# The ORCHESTRA Collaborative Data Sharing System

Zachary G. Ives Todd J. Green Grigoris Karvounarakis Nicholas E. Taylor Val Tannen
Partha Pratim Talukdar Marie Jacob Fernando Pereira*
Computer and Information Science Department
University of Pennsylvania
{zives,tjgreen,gkarvoun,netaylor,val,partha,majacob,pereira}@cis.upenn.edu

## ABSTRACT

Sharing structured data today requires standardizing upon a single schema, then mapping and cleaning all of the data. This results in a single queriable *mediated* data instance. However, for settings in which structured data is being collaboratively authored by a large community, e.g., in the sciences, there is often a *lack of consensus* about how it should be represented, what is correct, and which sources are authoritative. Moreover, such data is seldom static: it is frequently updated, cleaned, and annotated. The ORCHESTRA *collaborative data sharing system* develops a new architecture and consistency model for such settings, based on the needs of data sharing in the life sciences. In this paper we describe the basic architecture and implementation of the ORCHESTRA system, and summarize some of the open challenges that arise in this setting.

## 1 INTRODUCTION

Increasingly, progress in the sciences, medicine, academia, government, and even business is being facilitated through sharing large structured data resources. Examples include curated experimental data, student grades, census or survey data, customer reports, market projections, and so on. In general, these data resources are evolving over time, as they are extended and revised in collaborative fashion by an entire community. Effective data-centric collaborations have a number of key properties: (1) they generally benefit all parties, without imposing undue work or restrictions on anyone; (2) they include parties with diverse perspectives, both in terms of how information is modeled or represented, and what information is believed to be correct; (3) they may involve differences of authoritativeness among contributors; (4) they support an evolving understanding of a dynamic world, and hence include data that changes.

As an example of this type of collaboration in the sciences, consider the field of bioinformatics. Here there are a plethora of different databases, each focusing on a different aspect of the field — organisms, genes, proteins, diseases, etc. — from a unique perspective. Associations exist between the different databases' data (e.g., links between genes and proteins, or gene homologs between species). Multiple standardization efforts have resulted in large data warehouses (e.g., GenBank, SWISS-PROT, InterPro, etc.), each of which seeks to be the definitive portal for a particular bioinformatics sub-community. Each such warehouse provides three services to its community:

1. A conceptual model, in the form of a custom schema with terminology matched to the community;
2. Access to data, both in the form of raw measurements and also derived possible associations, e.g., a gene that appears to be correlated with a disease;
3. Cleaning and curation of data produced locally, as well as data that has possibly been imported from elsewhere.

Different sub-communities may occasionally disagree about which data is correct! Yet, some of the databases import data from one another (typically using custom scripts); and each warehouse is being constantly updated, with corrections and new data typically published (in the form of *deltas* describing changes) on a weekly, monthly, or on-demand basis.

Currently, there is no principled infrastructure for supporting collaborations along these lines: at best, scientists use ad hoc collections of scripts to exchange their data. We observe that their usage model is *update-centric* and requires support for *multiple schemas* and *multiple data versions*. Tools for managing heterogeneous structured data — e.g., those developed for data integration and warehousing — are *query-centric*, tend to assume a *single global schema* to which all data gets mapped, and strive to define a single clean global data instance. Even recent *peer data management systems* [5, 25, 32], while supporting multiple schemas, are not flexible enough to meet life scientists' needs for managing data importation, updates, and inconsistent data. Recent proposals for probabilistic database systems [2, 4, 11, 34] manage uncertainty within a single database instance, but do not help with integration across multiple databases or management of consistency and reconciliation of conflicts.

In order to provide collaborating scientists, organizations, and end users with the tools they need to share and revise structured data, we have been developing a new architecture we term *collaborative data sharing systems* [28] (CDSSs), and the first implementation of a CDSS in the form of the ORCHESTRA system. The CDSS provides a principled semantics for exchanging data and updates among autonomous

* Currently on leave at Google.

sites, which extends the data integration approach to encompass scientific data sharing practices and requirements — in a way that also generalizes to many other settings. The CDSS models the exchange of data among sites as *update propagation among peers*, which is subject to transformation (schema mapping), filtering (based on policies about source authority), and local revision or replacement of data.

