ORCHESTRA: Rapid, Collaborative Sharing of Dynamic Data

Zachary Ives Nitin Khandelwal Aneesh Kapur
University of Pennsylvania
zives@cis.upenn.edu, {khandeln,aneeshk}@seas.upenn.edu

Murat Cakir*
Drexel University
mpc48@drexel.edu

Abstract

Conventional data integration techniques employ a *"top-down"* design philosophy, starting by assessing requirements and defining a global schema, and then mapping data sources to that schema. This works well if the problem domain is well-understood and relatively static, as with enterprise data. However, it is **fundamentally mismatched** with the *"bottom-up"* model of scientific data sharing, in which new data needs to be rapidly developed, published, and then assessed, filtered, and revised by others.

We address the need for bottom-up *collaborative data sharing*, in which independent researchers or groups with different goals, schemas, and data can share information in the absence of global agreement. Each group independently curates, revises, and extends its data; eventually the groups compare and *reconcile* their changes, but they are not required to agree. This paper describes our initial design and prototype of the ORCHESTRA system, which focuses on *managing disagreement* among multiple data representations and instances. Our work represents an important evolution of the concepts of peer-to-peer data sharing [23], which considers *revision*, *disagreement*, *authority*, and *intermittent participation*.

1 Introduction

Established techniques for data integration follow a *"top-down,"* global-schema-first design philosophy, starting by assessing requirements and defining a global schema, and then mapping data sources to that schema. This works well if the problem domain is well-understood and relatively static, as with business data. However, there is an urgent need to address data sharing problems beyond this context — especially in supporting large-scale scientific data sharing or managing community-wide information. Prominent examples of such data sharing occur in the emerging fields of comparative genomics and systems biology, whose focus is the integrated analysis of the large, dynamic, and growing body of biological data. The success of these fields will be largely determined by the quantity and quality of the data made available to scientists.

Unfortunately, attempts to apply data integration techniques — standardizing data formats and schemas, and then creating translators, warehouses, or virtual mediator systems to combine the data — have had limited success. In bioinformatics today there are many "standard" schemas rather than a single one, due to different research needs, competing groups, and the continued emergence of new kinds of data. These efforts not only fail to satisfy the goal of integrating **all** of the data sources needed by biologists, but they result in standards that must repeatedly be revised, inconsistencies between different repositories, and an environment that restricts an independent laboratory or scientist from easily contributing "nonstandard" data.

The central problem is that science evolves in a *"bottom-up"* fashion, resulting in a fundamental mismatch with top-down data integration methods. Scientists make and publish new discoveries, and others accept, build upon, and refine the most convincing work. Science does not revisit its global models after every discovery: this is time-consuming, requires consensus, and may not be necessary depending on the long-term significance of the discovery. Data sharing approaches reliant on globally designing/redesigning integrated schemas are inappropriate for the same rea-

* Work done while an M.S. student at the Univ. of Pennsylvania.

Proceedings of the 2005 CIDR Conference

sons. Moreover, today's integration techniques fail to recognize that scientific discovery is inherently a process of *revision* and *disagreement*. A common practice is *collaborative data sharing*: independent researchers or groups with different goals, schemas, and data agree to share data with one another; each group independently curates, revises, and extends this shared data; at some point the groups compare and *reconcile* their changes. Even after reconciliation, some groups may disagree about certain aspects of the data, and some may yield to others with greater *authority*.

The Web, while not sufficient to address collaborative data sharing, provides a model of an existing and incredibly successful bottom-up data sharing infrastructure. Anyone can contribute new content as desired or link to it; useful data is filtered by trusted, value-added "portal" sites that link to content consistent with some theme. The Web evolves rapidly, and it is self-organizing and self-maintaining. Peer-to-peer approaches [37, 41, 39] also adopt a bottom-up, self-organizing approach, but to this point peer data management only considers query answering over multiple schemas [23, 29].

Science and academia are in need of a Web-like, bottom-up infrastructure to address their collaborative data sharing needs. With such infrastructure, one could rapidly contribute new schemas, data, and *revisions*; specify schema mappings to others' data; control and *filter* what data is exchanged, based on its source; and publish and *reconcile* changes to shared data. Shared data must be *always available*, even if a given contributor is not. This infrastructure could support a number of important classes of applications.

Bioinformatics and scientific data exchange. Bioinformatics is an example of a science that needs to support data exchange among multiple schemas: numerous standards exist for each of a variety of data types, including microarray, proteomics, and neuroinformatics data. Unfortunately, individual biologists may not have the resources to influence the standard schemas or easily share their proprietary data (which may include annotations and analysis results valuable to the community as a whole). Rapid advances in biology require data sharing techniques that are more responsive to new developments, and that promote effective data exchange among collaborators and groups with related interests.

Academic community resources. A common problem in academia is sharing and maintaining community resources such as citation entries, software listings, or research group information. Typically this is accomplished via repositories maintained by individual groups, e.g., DBLP (`dblp.uni-trier.de`), ACM's Digital Library, SIGMOD's list of database systems (`www.acm.org/sigmod/databaseSoftware/`), or by systems that allow distributed submission of entries. Keeping such databases current, duplicate-free, and "clean" is a major challenge, requiring either significant human input or automated and not-always-reliable techniques for finding and matching entries. Further exacerbating the problem, there may also be a desire to represent the data in multiple forms.

Other group-oriented applications. Discussion groups and blogs, group calendars, and version control systems are typically built using a centralized client-server architecture and a single schema or representation. As such tools are deployed across multiple departments, organizations, and open-source efforts, there is increasingly a need to customize them for each organization, and perhaps even to separately administer each location. Even within a single organization, there may be multiple formats to be managed, e.g., when a PDA adds special fields to groupware data. Current tools rely on opaque "custom fields" that cannot be mapped, e.g., between a PDA and a cell phone. Data would be more effectively managed if each organization could separately extend and manage its data and schema, while mapping information to other groups.

Contributions and Road Map

In the ORCHESTRA project, we are developing a collaborative data sharing infrastructure to facilitiate scientific and academic data sharing in a bottom-up, rapid-to-change fashion: ORCHESTRA emphasizes *managing disagreement* at the schema and instance levels, and it supports rapidly changing membership. This paper makes the following contributions:

- To address disagreement between *different data instances*, we propose a novel data model that focuses on accommodating data from many disagreeing viewpoints.
- To address the problem of propagating updates despite *rapidly changing participation*, we implement ORCHESTRA over a peer-to-peer, distributed hash table substrate [39] and replicate data among active members.
- To address the fact that different members may prefer *different schemas*, we extend the techniques of peer data management systems [23, 20] to translate *updates* from one schema to another.

