Update Exchange with Mappings and Provenance

Todd J. Green Grigoris Karvounarakis Zachary G. Ives Val Tannen
Computer and Information Science Department
University of Pennsylvania
{tjgreen,gkarvoun,zives,val}@cis.upenn.edu

ABSTRACT

We consider systems for data sharing among heterogeneous peers related by a network of schema mappings. Each peer has a locally controlled and edited database instance, but wants to ask queries over related data from other peers as well. To achieve this, every peer's updates propagate along the mappings to the other peers. However, this **update exchange** is filtered by **trust conditions** — expressing what data and sources a peer judges to be authoritative — which may cause a peer to reject another's updates. In order to support such filtering, updates carry **provenance** information. These systems target scientific data sharing applications, and their general principles and architecture have been described in [20].

In this paper we present methods for realizing such systems. Specifically, we extend techniques from data integration, data exchange, and incremental view maintenance to propagate updates along mappings; we integrate a novel model for tracking data provenance, such that curators may filter updates based on trust conditions over this provenance; we discuss strategies for implementing our techniques in conjunction with an RDBMS; and we experimentally demonstrate the viability of our techniques in the ORCHESTRA prototype system.

1. INTRODUCTION

One of the most elusive goals of the data integration field has been supporting sharing across large, heterogeneous populations. While data integration and its variants (e.g., data exchange [12] and warehousing) are being adopted in corporations or small confederations, little progress has been made in integrating broader communities. Yet the need for sharing data across large communities is increasing: most of the physical and life sciences have become data-driven as they have attempted to tackle larger questions. The field of bioinformatics, for instance, has a plethora of different databases, each providing a different perspective on a collection of organisms, genes, proteins, diseases, and so on. Associations exist between the different databases' data (e.g., links between genes and proteins, or gene homologs between species). Unfortunately, data in this domain is surprisingly difficult to integrate, primarily because conventional data integration techniques require the development of a single global schema and complete global data consistency. Designing one schema for an entire community like systems biology is arduous, requires many revisions, and requires a central administrator[1]. Even more problematic is the fact that the data or associations in different databases are often contradictory, forcing individual biologists to choose values from databases they personally consider most authoritative [28]. Such inconsistencies are not handled by data integration tools, since there is no consensus or "clean" version of the data. Thus, scientists simply make their databases publicly downloadable, so users can copy and convert them into a local format (using custom Perl scripts or other ad hoc measures). Meanwhile the original data sources continue to be edited. In some cases the data providers publish weekly or monthly lists of updates (*deltas*) to help others keep synchronized. Today, few participants, except those with direct replicas, can actually exploit such deltas — hence, once data has been copied to a dissimilar database, it begins to diverge from the original.

To address the needs of scientists, we have previously proposed an extremely flexible scheme for sharing data among different participants, which, rather than providing a global view of all data, instead facilitates import and export among autonomous databases. The *collaborative data sharing system* (abbreviated CDSS) [20] provides a principled architecture that extends the data integration approach to encompass today's scientific data sharing practices and requirements: Our goal with the CDSS is to provide the basic capability of *update exchange*, which publishes a participant's updates to "the world" at large, and then maps others' updates to the participant's local schema — also filtering which ones to apply according to the local administrator's unique trust policies. Data sharing occurs among *loosely* coupled confederations of participants (peers). Each participant controls a local database instance, encompassing all data it wishes to manipulate, including data imported from other participants. As edits are made to this database, they are logged. Declarative *schema mappings* specify one database's relationships to other participants, much as in peer data management systems [18]; each peer operates in "offline" mode for as long as it likes, but it occasionally performs an update exchange operation, which propagates updates to make its database consistent with the others according to the schema mappings and local trust policies. The update exchange process is bidirectional and involves publishing any updates made locally by the participant, then importing new updates from other participants (after mapping them across schemas and filtering them according to trust conditions). As a prerequisite to assessing trust, the CDSS records the derivation of the updates, i.e., their provenance or lineage [4, 8, 7,

VLDB '07, September 23-28, 2007, Vienna, Austria.
Copyright 2007 VLDB Endowment, ACM 978-1-59593-649-3/07/09.

[1] Peer data management [3, 22, 27, 18] supports multiple mediated schemas, thus relaxing some aspects of administration, but it assumes data is consistent.

2]. After the update exchange, the participant has an up-to-date data instance, incorporating trusted changes made by participants transitively reachable via schema mappings. Update exchange introduces a number of novel challenges that we address in this paper:

- Building upon techniques for exchanging **data** using networks of schema mappings, we map **updates** across participants' schemas: this facilitates incremental view maintenance in a CDSS, and it also generalizes peer data management [18] and data exchange [31, 12].
- We allow each participant to perform modifications that "override" data imported from elsewhere (i.e., remove or replace imported data in the local instance), or add to it (i.e., add new data to the local database instance).
- In order to allow CDSS participants to specify what data they trust and wish to import, we propose the use of *semiring provenance* [16] to trace the derivation of each mapped update, and we filter updates with participant-specified *trust conditions*.
- We develop a complete implementation of update exchange in our ORCHESTRA CDSS prototype, with novel algorithms and encoding schemes to translate updates, maintain provenance, and apply trust conditions. We provide a detailed experimental study of the scalability and performance of our implementation.

This paper is part of an ongoing effort to realize the CDSS proposal of [20]. It complements our recent work in [32], which focuses on algorithms to reconcile conflicting updates made over the **same** schema. Here we not only develop a complete semantics for translating updates into a target schema, maintaining provenance, and filtering untrusted updates, but also a concrete implementation. This paper also builds upon [16], which presents a formal study of the provenance model that we implement in this paper. Finally, we note that our algorithms are of independent interest for incremental maintenance with provenance in other data integration and data exchange settings besides the CDSS.

Roadmap. Section 2 provides an overview of the CDSS update exchange process. Section 3 formalizes system operation and our data model. Section 4 presents algorithms for supporting update exchange, and Section 5 describes how we implemented them within the ORCHESTRA system. We experimentally evaluate our work in Section 6, describe related work in Section 7, and present our conclusions and future plans in Section 8.

2. CDSS UPDATE EXCHANGE

The CDSS model builds upon the fundamentals of data integration and peer data management [18], but adds several novel aspects. As in a PDMS, the CDSS contains a set of *peers*, each representing an autonomous domain of control. Each peer's administrator has full control over a local DBMS, its schema, and the conditions under which the peer trusts data. Without loss of generality, we assume that each peer has a schema disjoint from the others. In this paper we assume relational schemas, but our model extends naturally to other data models such as XML or RDF.

EXAMPLE 1. *Figure 1 illustrates an example bioinformatics collaborative data sharing system, based on a real application and databases of interest to affiliates of the Penn Center for Bioinformatics. GUS, the Genomics Unified Schema, contains gene expression, protein, and taxon (organism) information; BioSQL, affiliated with the BioPerl project, contains very similar concepts; and a third schema, uBio, establishes synonyms and canonical names for taxa. Instances of these databases contain taxon information that is autonomously maintained but of mutual interest to the others. Suppose that a peer with BioSQL's schema, $\mathbf{P}_{BioSQL}$, wants to import data from another peer, $\mathbf{P}_{GUS}$, as shown by the arc labeled m_1, but the converse is not true. Similarly, peer $\mathbf{P}_{uBio}$ wants to import data from $\mathbf{P}_{GUS}$, along arc m_2. Additionally, $\mathbf{P}_{BioSQL}$ and $\mathbf{P}_{uBio}$ agree to mutually share some of their data: e.g., $\mathbf{P}_{uBio}$ imports taxon synonyms from $\mathbf{P}_{BioSQL}$ (via m_3) and $\mathbf{P}_{BioSQL}$ uses transitivity to infer new entries in its database, via mapping m_4. Finally, each participant may have a certain* trust policy *about what data it wishes to incorporate: e.g., $\mathbf{P}_{BioSQL}$ may only trust data from $\mathbf{P}_{uBio}$ if it was derived from $\mathbf{P}_{GUS}$ entries. The CDSS facilitates dataflow among these systems, using mappings and policies developed by the independent participants' administrators.*

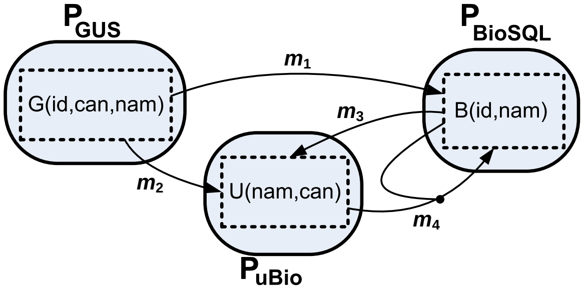


Figure 1: Example collaborative data sharing system for three bioinformatics sources. For simplicity, we assume one relation at each participant ($\mathbf{P}_{GUS}$, $\mathbf{P}_{BioSQL}$, $\mathbf{P}_{uBio}$). Schema mappings are indicated by labeled arcs.

