18 Integration in Support of Collaboration

Up to this point in the book, we have focused on data integration scenarios where data creation and editing (at the sources) and data *usage* (via querying) are separate: those who query the data through the integration system are generally not able to *create* and update data. Under such scenarios, the users are primarily concerned with being able to pose queries, get results, and act upon those results. In the business world, the users might be account representatives who contact clients; in the scientific realm, the users might be biologists who need to consult protein and gene databases to conduct an experiment.

Of course, Web 2.0 applications, particularly Wikipedia, have shown an entirely different class of usage scenario, in which the users are themselves the primary data creators, curators, and maintainers. In fact, such scenarios are also commonplace in more focused settings, such as scientific research communities — where scientists publish data, import data from one another as well as large organizational repositories, and make revisions to this data. Under these *collaborative* scenarios, we must consider a variety of issues beyond those in traditional data integration, ranging from handling of updates and annotations to handling conflicting data and different user perspectives.

18.1 What Makes Collaboration Different

We begin with an overview of the major challenges encountered when supporting collaboration. Many of these are not exclusive to the collaborative setting and exist to a limited extent in other contexts — but they must be dealt with in a much more systematic fashion in collaborative settings, because of the greater diversity and dynamicity in the system.

EMPHASIS ON DATA CREATION/EDITING

Often the goal of a collaboration is to allow different users to jointly add and edit data, possibly in response to what they see in their existing query results. Sometimes the data to be shared simply come from tables or XML documents contributed by a single user, but in many cases the users need to be able to edit one another's contributed data. Ultimately, this property leads to issues such as concurrency and conflict management (different users may make incompatible updates to the same data), incremental change propagation and even data provenance (a topic discussed in Chapter 14).

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00018-1

REPAIR VIA USER FEEDBACK
Individual actions from a particular user may be subject to error or even malicious behavior. Many collaborative tools, borrowing from the ideas underlying popular services such as Wikipedia, support correction and undo mechanisms for edits.

NEED FOR DATA ANNOTATION
In sharing data, collaborators will often want to make comments or *annotations*, attached to particular subsets of the data. Such annotations can be arbitrarily complex, depending on the situation, including attachments, discussion threads, and provenance. They are not direct modifications to the database data, but rather additional metadata to be overlaid.

DIFFERENT INTERPRETATIONS, BELIEFS, AND VALUE ASSESSMENTS
In a collaborative setting, different users may have different beliefs about the data domain, different standards for believing or disbelieving a claim, and different hypotheses about observed phenomena. For instance, in science there are typically many competing theories. Likewise, in intelligence or business forecasting there may be multiple perspectives or predictions. A system for facilitating collaboration must take this into account, for instance, by assigning relevance or trustworthiness scores to data based on user perspective or authority, or by managing conflicts in a standardized way.

A variety of approaches have been attempted to address these issues. We divide our discussion of these topics into three broad categories. In Section 18.2, we focus on systems where the data are primarily obtained through automated Web crawling or from external sources but are repaired via user feedback or input. Next, in Section 18.3, we consider situations where users primarily add curation, in the form of annotations and discussions. Finally, Section 18.4 describes the handling and propagation of user updates to the data.

18.2 Processing Corrections and Feedback

There are a variety of settings in which information is widely available across the Internet, from sources of differing degrees of quality or ease of extraction, and users would like to see a bird's-eye view of the data. Of particular note are the following:

- **Topic-specific portals**, where data are automatically extracted from different sources and used to generate pages, such as a portal on the database research community or a portal that provides access to thousands of job search sites (see Sections 15.2.1 and 15.3 for techniques used in generating these portals).
- **Local search services** provided by various search engines such as Google, Yahoo!, and Bing, which in fact are looking increasingly like topic-specific portals in the sense that they present restaurants, businesses, tourist attractions, and other items that are oriented around a specific locale. Moreover, the search providers increasingly are trying to take semantics into account for local search.

- **Bioinformatics or medical informatics repositories**, which often incorporate data from a variety of public sources and provide a Web interface around a query system.

In all of these contexts, some of the data shown to the user population is likely to be incorrect — whether because the source data included errors (as is often the case with life sciences data), because the data were automatically extracted through an imperfect process, or because the data were combined through mappings or joins that were incorrectly formulated. We describe two different models for making corrections to such data. In both of these models, we assume that updates can only be made by individuals with authority to do so, or that there is another module that verifies the truth or popularity of an update before applying it.