Each participant or peer in a CDSS controls a local database instance, encompassing all data it wishes to manipulate (possibly including data that originated elsewhere). The participant normally operates in "disconnected" mode for a period, making local modifications to data stored in a local DBMS. As edits are made to this database, they are logged. At the users' discretion, the *update exchange* capability of the CDSS is invoked, which publishes the participant's previously-invisible updates to "the world" at large, and then translates others' updates to the participant's local schema — also filtering which ones to apply, and reconciling any conflicts, according to the local administrator's unique trust policies, before applying them to the local database.

Declarative *schema mappings* specify one participant's schema-level relationships to other participants, in a compositional way resembling the peer data management system [25] model[1]. Schema mappings may be annotated with *trust policies*: these specify filter conditions about *which* data should be imported to a given peer, as well as precedence levels for reconciling conflicts. Trust policies take into account the *provenance* or lineage [4, 6, 7, 9] of data.

EXAMPLE 1. Figure 1 shows a screen shot from the ORCHESTRA management interface, featuring a simplified version of a bioinformatics collaborative data sharing setting for 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 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 BioSQL wants to import data from GUS, as shown by the arc labeled $m_1$, but the converse is not true. Similarly, uBio wants to import data from GUS, along arc $m_2$. Additionally, BioSQL and uBio agree to mutually share some of their data: e.g., uBio imports taxon names from BioSQL (via $m_3$) and BioSQL uses mapping $m_4$ to add entries for synonyms to any organism names it has in its database. Finally, each participant may have a certain *trust policy* about what data it wishes to incorporate: e.g., BioSQL may only trust data from uBio if it was derived from GUS entries. The CDSS facilitates dataflow among these systems, using mappings and policies developed by the independent participants' administrators. □

In this paper, we provide an overview of the basic operation of the CDSS, describe our existing prototype implementation (demonstrated at the SIGMOD 2007 conference [21]), and describe some of the open research problems that arise when using ORCHESTRA as a data sharing platform.

[1]These schema mappings may also include record linking tables translating terms or IDs from one database to another [32].

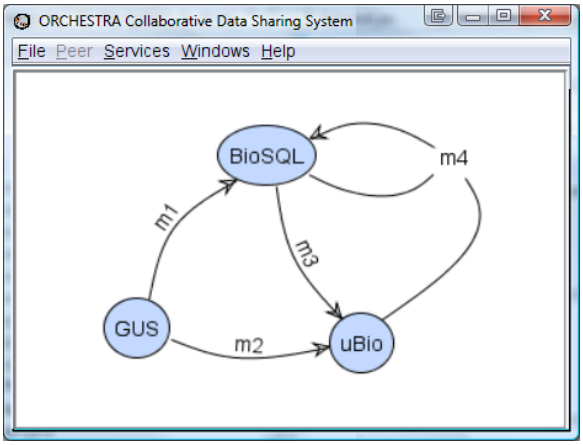

Figure 1: Example collaborative data sharing system for three bioinformatics sources. For simplicity, we assume one relation at each participant (GUS, BioSQL, uBio). Schema mappings are indicated by labeled arcs.

## 2 ORCHESTRA OVERVIEW

The ORCHESTRA CDSS is a fully peer-to-peer architecture with no central server. An ORCHESTRA runtime sits above an existing DBMS on every participant's machine (peer) P, and manages the exchange and permanent storage of updates. In general, each peer represents an autonomous domain with its own unique schema and associated *local data instance* (managed by the DBMS). The users located at P typically query and update the local instance in a "disconnected" fashion. Periodically, upon the initiative of P's administrator, P invokes the CDSS. This *publishes* P's local edit log — making it globally available. This also subjects P to the effects of *update exchange*, which fetches, translates and applies updates that other peers have published (since the last time P invoked the CDSS). After update exchange, the initiating participant will have a data instance incorporating the most-trusted changes made by participants transitively reachable via schema mappings. Any updates made locally at P can modify data imported (by applying updates) from other peers.