The arcs between participants in the example are formally a set of *schema mappings*. These are logical assertions that relate multiple relations from different peer schemas. We adopt the well-known formalism of *tuple-generating dependencies* (tgds). Tgds are a popular means of specifying constraints and mappings [12, 9] in data sharing, and they are equivalent to so-called *global-local-as-view* or *GLAV* mappings [14, 18], which in turn generalize the earlier global-as-view and local-as-view mapping formulations [23]. A tgd is a logical assertion of the form:

$$\forall \bar{x}, \bar{y}\ (\phi(\bar{x}, \bar{y}) \rightarrow \exists \bar{z}\ \psi(\bar{x}, \bar{z}))$$

where the left hand side (LHS) of the implication, ϕ, is a conjunction of atoms over variables $\bar{x}$ and $\bar{y}$, and the right hand side (RHS) of the implication, ψ, is a conjunction of atoms over variables $\bar{x}$ and $\bar{z}$. The tgd expresses a constraint about the existence of a tuple in the instance on the RHS, given a particular combination of tuples satisfying the constraint of the LHS.

Notation. We use Σ for the **union** of all peer schemas and $\Sigma(\mathbf{P})$ for the schema of peer **P**. We use $\mathcal{M}$ for the set of all mappings, which we can think of as logical constraints on Σ. When we refer to mappings we will use the notation of tgds. For readability, we will omit the universal quantifiers for variables in the LHS. When we later refer to queries, including queries based on mappings, we will use the similar notation of datalog (which, however, reverses the order of implication, specifying the output of a rule on the left).

EXAMPLE 2. *Refer to Figure 1. Peers $\mathbf{P}_{GUS}$, $\mathbf{P}_{BioSQL}$, $\mathbf{P}_{uBio}$ have one-relation schemas describing taxa IDs, names, and canonical names: $\Sigma(\mathbf{P}_{GUS}) = \{G(id, can, nam)\}$, $\Sigma(\mathbf{P}_{BioSQL}) = \{B(id, nam)\}$, $\Sigma(\mathbf{P}_{uBio}) = \{U(nam, can)\}$.*

Among these peers are mappings $\mathcal{M} = \{m_1, m_2, m_3, m_4\}$, *shown as arcs in the figure. The mappings are:*

$$\begin{array}{ll} (m_1) & G(i,c,n) \rightarrow B(i,n) \\ (m_2) & G(i,c,n) \rightarrow U(n,c) \\ (m_3) & B(i,n) \rightarrow \exists c\, U(n,c) \\ (m_4) & B(i,c) \wedge U(n,c) \rightarrow B(i,n) \end{array}$$

Observe that m_3 has an existential variable: the value of c is unknown (and not necessarily unique). The first three mappings all have a single source and target peer, corresponding to the LHS and the RHS of the implication. In general, relations from multiple peers may occur on either side, as in mapping m_4, which defines data in the BioSQL relation based on its own data combined with tuples from uBio.

CDSS operation. Each peer **P** represents an autonomous domain with its own unique schema and associated *local data instance*. The users located at **P** query and update the local instance in an "offline" fashion. Their updates are recorded in a *local edit log*. Periodically, upon the initiative of **P**'s administrator, **P** requests that the CDSS perform an *update exchange* operation. This *publishes* **P**'s local edit log — making it globally available via central or distributed storage [32]. This also subjects **P** to the effects of the updates that the other peers have published (since the last time **P** participated in an update exchange). To determine these effects, the CDSS performs *update translation* (overview in Section 2.1), using the schema mappings to compute corresponding updates over **P**'s schema. As the updates are being translated, they are also filtered based on **P**'s *trust* conditions that use the *provenance* (overview in Section 2.2) of the data in the updates. As a result, only trusted updates are applied to **P**'s database, whereas untrusted data is *rejected*. Additional rejections are the result of manual curation: If a local user deletes data that was *not* inserted by **P**'s users (and hence must have arrived at **P** via update exchange), then that data remains rejected by **P** in future update exchanges of the CDSS.

After update exchange, **P**'s local instance will intuitively contain data mapped from other peers, "overlaid" by any updates made by **P**'s users and recorded in the local edit log. If **P**'s instance is subsequently updated locally, then **P**'s users will see the effects of these edits. Other peers in the CDSS will only see data that resulted from the **last** update exchange, i.e., they will not see the effects of any unpublished updates at **P**. This situation continues until **P**'s next update exchange.

Intuitively, this operating mode resembles deferred view maintenance across a set of views — but there are a number of important differences, in part entailed by the fact that instances are related by schema mappings rather than views, and in part entailed by peers' ability to specify local edits and trust policies. In the remainder of this section, we provide a high-level overview of these characteristic aspects of update exchange. Section 3 will revisit these aspects in order to define our model formally.

Application of updates. The result of update translation, once trust policies have been enforced over provenance and data values, is a set of updates to the local instances. In this paper, we assume that these updates are mutually compatible. A more realistic approach would treat them as *candidate* updates and further use the prioritization and conflict reconciliation algorithm of [32] to determine which updates to apply. In fact, we do so in our ORCHESTRA prototype implementation, but for simplicity of presentation we ignore this aspect in the model formalization that follows.

2.1 Update Translation and Query Answers

The description of the peer instances during CDSS operation that we give here is *static*, emphasizing what data should end up where, rather than *when* it arrives. The same approach is taken in the formalization in Section 3.1. This is essential in order to provide an intelligible semantics for query answers. However, the practical approach to CDSS implementation does rely on a dynamic, *incremental* way of achieving this semantics (see Section 4.2).

The "source" or "base" data in a CDSS, as seen by the users, are the local edit logs at each peer. These edit logs describe local data creation and curation in terms of insertions and deletions/rejections. Of course, a local user submitting a query expects answers that are fully consistent with the local edit log. With respect to the *other* peers' edit logs the user would expect to receive all *certain* answers inferable from the schema mappings and the tuples that appear in the other peers' instances [18]. Indeed, the certain answers semantics has been validated by over a decade of use in data integration and data exchange [24, 14, 12, 22, 18].

Queries are answered in a CDSS using only the local peer instance. Hence, the content of this instance must be such that all and only answers that are certain (as explained above) and consistent with the local edit log are returned. As was shown in the work on data exchange [12, 25], the certain answer semantics can in fact be achieved through a form of "data translation", building peer instances called *canonical universal solutions*. In our case, the source data consists of edit logs so we generalize this to *update translation*. A key aspect of the canonical universal solutions is the *placeholder values* or *labeled nulls* for unknown values that are nonetheless needed in order to validate mappings with existentials (such as m_3 in Example 2). The labeled nulls are internal bookkeeping (e.g., queries can join on their equality), but tuples with labeled nulls are discarded in order to produce certain answers to queries. (We can additionally return tuples with labeled nulls, i.e., a superset of the certain answers, which may be desirable for some applications.)

EXAMPLE 3. *Continuing our example, assume that the peers have the following local edit logs (where '$+$' signifies insertion):*

ΔG

$+$	1	2	3
$+$	3	5	2

ΔB

$+$	3	5

ΔU

$+$	2	5

The update translation constructs local instances that contain:

G

1	2	3
3	5	2

B

3	5
3	2
1	3
3	3

U

2	5
3	2
5	c_1
2	c_2
3	c_3

Examples of certain answers query semantics at $\mathbf{P}_{BioSQL}$:

- query $ans(x,y)$:- $U(x,z), U(y,z)$ returns $\{(2,2),(3,3),(5,5)\}$*;*
- query $ans(x,y)$:- $U(x,y)$ returns $\{(2,5),(3,2)\}$.

Moreover, if the edit log ΔB would have also contained the curation deletion $(- \mid 3 \quad 2)$ *then after update translation, B would not only be missing* $(3,2)$*, but also* $(3,3)$*; and U would be missing* $(2,c_2)$.

Finally, this example suggests that the set semantics is not telling the whole story. For example the tuple $U(2,5)$ has two different "justifications": it is a local insertion as well as the result of update translation via (m_2). The tuple $B(3,2)$ comes from two different update translations, via (m_1) and via (m_4).

The challenge in the CDSS model is that peer instances cannot be computed merely from schema mappings and data instances. The ability of all peers to do curation deletions and trust-based rejections requires a new formalization that takes edit logs and trust policies into account. We outline in Section 3.1 how we can do that and still take advantage of the techniques for building canonical universal solutions.

2.2 Trust Policies and Provenance

In addition to schema mappings, which specify the relationships between data elements in different instances, the CDSS supports *trust policies*. These express, for each peer **P**, what data from update translation should be trusted and hence accepted. The trust policies consist of *trust conditions* that refer to the other peers, to the schema mappings, and even to selection predicates on the data itself. Different trust conditions may be specified separately by each peer, and we discuss how these compose in Section 3.3.