18.2.1 Direct User Updates Propagated Downstream

One approach that is commonly used in topic-specific portals generalizes the direct-edit model used in Wikis. At selected stages in the data gathering and integration pipeline, e.g., after various information extraction routines are run, views are presented to the end user. These views are predefined by an administrator and limited to selection and projection operations, and these views are constrained to include any key attributes in the base relations. This set of restrictions ensures that the views are updatable.

Now the user may make direct updates to the content of the views. These updates will then get materialized and also propagated "downstream" to any derived data products. The source data remain unchanged, but relevant updates will continue to be applied even as the source data get refreshed.

This basic approach allows for "persistent" user-provided updates, whenever the administrator chooses to enable such a capability. However, the updates are not seen by the original data provider or the author of the information extractor.

18.2.2 Feedback or Updates Propagated Back Upstream

Naturally, an alternative approach is to try to make revisions to the actual source data — or the formulation of the source queries — in response to user feedback. If the user feedback comes in the form of updates to data values, then one common approach is to support *view update* operations, where an update to a query's output is transformed to a series of operations over the view's data sources. The view update problem has been studied extensively since the 1980s, with the challenge being that updates might cause side effects. In essence a side effect refers to a situation in which, if we are to accomplish the desired change to a view's output, we must change at least one of the source tuples. Yet if the view is recomputed in response to this change, additional tuples will be changed in the output (not in accordance with the user's feedback). For example, tuples that are created as a result of joining the modified tuple may also be affected. Early work explored the constraints under which views are always updatable without unpredictable side effects. More recently, data provenance (Section 14) was shown to be effective in determining whether

a specific update (even to a generally nonupdatable view) will actually cause side effects. This enables a "best effort" model of update propagation.

In some cases, a user deletion should not be interpreted merely as the desire to remove a source tuple but, rather, as feedback that a specific set of results should never have been generated. For instance, when automatic schema alignments or extraction rules are used to produce output, a bad alignment or rule will result in invalid results. Here the system should learn not to use the specific alignment or rule that produced the output result. This can be accomplished as follows. Suppose each rule or alignment is given a score or cost, and query results are each given scores composed from these underlying values. Given feedback on the query results, it is possible to use online learning algorithms to adjust the base scores, ultimately moving them beyond some threshold so they do not appear in the output.

18.3 Collaborative Annotation and Presentation

There are a variety of settings where the assembly of content is done in a highly distributed way, possibly even over time. This is especially true in the *pay-as-you-go* information integration context, where the system starts with very limited ability to extract structured information from the sources, find mappings to semantically related sources, and query the data.

Perhaps over time, multiple users extract and refine data from certain sources. Others may collect data or suggest sources. Still others may provide comments about specific data. In this section we consider scenarios supporting pay-as-you-go integration through collaborative annotations, including annotations that define mappings as well as those that are simply comments. Other aspects of pay-as-you-go and lightweight data integration are discussed in Chapter 15.

18.3.1 Mappings as Annotations: Trails

Within any collection of data, some specific data items are likely to be of use to many users, even if those users have different information needs. Intuitively, if one user defines a way of extracting or mapping such a data item into their query, it would be nice to be able to "capture" this, such that a future user might be able to make use of the work done by the first user. This is the intuition behind *trails*.

A trail is a Global-as-View mapping over XML data in an extended dialect of XPath that supports both keywords and path expressions. One can use it to map a keyword search to a path query and vice versa, or to map from one path to another. Trails might be created by using mining techniques or by looking at how users are mapping data items through a graphical tool.

We illustrate trails with a few examples, rather than presenting a full language syntax. Recall that basic XPath is discussed in Section 11.3.2.

Example 18.1

Consider an repository of photos that is either on the Web or part of a personal collection and assume that the metadata about the photos are stored as XML elements. Suppose we want to query for all data about digital photos received yesterday. The following trails might be useful.

First, the keyword "photo" might refer to any filename ending with ".jpeg":

$$photo \rightarrow //*.jpeg$$

Second, the keyword "yesterday" should be macro-expanded into a query over the date property and the yesterday() function:

$$yesterday \rightarrow date = yesterday()$$

Finally, we might declare that the property "date" should also be expanded to mean the "modified" or "received" properties. In trails, the attributes or properties associated with an XML element can be specifically requested using the *.tuple.*{*attribute*} syntax. Hence we can enter the following:

$$//*.tuple.date \rightarrow //*.tuple.received$$

which means that we will macro-expand the "date" property into a "received" property.