ORCHESTRA's features are grouped into three main modules, each of which is described in more detail later in this paper and in the references.

**Publishing and Archiving Update Logs (Section 3).** The first stage of sharing updates with other peers in ORCHESTRA is to *publish* data. Following the philosophy that any data, once published, should remain part of a permanent record, ORCHESTRA provides "zero administration," versioned, replicated storage for published updates — maximizing the likelihood that data (whether current or archived) will be available in the system. This is based on peer-to-peer replication and storage techniques [41].

**Transforming and Filtering Updates (Sections 4–5).** Perhaps the most complex aspect of the CDSS model, and of the ORCHESTRA implementation, revolves around how updates are processed, filtered, made consistent, and applied to a given participant's database instance. Figure 2 shows the basic data processing "pipeline" from the perspective of a given peer. Initially, all updates not-yet-seen by the peer are fetched. Next, update exchange (Section 4) is performed,

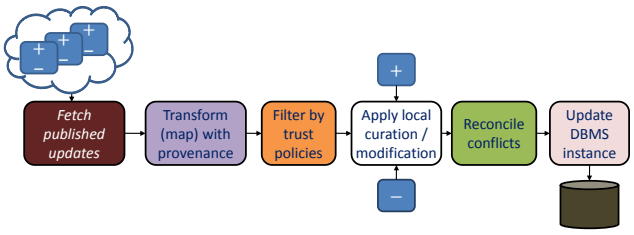


Figure 2: ORCHESTRA stages for importing updates to a peer.

consisting of two aspects: transforming or mapping the updates using schema mappings, while recording the mapping steps as data provenance [6, 9]; then, filtering trusted vs. untrusted updates based on their provenance, according to local trust policies. Now, any modifications made by local users are additionally considered, forming a set of *candidate updates*. These candidate updates may be grouped into transactions, and they may have data dependencies. The *reconciliation* process (Section 5) arbitrates among the possible updates and determines a consistent set to apply to the peer's database instance.

**Querying across Peers (Section 6).** ORCHESTRA's primary data sharing mechanisms are oriented around local data instances, and the user of any peer's database may never need to directly interact with ORCHESTRA. However, in some cases we would like to query *across* different peers, perhaps in different sub-fields. A scientist or other user in a CDSS may not know which peers are most relevant, nor how to write queries in SQL. ORCHESTRA's query system, Q [40], provides a facility through which non-expert users can author queries (or, more specifically, query templates that generate Web forms) over the relations on any peers in the system. Q is initially given a keyword query, which it attempts to match against schema elements. From the matching, it constructs a ranked list of potential conjunctive queries that meet the user's information need, executes the top queries, and returns answers. The user may provide feedback on the answers, which are used to re-rank the queries and generate new, more relevant results.

# 3 ARCHIVING UPDATES PERSISTENTLY

The act of *publishing* updates to ORCHESTRA is intended to maintain a permanent record of a peer's changes to the data, which is accessible to all users even if future changes are made. In some sense this resembles a version control system, except that each peer's updates occur over a different schema and they must later be merged by each individual peer as it refreshes its data instance during update exchange.

The initial step in publishing a peer's updates is to extract a log of changes from the peer's DBMS. Here ORCHESTRA uses a DBMS-specific wrapper that may use one of several different techniques. In many higher-end DBMSs, the wrapper hooks into the queuing system used for distributed replication; this avoids costly data analysis or transaction log crawling. If the DBMS does not support such capabilities, we can compare old and new data snapshots, or in some cases crawl the transaction log (when enough semantic information is preserved).

Once obtained, updates are published to a fully decentralized, peer-to-peer *update store* — a persistent, highly available storage subsystem, which allows updates to be grouped into transactions, and which records data dependencies among transactions. Transactions are logically globally timestamped according to when they are published. In [41] we describe how a distributed hash table [39] is used to partition and replicate data across all of the currently-available participants. The advantages of this architecture are that (1) no dedicated machine is required, (2) no administration is required, and (3) most importantly, as machines in the CDSS setting are replaced or upgraded, data will automatically migrate to these machines.