This paper is organized as follows. Section 2 discusses how the collaborative data sharing problem goes beyond the domains of past work on data integration and distributed databases. Section 3 presents an overview of the ORCHESTRA collaborative data sharing system we are building. Section 4 describes our approach to representing and reconciling conflicting updates. Section 6 describes how we recast the cross-schema update

translation problem to leverage standard query reformulation techniques. We conclude and discuss future work in Section 7.

2 Rapid, Collaborative Data Sharing

Approaches to distributed data sharing today can be divided into two categories. *Distributed databases*, exemplified by Mariposa [42] and recent peer-to-peer data processing engines and storage systems [24, 1], generally assume one schema with relations (or fragments of relations) that are distributed and replicated. They provide high-performance queries and updates over distributed and replicated data under a single administrative domain.

Data integration ties together pre-existing, heterogeneous data sources in a distributed context. This typically involves defining a global virtual schema or materialized data warehouse as a uniform query-only interface over the data sources, then creating *schema mappings* to relate the sources to this schema. Such mappings are usually specified as conjunctive queries, and *query reformulation* techniques [32] are used to compose user queries with mappings to return results. Today, the focus in data integration has been on designing tools to assist in defining mappings [36] and on more flexible, peer-to-peer models of integration [23, 22, 29].

2.1 Scientific Data Exchange

Such problems are of great importance in many situations, but we argue that there is a class of important data sharing problems not addressed by either the distributed database or data integration models. This class of applications revolves around *exchanging* data among different databases or warehouses, each with its own schema, and each of which is independently evolving. Data sharing for scientific collaboration is an exemplar of this class, as it must facilitate the exchange of rapidly evolving, sometimes-disputed data across different schemas and viewpoints.

Example 2.1 *The Penn Center for Bioinformatics (PCBI) administers a local genomics microarray data repository, RAD [40], a subset of a larger schema called GUS. PCBI needs to share RAD data with other standard microarray repositories, including ArrayExpress*[1]*, based on the MAGE-ML schema*[2]*, and, ultimately, higher-level data sources such as the Gene Ontology database*[3]*. All of these sources have overlapping data, and they all want access to a complete set of the data; yet none wants to adopt a new schema (as required by the top-down integration approach).*

Figure 1 shows how the three schemas (sets of "shared relations") can be mapped together in an envisioned collaborative data sharing environment. Each schema is shared by some number of participants*: in this system there are four, including two sites, RAD1 and RAD2, which replicate the same RAD schema. Each participant has a local data instance matching its schema, which it independently modifies. We indicate potential dataflows in this figure with directed edges; note that actual mappings may not be provided in both directions by the user, nor are there mappings specified between every pair of schemas. As in [23, 22, 10], it may be necessary to* invert *or* compose *mappings to convey information from one schema to another.*

At some point, a participant such as RAD1 will reconcile *the changes it made, i.e., it will publish all new updates it has made to the system, and it will receive any non-conflicting updates published recently by participants it trusts (perhaps ArrayExpress and Gene Ontology, but not RAD2). It is also possible for an entity such as Gene Ontology to only be a* one-way *participant, not accepting data from other sources but posting any modifications made to its internal database on a periodic basis. Gene Ontology's updates will be applied to any participant with a RAD or MAGE-ML schema the next time it reconciles. Those updates will be propagated to ArrayExpress even though there is no direct mapping between Gene Ontology and MAGE-ML: a transitive mapping will be derived by composing the Gene Ontology/RAD and RAD/MAGE-ML mappings.*

The data sharing environment must be rapidly extensible in a bottom-up way. If a new participant with a different schema and data wishes to join, it need only define a mapping to an existing schema. Alternatively, if PCBI updates the RAD schema, it will leave the old version in place and add a mapping from the new schema version to the old one; participants may use either schema version.

In scientific data sharing scenarios, like the one above, the predominant paradigm is to provide a *copy* of an experimental or analytical data instance to others. That data is mapped into another database with a different schema and then curated, refined, and modified independently. At some stable point, the parties will want to share, compare, and reconcile their changes. They may elect to do so at different points of time; they may reserve the right to override, or even remove, tuples or data values provided by the other party. This mode of *collaborative data sharing*, due to its dynamic nature and lack of central authority, exacerbates a number of issues that have not been systematically addressed in past data sharing research, and these problems are the focus of what we term a *collaborative data sharing system* (CDS).

[1] www.ebi.ac.uk/arrayexpress
[2] www.mged.org/
[3] www.geneontology.org

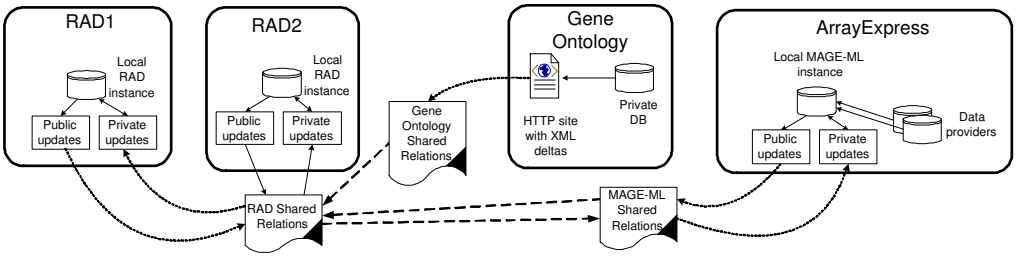

Figure 1: Collaborative data sharing among four bioinformatics participants, with three schemas (sets of virtual "shared relations"). Dotted lines represent mappings from shared data to local instances, and dashed lines represent update translation mappings between different schemas.

2.2 Collaborative Data Sharing

At the heart of the collaborative data sharing problem is the need to accommodate both *disagreement* and *dynamic participation of members*: schemas and data instances will constantly be revised and extended; conflicting assertions about facts will abound; participants will join and leave at will; different participants will use different schemas. We briefly outline the problem and our basic approach in this section; the remainder of the paper provides further details.