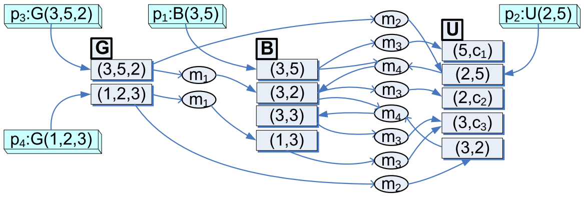
EXAMPLE 4. *Some possible trust conditions in our CDSS example:*

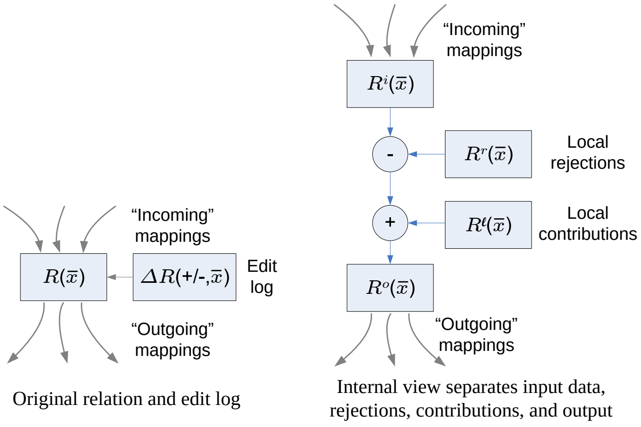
- *Peer* $\mathbf{P}_{BioSQL}$ *distrusts any tuple* $B(i, n)$ *if the data came from* $\mathbf{P}_{GUS}$ *and* $n \geq 3$*, and trusts any tuple from* $\mathbf{P}_{uBio}$*.*
- *Peer* $\mathbf{P}_{BioSQL}$ *distrusts any tuple* $B(i, n)$ *that came from mapping* (m_4) *if* $n \neq 2$*.*

Adding these trust conditions to the update exchange in Example 3 we see that $\mathbf{P}_{BioSQL}$ *will reject* $B(1, 3)$ *by the first condition. As a consequence,* $\mathbf{P}_{uBio}$ *will not get* $U(3, c_3)$*. Moreover, the second trust condition makes* $\mathbf{P}_{BioSQL}$ *reject* $B(3, 3)$*. Note that formulations like "comes from* $\mathbf{P}_{GUS}$*" need a precise meaning. We give this in Section 3.2.*

Since the trust conditions refer to other peers and to the schema mappings, the CDSS needs a precise description of how these peers and mappings have contributed to a given tuple produced by update translation. Information of this kind is commonly called *data provenance*. As we saw at the end of Example 3, provenance can be quite complex in a CDSS. In particular, we need a more detailed provenance model than why-provenance [4] and lineage [8] (including the extended model recently proposed in [2]). We discuss our needs more thoroughly in Section 7, but informally, we need to know not just from which tuples a tuple is derived, but also *how* it is derived, including separate alternative derivations through different mappings. We present this model more formally in Section 3.2 while here we illustrate our running example with a graphical representation of provenance.

EXAMPLE 5. *Consider the update translation from Example 3. Build a graph with two kinds of nodes: tuple nodes, shown as rectangles below, and mapping nodes, shown as ellipses. Arcs connect tuple nodes to mapping nodes and mapping nodes to tuple nodes. In addition, we have nodes for the insertions from the local edit logs. This "source" data is annotated with its own id (unique in the system)* $p_1, p_2, \ldots$ *etc., and is connected by an arc to the corresponding tuple entered in the local instance.*





Original relation and edit log

Internal view separates input data, rejections, contributions, and output

Figure 2: To capture the effects of the edit log on each relation (left), we internally encode them as four relations (right), representing incoming data, local rejections and local contributions, and the resulting ("output") table.

From this graph we can analyze the provenance of, say, $B(3, 2)$ *by tracing back paths to source data nodes — in this case through* (m_4) *to* p_1 *and* p_2 *and through* (m_1) *to* p_3*.*

3. UPDATE EXCHANGE FORMALIZED

After the informal overview in the previous section, we now provide a formal discussion of how local edits, schema mappings, and trust conditions work together in a CDSS. In particular, we extend the model proposed in the data exchange literature [12], which specifies how to compute peer instances, given data at other peers. We discuss how to incorporate updates into the computation of data exchange solutions, which we simply term *update translation*; and how to integrate trust conditions and provenance into the computation.

3.1 Update Translation

We first explain how the system automatically expands the user-level schemas and mappings into "internal" schemas and mappings. These support data exchange and additionally capture how edit log deletions and trust conditions are used to reject data translated from other peers. First, we state two fundamental assumptions we make about the form of the mappings and the updates.

We allow the set of mappings in the CDSS to only form certain types of *cycles* (i.e., mappings that recursively define relations in terms of themselves). In general, query answering is undecidable in the presence of cycles [18], so we restrict the topology of schema mappings to be at most *weakly acyclic* [12, 10]. Mapping (m_3) in Example 2 completes a cycle, but the set of mappings is weakly acyclic.

We also assume that within the set of updates published at the same time by a peer, no data dependencies exist (perhaps because transient operations in update chains were removed [20]). These updates are stored in an *edit log*. For each relation $R(\bar{x})$ in the local instance we denote by $\Delta R(d, \bar{x})$ the corresponding edit log. ΔR is an ordered list that stores the results of manual curation at the peer, namely the inserted tuples whose d value is '$+$' and the deleted tuples whose d value is '$-$'.

Internal peer schemas. For each relation R in Σ, the user-level edit log ΔR and the local instance R are implemented internally by four different relations, all with the same attributes as R. We illustrate these relations in Figure 2. Their meaning is as follows:

- R^ℓ, the peer's *local contributions table*, contains the tuples inserted locally, unless the edit log shows they were later deleted.
- R^r, the peer's *rejections table*, contains tuples that were not inserted locally and that are rejected through a local curation deletion. (Deletions of local contributions are dealt with simply by removing tuples from R^ℓ).
- R^i is the peer's *input table*. It contains tuples produced by update translation, via mappings, from data at other peers.
- R^o is the peer's curated table and also its *output table*. After update exchange, it will contain the local contributions as well as the input tuples that are not rejected. This table is the source of the data that the peer exports to other peers through outgoing mappings. This is also the table that the peer's users query, called the local instance in Section 2.1 and Example 3.

Internal schema mappings. Along with expanding the original schema into the internal schema, the system transforms the original mappings $\mathcal{M}$ into a new set of tgds $\mathcal{M}'$ over the internal schema, which are used to specify the effects of local contributions and rejections.

- For each tgd m in $\mathcal{M}$, we have in $\mathcal{M}'$ a tgd m' obtained from m by replacing each relation R on the LHS by R^o and each relation R on the RHS by R^i;
- For each R in Σ, $\mathcal{M}'$ is extended with rules to remove tuples in R^r and add those in R^ℓ:

$$\begin{array}{ll} (i_R) & R^i(\bar{x}) \wedge \neg R^r(\bar{x}) \rightarrow R^o(\bar{x}) \\ (\ell_R) & R^\ell(\bar{x}) \rightarrow R^o(\bar{x}). \end{array}$$

Note that the previous description and Figure 2 do not incorporate the treatment of trust conditions. We show how to incorporate them at the end of Section 3.3.

Some of the new mappings in $\mathcal{M}'$ contain negation, and thus one might wonder if this affects the decidability of query evaluation. Note that the negated atoms occur only on the LHS of the implication and every variable in the LHS also occurs in a positive atom there: we call such an assertion a *tgd with safe negation*. The results of [12, 25] generalize to tgds with safe negation.

THEOREM 3.1. *Let $\mathcal{M}$ be weakly acyclic and I be an instance of all local contributions and rejections tables.Then $\mathcal{M}'$ is also weakly acyclic, hence chase$_{\mathcal{M}'}(I)$ terminates in polynomial time, and moreover it yields an instance of the of input and output tables that is a canonical universal solution for I with respect to $\mathcal{M}'$.*

To recompute the relation input and output instances based on the extensional data in the local contributions and rejection tables, we use a procedure based on the chase of [12].

DEFINITION 3.1 (CONSISTENT SYSTEM STATE). *An instance (I, J) of Σ', where I is an instance of the local rejections and contributions tables and J of the input and output tables, is* **consistent** *if J is a canonical universal solution for I with respect to $\mathcal{M}'$.*

To recap, *publishing* a peer **P**'s edit log means producing a new instance of **P**'s local rejections and contributions tables, while holding the other peers' local rejections and contributions tables the same[2]. *Recomputing* **P**'s instance means computing a new instance of **P**'s input and output tables that is a canonical universal solution with respect to the internal mappings $\mathcal{M}'$. By definition, after each update exchange the system must be in a consistent state. (The initial state, with empty instances, is trivially consistent.)

3.2 Provenance Expressions and Graphs

As explained in Section 2.2, CDSS operation needs a data provenance model that is more expressive than why-provenance or lineage. We describe the essential aspects of the model in this section, relying on some properties that were shown in our theoretical study of the model [16], which also justifies the choice of "semirings" for the formalism.

