.
14. B. Leuf and W. Cunningham. *The Wiki Way – Quick Collaboration on the Web.* Addison-Wesley, 2001.
15. B. N. Levine, C. Shields, and N. B. Margolin. A Survey of Solutions to the Sybil Attack. Tech report 2006-052, University of Massachusetts Amherst, October 2006.
16. A. D. Sarma, M. Theobald, and J. Widom. Live: A lineage-supported versioned dbms. In *Proc. of SSDBM 2010*, volume 6187 of *Lecture Notes in Computer Science*, pages 416–433. Springer, 2010.
17. S. Singh, C. Mayfield, S. Mittal, S. Prabhakar, S. Hambrusch, and R. Shah. Orion 2.0: native support for uncertain data. In *SIGMOD '08:*, pages 1239–1242, New York, NY, USA, 2008. ACM.