2.2.1 Conflicting Data and Updates

The traditional emphasis in distributed data sharing has been on providing (at least eventual) consistency. Examples of previous work include data warehouse maintenance [14]; distributed concurrency control and replication, such as synchronization of disconnected replicas [11] and weak, eventual-consistency models [16]; and file sharing [35, 18, 30, 17]. The goal of these approaches is to merge update sequences in an ordered and consistent way, yielding a globally consistent data instance. When a consistent state cannot be determined, a human must resolve the conflict.

With scientific data, a given data item (e.g., a tuple, a tree, a series of data items related by foreign keys) may be the subject of disagreement among different sources. For instance, one bioinformatics data source may postulate a relationship between a gene sequence and a disease, whereas another may feel there is insufficient data to support this hypothesis. Variations on the relational data model have been proposed in order to support uncertainty or probability [33, 3] and inconsistency [4, 31], and these initially seem promising as solutions. Probabilistic databases allow one to attach probabilistic information to data items (usually tuples); the study of inconsistency in databases generally revolves around trying to "repair" a data instance where constraints are not satisfied. However, it is difficult to assign a probability to each representation, or to determine how to repair the data.

Our approach. We propose a model that intuitively resembles that of incomplete information [2], which represents multiple database instances within the same table by annotating each tuple with a condition specifying when it is part of an instance. Our case is a slightly simpler variation: tuples are "tagged" with information about which participants *accept* them. Each participant has an internally consistent database instance, formed by the set of tuples that it accepts. Importantly, no participant is required to modify its data instance to reach agreement with the others, although it has the option if it so chooses.

In large-scale data sharing scenarios, especially ones in which data may be frequently revised or conflicting, it is essential that the data versions can be "filtered" by data consumers. Scientists are willing to trust data from others whom they view as authorities; they may even delegate decisions of trust to others. In effect, they might decide whether to accept a data entry based on its *provenance* [9]. We allow each participant to specify precisely whom to trust data from, and under which conditions.

We are concerned not merely with representing different data instances, but with propagating and applying updates to those data instances. We tag and "filter" updates in a similar manner to the way we tag data items: an update is propagated only to those participants who trust its source and/or value. Our model of supporting multiple concurrent versions leads to an important difference from distributed concurrency control algorithms: we do not need to *minimize* conflict sets; rather, each participant receives only those transactions that it trusts and that do not conflict with its existing instance.

2.2.2 Dynamic Membership

In the data integration world, if a data source is unavailable for any reason, its data typically becomes unavailable to others. Scientists are very concerned about losing access to crucial experimental data, simply because its originator is unavailable or experiences a system crash. Any shared data must be permanently available, although individual machines may join and leave at will.

Our approach. We *replicate* all information (i.e., the

original source data and the update sequences applied to it) across the machines currently in the collaborative data sharing environment. We also *partition* the data before replicating it, allowing us to "stripe" the data and computation across multiple nodes. Our implementation is based on a peer-to-peer distributed hash table substrate [37, 41, 39], which provides effective replication and partitioning capabilities and does not rely on a central server.

2.2.3 Multiple Schemas

Conventional data integration techniques work well in enterprise information integration because the setting is well understood, and user needs do not change rapidly. In the sciences, there are diverse groups and needs, and these change frequently, so it is difficult to create and evolve a single global schema. Moreover, users frequently prefer their familiar local schema to a global, centralized mediated schema [38].

This has led to interest in more flexible, peer-to-peer methods of data sharing [23, 34, 29]. In past work, we and others proposed *peer data management*, which generalizes and decentralizes data integration: participants autonomously choose their own schemas and define mappings to other schemas. No single entity controls the schema for a domain, and cooperating entities each have the ability to adopt a schema customized to their individual needs. This provides a voluntary-participation, "best effort" semantics, similar to peer-to-peer file sharing systems [37, 41, 39]. A potential pitfall of this approach is that it must be possible to map between *any pair* of data sources; since that number of mappings may be unavailable, peer data management provides algorithms to *compose and combine* mapping paths to return the maximum set of inferrable answers [23, 22, 43, 10, 7, 29]. Peer data management provides a very effective model for sharing data among heterogeneous collaborators with different needs, and it meshes with the bottom-up data sharing model of the sciences. However, it focuses purely on query answering; like the broader data integration field, it does not consider the need for update propagation.

Our approach. Based on the ideas described above, we develop techniques for propagating *updates* across schemas. Intuitively, we *query for updates* to apply to the target schema, based on any updates made to the source schemas. New challenges arise from the fact that updates are sequential rather than independent.

3 The ORCHESTRA System

We are addressing the needs of collaborative data sharing in a system called ORCHESTRA. ORCHESTRA coordinates a set of autonomous *participants* who make updates to local relation instances (the rounded rectangles of Figure 1) and later *publish* them for others to access. More than one participant may have local relation

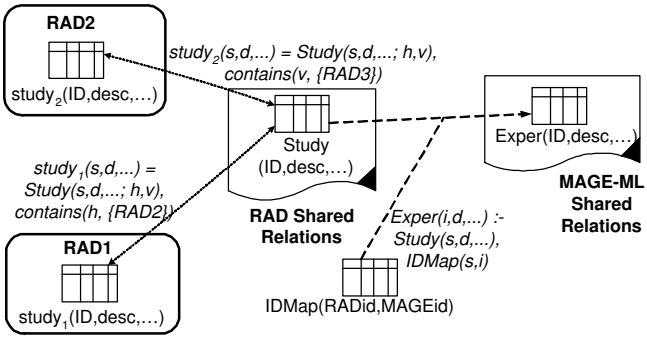

Figure 2: More detailed subset of the bioinformatics environment of Figure 1. Shared relations are capitalized; local relations are subscripted with the ID of the participant upon which they appear. Mappings are italicized, and in the form of conjunctive queries. The IDMap table is used only as part of a mapping, and it can be considered the sole member of its own schema.

instances that share the same schema; intuitively, these instances capture the same information about the data, and we define *shared relations* to encode the "union" of the published (possibly conflicting) data, as well as information about where that data originated.

Between a shared relation and each participant's *local relation* that "replicates" it, there is an *instance mapping* that specifies precisely what data from the shared relation is trusted by the participant, and should be replicated by it. In turn, data between shared relations in different schemas is interrelated through schema mappings. "Chains" of mappings can be followed between connected participants in the system.