In our model the provenance of a source (base) tuple is represented by its own tuple id (we call this a *provenance token*), and that of a tuple computed through update translation is an expression over a *semiring*, with domain the set of all provenance tokens; two operations, $+$ and $\cdot$; and one unary function for every mapping. Intuitively, $\cdot$ is used to combine the provenance of tuples in a join operation, mapping functions reflect the fact that the corresponding mapping was involved in the derivation, and $+$ is used in cases when a tuple can be derived in multiple ways (e.g. through different mappings). Using the terminology of proof-theoretic datalog evaluation semantics, every summand in a provenance expression corresponds to a *derivation tree* for that tuple in the result of the "program" formed by the mappings.

Notation. We write $\mathsf{Pv}(R(t))$ for the provenance expression of the tuple t in relation R. When the relation name is unimportant, we simply write $\mathsf{Pv}(t)$. We sometimes omit $\cdot$ and use concatenation.

EXAMPLE 6. *To illustrate how the provenance expressions are formed, consider the mappings from Example 2:*

$$\begin{array}{ll} (m_1) & G(i,c,n) \rightarrow B(i,n) \\ (m_3) & B(i,n) \rightarrow \exists c\, U(n,c) \\ (m_4) & B(i,c) \wedge U(n,c) \rightarrow B(i,n) \end{array}$$

If $\mathsf{Pv}(B(3,5)) = p_1$, $\mathsf{Pv}(U(2,5)) = p_2$, *and* $\mathsf{Pv}(G(3,5,2)) = p_3$ *then* $\mathsf{Pv}(B(3,2)) = m_1(p_3) + m_4(p_1 p_2)$.

Moreover, $\mathsf{Pv}(U(3,c_3)) = m_3(\mathsf{Pv}(B(3,2))) = m_3(m_1(p_3)) + m_3(m_4(p_1 p_2))$

Note that the expressions allow us to detect when the derivation of a tuple is "tainted" by a peer or by a mapping. Distrusting p_2 and m_1 leads to rejecting $B(3,2)$ but distrusting p_1 and p_2 does not.

When the mappings form cycles, it is possible for a tuple to have infinitely many derivations, as well as for the derivations to be arbitrarily large. In general, as it was explained in [16], the provenance of a tuple is akin to infinite *formal power series*. Nonetheless the provenances are finitely representable through a *system of equations*: on the head of each equation we have a unique provenance variable $\mathsf{Pv}(t)$ for every tuple t (base or derived), while in the body we have all semiring expressions that represent derivations of this tuple as an *immediate consequent* from other tuples (or provenance tokens, if t is a base tuple). Then, as it was proven in [16], the provenance of a tuple t is the value of $\mathsf{Pv}(t)$ in the solution of the system formed by all these equations.

We can alternatively look at the system of provenance equations as forming a graph just like the one in Example 5.

DEFINITION 3.2 (PROVENANCE GRAPH). *The graph has two types of nodes:* tuple nodes*, one for each tuple in the system and*

[2] This easily generalizes to updates over multiple instances.

mapping nodes *where several such nodes can be labeled by the same mapping name. Each mapping node corresponds to an instantiation of the mapping's tgds. This instantiation defines some tuples that match the LHS of the tgd and we draw edges from the corresponding tuple nodes to the mapping node (this encodes conjunction among source tuples). It also defines some tuples that match the RHS of the tgd and we draw edges from the mapping node to the tuple nodes that correspond to them. Multiple incoming edges to a tuple node encode the fact that a tuple may be derived multiple ways. Finally, some tuples are the result of direct user insertions; these are annotated with globally unique* ***provenance tokens****, which we represent as extra labels on the tuple nodes.*

One can generate provenance expressions from the provenance graph, by traversing it recursively backwards along the arcs as in Example 5. It is easy to prove formally that these expressions are the same as the solutions of the system of equations.

3.3 Assigning Trust to Provenance

In Section 2, we gave examples of *trust conditions* over provenance and data. Now that we have discussed our model for provenance, we specify how to determine trust assignments for tuples.

Within a CDSS, every tuple derives from base insertions within local contributions tables. We assume that each peer specifies a set of tuples to initially trust; in effect, each tuple is annotated with **T**, representing that it is trusted, or **D**, indicating it is not. Additionally, each peer annotates each schema mapping m_i with a *trust condition*, Θ_i. As tuples are derived during update exchange, those that derive only from trusted tuples and satisfy the trust conditions Θ_i along every mapping are marked as trusted. All other tuples are distrusted.

A finite provenance expression can simply be evaluated for "trustworthiness" as follows. For expressions of the form $m(p_1 \cdot p_2)$, we can simply map **T** to boolean **true** and **D** to **false**, then evaluate the provenance expression as a boolean equation, where $\cdot$ represents conjunction, $+$ represents disjunction, and the application of mapping m represents a conjunction of the mapping's trust assignment with that of its arguments.

Trust composes along a mapping path as follows: if peer P_i has a schema mapping (m_{ij}) from its neighbor P_j, then P_i will receive updates via mapping (m_{ij}) if (1) they derive from updates trusted by P_j, and (2) the updates additionally satisfy the trust conditions imposed by P_i. In essence, the trust conditions specified by a given peer are combined (ANDed) with the additional trust conditions specified by anyone mapping data from that peer. A peer delegates the ability to distrust tuples to those from whom it maps data.

EXAMPLE 7. *In Example 6 we calculated the provenances of some exchanged tuples. Suppose now that peer $\mathbf{P}_{BioSQL}$ trusts data contributed by $\mathbf{P}_{GUS}$ and itself, and hence assigns* **T** *to the provenance token p_3 and p_1, but does not trust $\mathbf{P}_{uBio}$'s tuple* $(2, 5)$ *and so assigns* **D** *to p_2. Assuming that all mappings have the trivial trust conditions* **T***, the provenance of* $(3, 2)$ *evaluates as follows:*

$$\mathbf{T} \cdot \mathbf{T} + \mathbf{T} \cdot \mathbf{T} \cdot \mathbf{D} = \mathbf{T}$$

therefore $\mathbf{P}_{BioSQL}$ should indeed have accepted $(3, 2)$.

We can now fully specify the mappings necessary to perform update exchange — which combines update translation with trust. For each relation R, the trust conditions are applied during update exchange to the input instance R^i of the peer, thus selecting the trusted tuples from among all the tuples derived from other peers. The result is an internal relation we can denote R^t. So instead of the internal mappings (i_R) described in Section 3.1 we actually have in $\mathcal{M}'$:

$$\begin{array}{ll} (i_R) & R^t(\bar{x}) = trusted(R^i(\bar{x})) \\ (t_R) & R^t(\bar{x}) \wedge \neg R^r(\bar{x}) \rightarrow R^o(\bar{x}) \end{array}$$

and the definition of consistent state remains the same.

4. PERFORMING UPDATE EXCHANGE

We now discuss how to actually compute peer data instances in accordance with the model of the previous section. We begin with some preliminaries, describing how we express the computations as queries and how we model provenance using relations. Then we describe how incremental update exchange can be attained.

In order to meet participants' needs for anonymity (they want all data and metadata to be local in order to prevent others from snooping on their queries), our model performs all update exchange computation locally, in auxiliary storage alongside the original DBMS (see Section 5). It imports any updates made directly by others and incrementally recomputes its own copy of all peers' relation instances and provenance — also filtering the data with its own trust conditions as it does so. Between update exchange operations, it maintains copies of all relations, enabling future operations to be incremental. In ongoing work, we are considering more relaxed models in which portions of the computation may be distributed.

4.1 Computing Instances with Provenance

In the literature [12, 25], chase-based techniques have been used for computing candidate universal solutions. However, these are primarily of theoretical interest and cannot be directly executed on a conventional query processor. In constructing the ORCHESTRA system, we implement update exchange using **relational query processing** techniques, in order to take advantage of robust existing DBMS engines, as well as to ultimately leverage multi-query optimization and distributed query execution. We encode the provenance in relations alongside the data, making the computation of the data instances and their provenance a seamless operation. The Clio system [31] used similar techniques to implement data exchange but did not consider updates or provenance.

4.1.1 Datalog for Computing Peer Instances

Work in data integration has implemented certain types of chase-like reasoning with relational query processors for some time [11, 31]: datalog queries are used to compute the certain answers [5] to queries posed over integrated schemas. Our goal is to do something similar. However, when computing canonical universal solutions for the local peer instances, we face a challenge because the instance may contain *incomplete* information, e.g., because not all attributes may be provided by the source. In some cases, it may be known that two (or more) values are actually **the same**, despite being of unknown value[3]. Such cases are generally represented by using existential variables in the target of a mapping tgd (e.g., (m_3) in Example 2). This requires *placeholder values* in the canonical universal solution. Chase-style procedures [12, 25] use *labeled nulls* to encode such values. In our case, since we wish to use datalog-like queries, we rely on *Skolem functions* to specify the placeholders, similarly to [31]. Each such function provides a unique placeholder value for each combination of inputs; hence two placeholder values will be the same if and only if they were generated with the same Skolem function with the same arguments.

