Efficiently Incorporating User Feedback into Information Extraction and Integration Programs

Xiaoyong Chai, Ba-Quy Vuong, AnHai Doan, Jeffrey F. Naughton
University of Wisconsin-Madison

ABSTRACT

Many applications increasingly employ information extraction and integration (IE/II) programs to infer structures from unstructured data. Automatic IE/II are inherently imprecise. Hence such programs often make many IE/II mistakes, and thus can significantly benefit from user feedback. Today, however, there is no good way to automatically provide and process such feedback. When finding an IE/II mistake, users often must alert the developer team (e.g., via email or Web form) about the mistake, and then wait for the team to manually examine the program internals to locate and fix the mistake, a slow, error-prone, and frustrating process.

In this paper we propose a solution for users to directly provide feedback and for IE/II programs to automatically process such feedback. In our solution a developer U uses hlog, a declarative IE/II language, to write an IE/II program P. Next, U writes declarative user feedback rules that specify which parts of P's data (e.g., input, intermediate, or output data) users can edit, and via which user interfaces. Next, the so-augmented program P is executed, then enters a loop of waiting for and incorporating user feedback. Given user feedback F on a data portion of P, we show how to automatically propagate F to the rest of P, and to seamlessly combine F with prior user feedback. We describe the syntax and semantics of hlog, a baseline execution strategy, and then various optimization techniques. Finally, we describe experiments with real-world data that demonstrate the promise of our solution.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems

General Terms

Design, Management

Keywords

information extraction, information integration, user feedback, provenance, incremental execution

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

1. INTRODUCTION

Over the past decade, the topic of extracting and integrating structures from unstructured data (e.g., Web pages, emails, and blogs) has received much attention [20, 18, 33, 16, 28]. Earlier works [33] have focused on developing individual extraction/integration techniques (henceforth *IE/II techniques* for short). Some recent works [16] then consider how to combine such techniques to form larger and more complex IE/II programs, those typically required by real-world applications.

To write such programs, these recent works have proposed several declarative IE/II languages, such as UIMA [18], GATE [11], AQL [32], and xlog [35]. They show that IE/II programs written in such languages are easier to develop, debug, and maintain than those written in procedural languages such as Perl and Java. Many other recent works then examine how to optimize programs written in such languages [26, 35, 32], to execute them effectively over evolving data [7, 8], to make them best-effort [34], to add provenance [3, 25], among others. Such IE/II programs have started to make their way into large-scale real-world applications, in both academic and industrial settings [16].

In these settings, such IE/II programs have proven highly promising, but they still suffer from a glaring limitation: *there is no easy way for human users to provide feedback into the programs.* To understand why user feedback is critical, consider DBLife, a real-world IE/II application that we have maintained for over three years [13, 14]. DBLife regularly crawls a large set of data sources, extracts and integrates information such as researchers' names, publications, and conferences from the crawled Web pages, then exposes the structured information to human users in the form of a structured Web portal. Since automatic IE and II are inherently imprecise, an application like this often contains many inaccurate IE/II results, and indeed DBLife does. For example, a researcher's name may be inaccurately extracted, or the system may incorrectly state that X is chairing conference Y. Being able to flag and correct such mistakes would significantly help improve the quality of the system. And given that at least 5-10 developers work on the system at any time (a reasonable-size team for large-scale IE/II applications), the developer team *alone* can already provide a considerable amount of feedback. Even more feedback can often be solicited from the multitude of users of the system, in a Web 2.0 style.

The problem, however, is that there is no easy way to provide such feedback. The current "modus operandi" is that whenever one of us (developers) finds a mistake (e.g.,

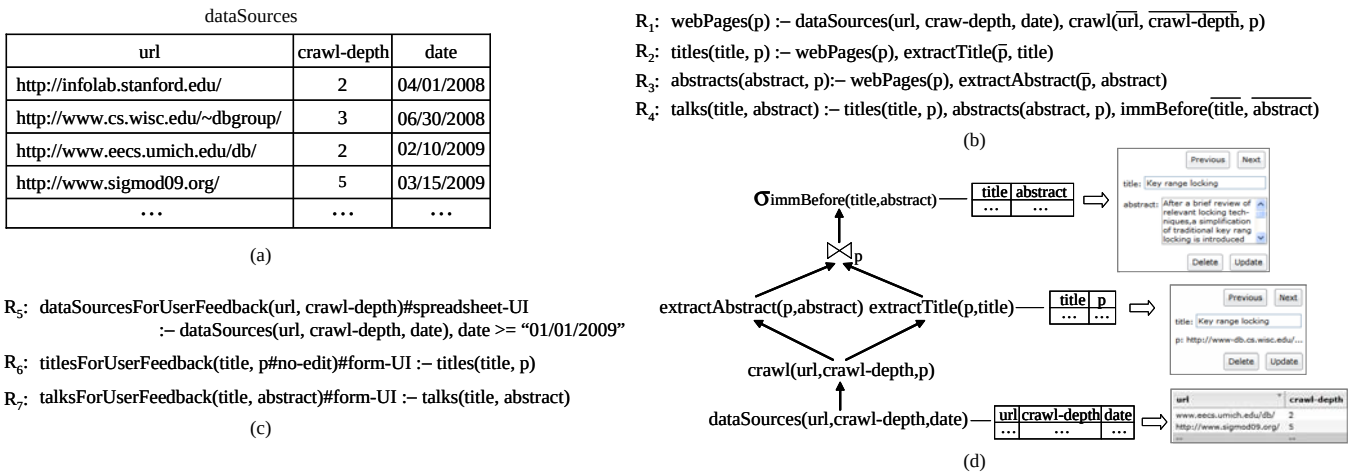

Figure 1: An illustration of our approach: given the set of data sources in (a), a developer U writes the IE/II program P in (b) to extract titles and abstracts of talks from the data sources; next U writes the user feedback rules in (c) to specify which parts of P users can edit and via which UIs; the system then executes P and exposes the specified data portions for users to edit, as shown in (d).

the name "D. Miller R" should actually be just "Miller R"), we email a designated developer. If we receive emails from users about mistakes, we forward them to the designated developer as well. The "victim" developer then delves into the internals of the IE/II program to locate and correct the mistake, and then restarts the program to propagate the correction. Needless to say, this developer soon becomes overwhelmed and resented at having to do all of the mundane work, and this solution obviously does not scale with the amount of user feedback.

A better solution then is to provide *automatic ways* for developers and users alike to provide feedback, like many current applications have done. For example, if a user finds an IE/II mistake, he or she can report it using a form interface, then the system can automatically incorporate the report.

This is also the approach we take in this paper, but our goal is to develop a general and efficient solution. Our basic idea is as follows. After writing an IE/II program P, a developer U writes a set of declarative *user feedback rules* to specify which data portions D of P (e.g., input data, intermediate data, or output data) users can edit[1], via which user interfaces (UIs). Then when executing P for the first time, the system materializes and exposes D via these UIs. Given a user edit, the system updates D, propagates the update to the rest of the program P, then waits for the next user edit. The following tiny example illustrates the above idea.

EXAMPLE 1.1. *Suppose developer U wants to crawl the set of data sources listed in Table dataSources of Figure 1.a to discover research talks. Then U may start by writing a program P in a declarative IE/II language. Figure 1.b shows such a sample program in the* xlog *language (see Section 2.1 for details). Roughly speaking, this program crawls the data sources (each to the specified crawl depth) to obtain a set of Web pages (in relation webPages(p); see Rule R_1). Next, it extracts titles and abstracts from the Web pages (Rules R_2 and R_3, respectively). Finally, it outputs only those (title,abstract) pairs where the title appears immediately before the abstract (Rule R_4).*

Next, U may write a set of user feedback rules, such as Rules R_5-R_7 in Figure 1.c. Rule R_5 creates a view dataSourcesForUserFeedback(url,crawl-depth) from Table dataSources(url,crawl-depth,date), then exposes this view via a spreadsheet UI for users to edit. Note that this view does not allow users to edit data sources added to the system before 1/1/2009 (e.g., because those data sources have been vetted by the developers). Note also that users can easily add new data sources by adding new tuples to the view. Similarly, Rules R_6 and R_7 allow users to edit titles and (title,abstract) pairs, respectively, using a form UI. Here, notation p#no-edit in Rule R_6 states that p (i.e., the Web page in which a title appears) can be inspected but not edited by users.