Example 3.1 *Refer to Figure 2 for a more detailed example of our bioinformatics collaborative data sharing environment. Suppose* `RAD1` *wishes to reconcile its changes with any other recent updates. It first determines what updates have been published to the system by others — in this case, participants who use either the* `RAD` *schema or the* `MAGE-ML` *one.* `RAD1` *first reformulates all updates in terms of its shared relations in* `Study`*, where necessary using the mapping between* `Exper` *(in MAGE-ML) and* `Study` *to translate the updates. Then the updates applied to* `Study` *are propagated to* `RAD`*, so long as they meet the conditions of its instance mapping and they do not violate its consistency. Finally,* `RAD1` *publishes its own updates to the* `RAD` *shared relations; these are automatically tagged with information specifying that they originate from* `RAD1`*.*

The general mode of operation in ORCHESTRA is to operate in disconnected fashion, then to reconcile. A participant p reconciles its updates with those made by others through the following steps:

1. Determine all updates *published* by participants to

ORCHESTRA since the last reconciliation.

2. Compute the effects of these updates on all shared relations, according to the possibly-disagreeing *viewpoints* of the different participants.

3. Use the set of instance mappings to determine which updates would be *accepted* by p and remove those that *conflict*.

4. Propagate to p's relation those updates that are accepted and non-conflicting.

5. Record the updates originating from p, or accepted by it, in ORCHESTRA for future reconciliation operations.

In subsequent sections, we describe in detail how ORCHESTRA accommodates multiple viewpoints in a consistent way, how it performs update propagation and reconciliation, and how it translates updates from one schema to another.

4 Consistency with Conflicting Data

We begin by defining the specifics of the data model what it means for an instance to be consistent with a set of schema mappings. ORCHESTRA merges the results of multiple, conflicting database instances into *shared relations*. In our bioinformatics example, these are the RAD, MAGE-ML, and Gene Ontology shared relations. Shared relations accommodate conflicting values by "tagging" each tuple with information specifying where it originated and who "believes" it; each participant filters the data it accepts based on value, origin, or who else accepts it. (This is a specialized and limited form of data provenance [9].)

4.1 Multi-Viewpoint Tables

Each participant p is said to have a *viewpoint*, V_p, which associates with each shared relation S an instance consistent with p's published updates. A viewpoint represents *hypotheses* held by p rather than *facts*, i.e., there can be conflicting information or information that is later determined to be incorrect[4]. A shared relation S, then, contains many overlapping data instances describing the hypotheses of each participant. Such a set of possible instances matches closely with models of *incomplete information* [25, 2], for which a number of formalisms, most notably *conditional tables*, have been proposed. Conditional tables represent a set of possible instances, and they attach to each tuple a condition that must be satisfied for the tuple to exist in a particular instance.

[4]Our model generalizes the standard *open-world assumption* made in data integration: in the open-world model, every instance of a relation (collection) is assumed to be a potentially incomplete subset of the overall set of tuples (objects, etc.) that may exist.

To encode multiple viewpoints' instances in conditional tables, we could assign each participant an identifier i, define a variable v representing the viewpoint, and attach the condition $(v = i)$ to each tuple that exists according to P_i's perspective. Unfortunately, conditional tables are seldom used in practice because in the worst case the cost of querying for certain answers (or of performing updates) is exponential in the size of the data [2].

Since we only need a restricted subset of conditional tables' capabilities, we propose a novel subset we term a *multi-viewpoint table* or MVT, which ensures polynomial data complexity in query answering and updates. Similar models have been proposed previously [33, 3], but the MVT is novel in that it encodes not only the data's original viewpoint, but also what other participants trust the data. That property allows us to delegate trust: e.g., participant P_i trusts anything that P_j believes.

Definition 1 *Given a relation $R(\bar{X})$, $\bar{X} = (x_1, \ldots, x_m)$, assume that there exist n possible instances of R, $I_1(R), \ldots, I_n(R)$, according to n different viewpoints. We will distinguish between tuples $t \in I_i(R(\bar{X}))$ that* originated *(or partly originated) in I_i, versus those that have been* accepted *into I_i.*

A multi-viewpoint table *that accommodates n viewpoints of R, $M(\bar{X}, \hbar, \bar{v})$, where s is an integer and $\hbar$ and $\bar{v}$ are n-bit boolean vectors, is defined as follows. For each tuple t in instance $I_i(R(\bar{X}))$, there exists a tuple t' in M with the following properties:*

- *For each attribute $x_a \in \bar{X}$, $M(x_a) = R(x_a)$.*
- *For every $1 \leq j \leq n$, if t is derived from a tuple originating in instance I_j, then the jth bit in the $\bar{h}$* origin vector *is set to true, otherwise it is false. This encodes a limited form of* data provenance *[9, 21]: the viewpoints from which the tuple was derived. (A tuple that is the result of a join may have more than one origin.)*
- *For every instance $I_j(R)$ that also contains t, the jth bit of $\bar{v}$ in M is set to true, otherwise the jth bit is false. This forms the* viewpoint vector *for each tuple; it is a more compact representation of the conditional table expression $(v = j_1) \vee \ldots \vee (v = j_m)$, where $j_1, \ldots, j_m$ are the bits set in v. This indicates what viewpoints include the tuple.*

It is straightforward to extend the standard relational algebra in terms of these operators[5]. Selection and join predicates may test not only the standard relational attributes, but two additional attributes, the viewpoint and origin vectors. The selection operator "passes through"

[5]Such extensions have been proposed for the more general conditional table in [25].

any tuple t satisfying its predicates. Projection over an MVT tuple t returns a tuple t' with t's viewpoint and origin vectors; if two returned tuples t', t'' match in attributes and origin, then one will be removed and the other's viewpoint vector will become the union of the two tuples' viewpoint vectors. A join of tuples t_1, t_2 returns a tuple t' only if there is at least one overlapping bit in the viewpoint vectors of t_1, t_2; t' will have bits set in its origin vector for each origin of t_1 or t_2, and bits set in its viewpoint vector for each viewpoint in common between the source tuples.