# 4 TRANSFORMING UPDATES

The update store is responsible for making data available to other peers; however, in the common case, these updates will not be in the same schema, using the same identifiers. Moreover, not every peer will consider every update to be of equal authority or quality. The *update exchange* operation involves *translating* updates across schema mappings (and possibly identifiers); *tracking provenance* of those updates; and *filtering according to trust policies*. Moreover, the peer's users may *override* data imported by update exchange, through *local curation* (updates). Finally, the set of imported and local updates may not in fact be mutually compatible; thus, update exchange is followed by *reconciliation* (Section 5).

## 4.1 Basic Update Exchange

Logically, the process of translating updates in ORCHESTRA is a generalization of *data exchange* [16]. If we take the data locally inserted by each peer to be the source data in the system, then (in the absence of deletions or trust conditions) ORCHESTRA computes at every peer a database instance that is a *canonical universal solution* [16]. The canonical universal solution is a materialized data instance from which all of the *certain answers* [24] to a query can be computed — the user will get back a set of query answers following the semantics used in over a decade of virtual data integration research, and matching the results returned by peer data management systems [25] with the same mappings.

Of course, there are many additional subtleties introduced by deletions, the computation of provenance, and trust conditions. We provide a brief overview of the update exchange process here, and refer the reader to [19] for full details.

**Schema mappings.** ORCHESTRA uses tuple generating dependencies (tgds) to express schema mappings as constraints between data instances. Tgds are a popular means of specifying constraints and mappings [13, 16] in data sharing, and can also be viewed as *global-local-as-view* or *GLAV* mappings [24], which in turn generalize the earlier global-as-view and local-as-view mapping formulations [35]. 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, $\phi$, is a conjunction of atoms over variables $\bar{x}$ and $\bar{y}$, and the right hand side (RHS) of the implication, $\psi$, is a conjunction of atoms over variables $\bar{x}$ and $\bar{z}$. The tgd expresses a constraint about the existence of tuples in the instance on the RHS, given a

particular combination of tuples satisfying the conjunctive query on the LHS.

EXAMPLE 2. Refer to Figure 1. Peers GUS, BioSQL, uBio have one-relation schemas describing taxon IDs, names, and canonical names: $G(id, can, nam)$, $B(id, nam)$, $U(nam, can)$. Among these peers are mappings:

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

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. □

**Data Exchange Programs.** Let us focus initially on how ORCHESTRA would compute data instances given data locally contributed by peers; we will then discuss how to extend this to updates. ORCHESTRA builds upon the model of data exchange, where tgds are typically used with a procedure called the *chase* [1] to compute a canonical universal solution. Importantly, this solution is not a standard data instance, but rather a *v-table*, a representation of a set of possible database instances. For instance, in $m_3$ in the above example, the variable $c$ may take on many different values, each resulting in a different instance. Rather than apply the chase procedure directly, ORCHESTRA instead translates the mappings into a program in an extended version of Datalog, which includes support for Skolem functions (these take the place of existential variables like $c$). The resulting (possibly recursive) program computes a canonical universal solution as well, but has benefits arising from the fact that it is a query as opposed to a procedure. The program greatly resembles that of the *inverse rules* query answering scheme [15], and also the XQuery rules used in the Clio system [38]. We note that the set of mappings must be *weakly acyclic* [14] in order for the program to terminate.

EXAMPLE 3. The update exchange Datalog program for our running example includes the following rules (note that the order of the source and target is reversed from the tgds):

$$\begin{aligned} & B(i,n) \text{ :- } G(i,c,n) \\ & U(n,c) \text{ :- } G(i,c,n) \\ & U(n,f(i,n)) \text{ :- } B(i,n) \\ & B(i,n) \text{ :- } B(i,c), U(n,c) \end{aligned}$$

This program is recursive (specifically, with respect to $B$), and must be run to fixpoint. □