Of course, multiple trails may match a given query, and in general each must be expanded to produce candidate answers. In any practical setting, trails may be given different scores, and we may wish to return the top-scoring answers rather than all answers.

Assuming that users are willing to share their query or mapping operations (i.e., they do not have any expectation of confidentiality), trails are a promising mechanism for pay-as-you-go integration, in that they let future users benefit from the work of past users.

18.3.2 Comments and Discussions as Annotations

Sometimes, the task of integrating data involves contributions by different data owners and users. It may be that the data need to be understood and assessed by multiple people in order to be integrated, or certain data items need to be corrected along the way. One of the necessary capabilities in such scenarios is the ability to annotate data — attributes, tuples, trees — with comments and possibly even discussion threads. Such annotations would be combined with provenance information (as described in Chapter 14) to help the collaborators with their tasks. We briefly discuss two different contexts in which these capabilities have been offered.

WEB END USER INFORMATION SHARING

One setting in which annotations are employed is that of supporting end user data integration and sharing on the Web. Here, the system provides an intuitive AJAX-based browser interface that enables users to import and merge multiple spreadsheets or tables on the Web, through joins and unions. Within this interface, a user may highlight a specific cell, row, or column and add a discussion thread to that item. The annotation will show up as a highlight in the Web interface and includes the time of the post as well as the user.

In the Web context, discussion items are not propagated to derived (or base) data, rather, annotations are kept local to the view in which they were defined. The rationale for this is that other versions of the data may be for very different purposes; hence it might be undesirable to see the comments. In essence, annotations are associated with the *combination* of the view and the data.

SCIENTIFIC ANNOTATION AND CURATION

In the scientific data sharing setting, annotations are generally closely associated with the data, and the view simply becomes a means of identifying the data. Scientific data management systems thus follow a more traditional database architecture and include the ability to propagate annotations to derived views in a controlled fashion.

To help a database administrator control annotation propagation from base to derived data, language extensions to SQL, called pSQL, have been developed. In some cases, attributes from multiple source relations might be merged into the same column (e.g., during an equijoin), and there are several options for which annotations are shown.

pSQL has three modes of propagation, specifiable through a new `PROPAGATE` clause: *default*, which only propagates annotations from the attributes specifically named to be output in the query; *default-all*, which propagates annotations from *all* equivalent query formulations; and *custom*, where the user determines which annotations to propagate.

Example 18.2

The two SQL queries

```
SELECT  R.a                    SELECT  S.b
FROM    R, S                   FROM    R, S
WHERE   R.a = S.b              WHERE   R.a = S.b
```

are equivalent in standard semantics. However, according to pSQL they are not, because *R.a* might have different annotations from *S.b*.

In pSQL:

- The `PROPAGATE DEFAULT` clause will propagate only *R.a*'s annotation for the query on the left and *S.b*'s annotation for the query on the right.
- The `PROPAGATE DEFAULT-ALL` clause will copy annotations from both sources.
- If the `SELECT` clause were changed to "`SELECT X`" and a custom clause such as `PROPAGATE R.a TO X` were added, then only *R.a* will be output.

While this syntax is relatively straightforward, pSQL queries must be rewritten into common SQL in a way that brings together all of the potential annotations. This typically requires a union of multiple possible rewritings (each of which retrieves a subset of the annotations).

18.4 Dynamic Data: Collaborative Data Sharing

Increasingly, progress in the sciences, medicine, academia, government, and even business is being facilitated through sharing large structured data resources, such as databases and object repositories. Some common examples include highly curated experimental data, public information such as census or survey data, market forecasts, and health records. A major challenge lies in how to enable and foster the sharing of such information in a collaborative fashion. Most effective data-centric collaborations have a number of key properties that one would like to promote with any technical solution:

1. They generally **benefit all parties**, without imposing undue work or restrictions on anyone. In other words, there is a low barrier to entry and a noticeable payoff.
2. They include parties with **diverse perspectives**, in terms of both how information is modeled or represented and what information is believed to be correct.
3. They may involve **differences of authoritativeness** among contributors. In some sense, this is a generalization of the notions of authoritativeness considered by PageRank and other link analysis mechanisms in search engines — with the extra wrinkle being that a result in a database system may come from multiple joined or unioned sources.
4. They support an **evolving understanding of a dynamic world** and hence include data that change.