Under this semantics, multi-viewpoint tables are a practical way of representing instances for multiple viewpoints, while maintaining query answering tractability:

Theorem 1 *Answering datalog queries with multi-viewpoint tables is polynomial in the size of the data and the number of participants' viewpoints.*

4.2 Mappings from Shared to Local Relations

Key to ORCHESTRA's data sharing model is the fact that multiple local relation instances are mapped into a single shared MVT relation. Participant p is willing to trust and accept certain data from its associated shared MVT relations; *instance mappings* define the constraints under which local (non-MVT) tuples match those of shared relations. Specifically, they provide a mapping between the *set of instances* of a shared MVT relation and a *single local instance* at p, based on tuples' values, origins, and realm of trust. An instance mapping contains selection conditions over the origin or viewpoint vectors (as well as potentially the data), defining a correspondence between a subset of the MVT tuples and those of the local instance. Figure 2 includes instance mappings for `RAD1` and `RAD2`, where the predicates over the origin and viewpoint vectors are specified as set-containment constraints. In this case, `RAD1` trusts data originating from `RAD2`, and `RAD2` *delegates its trust* by accepting anything in the viewpoint of node $RAD3$ (not shown).

When a node p reconciles, its published updates are incorporated into the shared MVT relations with their origin and viewpoint vectors set to include p. Likewise, when p accepts a value into its instance, p's bit is set in the MVT's viewpoint vector.

Example 4.1 *Suppose* `RAD2` *publishes the tuple* $\texttt{study}_2(12, \text{“}test''\text{”}, ...)$. *This will show up in the shared relation* $\texttt{Study}(12, \text{“}test''\text{”}, ...; \{RAD2\}, \{RAD2\})$, *where the attributes after the semicolon represent (in set form) the contents of the origin vector and viewpoint vector, respectively. After* `RAD1` *reconciles, it will accept this new tuple, based on the constraints of its instance mapping. Then the MVT will be updated to include* `RAD1` *in its viewpoint:* $\texttt{Study}(12, \text{“}test''\text{”}, ...; \{RAD2\}, \{RAD1, RAD2\})$.

4.3 Mappings between Shared Relations

As in a peer data management system [23, 10], it is possible to express *schema mappings* between shared MVT relations, and to formulate a query answering problem in this model. We defer a discussion of such details to Section 6.

5 Publishing Updates

Thus far, we have described the semantics of mappings between conflicting *data instances*. In reality, ORCHESTRA must manipulate, compare, and apply logs of *update sequences*, because each participant p reconciles according to its own schedule, and reconciliation must compare modifications done locally at p against *all updates published since p last reconciled*.

The fundamental unit of storage and propagation in ORCHESTRA is an atomic *delta* over a single relation, representing a minimal encoding for the insertion, deletion, or replacement of a single tuple. Our approach bears great similarity to that of incremental view maintenance [8] and hypothetical updates [19], except for our inclusion of an atomic *replace* operation that changes the values of a tuple. We need an atomic replacement operation in order to propagate a change "through" a mapping with a projection: this allows us to maintain any projected-out values.

Example 5.1 *Insertion, replacement, and deletion of tuples from MAGE-ML relation* $\texttt{Experiment}(id, desc, date)$ *can be specified with deltas such as the following.*

$\texttt{+Exper}(12, \text{“}initial\text{”}, 1/04)$
$\rightarrow\texttt{Exper}(12, \text{“}initial\text{”}, 1/04 : 45, \text{“}new\text{”}, 2/04)$
$\texttt{-Exper}(45, \text{“}new\text{”}, 2/04)$ *(The "replace" operation, $\rightarrow$, takes both old and new values, separated by a ":" in our notation.)*

5.1 Reconciling Published Updates

A fundamental question about reconciliation is *how deltas from different participants should interact* — which updates we consider to conflict. We adopt the *principle of least surprise* here: "intermediate" updates, i.e., those whose value gets changed by subsequent updates from the same source, should not interfere with updates from other sources. Two updates only conflict if their effects would both become visible in contradictory ways at the end of a reconciliation operation. This semantics implies that ORCHESTRA can *minimize* or "flatten" update sequences, eliminating any intermediate steps, before it checks for conflicts. For instance, the update sequence in the above example should "cancel out" since the same tuple is inserted, modified, and finally deleted.

Our definition of conflicts also affects what it means for transactions to conflict: two transactions conflict during

a reconciliation stage if any of their individual operations conflict, or if either transaction uses results from previous transactions that conflict.

5.2 Flattening Dependency Sequences

Based on the observations made above, we can define a *flattening* operation that reduces an update sequence to the minimal number of steps. We assume that all operations in an update sequence have been validated by the underlying DBMS, that there are no implicit dependencies between transactions. For each operation, let the *target* of a deletion or replacement be the key of the value to be removed or replaced; let the target of an insertion be the key of the new tuple. Let the *new key* of a replacement be the key of the new tuple, and the new key of an insertion be the key of the inserted tuple. For each transaction, create an *operation set* and insert into it every operation in that transaction.

Each dependency sequence forms the start of a chain in which the new key of one update potentially becomes the target of some subsequent update. We repeatedly apply the transformation rules of Figure 3, until each update sequence is reduced to at most one operation. Whenever a transformation merges two operations to create a new operation, we add all operations from the operation set of the first transaction to the operation set of the second transaction: the second transaction can only complete if the first succeeds.

After this merge operation completes, the only remaining dependencies are those resulting from "blind writes" in which an item is removed and then replaced (e.g., deleting an item and then inserting a new item with the same key). If we treat the set of updates as follow an ordering across the different *types* (applying deletions before replacements, and deferring insertions until last), then we can achieve the effect of the original update sequence in three successive steps of applying flattened sets of updates.