[3] This enables joins on tuples on unknown values that may result in additional answers.

Normal datalog does not have the ability to compute Skolem functions; hence, rather than converting our mapping tgds into standard datalog, we instead use a version of datalog extended with Skolem functions. (Section 5 discusses how these queries can in turn be executed on an SQL DBMS.)

Notation. Throughout this section, we will use the syntax of datalog to represent queries. Datalog rules greatly resemble tgds, except that the output of a datalog rule (the "head") is a single relation and the head occurs to the **left** of the body. The process of transforming tgds into datalog rules is the same as that of the *inverse rules* of [11]:

$$\phi(\bar{x},\bar{y}) \rightarrow \exists \bar{z}\ \psi(\bar{x},\bar{z}) \text{ becomes } \psi(\bar{x},\bar{f}(\bar{x})) \text{ :- } \phi(\bar{x},\bar{y})$$

We convert the RHS of the tgd into the head of a datalog rule and the LHS to the body. Each existential variable in the RHS is replaced by a Skolem function over the variables in common between LHS and RHS: $\bar{z}$ is replaced by $\bar{f}(\bar{x})$.[4]

Our scheme for parameterizing Skolem functions has the benefit of producing universal solutions (as opposed, e.g., to using a subset of the common variables) while guaranteeing termination for weakly acyclic mappings (which is not the case with using all variables in the LHS, as in [19]). If ψ contains multiple atoms in its RHS, we will get multiple datalog rules, with each such atom in the head. Moreover, it is essential to use a **separate** Skolem function for each existentially quantified variable in each tgd.

EXAMPLE 8. *Recall schema mapping (m_3) in Example 2. Its corresponding internal schema mapping:*

$$B^o(i,n) \rightarrow \exists c\, U^i(n,c) \textit{ becomes } U^i(n,f(n)) \text{ :- } B^o(i,n)$$

We call the resulting datalog rules **mapping rules** and we use $P_{\mathcal{M}}$ to denote the datalog program consisting of the mapping rules derived from the set of schema mappings $\mathcal{M}$.

PROPOSITION 1. *If $\mathcal{M}$ is a weakly acyclic set of tgds, then $P_{\mathcal{M}}$ terminates for every edb instance $\mathcal{I}$ and its result, $P_{\mathcal{M}}(\mathcal{I})$, is a canonical universal solution.*

This basic methodology produces a program for recomputing CDSS instances, given a datalog engine with fixpoint capabilities.

4.1.2 Incorporating Provenance

We now show how to encode the provenance graph *together* with the data instances, using additional relations and datalog rules. This allows for seamless recomputation of both data and provenance and allows us to better exploit conventional relational processing.

We observe that in a set-based relational model, there exists a simple means of generating a unique ID for each base tuple in the system: within any relation, a tuple is uniquely identified by its values. We exploit this in lieu of generating provenance IDs: for the provenance of $G(3,5,2)$, instead of inventing a new value p_3 we can use $G(3,5,2)$ itself.

Then, in order to represent a product in a provenance expression, which appears as a result of a join in the body of a mapping rule, we can record all the tuples that appear in an instantiation of the body of the mapping rule. However, since some of the attributes in those tuples are always equal in all instantiations of that rule (i.e., the same variable appears in the corresponding columns of the atoms in the body of the rule), it suffices to just store the value of each unique variable in a rule instantiation. To achieve this, for each mapping rule

$$(m_i) \quad R(\bar{x},\bar{f}(\bar{x})) \text{ :- } \phi(\bar{x},\bar{y})$$

we introduce a new relation $P_{Ri}(\bar{x},\bar{y})$ and we replace (m_i) with the mapping rules

$$\begin{array}{ll} (m_i') & P_{Ri}(\bar{x},\bar{y}) \text{ :- } \phi(\bar{x},\bar{y}) \\ (m_i'') & R(\bar{x},\bar{f}(\bar{x})) \text{ :- } P_{Ri}(\bar{x},\bar{y}) \end{array}$$

Note that (m_i') mirrors m_i but *does not project any attributes*. Note also that (m_i'') derives the actual data instance from the provenance encoding.

EXAMPLE 9. *Since tuples in B (as shown in the graph of Example 5) can be derived through mappings m_1 and m_4, we can represent their provenance using two relations P_{B1} and P_{B4} (named based on their origins of B, and (m_1) and (m_4), respectively), using the mapping rules:*

$$\begin{array}{l} P_{B1}(i,c,n) \text{ :- } G(i,c,n) \\ B(i,n) \text{ :- } P_{B1}(i,c,n) \\ P_{B4}(i,n,c) \text{ :- } B(i,c), U(n,c) \\ B(i,n) \text{ :- } P_{B4}(i,n,c) \end{array}$$

Recall from example 6 that $\mathsf{Pv}(B(3,2)) = m_1(\mathsf{Pv}(G(3,5,2))) + m_4(\mathsf{Pv}(B(3,5))\mathsf{Pv}(U(2,5)))$. *The first part of this provenance expression is represented by the tuple $P_{B1}(3,5,2)$ and the second by the tuple $P_{B4}(3,2,5)$. (Both can be obtained by assigning $i = 3, n = 2, c = 5$ in each of the rules above.)*

As we detail in Section 5, we can further optimize this representation in order to efficiently implement it in an RDBMS.

4.1.3 Derivation Testing

In addition to computing instances and their provenance, we can use datalog rules to determine *how a tuple was derived.* This is useful in two contexts: first, to assess whether a tuple should be trusted, which requires that it be derivable only from trusted sources and along trusted mappings; and second, to see if it is derivable even if we remove certain tuples, which is necessary for performing incremental maintenance of CDSS instances.

The challenge is to compute the set of base insertions from which a tuple (or set of tuples) was derived in a *goal-directed* way. In essence, this requires reversing or inverting the mappings among the provenance relations. We do this by first creating a new relation R_{chk} for each R, which we populate with the tuples whose derivation we wish to check. Then we create a relation R' for every additional R; R' will hold the tuples from the corresponding R that are part of the derivation tree. Now, for each mapping rule specifying how to derive a relation R from P_{Ri}:

$$(m_i'') \quad R(\bar{x},\bar{f}(\bar{x})) \text{ :- } P_{Ri}(\bar{x},\bar{y})$$

we define an inverse rule that uses the existing provenance table to fill in the possible values for $\bar{f}(\bar{x})$, namely the $\bar{y}$ attributes that were projected away during the mapping. This results in a new relation P_{Ri}' with exactly those tuples from which R can be derived using mapping P_{Ri}:

$$P_{Ri}'(\bar{x},\bar{y}) \text{ :- } P_{Ri}(\bar{x},\bar{y}), R_{chk}(\bar{x})$$

Now, for each mapping rule:

$$(m_i') \quad P_{Ri}(\bar{x},\bar{y}) \text{ :- } \phi(\bar{x},\bar{y})$$

[4] Although the term has been occasionally abused by computer scientists, our use of *Skolemization* follows the standard definition from mathematical logic.

first expand the RHS using the rest of the rules until the RHS has only relations of the form P_{Rj}. Let ϕ' be the result, Then derive the set of inverse rules:

$$\phi'(\bar{x}, \bar{y}) \text{ :- } P'_{Ri}(\bar{x}, \bar{y})$$

(one rule with each of the atoms of ϕ' in the head).

Finally, again for each rule of the form

$$(m''_i) \quad R(\bar{x}, \bar{f}(\bar{x})) \text{ :- } P_{Ri}(\bar{x}, \bar{y})$$

derive a corresponding version to relate R' and P'_{Ri}:

$$R'(\bar{x}, \bar{f}(\bar{x})) \text{ :- } P'_{Ri}(\bar{x}, \bar{y})$$

If we take the resulting program and run it until it reaches fixpoint in computing the R' relations, then we will have the set of tuples from which the original R_{chk} relations could have been derived. In general, we will perform one final step, which is to filter the R' relations (e.g., to only include values we trust, or values from local contributions tables) and validate that the R_{chk} tuples can indeed be re-derived if we run the original datalog program over the R' instances.

4.2 Incremental Update Exchange

One of the major motivating factors in our choice of provenance formalisms has been the ability to *incrementally maintain* the provenance associated with each tuple, and also the related data instances. We now discuss how this can be achieved using the relational encoding of provenance of Section 4.1.2.

Following [17] we convert each mapping rule (after the relational encoding of provenance) into a series of *delta rules*. For the insertion delta rules we use new relation names of the form R^+, P^+_{Ri}, etc. while for the deletion delta rules we use new relation names of the form R^-, P^-_{Ri}, etc.