The system now proceeds to execute program P for the first time. Conceptually, it compiles P into the execution plan in Figure 1.d, then evaluates this plan "bottom up", in a fashion similar to evaluating relational execution plans [35] (see also Section 2.3). During the evaluation, the system also materializes and exposes the views specified by Rules R_5-R_7 via the appropriate UIs, for subsequent user feedback (see Figure 1.d).

After the initial execution, the system then enters a loop of obtaining and incorporating user feedback. For example, after a user has modified the crawl depth of a url in the view dataSourcesForUserFeedback(url,crawl-depth), the system would modify the base table dataSources(url,crawl-depth, date) accordingly, then propagate this modification by re-evaluating the execution plan. It then waits for the next user feedback, and so on.

As described, the above approach provides an automatic way to incorporate user feedback. Users can now provide feedback directly to the system instead of waiting for developers to manually incorporate it. Realizing this approach,

[1]Henceforth we use "users" to refer to both developers and system users.

however, raises many challenges. In this paper, we identify these challenges and provide initial solutions.

We begin by considering how to model IE/II programs and user feedback, and how to incorporate such feedback. To address these issues, we develop hlog, a declarative language for writing "user feedback aware" IE/II programs (such as the one in Figure 1). hlog builds on xlog [35], and thus can be viewed as a Datalog extension, equipped with *declarative user feedback rules*. Incorporating user feedback into hlog programs turned out to be quite tricky. To see why, consider again program P in Figure 1.b. Suppose a user X has deleted a tuple (t, a) from the output table *talks*. Suppose later a user Y inserts a new data source tuple (u, c, d) into the input table *dataSources*. Consequently, we re-execute P to propagate Y's update from table *dataSources* to table *talks*. A straightforward re-execution however will reintroduce the deleted tuple (t, a), "wiping out" the update of X. Furthermore, what if when processing the new data source tuple (u, c, d), program P discovers the same talk (t, a)? Should we keep this tuple, or delete it according to X's feedback? To address these problems, we develop a *provenance-based* solution for interpreting and incorporating user feedback.

After defining the syntax and semantics of hlog, we develop a baseline solution for executing hlog programs. We show how to store and manipulate tuple provenances, as well as user feedback. We discuss in particular the trade-offs between maximizing the amount of user feedback incorporated into a program P and minimizing the execution time of P.

Finally, we develop a set of optimization techniques to speed up the baseline solution for executing hlog programs. First we examine how to execute such programs incrementally, after each user feedback. Incrementally updating relational operators (e.g., σ, π and $\bowtie$) has been studied extensively (see the related work section). Incrementally updating IE/II operators is more difficult, due to their "black-box" nature. To address this problem, we identify a set of incremental properties that the developer can use to characterize IE/II operators. Once the developer has identified such properties, we can automatically construct incremental update versions for these operators.

The second set of optimization techniques that we develop concerns concurrency control. Multiple users may happen to view and update the exposed program data at the same time. To ensure the consistency of the program data, we need to enforce concurrency control (CC). To do that, we could require developers to force all data and IE/II computation into an RDBMS and use its CC capabilities. In practice, however, developers usually choose not to do so (at least not today) for ease-of-development or various performance reasons. Hence we seek to develop CC solutions outside RDBMS. In this paper, we show how to explore the graph structure of an IE/II program to design efficient CC mechanisms.

In summary, we make the following contributions:

- Introduce the problem of allowing users to edit the data in an IE/II program to improve the quality of the program results.
- Develop a declarative language (hlog) with well-defined semantics that allows developers to quickly write IE/II programs with the capabilities of incorporating user feedback.
- Develop a solution to execute programs written in hlog.
- Propose optimization techniques for enhancing the performance of IE/II programs in terms of runtime and concurrency degree.
- Conduct extensive experiments over real-world data that demonstrate the promise of the proposed approach.

2. SYNTAX AND SEMANTICS

In this section we describe the syntax and semantics of hlog, our proposed declarative language to write "user feedback aware" IE/II programs. We first describe xlog, a recently developed Datalog variant for writing declarative IE programs [35], then build on it to describe hlog. For ease of exposition, we will focus on IE programs, deferring the discussion of II aspects to Section 2.4.

2.1 The xlog Language

We now briefly describe xlog (see [35] for more details). Like in traditional Datalog, an xlog program P consists of multiple *rules*. Each rule is of the form $p :- q_1, \cdots, q_n$, where p and q_i are predicates, p is the head of the rule, and q_i's form the body. Each predicate in a rule is associated with a relational table. A predicate is *extensional* if its table is provided to program P, and is *intensional* if its table must be computed using rules in P.

xlog extends Datalog by supporting procedural predicates (*p-predicates*) and functions (*p-functions*), as real-world IE often involves complex text manipulations that are commonly implemented as procedural programs. A p-predicate p is of the form $p(\overline{a_1}, \cdots, \overline{a_n}, b_1, \cdots, b_m)$, where a_i and b_j are variables. Predicate p is associated with a procedure g (e.g., written in Java or Perl) that takes as input a tuple $(u_1, \cdots, u_n)$, where u_i is bound to a_i, $i \in [1, n]$, and produces as output a set of tuples $(u_1, \cdots, u_n, v_1, \cdots, v_m)$. A p-function $f(\overline{a_1}, \cdots, \overline{a_n})$ takes as input a tuple $(u_1, \cdots, u_n)$ and returns a scalar value. The current version of xlog does not yet support recursion nor negation.

EXAMPLE 2.1. *Figure 1.b shows an* xlog *program P with four rules $R_1 - R_4$ that finds talks from a set of data sources. P has one extensional predicate (dataSources), four intensional predicates (webPages, titles, abstracts, and talks), three p-predicates (crawl, extractTitle, and extractAbstract), and one p-function (immBefore).*

The p-predicate crawl($\overline{url}$,$\overline{crawl\text{-}depth}$,p) for example takes as input a url u and a crawl depth d (e.g., 3), crawls u to the specified depth, then returns all tuples (u, d, p) where p is a page found while crawling u. As another example, the p-function immBefore($\overline{title}$,$\overline{abstract}$) returns true only if the input title appears immediately before the input abstract.

The output of an xlog program P is then the relation computed for a designated head predicate, as illustrated in the following example:

EXAMPLE 2.2. *Consider again program P in Figure 1.b with talks being the designated head predicate. Conceptually, for each url in dataSources, P applies the (procedure associated with) crawl predicate to crawl that url to a prespecified depth. This yields a set of Web pages p (see Rule R_1). Next, P applies the extractTitle predicate to each Web page p to extract talk titles (Rule R_2). Similarly, P applies*

extractAbstract to extract talk abstracts (Rule R_3). Finally, P applies immBefore to each pair of title and abstract and outputs only those pairs where immBefore evaluates to true (Rule R_4).

2.2 The hlog Language: Syntax

We now describe hlog. To write an hlog program, a developer U starts by writing a set of IE rules that describes how to perform the desired IE task. These rules form an xlog program P. U then writes a set of user feedback rules, or *UF rules* for short, that describes how to provide feedback to the data of program P.

The data of P falls into three groups, input, intermediate, and output data, as captured in the tables of the extensional, intensional, and head predicates, respectively. Today many IE applications allow editing only the input and output data. We found however that editing certain intermediate data can also be highly beneficial, because correcting an error early can drastically improve IE accuracy "down the road".