Proposition 1 *Given set semantics and our assumption that all operations without data dependencies are independent, we can achieve the results of the original update sequence over a relation R by applying the flattened updates in three successive steps. The first step removes all tuples that are removed in the original update sequence; the second step modifies tuples that were modified in the original update sequence; the final step inserts new tuples that were created in the original update sequence.*

Since the results of the "flattened" update sequence over relation $r(\bar{X})$ are three (unordered, independent) sets of deltas, we define three *delta relations over* r, one for each type of operation. These are $-r(\bar{X})$, $\rightarrow r(\bar{X}, \bar{X}')$, and $+r(\bar{X})$, representing the collections of tuples to be deleted, the collection of modifications to be applied,

$$
\begin{array}{l}
[+A, +A, \ldots] \Rightarrow [+A, \ldots] \\
[+A, -A, \ldots] \Rightarrow [\ldots] \\
[+A, \rightarrow A:B, \ldots] \Rightarrow [+B, \ldots] \\
[-A, -A, \ldots] \Rightarrow [-A, \ldots] \\
[\rightarrow A:A, \ldots] \Rightarrow [\ldots] \\
[\rightarrow A:B, +B, \ldots] \Rightarrow [A \rightarrow B, \ldots] \\
[\rightarrow A:B, -B, \ldots] \Rightarrow [-A, \ldots] \\
[\rightarrow A:B, \rightarrow A:B, \ldots] \Rightarrow [\rightarrow A:B, \ldots] \\
[\rightarrow A:B, \rightarrow B:C, \ldots] \Rightarrow [\rightarrow A:C, \ldots]
\end{array}
$$

Figure 3: Transformation rules for flattening dependency sequences. A, B, C represent the keys of different updates over the same relation.

and the collection of tuples to be inserted, respectively. We use these delta relations as the basis of defining *update mappings* in Section 6.

5.3 Reconciliation in a Dynamic Environment

Given a set of "flattened" updates from a reconciling participant p, ORCHESTRA must compare these updates to those performed elsewhere since p's last reconciliation. Not all participants may be available, so we do not perform pairwise reconciliations. Instead, our solution is to record and replicate all published updates in a peer-to-peer distributed hash table, which allows us to both replicate data and distribute computation.

Every update published by a participant is routed to a set of nodes in the P2P substrate according to its key (and relation name). Once all updates have been published, each peer will have received all updates, from any participant, which share the same key. It can quickly determine which updates are trusted by the reconciling participant (by filtering against the instance mapping), remove all conflicting updates (and all updates from conflicting transactions), and then return the remaining updates to the reconciling participant to be applied to its state. The major challenge in reconciliation lies in ensuring that the process is atomic and resilient to the failure of any node. We rely on redundant computation, done at each replica, to ensure that reconciliation completes even if a peer disconnects. We also designate a specific peer to be the *coordinator* of a reconciliation operation for a specific shared relation: like a distributed lock, it prevents concurrent reconciliations over the same relation.

5.4 Implementation Status

We have an early implementation of the distributed reconciliation engine, built over the Pastry distributed hash table substrate. We are in the process of constructing a synthetic workload generator and testbed to evaluate the implementation with a large number of peers and large data sets.

6 Mapping Updates between Schemas

ORCHESTRA allows update propagation *across* different schemas, meaning that reconciliation typically re-

quires more work than simply comparing delta recorded within the peer-to-peer substrate. The updates must first be translated into the schema of reconciling participant p. Unfortunately, there may not be direct mappings from every schema to that of p. Here we leverage techniques developed for peer data management [23, 43]: participant p will *query* for all deletions, replacements, and insertions made to its schema, and it will compare those with the updates it has made. The updates it receives will be based on the transitive closure of the constraints of all schema mappings in the system.

We briefly describe the salient features of peer data management [23, 43]. The goal is to reformulate a query posed over one schema into queries over all other schemas, based on *compositions* of mappings between pairs of schemas. This enables the peer data management system to use a data source to provide certain answers to a query even when a direct mapping to that source is unavailable. The result is a union of conjunctive queries, which provide the maximally complete set of certain answers that satisfy all mappings in the system, provided that there are no cycles in the mappings. If cycles are present, then the algorithm returns sound (but not complete) answers.

6.1 Update Translation Mappings

In order to leverage the query reformulation approach discussed above, in ORCHESTRA we take the mappings between relations and convert them into mappings between "delta relations" (described in the previous section). Essentially, we recast the update translation problem into one of translating unordered sets of updates from one schema to another.

The update translation process differs subtly from query unfolding and reformulation. Given a standard schema mapping from schema S_1 to a relation in schema S_2, it is possible to easily derive an update over S_2 from an update over S_1; this forms the basis of incremental view maintenance [8, 13]. However, translating in the *inverse direction*, from the output of a view to its base relations, is not deterministic: the same deletion of a tuple in the `Exper` relation of Figure 2 might be accomplished by deleting a tuple from *either* `Study` or `IDMap`, or by deleting a tuple from *both* relations — or by changing the value of one of the relations' join attributes. Moreover, depending on the key and functional dependency constraints on the schemas, some of these alternatives may not always achieve correct, "side effect free" translation. In fact, there may be no way to propagate the update at all! Finally, note that one possible translation accomplishes an operation (delete) by executing an *entirely different class of operation* (replacement of a join key). Different aspects of this problem have been addressed in [5, 15, 28].

A mapping for updates must be able to specify how to take *a particular type of update to the output of a view* and translate it into a set of updates over base relations. In traditional schema mappings, a similar effect may be achieved using *inverse rules* [32]: an inverse rule defines how to derive a tuple in a base relation from tuples in a view, given that a view defines a constraint over the base relation(s) and view. Here, the problem is to constrain the relationship between an *update* over a base relation and an update over a view. If update operations are represented as delta relations, then an update translation is an "inverse rule" that maps a delta relation for the view's output into a delta relation over a base relation. Moreover, simple variations of this rule (e.g., changing from one type of delta to another, or joining with data in a base relation in order to supply additional attribute values) can deterministically express the classes of update translation proposed in [28]. The following example illustrates.

Example 6.1 *Consider the schema mapping between* `Exper` *and* `Study` *provided in Figure 2. Given an update to* `Exper`*, valid update translations could either delete a tuple from* `Study`*, a tuple from* `IDMap`*, or both. This must be disambiguated by specifying a single deletion mapping such as:*

$$\texttt{-Study}(s, \ldots) \texttt{ :- -Exper}(s2, \ldots), \texttt{IDMap}(s, s2)$$

which deletes tuples from `Study` *but* not *from the ID mapping table.*

One drawback to "inverting" a view is that quite frequently, certain information from the base relations is projected away by the view. A common technique in data integration is to use *Skolem functions* to represent "unknown" values. An important benefit of Skolem functions over traditional null values is that the same Skolem value can be used in more than one place — enabling the query processor to determine that it can join or merge tuples. In some data translation scenarios, we can go one step beyond: there may exist a mapping table that, given a key, supplies values for the unknown information. The `IDMap` is an example of such a mapping table: given a key in either the `RAD` or `MAGE` relations, it can supply the other key.