For the case of **incremental insertion** in the absence of peer-specific trust conditions, the algorithm is simple, and analogous to the **counting** and DRed incremental view maintenance algorithms of [17]: we can directly evaluate the insertion delta rules until reaching a fixpoint and then add R^+ to R, P^+_{Ri} to P_{Ri}, etc. Trust combines naturally with the incremental insertion algorithm: the starting point for the algorithm is already-trusted data (from the prior instance), plus new "base" insertions which can be directly tested for trust (since their provenance is simply their source). Then, as we derive tuples via mapping rules from trusted tuples, we simply apply the associated trust conditions to ensure that we only derive new trusted tuples.

Incremental deletion is significantly more complex. When a tuple is deleted, it is possible to remove any provenance expressions and tuples that are its immediate consequents and are no longer directly derivable. However, the provenance graph may include cycles: it is possible to have a "loop" in the provenance graph such that several tuples are mutually derivable from one another, yet none are derivable from edbs, i.e., local contributions from some peer in the CDSS. Hence, in order to "garbage collect" these no-longer-derivable tuples, we must test whether they are derivable from trusted base data in local contributions tables; those tuples that are not must be recursively deleted following the same procedure.

Suppose that we are given each list of initial updates $\bar{R}^-$ from all of the peers. Our goal is now to produce a set of R^i update relations for the peer relations and a corresponding set P^-_{Ri} to apply to each provenance relation. Figure 3 shows pseudocode for such an algorithm. First, the algorithm derives the deletions to apply to the provenance mapping relations; based on these, it computes a new version of the peer schema relations and their associated provenance relations (Lines 4–8). Next, it must determine whether a tuple in the instance is no longer derivable (Lines 10-16): such tuples must also be deleted. The algorithm first handles the case where the tuple is not directly derivable (Line 13), and then it performs a more extensive test for derivability from edbs (Line 16). The "existence test" is based on the derivation program described in Section 4.1.3, which determines the set of edb tuples that were part of the original derivation. Given that set, we must actually validate that each of our R_{chk} tuples are *still* derivable from these edbs, by re-running the original set of schema mappings on the edb tuples.

```
Algorithm PropagateDelete
1.  for every $P_{Ri}$, let $R^0 \leftarrow R$
2.  Initialize $c \leftarrow 0$
3.  repeat
4.      Compute all $P^-_{Ri}$ based on their delta rules
5.      ($*$ Propagate effects of deletions $*$)
6.      for each idb $R$
7.          do update each associated $P_{Ri}$, by applying $P^-_{Ri}$ to it
8.              Define new relation $R^{c+1}$ to be the union of all
                $P_{Ri}$, projected to the $\bar{x}$ attributes.
9.      ($*$ Check tuples whose provenance was affected $*$)
10.     for each idb $R$
11.         do Let $R_{chk}$ be an empty temporary relation with $R$'s
                schema
12.             for each tuple $R^c(\bar{a})$ not in $R^{c+1}(\bar{a})$
13.                 do if in there exists, in any provenance relation
                        $P_{Ri}$ associated with $R$, a tuple $(\bar{a}, \bar{y}_i)$
14.                     then add tuple $(\bar{a})$ to $R_{chk}$
15.                     else  add tuple $(\bar{a})$ to $R^-$
16.             Test each tuple in $R_{chk}$ for derivability from edbs;
                add it to $R^-$ if it fails
17.     Increment $c$
18. until no changes are made to any $R^-$
19. return the set of all $P^-_{Ri}$ and $R^-$
```

Figure 3: Deletion propagation algorithm.

These two steps may introduce further deletions into the system; hence it is important to continue looping until no more deletions are derived.

EXAMPLE 10. *Revisiting Example 5 and the provenance graph there, suppose that we wish to propagate the deletion of the tuple* $T(3, 2)$. *This leads to the invalidation of mapping node labeled* m_4 *and then the algorithm checks if the tuple* $S(1, 2)$ *is still derivable. The check succeeds because of the inverse path through* (m_1) *to* $U(1, 2, 3)$.

We note that a prior approach to incremental view maintenance, the DRed algorithm [17], has a similar "flavor" but takes a more pessimistic approach. (DRed was formulated for view maintenance without considering provenance, but it can be adapted to our setting.) Upon the deletion of a set of tuples, DRed will pessimistically remove all tuples that can be transitively derived from the initially deleted tuples. Then it will attempt to re-derive the tuples it had deleted. Intuitively, we should be able to be more efficient than DRed on average, because we can exploit the provenance trace to test derivability in a goal-directed way. Moreover, DRed's re-derivation should typically be more expensive than our test for derivability, because insertion is more expensive than querying. In Section 6 we validate this hypothesis.

5. IMPLEMENTATION

The ORCHESTRA system is the first real-world implementation of a CDSS. Our initial prototype version in [32] focused on issues unrelated to update exchange (primarily those related to peer-to-peer communication and persistent storage), and hence for this paper we had to extend the system in many fundamental ways. Atop the existing catalog, communications, and persistence layers, we developed the following components in order to support incremental update exchange:

- *Wrappers* connect to RDBMS data sources, obtain logs of their updates, and apply updates to them.
- *Auxiliary storage* holds and indexes provenance tables for peer instances.
- The *update exchange engine* performs the actual update exchange operation, given schemas, mappings, and trust conditions.

ORCHESTRA performs two different but closely related tasks. When a peer first joins the system, the system *imports* its existing RDBMS instance and logs, and creates the necessary relations and indices to maintain provenance. Later, when the peer actually wishes to share data, we (1) obtain its recent updates via a wrapper and publish these to the CDSS, and (2) perform the remaining steps of update exchange in an incremental way. To review, these are update translation, provenance recomputation, and application of trust conditions. The final resulting instance is written to ORCHESTRA's auxiliary storage, and a derived version of it is recorded in the peer's local RDBMS.

Provenance storage. The majority of our technical contributions were in the update exchange engine, which comprises 34,000 lines of Java code. The engine parses tgds and transforms them into inverse rules, then delta rules, as described in Section 4. These rules are then precompiled into an executable form, resembling datalog which we map down to a query engine through a pluggable back-end. Provenance relations were initially encoded along the lines of 4.1.2, but we found that we needed to find strategies for reducing the number of relations (and thus the number of operations performed by the query engine). After experimenting with the so-called "outer union" approach of [6] — which allows us to union together the output of multiple rules even if they have different arity — we found that in practice, an alternate approach, which we term the *composite mapping* table, performed better. Rather than creating a separate provenance table for each source relation, we instead create a single provenance table per mapping tgd, even if the tgd has multiple atoms on its RHS.

Our initial implementation used a SQL-based back-end (described below); however, the commercial DBMSs we tried required extensive tuning to achieve good query optimizer behavior. Hence, we developed an alternative engine using a customized version of our locally-developed Tukwila query engine, running on top of the Oracle Berkeley DB storage system. In the remainder of this section we discuss and compare these two implementation approaches; in Section 6 we compare them experimentally.

5.1 RDBMS-based Implementation

Using an off-the-shelf RDBMS as a basis for the update exchange component is attractive for several reasons: (1) each peer is already running an RDBMS, and hence there is a resource that might be tapped; (2) much of the data required to perform maintenance is already located at the peer (e.g., its existing instance), and the total number of recent updates is likely to be relatively small; (3) existing relational engines are highly optimized and tuned.

However, there are several ways in which our requirements go beyond the capabilities of a typical RDBMS. The first is that the datalog rules are often mutually recursive, whereas commercial engines such as DB2 and Oracle only support *linearly* recursive queries. The second is that our incremental deletion algorithm involves a series of datalog computations and updates, which must themselves be stratified and repeated in sequence. Finally, RDBMSs do not directly support Skolem functions or labeled nulls.

Hence, we take a datalog program and compile it to a combination of Java objects and SQL code. The control flow and logic for computing fixpoints is in Java,[5] using JDBC to execute SQL queries and updates on the RDBMS (which is typically on the same machine). To achieve good performance, we make heavy use of prepared statements and keep the data entirely in RDBMS tables. Queries return results into temporary tables, and the Java code only receives the number of operations performed, which it uses to detect fixpoint. Support for Skolem functions was implemented by adding user-defined functions and extending the schema with extra columns for each function and parameter.

We found that the RDBMS-based approach was limited in fundamental ways. First, update translation requires many round-trips between the Java and SQL layers. While this might be reduced by using the stored procedure capabilities available in one of the DBMSs, there is still a fundamental impedance mismatch. More challenging was the fact that getting good and consistent performance required extensive tuning, as the query optimizer occasionally chose poor plans in executing the rules.

As part of the process of building the system, we experimented with several different DBMSs; the one with the best combination of performance and consistency was DB2, so we report those numbers in Section 6.