Furthermore, the boundary between intermediate and output data is often blurred. Consider for instance an IE program that extracts entities from text, discovers relations among entities, then outputs both entities and relations as the final results. Here entities are both intermediate results (since the program builds on them to discover relations) and final results. Clearly, correcting the entities can significantly improve the subsequent relation discovery process.

Consequently, in hlog we allow users to edit all three groups of data, that is, the extensional, intensional, and head predicates. Suppose U has decided to let users edit such a predicate p. Then U writes a UF rule of the form

$$v\#w :- p, q_1, \cdots, q_n.$$

This rule specifies a view $v :- p, q_1, \cdots, q_n$ over p, so that users can only inspect and edit p's data via the view v. Here each q_i is a built-in predicate "a *op* b", where a is an attribute of p, *op* is a primitive operator (e.g., "=", ">"), and b is an attribute of p or a constant. As such, v is a combination of selections and projections over p. Currently we consider only such views because they are *updatable*: user edits over them can be unambiguously and efficiently translated into edits over p.

Using the notation $v\#w$, the UF rule also specifies that users can edit view v via a user interface (UI) w. We assume that the system has been equipped with a set of UIs (e.g., spreadsheet, form, wiki, and graphical), and that w comes from this set. Formally, we define a user interface w as a pair $\langle f_{out}, f_{in} \rangle$, where f_{out} is a function that renders the data of a view v into the format that w can display, and f_{in} is a function that translates actions users perform on w into operations (queries and updates) over v. Consider the spreadsheet interface for example. Here the f_{out} function converts view data into a spreadsheet file that the interface can read and display. The f_{in} function then translates user actions, such as deleting a row from a spreadsheet, into view updates, such as deleting a tuple from v. We discuss user actions in more details in Section 2.3.2.

EXAMPLE 2.3. *UF rule R_5 in Figure 1.c specifies view dataSourcesForUserFeedback over predicate dataSources. The view allows users to edit only data sources added to the system on or after 1/1/2009 (e.g., possibly because all data sources added earlier have been vetted by the developer), via a spreadsheet UI. UF rule R_6 allows users to edit extracted titles via a form UI. Here p#no-edit means that users can inspect but cannot edit the url p.*

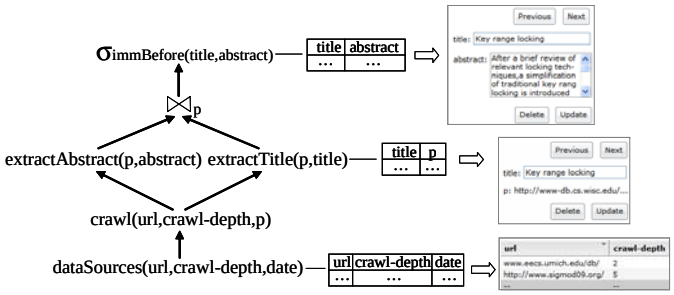

Figure 2: Execution graph of the hlog program in Figures 1.b-c, as reproduced from Figure 1.d.

2.3 The hlog Language: Semantics

In the following, we first describe a baseline semantics that is straightforward but fails to incorporate previous user feedback. Then we describe a better semantics that removes this limitation.

2.3.1 Baseline hlog Semantics

Recall from the introduction that after developer U has written an hlog program P, the system executes P for the first time, materializing and exposing certain data portions D of P for user edits. It then enters a loop of waiting for and processing user feedback. In what follows we describe these steps in details.

Initial Execution: Given an hlog program P, first we compile the xlog portion of P (*i.e.*, the IE rules) into an execution plan G. We omit the details of this compilation; see [35] for a complete description. Since the current version of xlog does not yet support recursion nor negation, the execution plan G can be viewed as a directed acyclic graph (DAG), with "leaf nodes" at the bottom and a "root node" at the top. Figure 2 shows a sample execution plan (reproduced from Figure 1.d) for the xlog program in Figures 1.b. Note that the internal nodes of such a plan are either relational or (procedural) IE operators.

Next, we evaluate the execution plan G in a "bottom up" fashion, starting with the leaf nodes. During the evaluation we materialize and expose certain data portions D of P via certain UIs. Specifically, if the hlog program P contains a UF rule that involves view v over predicate p and UI w, we materialize the data of p and the data of v, and then expose the data of v via UI w. Note that this differs from a traditional xlog (or RDBMS) execution, where intermediate data typically is not materialized (unless for optimization purposes). Here, we must materialize the data of p and v so that later we can incorporate user feedback. Figure 2 shows how the data of three predicates has been materialized and exposed, according to the UF rules R_5–R_7 in Figure 1.c.

Loop of Waiting for and Processing User Feedback: After the initial execution, we enter a loop of waiting for and processing user feedback. Suppose a user performs an update M over a view v (e.g., modifying, inserting, or deleting a tuple; see Section 2.3.2). Then we translate this update M over view v into an update N over the "base" predicate

p. In the next step, we start a *user feedback transaction*, or *transaction* for short. This transaction performs the update N on the (materialized) data of p. Next, it propagates the update "up" the execution graph G. That is, suppose the data from p is part of the input to an operator q of G, then we re-execute q with the newly revised p. Next we re-execute operators that depend on q, and so on. The transaction terminates after we have re-executed the root-node operator. We then wait for the next user transaction, and so on.

It is important to note that we consider propagating a user update only *up* the execution graph. Propagating update *down* the execution graph would require being able to "reverse" the input-output of IE blackboxes (*i.e.*, given an output, compute the input). We believe requiring the IE blackboxes to be "invertible" may be too strong a requirement.

Since multiple users may provide feedback at the same time, we need a way to enforce concurrent execution of the user transactions. To do that, we could require developers to force all data and IE/II computation into an RDBMS and use its CC capabilities. In practice, however, developers usually choose not to do so (at least not today) for ease-of-development or various performance reasons. Hence we seek to develop CC solutions outside RDBMS. For now, we adopt a simple CC solution: a user transaction T will x-lock the whole execution graph at the start (of its execution) and unlock the graph after the finish. Call this solution graph-locking. (See Section 4 for more efficient CC solutions.)

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of user transactions started and finished during the time period $[x, y]$. It is easy to see that the above algorithm guarantees a serial execution of the transactions. Hence, at the end of the time period, the final output of graph G (*i.e.*, the output of the root operation) incorporates all user updates encoded in $\mathcal{T}$, in some serial order. This semantics is well defined. However, its notion of incorporating user updates is severely limited, in that it often clobbers previous user feedback: when an operator p in the execution graph is re-executed, its new output will simply replace the old one; thus any previous user updates over p's output will be lost. In the next subsection, we consider how to address this problem.

2.3.2 Extending Baseline Semantics to Handle Previous User Feedback

Preliminaries: We start with three preliminaries. First, we observe that any user feedback F in our framework can be viewed as feedback over the *output* O of some (relational or IE) operator p in the execution graph G. (This is always true except when F is over a leaf node of G, that is, an extensional table. But we can easily handle this by pretending that each extensional table is the output of some dummy operator.)

Second, to be concrete, we will assume that a user update F (we use "user update" and "user feedback" interchangeably) can be only one of the followings: deleting a tuple $t \in O$, modifying a tuple $t \in O$ to t', or inserting a tuple t into O, where O is the output of an operator p. More complex types of user feedback exist, and our current framework can be extended to deal with many of these. But we will leave an in-depth examination of this issue as future work.

As a final preliminary, we will develop this subsection assuming that operator p is unary, that is, it takes as input a single table I. The notions we will develop can be generalized in a straightforward fashion to the case where p takes as input a set of tables.

Tuple Provenance: We are now ready to consider how to save user updates on O, and then apply them when p is re-executed. A simple solution is to save a user update F as an update *operation* (e.g., insertion, deletion, or modification), and then apply the operation when p is re-executed. This solution, however, may incorporate user updates incorrectly, as the following example illustrates.