6.2 Translating Updates Using Mappings

Over ordinary update sequences, the approach we describe above must be applied with care, because the order in which updates are applied and combined is important. We avoid most of those issues because we have eliminated all ordering dependencies between deltas of the same type: as discussed previously, we can translate and apply all deletions in one step, removing existing elements from the shared relations; then apply modifications; and finally apply insertions. A naive version of the update translation algorithm, triggered by participant p, proceeds as follows:

1. Traverse p's instance mappings to its shared relations. Collect information about the set of shared schemas and mappings that are reachable from p's shared relations.

2. For every shared MVT relation S, compute the set of tuples to be deleted from S, given the state of the shared MVT instances from the last reconciliation of p[6] and all published updates since that point.

3. For every shared MVT relation S, compute the set of tuple replacement operations over S, given the shared MVT instances that result after the deletions of the previous step have been applied, and all published updates since p's last reconciliation.

4. For every shared MVT relation S, compute the set of insertions over S, given the shared MVT instances that result after the deletions and replacements of the previous steps have been applied, and all published updates since p's last reconciliation.

Some mappings between schemas may only convey partial information, e.g., a subset of the attributes of the target relation. If more than one mapping path exists between a pair of schemas, it may be possible to merge tuples from both paths (e.g., on a key attribute) in order to recover complete tuples.

6.2.1 Implementation Status

More efficient approaches than the naive algorithm for update translation are a focus of ongoing work. Since there may be a large number of interacting update translation rules, we are focusing on eliminating irrelevant ones. Clearly, we can push the reconciling participant's instance mapping predicates into the reformulation process — eliminating irrelevant viewpoints from the process. However, there is also a great deal of potential for eliminating redundant expressions and for distributing (and caching) common subexpressions.

6.3 From Schema Mappings to Update Mappings

Our update translation approach is based on an assumption that custom mappings have been created to specify how updates can be propagated. This process occurs *offline*, and it can be customized by the participant who defines the mapping.

Given a traditional schema mapping (potentially derived from a schema matching tool such as those described in [36]), it is trivial to derive the update translation rules that "mirror" a traditional schema mapping from relations $s_1, \ldots, s_k$ to relation r. Since we separately consider deletions, replacements, and insertions, we need three views to derive the deletion, replacement, and insertion delta-relations over r. Each query which produces the delta-relation of type Δ is computed using the original relations $s_1, \ldots, s_k$ (after any previous update propagation steps) and the Δ delta-relations over them. We generate a rule for the Δr relation for each possible way of substituting Δs_i for s_i in the original schema mapping.

6.3.1 Generating Inverse Update Mappings

The problem of generating "inverse" update mappings is significantly more complex, particularly since not every schema mapping *has* a correct update translation that produces precisely the desired effect (and no others). Moreover, if *multiple* mappings exist between a pair of relations, it is possible that such mappings might interact. Our initial approach in ORCHESTRA is to consider only those cases where a sound and complete, instance-independent means of mapping an update exists, and the effects of individual mappings can be considered independently (i.e., either there is only a single mapping or different mappings' translations do not contradict one another).

We have developed a Mapping Wizard that takes a traditional mapping between schemas and creates a set of update translator mappings to propagate changes in the reverse direction. When necessary, it prompts the local administrator to choose among valid update mappings. Our approach builds upon the algorithm of Dayal and Bernstein [15], which determines which source relations can become targets of a modification; and on work by Keller [28], who enumerated alternative ways of performing an update (e.g., deleting a tuple in the view by modifying the value of a source attribute). The main novelty is that we generate translations as *rules*, rather than directly applying the update as part of the algorithm.

6.3.2 Number of Possible Translations

For our techniques to be scalable — and particularly for them to be manageable by users and administrators — the number of alternative update translations (and resulting rules) must be small. A close study of the approaches of [15, 28] reveals that a view must have a very specific structure in order to be updatable. Specifically, it must be possible to find one base relation that functionally determines the remaining source tuples in the view. It is rare to have more than one relation within the same view that has this characteristic, and thus an update typically has few alternative sources to which it may be propagated. Thus, the only remaining choices in terms of update propagation tend to be along the lines of those suggested by Keller.

Table 1 shows real-world results for real-world bioinformatics schema mappings, converting from the RAD database to the MAGE-ML interchange format. These views represent combinations of different aspects of experimental setups and data. Our results show that the

[6] Note that reconciliation is always done between a participant and the system *from the point that participant last reconciled*.

View	# Rel.	Targets	Var.	Rules
Deletion				
RAD-DESIGN	4	1(1)	3(2)	1
RAD-DESIGN-ASSAY	5	3(2)	1(1)	2
RAD-DESIGN-FACTOR	6	3(2)	8(3)	2
RAD-TREATMENT	4	1(1)	3(2)	1
RAD-ASSAY-BIOMAT	3	1(1)	1(1)	1
RAD-ARRAY	4	1(1)	3(2)	1
RAD-COMP-ELEM	6	3(2)	1(1)	2
RAD-SPOT	9	1(1)	15(4)	1
Insertion				
RAD-DESIGN	4	1(1)	-	5(4)
RAD-DESIGN-ASSAY	5	1(1)	-	7(5)
RAD-DESIGN-FACTOR	6	1(1)	-	8(6)
RAD-TREATMENT	4	1(1)	-	5(4)
RAD-ASSAY-BIOMAT	3	1(1)	-	4(3)
RAD-ARRAY	4	1(1)	-	5(4)
RAD-COMP-ELEM	6	1(1)	-	8(6)
RAD-SPOT	9	1(1)	-	10(9)

Table 1: Characterization of possible translations of updates over bioinformatics mappings, showing number of relations in the view, number of sources that may be modified to achieve the update, number of different ways such modifications may be performed, and maximum number of resulting rules for any translation. Numbers in parentheses indicate values when we assume a minimal number of update operations.

number of possible sources of deletion (3rd column) is typically small: it is only greater than 1 when the base table joins with another relation with which it has a 1:1 relationship. Within a given tuple, it may be possible to modify different attributes; this is the 4th column in the table, and the figure is only slightly larger. The total number of resulting update translation rules is shown in the last column. Note that the numbers generally represent *worst-case* numbers of user choices and numbers of rules; the numbers within parentheses represent the worst case if we limit ourselves to deleting only one source tuple for every deletion over the view. From this we tentatively conclude that our approach produces manageable numbers of rules.