5.2 Tukwila-based Implementation

Motivated by our experiences with the RDBMS-based implementation, we developed an alternative implementation of the system in which we could have total control over the query operations. We extended the Tukwila data integration engine of [21], which has many facilities focused on distributed data. We added operators to support local B-Tree indexing and retrieval capabilities via Oracle Berkeley DB 4.4, and we also added a fixpoint operator to the system. Finally, we developed a translation layer for ORCHESTRA that directly produced physical query plans from datalog rules. We used heuristics to choose join orderings (updates are assumed to be small compared to the size of the database).

This implementation strategy yielded several advantages. The first is that the translation of rules produces a *single* query plan which can be stored as a prepared statement and used to perform update translation with no round-trips. The second advantage is that with control over the physical query plan, we also get reliable and consistent performance (see Section 6).

The approach also has some disadvantages. The query plan we produce works well for small update loads but might be suboptimal in other cases, such as when a peer first joins the system. Tukwila is fast but still not as highly tuned as a commercial DBMS. Our fixpoint operator does not make use of more clever evaluation techniques such as magic sets [29].

Our current Tukwila back-end does not yet have a complete implementation of deletions, so we give experimental results in Section 6 only for insertions. This is sufficient to establish relative performance of the strategies.

[5] We chose to use Java over SQL rather than user-defined functions for portability across different RDBMSs.

6. EXPERIMENTAL EVALUATION

In this section, we investigate the performance of our incremental update exchange strategies, answering several questions. First, we study the impact of incremental update exchange (including our deletion strategy as well as that of DRed [17]) versus full instance recomputation. We then study performance relative to number of peers, base instance and update load size, and we show that our approach scales to realistic bioinformatics data sharing scenarios. We also investigate how the size of the computed instances is affected by the existence of cycles in the "graph" formed by the mappings, and we study the impact of existential variables in the mappings (which result in nulls).

6.1 Experimental CDSS Configurations

To stress test the system at scale, we developed a synthetic workload generator based on bioinformatics data and schemas to evaluate performance, by creating different configurations of peer schemas, mappings, and updates. The workload generator takes as input a single universal relation based on the SWISS-PROT protein database [1], which has 25 attributes. For each peer, it first chooses a random number i of relations to generate at the peer, where i is chosen with Zipfian skew from an input parameter representing the maximum number of schemas. It then similarly chooses j attributes from SWISS-PROT's schema, partitions these attributes across the i relations, and adds a shared key attribute to preserve losslessness. Next, mappings are created among the relations via their shared attributes: a mapping source is the join of all relations at a peer, and the target is the join of all relations with these attributes in the target peer. We emphasize that this was a convenient way to synthesize mappings; no aspect of our architecture or algorithms depends on this structure.

Finally, we generate fresh insertions by sampling from the SWISS-PROT database and generating a new key by which the partitions may be rejoined. We generate deletions similarly by sampling among our insertions. The SWISS-PROT database, like many bioinformatics databases, has many large strings, meaning that each tuple is quite large. We also experimented with the impact of smaller tuples (where we substituted integer hash values for each string). We refer to these in the rest of the discussion as the "string" and "integer" datasets, respectively.

6.2 Methodology

Our SQL engine-based experiments were run using the IBM DB2 UDB 9.1 database engine, running on a dual Xeon 5150 server with 8GB of RAM. We allocated 2GB of RAM to DB2. Our ORCHESTRA translation layer was written in Java 6. Each individual experiment was repeated seven times, with the final number obtained by discarding the best and worst results and computing the average of the remaining five numbers. The Tukwila implementation was running on a different (slower) computer, a dual 3GHz Xeon 5000-series machine — our experiments between the platforms focus more on the "scaling coefficient" of each implementation, rather than their direct performance relative to each other.

Terminology. We refer to the *base size* of a workload to mean the number of SWISS-PROT entries inserted initially into each peer's local tables and propagated to the other peers before the experiment is run. Thus, in a setting of 5 peers, a base size 1000 begins with 5000 SWISS-PROT entries, but as these are normalized into each of the peers' schemas, this results in about 20,000 tuples in the peers' R^ℓ local contributions tables, for a setting with 2 mappings per peer and no cycles. When we discuss *update sizes*, we mean the number of SWISS-PROT entries per peer to be updated (e.g., 200 deletions in the setting above translates to 1000 SWISS-PROT entries, or 4000 tuples total).

6.3 Incremental vs. Complete Recomputation

Our first experiment investigates where our incremental maintenance strategy provides benefits, when compared with simply recomputing all of the peers' instances from the base data. The more interesting case here is deletion (since incremental insertion obviously requires a subset of the work of total recomputation). Moreover, our rationale for developing a new incremental deletion algorithm, as opposed to simply using the DRed algorithm, was that our algorithm should provide superior performance to DRed in these settings.

Figure 4 shows the relative performance of recomputing from the base data ("Non-incremental"), our incremental deletion algorithm, and the DRed algorithm, for a setting of 5 peers, full mappings, and 2000 base tuples in each peer. We note several fairly surprising facts: first, our deletion algorithm is faster than a full recomputation even when deleting up to approximately 80% of the instance. Second, in comparison DRed performs worse in all measured settings — in fact, only outperforming a recomputation for delete settings of under 50%. We attribute most of these results to the fact that our algorithm does the majority of its computation while *only* using the keys of tuples (to trace derivations), whereas DRed (which does reinsertion) needs to use the complete tuples.

6.4 Scale-up

In our second experiment, we look at how the algorithms scale with respect to the number of peers. We begin with the data instances computed from 10,000 original base insertions, with $n-1$ mappings among n peers. For reference, the resulting instance sizes are shown in Figure 6. Figures 7 and 8 show the scaling performance for the DB2 and Tukwila-based engines for insertions. Since we do not have a Tukwila implementation of deletions, we show only the DB2 case of deletions in Figure 9.

For the original SWISS-PROT data, insertions are significantly more expensive than deletions (and grow at a higher rate), due to the overhead of carrying all tuple state. In contrast, the situations reverse when the tuples are small, because both operations carry approximately the same amount of data, but the number of queries executed in deletion is greater. In terms of scalability, we were able to perform insertions up to 10 peers with strings (after which our DBMS exceeded the 20GB storage reserved for the data instances), and with integers the approach scaled to upwards of 20 peers (already larger than most real bioinformatics confederations). We note that the DB2 version is faster when a peer joins the system (Figure 5) or when the number of updates is relatively large (e.g., for 10% updates and SWISS-PROT data, for which Tukwila's running times were beyond the scale of the graph), but the Tukwila implementation is better optimized for the common case, where the volume of updates is significantly smaller than the base size, as well as when the tuple size is small.

6.5 Impact of Cycles and Mappings

Our final experiment focuses on the impact of mappings' topology on scalability. In general, cyclic mappings increase the complexity of computation significantly, as the incremental update exchange process takes more recursive steps to reach fixpoint. We measured insertions in both the DB2 and Tukwila engines with 5 peers, averaging 2 neighbors each, and manually added cycles. Figure 10 shows the readings for both engines, compared versus the total number of tuples computed. We see that overall running times increase at a somewhat higher rate than the data instance does: not only are the instance sizes growing, but the actual number of itera-

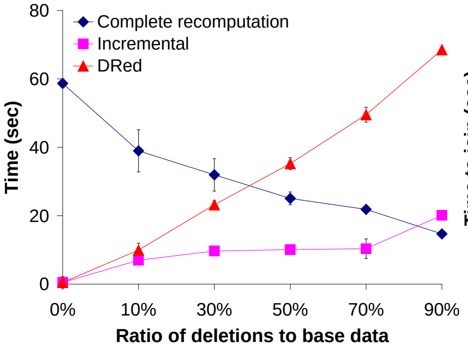


Figure 4: Deletion alternatives

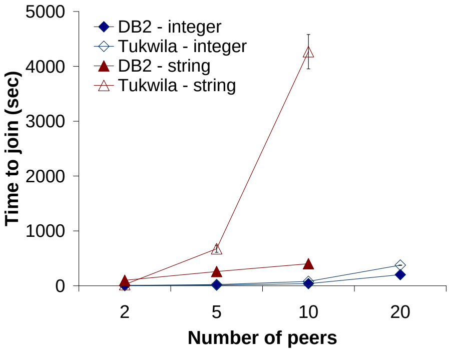


Figure 5: Time to join system

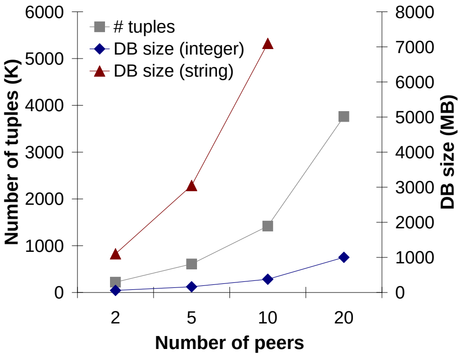


Figure 6: Initial instance size

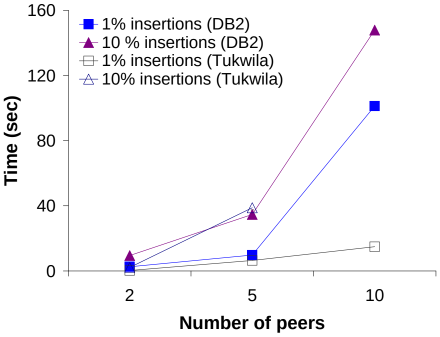


Figure 7: Scalability of incremental insertions (string dataset)