EXAMPLE 2.4. *Suppose that given input I, p produces output O, and that a user has deleted an incorrect tuple t from O. Now suppose that we modify input I into I', and that re-executing p given I' produces output O', which consists of a single tuple t. Then using the above solution we should delete t from O'. But what happens if t is indeed the correct output for $p(I')$? In this case we have incorrectly applied an old user update.*

This example suggests that we should interpret a user update on an output tuple based also on the *provenance* of that tuple from the input data. Specifically, suppose p takes as input a single table I and produces an output table O. Then we define the *provenance* of a tuple $t \in O$ to be the set S_t of tuples in I that p uses to produce t.

We require that given any input I, p produces not just the output O, but also the provenances of all tuples in O, such that each tuple t has a unique provenance (*i.e.*, a set S_t in input I). If p is a relational operator, then it is relatively easy to modify p to do so. If p is an IE operator, then it is more difficult, but often do-able, especially for the creator of p. In the worst-case scenario, we can take the whole input table to be the provenance for each tuple in the output.

We further require that each operator p is *monotonic*, in the sense that if $I \subseteq J$ then $p(I) \subseteq p(J)$. Most IE operators satisfy this requirement, and this requirement gives us a well-defined interpretation of user update, as we will see below.

Interpreting and Incorporating User Updates: Having defined provenance, we can now interpret user update as follows. Suppose a user deletes a tuple t from output O of operator p. Let $S_t \subseteq I$ be the provenance of t, and M be $p(S_t)$, the result of applying p to S_t.

Let M' be the result obtained after deleting tuple t from M. Then we assume that by deleting the tuple t from output O, the user means to state that $p(S_t)$, the output of applying p to S_t, should really be M', not M.

Under this interpretation, the above user update can be (conceptually) saved as a tuple (S_t, M'), and we can incorporate this update in case p is re-executed as follows. Suppose p is re-executed over a new input I', and produces output O'. Then we check to see if S_t appears in I'. If yes, we remove from O' all tuples whose provenance is S_t (since p is monotonic, $p(S_t)$ will appear in O'), then add to O' all tuples in M'.

We now discuss how to handle other types of user updates. Suppose a user modifies a tuple t in the output O into t', then we can interpret, save, and incorporate this modification update in a similar fashion. Suppose a user inserts a tuple t, then we ask the user for the provenance of t. Given the provenance, we can again interpret, save, and incorporate the insertion in the same manner. If the user does not give the provenance of t, then we create a special provenance

S^*, and use it as the provenance of t. We assume that S^* appears in any input set I.

The New hlog Semantics: We are now ready to reconsider hlog semantics. In this new semantics, we redefine the notion of a user feedback transaction. Here, given an update (S_t, M') on an output tuple t of an operator, a transaction T first incorporates the update (see the preceding two paragraphs), and then propagates it up the execution graph, exactly as in Section 2.3.1.

However, before propagating the update up the graph, T saves the update (S_t, M') as a tuple for operator p so that if a subsequent transaction T' re-executes p, T' can incorporate this update. Over time, many updates may be saved for p, and they should be saved in their arrival order. When transaction T' incorporates these updates, it should incorporate them in that order.

As for transaction T itself, whenever it re-executes an operator q (that depends on p in the execution graph), T checks to see if q has any saved updates, and then incorporates these updates in their arrival order.

Again, we assume that each user transaction T will x-lock the whole execution graph at the start (of its execution) and unlock the graph after the finish.

Let $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$ be a set of user transactions, as defined above, during a time period $[x, y]$. It is easy to see that the above graph-locking algorithm guarantees a serial execution of the transactions. Hence, at the end of the time period, the final output of graph G (*i.e.*, the output of the root operation) incorporates all user updates encoded in $\mathcal{T}$, in some serial order. Here, the notion of incorporating user updates is more expressive than that in Section 2.3.1, in the sense that a previous user update is clobbered only if there exists a new user update on the same tuple with the same provenance.

In the rest of the paper, we will use the above conceptual algorithm to express the semantics of hlog.

2.4 Extending hlog to Handle II

So far we have introduced hlog as a language for writing IE programs. Since real-world applications often involve II activities, such as schema matching and de-duplication, developer U also needs a language to write II programs. Furthermore, since II is often semantics-based, and automatic II techniques are error prone, U also wants to leverage user feedback to improve the quality of II results.

Consequently, we have extended hlog to support II operations, in the same way that we support IE operations. Specifically, to create an II operator p, developer U first models it as a p-predicate or a p-function. Then U implements a procedure g (e.g., in Perl) that carries out the II activity, and associates p with g. To enable users to provide feedback on p's results, U writes a user feedback rule in a way similar to those written for IE predicates. We are currently exploring the power and limitation of this hlog extension in handling real-world II applications.

3. EXECUTING HLOG PROGRAMS

We now describe a baseline solution to execute hlog programs, according to the user update semantics discussed in Section 2.3.2.

Let P be an hlog program. In Section 2.3.2, we already discuss a high-level algorithm to execute P, including the initial execution and the subsequent loop of waiting for and processing user feedback. In this section, we focus on two most difficult steps of this algorithm: how to store provenance and how to exploit it to incorporate user feedback. The remaining steps are relatively straightforward.

PO

tid	rid
1	1
2	2
3	2

PR

rid	sid1	sid2
1	1	1
2	1	2

PS_1

sid	tid
1	1
1	3

PS_2

sid	tid
1	4
2	5

Figure 3: Example provenance tables for a case where operator p takes as input two tables and produces as output a table with three tuples.

3.1 Storing Provenance Data

Let p be an operator that takes as input two tables I_1 and I_2 and produces as output a table O. Recall from Section 2.3.2 that the provenance R of each tuple $t \in O$ is then a pair (S_1, S_2), where $S_1 \subseteq I_1$ and $S_2 \subseteq I_2$. That is, p uses the tuples in S_1 and S_2 to produce tuple t. Our goal is to store the provenances of all tuples in O.

Assume that all tuples in I_1, I_2, and O come with unique IDs (it is relatively easy to modify p to do so). Then we can store the above provenances in four tables, as illustrated in Figure 3. Table PO stores for each tuple in O the ID of its provenance R. Thus the first row of PO states that output tuple with tid=1 in O has a provenance with rid=1.

Table PR then stores for each provenance the IDs of its component sets, one for each input table. The first row of PR states that provenance with rid=1 consists of sets with $sid1$=1 and $sid2$=1. Table PS_1 then stores for each component set all of its component tuples. The first row of PS_1 for example states that the set with $sid1$=1 consists of the two tuples with tids 1 and 3 in the input I_1.

In the case that p takes a single input table, or more than two input tables, we can store its provenances in a similar fashion. For each operator p whose output is subject to user feedback, we require p to output provenance tables as described above, after each execution. Whenever a user updates the output of p, the provenance tables must also be updated, to reflect the user update. Such updating is relatively straightforward, and we will not describe it further, for space reasons.

3.2 Storing and Incorporating User Feedback

Consider an operator p that takes as input a single table. We now use p to describe how we store and incorporate user feedback. (The algorithm below generalizes in a straightforward fashion to the case of multiple input tables.)

Suppose when executed for the first time, p takes input I_1 and produces output O_1, as well as provenance tables PO_1, PR_1, and PS_1, as illustrated in the left part of Figure 4.

Now suppose a user updates a tuple $t \in O_1$, with the provenance rid_1. Then we can just store this update as rid_1 (after we have updated output table O_1 and the provenance tables appropriately). Recall from Section 2.3.2 that conceptually we should store a user update as (S_t, M'), specifying that the value of $p(S_t)$ is M'. Here, S_t is represented by rid_1, and M' is captured inside O_1, that is, M' consists of exactly those tuples in O_1 that have provenance rid_1. Thus

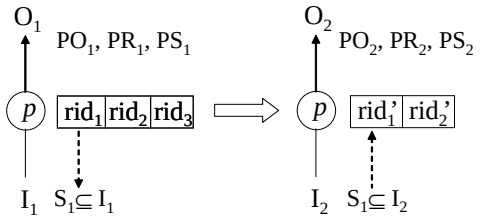

Figure 4: An example of incorporating user feedback.

as long as we store rid_1 and O_1, we have effectively stored (S_t, M').