**From Data to Update Exchange.** Update exchange requires the ability for each peer not simply to provide a relation with source data, but in fact to provide a set of *local updates* to data imported from elsewhere: insertions of new data as well as deletions of imported data. ORCHESTRA models the local updates as relations, as follows. It takes the *local update log* at each peer and first "minimizes it," removing insertion-deletion pairs that cancel each other out. Then it splits the local updates of each relation $R$ into two logical tables: a *local contributions table*, $R^l$, including all inserted data, and a *local rejections table*, $R^r$, including all deletions of external data. It then updates the Datalog rules for $R$ by adding a mapping from $R^l$ to $R$, and by adding a $\neg R^r$ condition to every mapping. For instance, the first mapping in our example would be replaced with:

$$\begin{aligned} & B(i,n) \text{ :- } B^l(i,n) \\ & B(i,n) \text{ :- } G(i,c,n), \neg B^r(i,n) \end{aligned}$$

Finally, for efficiency ORCHESTRA actually performs incremental propagation of insertions and deletions. This requires *incremental view maintenance* [23] techniques, which take the set of updates, plus the contents of the existing relations, and propagate the necessary changes to accomplish the results of the update. Our implementation is novel in that it exploits data provenance (discussed next) to significantly speed up deletion propagation. Specifically, provenance is used to determine whether view tuples are still derivable when some base tuples have been removed (see [19]).

For peers that require closer collaboration, e.g., that wish to mirror each other's data, we have also introduced *bidirectional* mappings and bidirectional update exchange [30]. The latter involves a generalization of the *view update* [12] problem, where removing a derived tuple also removes (some of) its source tuples. We provide algorithms that take advantage of provenance information to detect and avoid side effects at run-time, as explained in [30].

## 4.2 Data Provenance

One challenge in data integration — particularly peer-to-peer-style data integration — is that it becomes very difficult to determine *why* and *how* a tuple came into existence in a data instance. Such *provenance* information becomes particularly essential when not all sources are equally reliable. In ORCHESTRA, provenance is created and maintained as part of update exchange, and it is primarily used to allow each peer to assess how much it *trusts* a given update (discussed in the next subsection). Our provenance formalism describes how each tuple is introduced into a data instance as an *immediate consequence* of a mapping and a set of source tuples in other instances.

EXAMPLE 4. Consider the mappings from our running example. The provenance of the data in the peers' instances can be captured as a graph (Figure 3) with two kinds of nodes: tuple nodes, shown as rectangles below, and mapping nodes, shown as ellipses. Arcs connect tuple nodes to mapping nodes that use the tuples as input, and mapping nodes to tuple nodes representing derivations. The 3-D nodes in the figure represent 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.

From this graph we can analyze the provenance of, say, $B(3,2)$ by tracing back paths to source data nodes — in this

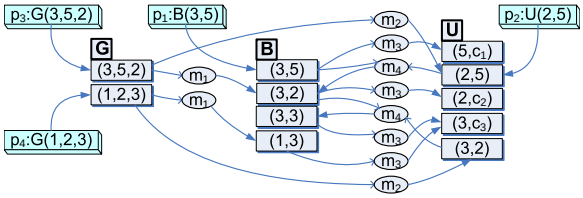


Figure 3: Provenance graph corresponding to example CDSS setting

case through $m_4$ to $p_1$ and $p_2$ and through $m_1$ to $p_3$. □

The provenance of each tuple in ORCHESTRA is formally an expression from the provenance semiring [20], but we encode it in relations, which can be updated incrementally with an extended version of our update exchange program; and they can be queried using Datalog. As discussed previously, the provenance graph is also used during incremental maintenance to speed performance [19].

## 4.3 Trust Policies and Provenance

Schema mappings describe the relationships between data elements in different instances. However, mappings are compositional, and not every peer wants to import all data that can be logically mapped to it. A peer may distrust certain sources, or favor some sources over others, e.g., because one source is more authoritative. *Trust policies*, specified for each peer, encode conditions over the data and provenance of an update, and associate a *priority* with the update. A priority of 0 means the update is untrusted.

EXAMPLE 5. As examples, $U$ may trust data from $G$ (giving it priority 2) more than $B$ (given priority 1). $B$ might not trust any data from mapping $m_3$ with a name starting with "a" (trust priority 0). □

During update exchange, ORCHESTRA will automatically filter out any updates with priority level 0.