In this section we consider one type of data that forms a collaboratively maintained, publicly available resource: scientific data from the field of bioinformatics.

Example Application

In bioinformatics there are a plethora of different databases, each focusing on a different aspect of the field from a unique perspective, e.g., organisms, genes, proteins, and diseases. Associations exist between the different databases' data, such as links between genes and proteins, or gene homologs between species. Multiple standardization efforts have resulted in large data warehouses, each of which seeks to be the definitive portal for a particular bioinformatics subcommunity. Each such warehouse provides three services to its community:

1. A data representation, in the form of a schema and query interface with terminology matched to the community.
2. Access to data, in the form of both raw measurements and statistically or heuristically derived diagnoses and links, 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 have possibly been imported from elsewhere.

There are occasionally disputes about which data are correct among the different warehouses. Yet some of the databases import data from one another (typically using custom scripts), and each warehouse is constantly updated, with corrections and new data typically published on a weekly, monthly, or on-demand basis.

Observe that the usage model here is update-centric and requires support for multiple schemas and multiple data versions, across different participating organizations. This is quite different from most of the data integration scenarios described in this book, even going beyond peer data management systems (Chapter 17), in that updates and data conflicts need to be managed. That has motivated the development of the *collaborative data sharing system* (CDSS), which builds upon data integration techniques to provide a principled semantics for exchanging data and updates among autonomous sites. The CDSS models the exchange of data among sites as *update propagation among peers*, which is subject to transformation (through schema mappings), content filtering (based on policies about source authority), and local, per-participant 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), in the format that the peer prefers. The participant normally operates in "disconnected" mode for a period of time, querying over the local DBMS and making local modifications. As edits are made to this database, they are logged. This enables users to make modifications without (yet) making them visible to the external world. There are many scenarios where individual participants ultimately want to share their data but need to keep it proprietary for a short time, e.g., to publish a paper on their results or to ensure the data are consistent and stable.

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.

Schema mappings, resembling those of the PDMS (Chapter 17), specify one participant's schema-level relationships to other participants. Schema mappings may be annotated with trust policies that 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 as well as the values of data.

Example 18.3

Figure 18.1 shows a simplified bioinformatics collaborative data sharing system. GUS, the Genomics Unified Schema, contains gene expression, protein, and taxon (organism)

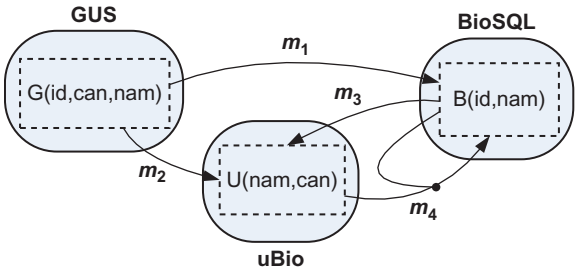

FIGURE 18.1 Example collaborative data sharing system for three bioinformatics sources. For simplicity, we assume one relation at each participant (**GUS, BioSQL, uBio**). Schema mappings are indicated by labeled arcs.

information; BioSQL 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. For example, BioSQL may only trust data from uBio if they were derived from GUS entries. The CDSS facilitates dataflow among these systems, using mappings and policies developed by the independent participants' administrators.

18.4.1 Basic Architecture

We begin with an overview of the CDSS architecture. The CDSS is designed to build over, rather than replace, collaborators' existing DBMS infrastructure. The CDSS runtime sits above an existing DBMS on every participant's machine (peer) P and manages the exchange and permanent storage of updates. It implements a fully peer-to-peer architecture with no central server. 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.