Suppose that two more user updates come in on the output of p, and that we further store these updates as rid_2 and rid_3 (see Figure 4). In general, then, the set of user updates stored at p is a set of provenance IDs (together with the latest output table of p). These provenance IDs are unique. That is, if we have two updates with the same provenance, then the latter will override the former, and so we only need to store each provenance ID once.

Now suppose a transaction T re-executes p with input I_2, and produces output O_2, together with provenance tables PO_2, PR_2, and PS_2 (see the right part of Figure 4). Transaction T must then incorporate user updates rid_1, rid_2, and rid_3 into the output of p.

Consider the first update rid_1 (we can process these updates in any order; the end results will be the same). To incorporate this update, we first use the provenance tables PR_1 and PS_1 to find the provenance $S_1 \subseteq I_1$ (see Figure 4). Next, we check to see if S_1 is also a provenance in I_2. If so, then we say that update rid_1 is applicable to I_2.

In this case, suppose the provenance ID of S_1 in I_2 is rid_1' (which can be different from rid_1, see Figure 4). Then to incorporate update rid_1, we remove from O_2 all tuples with provenance rid_1', then copy into O_2 all tuples of O_1 with provenance rid_1. Note that if update rid_1 conceptually specifies that $p(S_t)$ must be M' (as discussed earlier), then copying from O_1 to O_2 in effect sets the value of $p(S_t)$ in O_2 to be M'.

If S_1 is not a provenance in I_2, then we say that the update rid_1 is not applicable to I_2, and we ignore this update. We proceed in a similar fashion with updates rid_2 and rid_3.

After we have processed all of these updates, we revise the set of updates stored at operator p. To do so, we keep all updates that are applicable, but revise their rids, and drop all inapplicable updates. For example, since update rid_1 is applicable, we keep it, and revise its rid to rid_1' (since we now store I_2 and O_2, no longer I_1 and O_1, where rid is a valid rid). Suppose update rid_2 is applicable, then we do the same, and keep the new rid rid_2'. Suppose update rid_3 is not applicable, then we will drop it from the set of updates maintained at operator p (see Figure 4). In theory, we can keep the inapplicable updates around, in case later they become applicable. Doing so, however, would require us to store extra input and output tuples (*i.e.*, the S_t and M') for each inapplicable update. This may take up a significant amount of space over time. Hence, for now we choose not to keep the inapplicable updates around.

We have thus described an algorithm to process user updates. Only one minor issue remains. Earlier we state that given a provenance S_1 in I_1, we must check to see if it is also a provenance in I_2. Checking this by comparing the contents of the tuples is often expensive. We can speed up this checking using a variety of methods, including incremental ID maintenance, which we will discuss in Section 4.2.

4. OPTIMIZING HLOG EXECUTION

We now describe several optimization techniques to speed up program execution. First we describe how to incrementally execute an IE/II operator. Then we describe two concurrency control methods that exploit the graph structure of an IE/II program to achieve higher degrees of concurrency than the whole-graph-locking method in Section 2.3.

4.1 Incremental Execution

In the baseline solution, we execute an operator p from scratch each time. This is inefficient when only a small amount of p's input has been changed, and most of p's output remains the same. In this section, we describe how to execute an IE/II operator efficiently by incrementally updating its output. The basic idea is to exploit certain properties of the operator with respect to incremental update.

Incrementally updating the output of an operator after its input has been changed is not a new problem. In the relational setting, view maintenance [2, 23] considers a similar problem, where we would like to incrementally update a materialized view after its input relations have been changed. As one solution, we can use the distributive property [2] of the basic relational operators (*i.e.*, σ, π and $\bowtie$) to derive incremental updates to the view.

If we view an IE/II operator p as a view definition, then the input of p is the set of base relations in the view definition, and the output of p is the materialized view. However, unlike relational operators whose semantics are well-understood, the operator p in general is a blackbox, and the distributive property may not hold on p.

To incrementally execute an IE/II operator, we extend the ideas from relational view maintenance. Instead of using a single property to describe all operators, we introduce a set of fairly general properties. Although an operator can be a blackbox, we can make it less "black" by allowing the developer to specify which properties hold for the operator. Given these properties, we can then update the output of the operator incrementally.

In the following, we first present five such properties. Then we provide an algorithm that exploits these properties to construct incremental versions of an operator automatically.

Incremental-Update Properties: Let p be an operator which takes n tables $I_1, \cdots, I_n$ as input, and outputs table O. Each property below captures some incremental relationship between an input table $I_i \in \{I_1, \cdots, I_n\}$ and output O. For conciseness, we use $R_i(S)$ to denote $I_1, \cdots, I_n$ with I_i replaced by S. That is, $R_i(S) = I_1, \cdots, I_{i-1}, S, I_{i+1}, \cdots, I_n$.

Definition 1 (Closed-Form Insertion). *An operator p is closed-form insertable w.r.t. an input table I_i if and only if there exists a function f such that for any set ΔI of tuples to be inserted into I_i, the condition $p(R_i(I_i \cup \Delta I)) = f(R_i(\Delta I), O)$ holds.*

Definition 2 (Closed-Form Deletion). *An operator p is closed-form deletable w.r.t. an input table I_i if and only if there exists a function f such that for any set ΔI of tuples to be deleted from I_i, the condition $p(R_i(I_i - \Delta I)) = f(R_i(\Delta I), O)$ holds.*

The two closed-form properties state that instead of executing p over the updated I_i (together with other input tables) from scratch, we can execute the function f that examines ΔI, which is often much smaller than I_i, to compute the new output. The function f may invoke p on $R_i(\Delta I)$. For example, consider the *crawl* operator in Figure 1.b, which crawls the set of data sources in table *dataSources* to find Web pages. If we add new data source tuples into table *dataSources*, we can obtain the new output by executing the function f that first crawls the new sources and then inserts the crawled Web pages into the current output.

To define the next property, we first define the notion of partitioning function. We say a function f is a *partitioning function w.r.t. a table I* if and only if f partitions I into k disjoint subsets $\{S_1, \cdots, S_k\}$, where $k > 0$, and $I = \bigcup_{i=1}^{k} S_i$. Denote the application of f to I as $f(I) = \{S_1, \cdots, S_k\}$.

Definition 3 (Input Partitionability). *An operator p is input partitionable w.r.t. an input table I_i if and only if there exists a partitioning function f on I_i such that*

- $p(I_1, \cdots, I_n) = \bigcup_{S_k \in f(I_i)} p(R_i(S_k))$, *and*
- $f(I_i)$ *is a non-trivial partitioning, in that at least one subset S_j, $j \in [1, k]$, is a proper subset of I_i.*

Intuitively, a partitioning function f partitions an input table into disjoint sub-tables. The input partitionability property implies that the output of an operator on one sub-table is independent of those of the other sub-tables. That is, if the input table is changed, we can update the output by first identifying all the changed sub-tables in the input, applying the operator to them, and then combining the outputs of these sub-tables with the outputs of those unchanged sub-tables. For example, consider the operator *extractTitle* in Rule R_2 of Figure 1.b, which extracts titles from Web pages. A simple partitioning function on its input table *webPages* is to partition the table tuple by tuple. As a result, the output of the operator over the entire input table is the union of the titles extracted from each page. Note that *extractTitle* also has the two closed-form properties above.