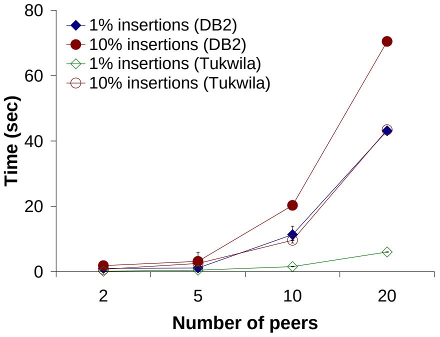


Figure 8: Scalability of incremental insertions (integer dataset)

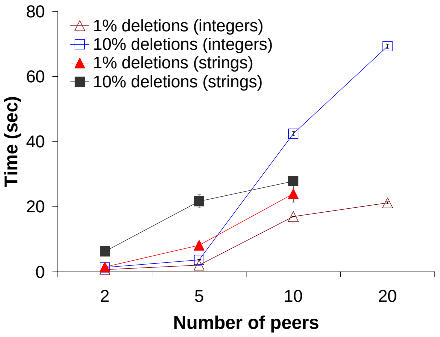


Figure 9: Scalability of incremental deletions

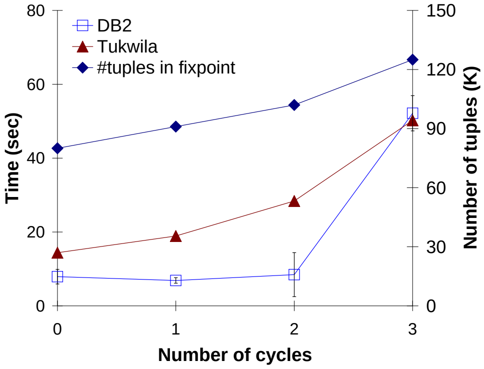


Figure 10: Effect of cycles in instance size

tions required through the cycle also increases.

Our final conclusion from these experiments is that the CDSS approach is amenable to incremental maintenance of both data and provenance. Our algorithms scale linearly to increased workloads and data sets, and performance is indeed acceptable for small data sharing confederations, as we are targeting for bioinformatics domains.

7. RELATED WORK

This paper takes advantage of previous work on PDMS (e.g., [18]) and on data exchange [12, 25] [31]. With our encoding in datalog, we reduce the problem of incremental updates in CDSS to that of recursive view maintenance where we contribute an improvement on the classic algorithm of [17]. Incremental maintenance of recursive views is also considered in [26] in the context of databases with constraints using the Gabrielle-Levi fixpoint operator; we plan to investigate the use of this technique for CDSS. The AutoMed system [27] implements data transformations between pairs of peers (or via a *public schema*) using a language called BAV, which is bidirectional but less expressive than tgds; the authors consider incremental maintenance and lineage [13] under this model. In [15], the authors use target-to-source tgds to express trust. Our approach to trust conditions has several benefits: (1) trust conditions can be specified between target peers or on mappings themselves; (2) each peer may express different levels of trust for other peers, i.e., trust conditions are not always "global"; (3) our trust conditions compose along paths of mappings. Finally, our approach does not increase the complexity of computing a solution.

We rely on a novel provenance model that is useful both for trust policies and for incremental deletion. Two recent papers develop among other things provenance models that bear a relationship to our approach. Like us, [7] identifies the limitations of why-provenance and proposes *route-provenance*, which is also related to derivation trees, but uses it for a different purpose—debugging schema mappings. Our model maintains a graph from which provenance can be incrementally recomputed or explored, whereas their model is based on recomputing the shortest routes on demand. [2] proposes a notion of lineage of tuples which is a combination of sets of relevant tuple ids and bag semantics. As best as we can tell, this is more detailed than why-provenance, but we can also describe it by means of a special commutative semiring, so our approach is more general. The paper also does not mention recursive queries, which are critical for our work. Moreover, [2] does not support any notion of incremental translation of updates over mappings or incompleteness in the form of tuples with labeled nulls. Our provenance model also generalizes the duplicate (bag) semantics for datalog [29] and supports generalizations of the results in [30].

8. FUTURE WORK

We believe that we have demonstrated the feasibility of the CDSS concept. Nevertheless much work remains to be done. We plan to develop a truly distributed implementation of CDSS, which will entail work on caching, replication, multi-query optimization, indexing of provenance structures, etc. The provenance model seems to allow a quite general approach to trust conditions that remains to be fully exploited (e.g., ranked trust models). Finally, we plan to consider complex transaction models that give rise to data dependencies and conflicts between the update policies of different peers. Solving such problems will likely require a *transaction provenance model* in which we also would annotate updates with transaction identifiers.

Acknowledgments

This work has been funded by NSF grants IIS-0477972, 0513778, and 0415810, and DARPA grant HR0011-06-1-0016. The authors would like to thank Sarah Cohen-Boulakia and the members of the SHARQ project for assistance with the biological datasets, Olivier Biton for assistance in design and coding of project modules, and the members of the Penn Database Group, Renée Miller and the anonymous reviewers for their feedback and suggestions.

9. REFERENCES

[1] A. Bairoch and R. Apweiler. The SWISS-PROT protein sequence database and its supplement TrEMBL. *Nucleic Acids Research*, 28, 2000.

[2] O. Benjelloun, A. D. Sarma, A. Y. Halevy, and J. Widom. ULDBs: Databases with uncertainty and lineage. In *VLDB*, 2006.

[3] P. A. Bernstein, F. Giunchiglia, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu. Data management for peer-to-peer computing: A vision. In *WebDB '02*, June 2002.

[4] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT*, 2001.

[5] D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Logical foundations of peer-to-peer data integration. In *PODS*, 2004.

[6] M. J. Carey, D. Florescu, Z. G. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, and S. Subramanian. XPERANTO: Publishing object-relational data as XML. In *WebDB '00*, 2000.

[7] L. Chiticariu and W.-C. Tan. Debugging schema mappings with routes. In *VLDB*, 2006.

[8] Y. Cui. *Lineage Tracing in Data Warehouses*. PhD thesis, Stanford University, 2001.

[9] A. Deutsch, L.Popa, and V. Tannen. Query reformulation with constraints. *SIGMOD Record*, 35(1), 2006.

[10] A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, 2003.

[11] O. M. Duschka and M. R. Genesereth. Answering recursive queries using views. In *PODS*, 1997.

[12] R. Fagin, P. Kolaitis, R. J. Miller, , and L. Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336, 2005.

[13] H. Fan and A. Poulovassilis. Using schema transformation pathways for data lineage tracing. In *BNCOD*, volume 1, 2005.

[14] M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *AAAI '99*, 1999.

[15] A. Fuxman, P. G. Kolaitis, R. J. Miller, and W.-C. Tan. Peer data exchange. In *PODS*, 2005.

[16] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.

[17] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, 1993.

[18] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *ICDE*, March 2003.

[19] A. Hernich and N. Schweikardt. CWA-solutions for data exchange settings with target dependencies. In *PODS*, 2007.

[20] Z. Ives, N. Khandelwal, A. Kapur, and M. Cakir. ORCHESTRA: Rapid, collaborative sharing of dynamic data. In *CIDR*, January 2005.

[21] Z. G. Ives, D. Florescu, M. T. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data integration. In *SIGMOD*, 1999.

[22] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *SIGMOD*, June 2003.

[23] M. Lenzerini. Tutorial - data integration: A theoretical perspective. In *PODS*, 2002.

[24] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB*, 1996.

[25] L. Libkin. Data exchange and incomplete information. In *PODS*, 2006.

[26] J. J. Lu, G. Moerkotte, J. Schue, and V. Subrahmanian. Efficient maintenance of materialized mediated views. In *SIGMOD*, 1995.

[27] P. J. McBrien and A. Poulovassilis. P2P query reformualtion over both-as-view data transformation rules. In *DBISP2P*, 2006.

[28] P. Mork, R. Shaker, A. Halevy, and P. Tarczy-Hornoch. PQL: A declarative query language over dynamic biological schemata. In *AMIA Symposium*, November 2002.

[29] I. S. Mumick, H. Pirahesh, and R. Ramakrishnan. The magic of duplicates and aggregates. In *VLDB J.*, 1990.

[30] I. S. Mumick and O. Shmueli. Finiteness properties of database queries. In *Fourth Australian Database Conference*, February 1993.

[31] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating web data. In *VLDB*, 2002.

[32] N. E. Taylor and Z. G. Ives. Reconciling while tolerating disagreement in collaborative data sharing. In *SIGMOD*, 2006.