# 5 RECONCILING CONFLICTS AMONG TRANSACTIONS

The previous section described how updates can be mapped into a common schema, and untrusted updates can be filtered. In [41], a model was proposed for *reconciliation*, ensuring that each peer receives a consistent (though perhaps unique) data instance. Here we consider the implications of *transactions* (e.g., a user might update an XML tree, which gets mapped to a set of relation updates, of which all or none of the updates should be applied). We define the trust priority of a transaction in terms of its constituent updates: a transaction is *untrusted* if any of its member updates is untrusted (since we consider it significant when an administrator says an item is *not* to be trusted); otherwise, it receives the *highest* trust priority of any contained update (since we otherwise want to ensure that the *most trusted* data is likely to be applied).

Transactions introduce several challenges that do not arise in a simple delete-insert update model: (1) data dependencies (one transaction may depend on the output of another); (2) atomicity (all updates, or none, may be applied); (3) serializability (some transactions can be applied in parallel, and others cannot). Our solution has the following properties:

**Peer-centric consistency model.** Every peer receives a set of updates according to its own policies. This includes all trusted updates that do not conflict; additionally, for each set of conflicting transactions, a peer receives the transaction it most trusts [41]. Each peer may reconcile as often as it wants, or as rarely. The transactions to be reconciled between a target peer and any other peer are those that occurred since both peers reconciled.

**Automatic reconciliation wherever possible.** Each transaction is assigned a priority as described above. If two incompatible transactions are given the same priority, then a user must arbitrate; but this intervention may be *deferred* as long as the user wishes; the system will continue to reconcile any portions of the data that do not "interact" with the deferred transactions.

**Reconciliation with "commit" semantics.** Once reconciliation occurs and a data instance is updated, subsequent reconciliation operations will not roll back the previous work. They may apply updates that modify its results, however.

**Scalable algorithm with simple rules.** ORCHESTRA's reconciliation algorithm [41] runs in time polynomial in the number of transactions, and uses rules that are simple for users to understand. Transactions are considered in order of priority, from highest to lowest; if they are to be applied and they depend on previous transactions, such transactions are also applied. However, if any transaction in a chain cannot be applied while satisfying database constraints, then neither it nor any transaction dependent upon it will be applied.

The ORCHESTRA reconciliation algorithm runs on the reconciling peer, and fetches the set of transactions it has not yet seen from the update store (Section 3). It assigns priorities to every transaction; then, in descending order of priority, it attempts to find the *latest* transactions of that priority that can be applied (together with any *antecedent* transactions needed in order to satisfy read-write or write-read dependencies). This runs in time polynomial in the number of priorities and updates and the length of the transaction chains. Details can be found in [41].

# 6 Q: SYSTEM-WIDE QUERYING

ORCHESTRA is primarily used to exchange data and updates among databases owned by different peers. However, in a large CDSS there will be need to query *across* different peers, e.g., if a user does not know which peer holds the most relevant information. This is where the Q system [40] serves an important role. Q takes a keyword query and turns it into a *view template* (a ranked union of conjunctive queries that may be parametrized at runtime), which is saved persistently along with ranking parameters. When the view template is executed, users see the top answers and provide feedback on these answers; the feedback is used to refine the ranking parameters, and thus the ranked query results.

Unlike prior keyword query systems for databases, Q targets *context-specific information needs*: different users from different communities (or with different goals, e.g., exploration vs. hypothesis confirmation) may ask queries that use similar terms, but they may value individual sources differently. For instance, a poorly curated source might be very useful in exploratory querying, but uninteresting for vali-

dating a hypothesis. Some users may value human-curated sources more (or less) than automatically curated ones. Q allows each view template to be custom-tailored to find the sources most appropriate for a specific information need.

Q starts with a *schema graph* describing all of the peers, relations, schema mappings, foreign keys, and other *associations* among tables. It may additionally have access to inverted indices and ontology (especially subclass and synonym) information. Relations are modeled as nodes in the graph (labeled with the relation and attribute names), and associations are modeled as weighted edges between nodes.