As another example, consider a simple IE/II program that discovers people entities from a set of Web pages. First, the program takes each seed name (e.g., "David Smith") from a dictionary, and generates name variants (e.g., "D. Smith" and "Smith, D."). It then finds the mentions of these variants in the Web pages. After that, it executes an II operator *getPeopleEntities* that groups the obtained mentions by their seed names, and then outputs an entity for each group. The operator exhibits the input partitionability property where the partitioning function partitions the input mentions by their seed names. If new mentions are inserted into the input of the operator, we can incrementally update its output by executing the operator over the partitions that the new mentions belong to, and then unioning the outputs with those of the other partitions. However, unlike in the previous example, this operator does not have the closed-form properties.

Definition 4 (Partition Correlation). *An operator p is partition correlated w.r.t. an input table I_i if and only if there exists a function f such that for an arbitrary partition $\langle S_1, S_2 \rangle$ of I_i where $I_i = S_1 \cup S_2$, the condition $p(I_1, \cdots, I_n) = p(R_i(S_1)) \cup p(R_i(S_2)) \cup f(R_i(S_1), R_i(S_2))$ holds.*

Unlike the input partitionability property where the output of an operator comprises the output of the operator over each input partition, the partition correlation property captures the cases where the output also contains results obtained from both partitions. Take a data matching operator p for example. Suppose that p takes a single table as input. For each pair of input tuples, p outputs their tuple IDs, together with a score measuring the similarity of the two tuples. Suppose a few more tuples are now inserted into the input table. For certain types of data matching operator p, we can incrementally update the output as follows. Let the old input table be partition S_1 and the set of inserted tuples be S_2. First, we execute p over S_2 to compute $p(R_1(S_2)) = p(S_2)$. Next, we apply a function f to S_1 and S_2. The function computes the similarity score of each pair of tuples, one from each partition. We then union $p(S_1)$, which is the old output O, with $p(S_2)$ and $f(S_1, S_2)$ to get the new output.

Definition 5 (Attribute Independence). *Let A be a proper subset of attributes in an input table I_i, and $\bar{A}$ be the rest of the attributes in I_i. An operator p is independent of $\bar{A}$ in I_i if and only if given any two instances S_1 and S_2 of I_i, $\pi_A(S_1) = \pi_A(S_2) \Rightarrow p(R_i(S_1)) = p(R_i(S_2))$.*

Some operators evaluate only a subset of attribute values of an input tuple. In this case, changes to the unused attributes have no effect on the output of such operators. The attribute independence property captures this case.

The incremental properties above are operational. Each property implies its own way of incrementally updating the output of an operator. Thus, to incrementally execute an operator p, we need to know which properties apply to p and for each property, the specific function f that realizes it (e.g., the function f that satisfies the condition $p(R_i(I_i \cup \Delta I)) = f(R_i(\Delta I), O)$ in Definition 1). Given these, we can construct incremental versions of p accordingly.

Property Specification: To allow the developer to specify the incremental-update properties of an operator p, we extend hlog by adding a language construct "$p(I_i)$:{(*type*,f)}". The construct lists a set of properties that p has with respect to its input table I_i. Each property is specified by a pair $(type, f)$, where *type* is the type name of the property, and f is the function that the developer needs to provide to instantiate the property (See Definitions 1-5). Here, *type* can take values from $\{ci, cd, ip, pc, ai\}$. The values in the set stand for closed-form insertion (ci), closed-form deletion (cd), input partitionability (ip), partition correlation (pc), and attribute independence (ai). The function f has different forms depending on the type of the property. For example, if $type = ci$, then f takes as input (1) a set of tuples inserted into an input table I_i, (2) all the other input tables, and (3) the old output table, and produces a new output table. If $type = ai$, then f is the function that projects out the set of attributes that are irrelevant to the operation of p (See Definition 5).

Figure 5 shows an example of property specifications for predicates *crawl*, *extractTitle*, and *extractAbstract* in Figure 1.b. In the figure, P_1 specifies two closed-form properties for predicate *crawl*. (The two properties must be specified in pairs. That is, if the developer U specifies the closed-form insertion property for an operator p, U must also specify the closed-form deletion property for p.) The closed-form insertion property, for example, is obtained from the observation that when new tuples are added to *dataSources*, we can update table *webPages* by first crawling pages from these new

```
# f1: crawls pages from the new data sources, and inserts them into webPages
# f2: deletes from webPages the pages crawled from the deleted data sources
P1:   crawl(dataSources) : {(ci, f1), (cd, f2)}
# f3: partitions webPages tuple by tuple
P2:   extractTitle(webPages) : {(ip, f3)}
P3:   extractAbstract(webPages) : {(ip, f3)}
```

Figure 5: An example of incremental-update property specification.

sources and then adding them to $webPages$. Furthermore, P_2 and P_3 specify input partitionability properties for predicates *extractTitle* and *extractAbstract*. They share the same partitioning function f_3 because table $webPages$ can be partitioned in the same way (*i.e.*, tuple by tuple) to satisfy the property for both predicates. Note that the developer does not need to specify all the applicable properties for an operator. For example, in addition to the closed-form properties, the operator *crawl* also has the input partitionability and attribute independence properties. Instead of specifying all the properties, the developer can specify those that he or she deems cost-efficient to execute.

Also note that each specification alone enables us to incrementally execute an operator p when among all the input tables of p, only the one given in specification has been changed. Therefore, to enable incremental update for any changes to the input of p, the developer needs to specify properties for each input table of p.

Algorithm: Once the developer has specified incremental properties for an operator p, we can create incremental versions of p accordingly. Figure 6 gives the pseudo-code of the algorithm that we currently use to realize incremental execution of p for each property specification. Briefly, the algorithm takes as input the specification, the old input and output tables, and the new input table I_i'. To compute the new output, the algorithm first computes I^+ and I^-, the set of tuples inserted into I_i and the set of tuples deleted from I_i. It then executes the function f given in the specification to update the old output.

Algorithm: Generic Incremental Update Algorithm
Input: Spec: $p(I_i) \leftarrow (type, f)$
$I_1, \cdots, I_n$ – the old input tables of p
I_i' – the new value of the ith input table
O – the old output table of p
Output: O' – the new output table
Process:
1. $O' = \emptyset$; $I^+ = I_i' - I_i$; $I^- = I_i - I_i'$;
2. **if** $(type = ip)$ **then**
3. $\{S_{1:u}\} = f(I_i)$; $\{S'_{1:v}\} = f(I_i')$;
4. **for each** $P \in \{S_{1:u}\} \cap \{S'_{1:v}\}$
5. $O' = O' \cup \{$output of partition P in $O\}$;
6. **for each** $P \in \{S'_{1:v}\} - \{S_{1:u}\}$
7. $O' = O' \cup p(R_i(P))$;
8. **else if** $(type = ci)$ **then**
9. $O' = f(R_i(I^+), O)$;
10. **else if** $(type = cd)$ **then**
11. $O' = f(R_i(I^-), O)$;
12. **else if** $(type = ai)$ **then**
13. **if** $(f(I^+) \neq f(I^-))$ **then**
14. $O' = p(R_i(I_i'))$;
15. **else** $O' = O$;
16. **else if** $(type = pc)$ **then**
17. $U = p(R_i(I^-))$;
18. $O' = O - U - f(R_i(I^-), R_i(I_i - I^-))$;
19. $U = p(R_i(I^+))$;
20. $O' = O' + U + f(R_i(I^+), R_i(I_i - I^-))$;
21. **return** O';

Figure 6: Generic Incremental Update Algorithm.

4.2 Incremental ID Maintenance

Consider a single-input operator p. Let I_1 and O_1 denote its input and output from its previous execution, and I_2 and O_2 denote its new input and output. Recall from Section 3.2 that to decide whether a user update F to O_1 is applicable to O_2, we check whether the provenance $R \subseteq I_1$ of F is also a provenance in I_2. One way to check this is to compare the contents of tuples in R with those in each provenance of I_2. But this is often expensive. If two tuples $t \in I_1$ and $t' \in I_2$ have the same ID if and only if they have the same content, then we only need to compare R with each provenance in I_2 by their tuple IDs, which is much more efficient. Call such a condition *ID consistency*. To ensure this condition, clearly we need to reconcile the tuple IDs whenever the input table I_1 of p has been re-computed to be another input table I_2, by some operator q (below p in the execution graph).