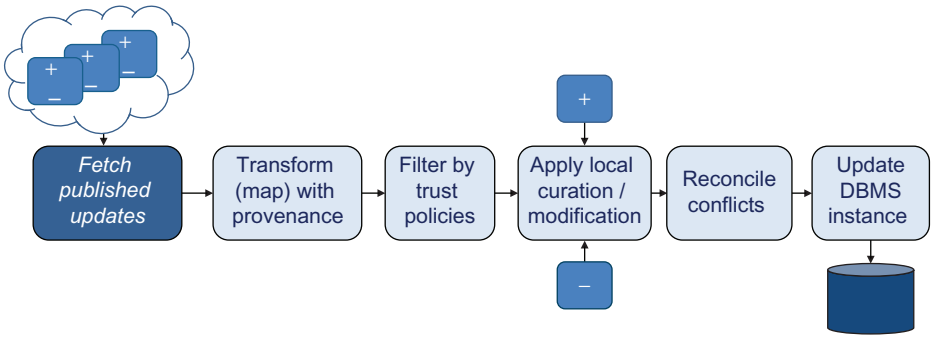

FIGURE 18.2 CDSS stages for importing updates to a peer.

Publishing and Archiving Update Logs

The first stage of sharing updates with other peers in a CDSS is to publish the updated data into a permanent, archived, always-available repository. The full version history of all data is kept in this repository, such that different versions can be compared and changes can be traced or even rolled back. In principle, the storage system could be any sort of highly available server infrastructure or a cloud storage system such as Amazon's SimpleDB. Some implementations even use reliable peer-to-peer storage infrastructure, such that the data are partitioned and replicated among the participants' own servers. That involves fairly complex engineering, and we refer the interested reader to the bibliographic references for more details.

Transforming and Filtering Updates

The most intricate aspects of the CDSS model revolve around how updates are processed, filtered, made consistent, and applied to a given participant's database instance. Figure 18.2 shows the basic data processing "pipeline" from the perspective of a given peer. Initially, all updates published by other peers, but not yet seen by the peer, are fetched from the aforementioned update log storage system. Next, update exchange (Section 18.4.2) is performed, consisting of two aspects: transforming or mapping the updates using schema mappings while recording the mapping steps as data provenance (Chapter 14), and 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, which may have data dependencies. The reconciliation process (Section 18.4.3) arbitrates among the possible updates and determines a consistent set to apply to the peer's database instance.

18.4.2 Mapping Updates and Materializing Instances

Updates published by the various participants will typically have different schemas and identifiers. Some of these updates may conflict with updates from other peers either due

to concurrent changes or due to disagreement about facts. Moreover, each participant may have some assessment of the authoritativeness of the other participants, and this may vary from one peer to another. 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 18.4.3).

The Update Exchange Process

Logically, the process of translating updates in a CDSS is a generalization of *data exchange* (see Section 10.2). 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) each peer should receive a materialized data instance capable of providing all of the certain answers entailed by the data in the CDSS and the constraints implied by the schema mappings. This is the same query answering semantics supported by the PDMS. To achieve this, we must compute a canonical universal solution, using a process similar to the chase, described in Section 10.2.

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 provide detailed references in the bibliographic notes.

The CDSS uses a set of schema mappings specified as tuple generating dependencies (tgds), as introduced in Section 3.2.5.

Example 18.4

Refer to Figure 18.1. Peers GUS, BioSQL, and uBio have one-relation schemas describing taxon IDs, names, and canonical names: GUS(id,can,nam), BioSQL(id,nam), and uBio(nam,can). Among these peers are mappings:

$$m_1 \quad \mathsf{GUS}(i,c,n) \rightarrow \mathsf{BioSQL}(i,n)$$

$$m_2 \quad \mathsf{GUS}(i,c,n) \rightarrow \mathsf{uBio}(n,c)$$

$$m_3 \quad \mathsf{BioSQL}(i,n) \rightarrow \exists c\, \mathsf{uBio}(n,c)$$

$$m_4 \quad \mathsf{BioSQL}(i,c) \wedge \mathsf{uBio}(n,c) \rightarrow \mathsf{BioSQL}(i,n)$$

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 left-hand side and the right-hand side 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 the CDSS model extends the data exchange paradigm to compute instances. Data exchange typically uses the schema mappings with a procedure called the *chase* 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.

In order to make the process of computing a canonical universal solution feasible using standard DBMS techniques, rather than a custom chase implementation, a relational-model CDSS translates the schema mappings into a program in an extended version of datalog, which includes support for Skolem functions that 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 (Section 2.4.5), with a key difference that we do not drop results that contain Skolem values in the distinguished variables. Instead, we must actually materialize the Skolem values.