6.3.3 Implementation Status

We have completed an early prototype of the Mapping Wizard tool discussed in this section. Our deinitial experience suggests that our approach is a feasible way of taking traditional schema mappings and allowing local administrators to rapidly and easily convert them into precise update translation mappings. Our current work focuses on reasoning about *interactions* between mapping translation rules.

7 Conclusions and Future Work

In this paper, we have shown how the problem of rapid, collaborative sharing of scientific data requires us to revisit our models of data sharing. We have described how our work on the ORCHESTRA system addresses this problem:

- It uses a novel data model and languages for *managing* conflicting information.
- It provides a peer-to-peer distributed hash table substrate to replicate and exchange updates.
- It extends query rewriting techniques from data integration to map updates between schemas.

Our current work is focused on refining and evaluating the peer-to-peer implementation of ORCHESTRA, based on the algorithms and techniques discussed. Our primary focus is on finding techniques to optimize the update translation process. Equally important, we are working on extending our data model to XML, which has become the standard format for exchanging structured data. Finally, we are considering means of applying our techniques when mappings do not satisfy the constraints described in Section 6: perhaps it is possible to define a clean mapping over *subsets* of the source data instances, enabling "best effort" update translation.

References

[1] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *CoopIS*, 2001.

[2] S. Abiteboul and G. Grahne. Update semantics for incomplete databases. In *VLDB '85*, pages 1–12, 1985.

[3] V. S. Alagar, F. Sadri, and J. N. Said. Semantics of extended relational model for managing uncertain information. In *CIKM '95*, pages 234–240, 1995.

[4] M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *PODS '99*, pages 68–79, 1999.

[5] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.

[6] T. Barsalou, A. M. Keller, N. Siambela, and G. Wiederhold. Updating relational databases through object-based views. In *SIGMOD '91*, pages 248–257, 1991.

[7] 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.

[8] J. A. Blakeley, P.-Å. Larson, and F. W. Tompa. Efficiently updating materialized views. In *SIGMOD '86*, pages 61–71, 1986.

[9] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT '01*, pages 316–330, 2001.

[10] D. Calvanese, G. D. Giacomo, M. Lenzerini, and R. Rosati. Logical foundations of peer-to-peer data integration. In *PODS '04*, pages 241–251, 2004.

[11] S. Ceri, M. A. W. Houtsma, A. M. Keller, and P. Samarati. Independent updates and incremental agreement in replicated databases. *Distributed and Parallel Databases*, 3(3):225–246, 1995.

[12] S. S. Chawathe and H. Garcia-Molina. Meaningful change detection in structured data. In *SIGMOD '97*, pages 26–37, 1997.

[13] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In *SIGMOD '96*, pages 469–480, 1996.

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

[15] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3):381–416, 1982.

[16] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *PODC '87*, 1987.

[17] W. K. Edwards, E. D. Mynatt, K. Petersen, M. J. Spreitzer, D. B. Terry, and M. M. Theimer. Designing and implementing asynchronous collaborative applications with Bayou. In *UIST '97*, pages 119–128, 1997.

[18] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. Technical Report MS-CIS-004-15, University of Pennsylvania, July 2004.

[19] S. Ghandeharizadeh, R. Hull, and D. Jacobs. Heraclitus: Elevating deltas to be first-class citizens in a database programming language. *TODS*, 21(3):370–426, 1996.

[20] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? In *WebDB '01*, June 2001.

[21] N. I. Hachem, K. Qiu, M. A. Gennert, and M. O. Ward. Managing derived data in the Gaea scientific DBMS. In *VLDB '93*, pages 1–12, 1993.

[22] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: Data management infrastructure for semantic web applications. In *12th World Wide Web Conference*, May 2003.

[23] A. Y. Halevy, Z. G. Ives, D. Suciu, and I. Tatarinov. Schema mediation in peer data management systems. In *ICDE '03*, March 2003.

[24] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Quering the Internet with PIER. In *VLDB '03*, pages 321–332, 2003.

[25] T. Imielinski and W. Lipski. Incomplete information in relational databases. *JACM*, 31(4):761–791, 1984.

[26] 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 '99*, pages 299–310, 1999.

[27] Z. G. Ives, A. Y. Halevy, and D. S. Weld. Adapting to source properties in processing data integration queries. In *SIGMOD '04*, June 2004.

[28] A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *SIGMOD '85*, pages 154–163, 1985.

[29] A. Kementsietsidis, M. Arenas, and R. J. Miller. Mapping data in peer-to-peer systems: Semantics and algorithmic issues. In *SIGMOD '03*, June 2003.

[30] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of diverging replicas. In *PODC '01*, August 2001.

[31] D. Lembo, M. Lenzerini, and R. Rosati. Source inconsistency and incompleteness in data integration. In *KRDB '02*, April 2002.

[32] M. Lenzerini. Tutorial - data integration: A theoretical perspective. In *PODS '02*, 2003.

[33] K.-C. Liu and R. Sunderraman. Indefinite and maybe information in relational databases. *TODS*, 15(1):1–39, 1990.

[34] W. S. Ng, B. C. Ooi, K.-L. Tan, and A. Zhou. PeerDB: A P2P-based system for distributed data sharing. In *SIGMOD '03*, 2003.

[35] B. C. Pierce, T. Jim, and J. Vouillon. UNISON: A portable, cross-platform file synchronizer, 1999–2001.

[36] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

[37] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of ACM SIGCOMM '01*, 2001.

[38] A. Rosenthal and E. Sciore. First-class views: a key to user-centered computing. *SIGMOD Rec.*, 28(3):29–36, 1999.

[39] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[40] C. Stoeckert, A. Pizarro, E. Manduchi, M. Gibson, B. Brunk, J. Crabtree, J. Schug, S. Shen-Orr, and G. C. Overton. A relational schema for both array-based and SAGE gene expression experiments. *Bioinformatics*, 17(4):300–308, 2001.

[41] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of ACM SIGCOMM '01*, 2001.

[42] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[43] I. Tatarinov and A. Halevy. Efficient query reformulation in peer-data management systems. In *SIGMOD '04*, June 2004.

[44] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal of Computing*, 18(6):1245–1262, 1989.