By using incremental update, we can save a lot of ID reconciliation effort. This is because when we incrementally execute q, the unchanged tuples are retained in the output table of q (which is the input table I_2 discussed above for p). Thus for these tuples, their IDs are consistent (*i.e.*, the same), and we only need to reconcile IDs for the newly generated output tuples. Specifically, for each new tuple t, we check whether t is present in the old output. If so, we set t's ID to be its ID in the old output. Otherwise, we assign t a new ID, which is guaranteed to be different from that of any other tuple. Therefore, by leveraging incremental execution, we can maintain ID consistency incrementally.

The above notion of ID consistency, however, requires the tables to have set semantics, that is, they cannot contain duplicates. This requirement may not work for certain IE/II operators in practice. To address this problem, we can employ a more relaxed notion of ID consistency. We say that tuple IDs are consistent across two tables I_1 and I_2 if whenever two tuples $t_1 \in I_1$ and $t_2 \in I_2$ have the same ID, they are duplicates. This consistency notion is more relaxed in that two tuples with different IDs can still be duplicates.

Using this relaxed ID consistency notion, when checking whether provenance $R \subseteq I_1$ is also a provenance of I_2, for each tuple $t \in R$, we check for its presence in I_2 by first checking if its ID appears in I_2. Only if we do not find its ID in I_2 would we resort to comparing its content against the content of tuples in I_2.

4.3 Improved Concurrency Control

The simple graph-locking policy in Section 2.3 requires a transaction to exclusively lock the entire execution graph before it starts. Since a transaction only executes one operator at any time, exclusively locking the whole graph excludes other transactions from executing other operators. By exploiting the fact that an execution graph is a DAG, we can design more efficient concurrency control solutions. In what follows we describe two such methods: table locking and operator skipping.

4.3.1 Table Locking

Recall that a user feedback transaction starts by updating the table whose view the user has edited. Refer to this table as the starting table. The transaction then re-executes

operators that depend on the starting table, and so on. It terminates after it has re-executed the root-node operator.

The table-locking policy does not require a transaction T to lock the entire execution graph G. Instead, it requires T to acquire locks on individual tables before it updates its starting table or executes an operator. Specifically:

- Before updating the starting table S, T requests an exclusive lock on S. Let p be the operator[2] that produces S. T also requests exclusive locks on the provenance tables and the update table (*i.e.*, the table stores all user updates to S) of p, in an all or nothing fashion. T releases these locks when the update is completed.
- Before executing an operator p, T requests share locks on all the input tables of p in an all or nothing fashion.
- Before writing the output of p, T acquires exclusive locks on p's output, provenance, and update tables, in an all or nothing fashion. After writing the output, T releases these locks, together with the share locks on the input tables of p.

Compared to the two-phase locking of the execution graph, the table-locking policy allows a transaction to interleave its lock acquisition and releasing activities. To execute an operator p, a transaction locks only the tables related to p. Furthermore, it releases these locks as soon as the output of p is written. Therefore, the policy allows other transactions to execute an operator q as long as there is no input dependency between p and q (*i.e.*, p and q are not connected in the execution graph). The following theorems state that the table-locking policy guarantees the consistency of the system and that concurrent execution is deadlock-free.

THEOREM 1 (CONSISTENCY). *Given a set of transactions $\{T_1, T_2, \cdots, T_n\}$, if all transactions follow the table-locking policy, then the system remains consistent after executing these transactions if it is consistent before.*

THEOREM 2 (DEADLOCK FREEDOM). *The table-locking policy cannot produce a deadlock.*

We omit the proofs for space reasons. See the extended technical report [6] for more details.

4.3.2 Operator Skipping

Consider a set of transactions $\{T_1, \cdots, T_n\}$ running concurrently on an execution graph G. Consider an operator p in G. Let T be the last transaction that executes p. Recall that a transaction executes operators in G upwards. Thus at the time T executes p, all the input tables of p are at their final states (*i.e.*, no other transactions will change the value of any input table of p). This suggests that once the input tables of p are at their final states, it is sufficient to have only one transaction execute p to update its output. In other words, transactions other than T can skip executing p, and let T execute p and write the output.

Suppose after T executes p, the output of p is changed. Then by definition, T must execute each operator q which takes p's output as input. Suppose for simplicity that q is a single-input operator. Then T also writes the final value of the output of q. This suggests that a transaction T' ($T' \neq T$) can also skip executing q since the final state of q is set by T. Applying this reasoning recursively, T' can skip executing those operators in G that are reachable from p. If T' can skip all the operators it needs to execute, T' can commit immediately given that T commits eventually.

Based on this idea, we extend the table-locking policy to allow a transaction to skip executing an operator if some other transaction will eventually read the final values of the input of the operator and overwrite the output. We call the extended policy operator skipping.

To implement operator skipping, we maintain a transaction ID list for each operator. When a transaction T starts, it adds its ID to the list of each operator it is going to execute. Before executing an operator p, T checks whether it is the only transaction in p's list. If so, T executes p; otherwise, T skips p. In either case, T removes its ID from p's list. Operator skipping also guarantees consistency and deadlock freedom since it follows the table-locking policy.

[2]If S is an extensible predicate, then we pretend that it is the output of a dummy operator (see Section 2.3.2). In this case, T x-locks S only.

5. EMPIRICAL EVALUATION

As a proof of concept, we conducted a preliminary case study by applying our user feedback solution to DBLife [13], a currently deployed IE and II application. Our goal is to evaluate the efficiency of the solution in incorporating user feedback into IE and II programs, focusing specifically on the optimization techniques proposed in Section 4. The experiments show that the incremental execution approaches can reduce the program execution time considerably, and that by exploiting the DAG structure of an execution graph, the concurrency control methods, table locking and operator skipping, can significantly improve the system performance in terms of both transaction throughput and response time.

Application Domain: DBLife [13] is a prototype system that manages the data of the database community using IE and II techniques. Given a set of data sources (e.g., homepages of database researchers and conference websites), DBLife crawls these data sources regularly to obtain data pages, and then applies various IE and II operators to the crawled pages to discover entities (e.g., people and organizations) and relationships between them (e.g, affiliated-with and give-talk). A variety of user services, including browsing and keyword search, are then provided over the obtained entities and relationships.

Methods: To evaluate the effectiveness of the framework, we implemented a modified version of the DBLife system under the framework. The modified DBLife system contains 13 operators, as listed in Table 1. These operators, ranging from crawling data sources to extracting mentions, to finding entities and relationships, cover the most essential operators in the entire DBLife workflow. Input to the modified DBLife program consists of four tables, which store data sources, researcher names, organization names, and publication titles, respectively. Output of the program consists of three entity tables and three relationship tables, produced by the last six operators in Table 1.

To evaluate the efficiency of the system in incorporating user feedback, we exposed the output of each operator, together with the four input tables, for user feedback. For simplicity, we used identity views to expose these 17 tables. We built a transaction simulator to generate user feedback transactions on one snapshot of the program data. The simulator

	Inc. Properties				
DBLife Operators	*ci*	*cd*	*ip*	*ai*	*pc*
Get Data Pages	✓	✓	✓	✓	✓
Get People Variations	✓	✓	✓		✓
Get Publication Variations	✓	✓	✓		✓
Get Organization Variations	✓	✓	✓		✓
Find People Mentions	✓	✓	✓		✓
Find Publication Mentions	✓	✓	✓		✓
Find Organization Mentions	✓	✓	✓		✓
Find People Entities			✓		
Find Publication Entities			✓		
Find Organization Entities			✓		
Find Related People	✓	✓	✓		✓
Find Authorship	✓	✓			✓
Find Related Organizations	✓	✓	✓		✓

Table 1: Incremental properties of DBLife operators.

generated a user transaction as follows. First, it randomly selected one of the 17 tables. Then it randomly deleted one tenth, inserted one tenth, and modified another one tenth of the tuples in the table to simulate user feedback.