## 6.1 From Keyword Search to Top Queries

In Q a user first poses a keyword query describing the concepts (schema elements such as relations or attributes) relevant to his or her information need. Q matches the keywords against the schema graph and finds join paths among the relations matching different search terms. It uses a Steiner tree algorithm to find the least costly trees containing nodes matching the terms (where the cost of the query is the sum of the edge weights). The *top-k* trees, by rank, are selected and used to generate conjunctive queries for the view template. Additionally, a Web form is generated as a front-end to this view template; this form allows a user to add selection criteria and to project out attributes.

## 6.2 Posing Queries and Returning Answers

The Web form can be made persistent for reuse by the query author and others. A user parametrizes the form's text fields and then executes the query. As answers are computed, they are annotated with data provenance by ORCHESTRA; provenance plays a role in the feedback stage discussed next. Results appear in ranked order, where each tuple receives a weight from the query(ies) that produced it.

## 6.3 View Template Refinement by Feedback

Now the user may provide *feedback* on individual answers (raising or lowering their ranking by confirming or refuting their relevance). The system will use this feedback to adjust the relative scores of the queries, and ultimately the edges in the schema graph. It does this by determining the provenance of the results, and the constraints that the user imposed on the relative ranking of results (e.g., a tuple output by Query 3 must score higher than a result from Query 1). A machine learning algorithm called MIRA [8] adjusts edge weights in a way that attempts to satisfy these constraints. Finally, Q uses the updated schema graph weights to compute a new set of top-k queries, and then a revised set of answers for the user. Over time, the system learns which relations are most relevant to the particular family of queries — and information needs — represented by the view template. The edge weights for this view template are stored with the template, and can even be made the defaults for the system.

The learning scheme in Q has been shown to be highly effective in learning real "gold standard" bioinformatics queries, over moderately large schemas; and it has been shown to scale to hundreds of relations [40].

# 7 RELATED WORK

Naturally, ORCHESTRA has connections to many existing efforts and systems in the literature. The peer-to-peer storage components of ORCHESTRA make heavy use of distributed hash table [39] techniques, including replication and transparent fail-over. In some ways this resembles peer-to-peer file systems like CFS [10].

Update exchange builds upon the foundations of PDMSs (e.g., [25, 32]), which support query reformulation over composable mappings, and data exchange [16, 17, 36, 38], which supports materialization of instances that support certain answers. An alternative mapping formalism with similar properties was proposed in [5]. Rather than simply propagating data, we implement view update [12] and view maintenance [23] behaviors; our implementations differ from prior techniques in that they exploit data provenance for reasoning about side effects (view update) and derivability (maintenance). Our work differs substantially from the data exchange, data cleaning [18], and distributed consistency [33] literature, whose goal is always a single unified, clean data instance: we support trust conditions (based on provenance) and a peer-centric model of consistency, in which many data items are common across instances, but each is allowed to diverge based on local updates or different trust priorities. Our scheme for modeling inconsistent data as a set of individually consistent, overlapping instances also contrasts with recent work on creating single uncertain and probabilistic databases [2, 4, 11, 34]. Our provenance model is based on the formalism of [20], which unifies several previous models [4, 6, 7, 9].

The Q system shares many high-level goals and techniques with keyword search engines over databases [26, 29], which also seek to model the *authority* of relations [3, 22, 31]. Our key difference is a *feedback and learning*-based approach, which allows rankings to be customized to a given view template and user information need.

# 8 ONGOING WORK

While we have developed a prototype ORCHESTRA system, work continues in many directions.

**Reliable distributed queries.** ORCHESTRA's update store employs peer-to-peer techniques to provide persistent archival that adapts to currently available machines and resources. We plan to take even further advantage of peer machines in the system: to actually *push* portions of update exchange query processing, or on-the-fly query answering over virtual views, directly to the nodes holding the stored updates. This should result in higher parallelism in computation, and in many cases less network utilization. However, new techniques must be developed to support *correct and complete* answers in peer-to-peer query processing: we cannot lose answers even if a node fails in mid-computation. Prior work on peer-to-peer query processing, such as [27, 37], assumes *best-effort* semantics and does not guarantee complete answers. New fail-over techniques, and new cost models for query optimization, must be developed.