Example 18.5

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):

$$\mathsf{BioSQL}(i,n) \text{ :- } \mathsf{GUS}(i,c,n)$$

$$\mathsf{uBio}(n,c) \text{ :- } \mathsf{GUS}(i,c,n)$$

$$\mathsf{uBio}(n,f(i,n)) \text{ :- } \mathsf{BioSQL}(i,n)$$

$$\mathsf{BioSQL}(i,n) \text{ :- } \mathsf{BioSQL}(i,c), \mathsf{uBio}(n,c)$$

This program is recursive (specifically, with respect to BioSQL) and must be run to fixpoint in order to fully compute the canonical universal solutions.

XML

As of the writing of this textbook, no XML implementation of a CDSS exists. However, given that data exchange, query reformulation, and incremental view maintenance techniques have all been shown to extend fairly easily to an XML model using the mapping formalisms of Section 11.5, there is a very natural path to such an implementation.

GENERALIZING 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. The CDSS models the local updates as relations. It takes the update log published by each peer and "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

$$\mathsf{BioSQL}(i,n) \text{ :- } \mathsf{BioSQL}^l(i,n)$$

$$\mathsf{BioSQL}(i,n) \text{ :- } \mathsf{GUS}(i,c,n), \neg\mathsf{BioSQL}^r(i,n)$$

INCREMENTAL UPDATE PROPAGATION
Finally, the CDSS uses *incremental view maintenance* techniques to more efficiently update the materialized instances at each peer. Given the set of updates made by each peer, described as *delta* relations (the sets of tuples inserted and deleted), plus the contents of the existing relations, the system can use the mappings to derive the set of deltas to apply to "downstream" relations. Additionally, it is possible in many circumstances to propagate updates "upstream" along mappings, in a variant of the *view update* problem, changing the source tuples from which the modified tuple was derived.

During incremental update propagation, the CDSS will typically filter the updates being propagated according to how trusted or authoritative they are; this is based in part on data provenance. It turns out that data provenance enables more efficient deletion propagation algorithms as well. Specifically, provenance is used to determine whether view tuples are still derivable when some base tuples have been removed. Provenance can also be used to facilitate more flexible solutions to the view update problem by making it possible to determine whether a modification to a base tuple will trigger side effects.

CYCLIC MAPPINGS
To guarantee termination of the data exchange process, the standard CDSS formulation requires that the set of mappings be weakly acyclic (see Section 10.2). Intuitively, this is because non-weakly-acyclic mappings may continue to introduce new labeled null values with each iteration through the mappings, never terminating. Recent work on CDSSs has investigated a user intervention phase, where the system lets the user decide whether the new labeled null values should be unified with the previous ones, i.e., assigned the same value. This enables computation even with cyclic mappings, at the expense of additional user input and a slightly different semantics.

Data Provenance

In any sort of data integration setting where sources have data of varying quality or represent different perspectives, a major challenge is determining *why* and *how* a tuple exists in a query answer or a materialized data instance. The integrated view hides the details from us. This is where *data provenance* plays an instrumental role in the CDSS.

A CDSS creates, materializes, and incrementally maintains a representation of the provenance semiring formalism described in Chapter 14; specifically, it takes the hypergraph representation and encodes it on disk. (Current implementations use relational tables, though other representations could equally well be used.)

Example 18.6

Consider the mappings from our running example. The provenance of the data in the peers' instances can be captured in the hypergraph representation shown in Figure 18.3. Note that "source" tuples (insertions in the local peers' R^l relations) are highlighted using 3-D nodes and "+" derivations. Labeled nulls are indicated with the $\perp$ character.

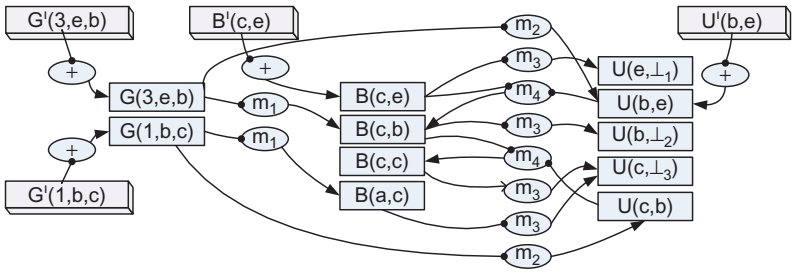

FIGURE 18.3 Provenance graph corresponding to example CDSS setting.