**Querying data provenance.** Data provenance is often useful for performing post-mortem analysis, understanding the roles of different contributors, etc. We are developing a query language and engine specifically for allowing allowing administrators and advanced users to query the *provenance* of data in the system, in order to debug, assess confidence

or determine authority, perform data forensics, or simply to understand the relationships among data values.

**Mapping evolution.** A key principle behind ORCHESTRA is that the system should be tolerant of constant change, not only at the data level, but also at the level of schemas, mappings, and even trust conditions. In ongoing work we are investigating how to efficiently update the data instances in the system when *mappings* are replaced, added, or removed.

# 9 CONCLUSIONS

The ORCHESTRA project represents a re-thinking of how data should be shared at large scale, when differences of opinion arise not only in the data representation, but also which data is correct. It defines new models and algorithms for transactional consistency, update exchange, provenance, and even ranking of keyword queries. Our initial prototype system demonstrates the feasibility of the concept, and we are in the process of developing a variety of real pilot applications in bioinformatics and medicine, soon to be followed by a release into open source.

We believe that many opportunities for further research are enabled by our platform. Not only is highly distributed query processing a natural fit for our setting, but there are many interesting avenues of exploration along derivations, conflicting data, data versions, etc. Ultimately we would like to explore probabilistic data models in our architecture.

# ACKNOWLEDGMENTS

This work is funded by NSF grants IIS-0477972, 0513778, and 0415810, and DARPA grant HR0011-06-1-0016. We thank Sarah Cohen-Boulakia for the biological data sets; Olivier Biton and Sam Donnelly for code development; and the Penn Database Group, Renée Miller, and the anonymous reviewers for their feedback and suggestions.

# 10 References

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

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

[3] A. Balmin, V. Hristidis, and Y. Papakonstantinou. ObjectRank: Authority-based keyword search in databases. In *VLDB*, 2004.

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

[5] 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.

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

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

[8] K. Crammer, O. Dekel, J. Keshet, S. Shalev-Shwartz, and Y. Singer. Online passive-aggressive algorithms. *Journal of Machine Learning Research*, 7:551–585, 2006.

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

[10] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[11] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. In *VLDB*, 2004.

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

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

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

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

[16] R. Fagin, P. Kolaitis, R. J. Miller, and L. Popa. Data exchange: Semantics and query answering. *Theoretical Computer Science*, 336:89–124, 2005.

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

[18] A. Fuxman and R. J. Miller. First-order query rewriting for inconsistent databases. *J. Comput. Syst. Sci.*, 73(4), 2007.

[19] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB*, 2007. Amended version available as Univ. of Pennsylvania report MS-CIS-07-26.

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

[21] T. J. Green, N. Taylor, G. Karvounarakis, O. Biton, Z. Ives, and V. Tannen. ORCHESTRA: Facilitating collaborative data sharing. In *SIGMOD*, 2007. Demonstration description.

[22] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. XRANK: Ranked keyword search over XML documents. In *SIGMOD*, 2003.

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

[24] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4), 2001.

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

[26] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, 2002.

[27] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Quering the Internet with PIER. In *VLDB*, 2003.

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

[29] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, 2005.

[30] G. Karvounarakis and Z. G. Ives. Bidirectional mappings for data and update exchange. In *WebDB*, 2008.

[31] G. Kasneci, F. M. Suchanek, G. Ifrim, M. Ramanath, and G. Weikum. Naga: Searching and ranking knowledge. In *ICDE*, 2008.

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

[33] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *TODS*, 6(2), 1981.

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

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

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

[37] D. Narayanan, A. Donnelly, R. Mortier, and A. Rowstron. Delay aware querying with Seaweed. In *VLDB*, 2006.

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

[39] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware*, pages 329–350, Nov. 2001.

[40] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to create data-integrating queries. In *VLDB*, 2008.

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