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

An advantage of encoding the provenance information as relations is that it can be incrementally maintained along with the database instances. In essence, we can split every CDSS schema mapping $\mathcal{M}$ into partial mappings: one that takes the left-hand side of $\mathcal{M}$ and computes a provenance relation from it (materializing all variables from the left-hand side) and a second that takes the provenance relation and projects the necessary variables to obtain the right-hand side of $\mathcal{M}$.

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 18.7

As examples, uBio may trust data from GUS (giving it priority 2) more than BioSQL (given priority 1). BioSQL might not trust any data from mapping m_3 with a name starting with "a" (trust priority 0).

Trust policies can be enforced directly during update exchange: as an update is propagated across a mapping, it is assigned a trust priority level for every peer. This priority level takes into account the peer's trust of the source tuples from the mapping, the trust level of the mapping itself, and the trust level of the target of the mapping. These are composed using operators conforming to one of the provenance semirings of Table 14.1 — generally the Boolean **trust** model or the **weight/cost** model.

At this stage, every tuple in every instance will conceptually be given a priority assignment for every peer. A challenge occurs if the tuples conflict with one another, especially if they are grouped into transactions: we must choose which tuples to accept and which to reject, and somehow treat all tuples within the same transaction as an atomic unit.

18.4.3 Reconciling Conflicts

The CDSS reconciliation process ensures that each peer receives a consistent (though perhaps unique) data instance. It considers not only conflicts but also transactions. The CDSS is not really intended for online transaction processing applications. However, transactions might arise when an atomic update gets mapped into a relational representation. For instance, a user of a particular application might update an XML tree, which gets mapped to a set of relation updates, of which all or none should be applied.

The CDSS defines the trust priority of a transaction in terms of its constituent updates. Many different policies are plausible. For instance, we might consider the priority of a transaction to be the minimum priority of its constituent updates or the maximum. The current state of the art considers a transaction to be untrusted if any of its member updates is untrusted (an untrusted update "taints" an entire transaction). Otherwise, it receives the highest trust priority of any contained update (we want to ensure that the most trusted data are applied).

Challenges of Transactions

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).

A common approach is to assign priorities to every transaction; then, in descending order of priority, 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.

Bibliographic Notes

The problems of supporting collaboration in database environments have been considered in a variety of contexts. The grassroots development of community portals is one of the major motivations, and the Cimple system [170, 184, 185] (and its accompanying DBLife portal) developed a variety of techniques for allowing user intervention and repair [113, 169]. This is also one setting considered in "in situ" pay-as-you-go data integration or *dataspaces* [233]. The iTrails system [504] discussed in this chapter was an attempt to construct a dataspace system. The notion of *data coordination*, where multiple users may wish to perform coordinated transactions (e.g., book a flight on the same plane or to the same destination), has been examined in the Youtopia system [278, 279].

A very common thread through many collaborative systems is the use of annotations and provenance. Google's Fusion Tables [260, 261], which is an implementation of an end user Web annotation system, uses both of these mechanisms. DBNotes [136], the first scientific data management system focusing on annotation propagation, and Orchestra [268, 342, 544], a collaborative data sharing system, also use annotations and provenance. Fusion Tables focuses primarily on the user-facing issues, whereas DBNotes focuses on how annotations interface with the query language, and Orchestra focuses on provenance and its interactions with update propagation. Storage schemes related to annotations have been studied in [209].

Provenance leads very naturally to work on two related notions useful in collaborative settings: trust and causality. We often wish to assign a score for how much to believe or trust a data value or a dataset, given knowledge of the trust levels of some of the input data, some of the results, or a combination thereof. A common approach is to use probabilistic models to compose trust, as discussed in Chapter 13, or machine learning techniques to determine the trust levels of different contributing factors, as discussed in Chapter 16. In many cases provenance helps us to distinguish between sources and to determine whether these sources are compatible with our beliefs. The notion of beliefs is much more deeply studied in the BeliefDB project [246], where a notion of "higher-order beliefs" (X believes Y believes something) is considered. Causality refers to finding the input "most responsible" for a given output result, e.g., an erroneous value. Recent work has examined how one might select tuples in a view and determine the input tuples that have the strongest level of causality for those results [425].

Finally, it is worth noting that both annotation and update propagation build heavily upon the ideas of view maintenance, including recursive view maintenance techniques proposed by Gupta, Mumick, and Subrahmanian [275], as well as the view update problem considered by Dayal and Bernstein [166], Keller [345], Bancilhon and Spyratos [53], and many others.