5.1 Incremental Execution

Broad Applicability of Incremental Properties: When developing DBLife [13], we did not expect its operators to be executed incrementally. Thus, we did not design them to be incrementally updatable. However, all the DBLife operators surprisingly have at least one incremental property presented in Section 4.1, and many of them have several. Table 1 lists the incremental properties (checkmarked in the table) of the operators we experimented with. This suggests that these properties may have a broad applicability to many real-world IE and II operators.

Efficiency of Incremental Execution: To evaluate the incremental update and incremental ID maintenance methods in Sections 4.1 and 4.2, we developed three versions of DBLife. They are (1) the basic version which executes each operator from scratch, (2) the incremental update version which executes the operators incrementally (the properties used for each operator are underlined in Table 1), and (3) the incremental update with ID maintenance version which leverages incremental update to maintain ID consistency.

In the experiment, we first initialized each version of the system by running the DBLife program over a given set of data sources. We varied the size of the set so that it gave 100-600 pages, at an interval of 100 pages. For each version, we then simulated 170 user feedback transactions, 10 on each table. Next, we executed these transactions separately. After each transaction completed, we restored the system to its initial state.

Figure 7 compares the average transaction execution time in the three versions as the size of data sources varies. As we expected, the basic version has the longest average execution time. As the size of data sources increases, the gap between the basic version and the incremental versions increases. Note that in the experiment, each transaction was simulated to update about one-third of the tuples in a table. Thus if a transaction updates only a few tuples in a table as a user is likely to do, the gain of incremental update will be even more significant. Figure 7 also suggests that leveraging incremental update to maintain ID consistency is a good strategy to reduce the execution time.

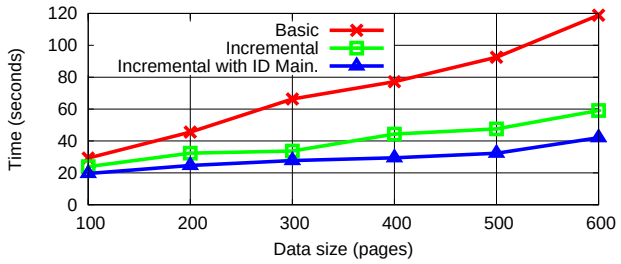

Figure 7: Transaction runtime in different DBLife versions.

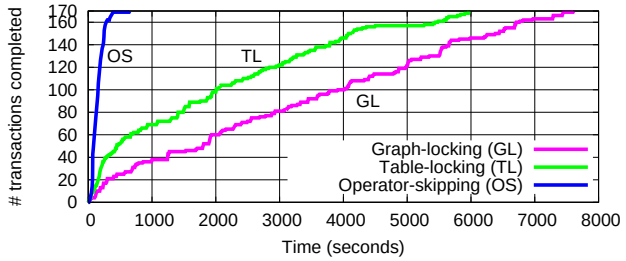

Figure 8: Comparison of transaction throughput.

5.2 Concurrency Control

In the next step, we evaluated the two concurrency control policies proposed in Sections 4.3.1 and 4.3.2. We used graph-locking policy which requires a transaction to exclusively lock the entire execution graph before execution as the baseline. In the experiment, we fixed the size of data sources to 500 pages, and used the incremental update with ID maintenance approach to execute the operators. Again, we simulated 170 user feedback transactions, 10 for updating each table. These transactions were then started one after another in a random order, at an interval of one second. The same order was used for comparing different policies in the experiment. We recorded the starting time and the commit time of each transaction, and also measured the space consumption of the system at different times.

Figure 8 shows the number of transactions completed at any given time. The total time to complete 170 transactions for the graph-locking (GL), table-locking (TL) and operator-skipping (OS) policies is 7605, 5965 and 641 seconds, respectively. It is clear that TL and OS outperform GL since they allow transactions to be executed concurrently. However, the improvement of TL over GL in terms of throughput is not as significant as we expected. This is because many of the operators are long-running and CPU-bounded. Thus although TL allows multiple transactions to execute different operators at the same time, these transactions have to compete for CPU intensively.

Figure 8 also demonstrates that OS outperforms the other two policies significantly in terms of transaction throughput in any period of time. This is not surprising because each transaction by definition must propagate updates on its starting table all the way to the end tables in the program. Thus two transactions may easily overlap in terms of the operators to execute. This is especially true for those

	min	max	average
Graph-locking	0s	7,584s	3,203s
Table-locking	1s	5,485s	1,841s
Operator-skipping	0s	457s	43s

Figure 9: Comparison of transaction response time.

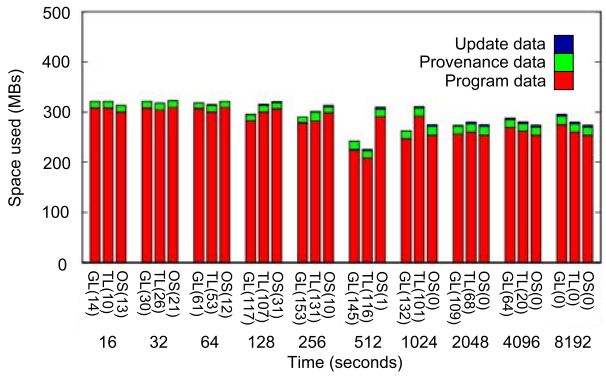

Figure 10: Space consumption under different concurrency control policies. Numbers in brackets give the number of active transactions at a certain time.

operators that produce the end tables. In particular, big transactions (e.g., those updating the input tables) usually subsume small transactions (e.g., those updating the end tables). Therefore, a transaction is likely to skip executing an operator p if some other transaction will eventually overwrite the output of p.

Table 9 lists the response time of the transactions under different policies. The average response time of table-locking is nearly one half of that of graph-locking, and the average response time of operator-skipping is only 1/40 of that of table-locking. Comparing the maximum response time of the three policies, we also observe much difference. Operator-skipping outperforms the other two significantly because small transactions tend to commit immediately when there are active transactions in the system that subsume them. As a result, fewer transactions compete for CPU or IO resources.

Figure 10 shows the amount of space used by DBLife at different time points during concurrent execution. As we can see, the total amount of space consumed remained at the same level, regardless of the number of active transactions in the system. Variations were mostly caused by the updates from the transactions.

By decomposing the total space consumption in Figure 10, we see that the extra amount of space incurred to store the provenance data and the user update data is reasonable. On average, provenance data only takes about 5.7% of the space that the program data needs. Furthermore, the growing rate of the space consumption is much lower than that of the data sources. This is because intermediate results take up most of the space, and they are independent of the data sources.

6. RELATED WORK

Soliciting and incorporating user feedback to improve IE/II results has received much attention in recent years. Recent works include using user feedback to correct schema matching results [15, 36], to generate integrated schemas [9], and to manage data in community information systems [17, 12, 29] and dataspace systems [19, 27]. While these works focused on leveraging user feedback to improve results of individual IE/II operations, our work aims to build an end-to-end framework where user feedback can be incorporated into various stages of a complex workflow. To allow developers to write programs for such workflows, we also proposed a declarative language hlog, which extends recently developed IE language xlog [35] by providing constructs for specifying user interactions.

Using the derivations of updates, *i.e.*, their provenance [5, 4] or lineage [10] for update reconciliation was also explored in [22]. In their work, the authors proposed a provenance model and used it for trust policies and incremental deletion. In contrast, we use provenance information to interpret the semantics of user updates and incorporate updates into execution results.

Many early works [2, 23, 24] have proposed and studied incremental view maintenance algorithms. Recent works have also considered how to incrementally execute IE/II operators [1, 7]. The proposed approaches, however, are specific to schema matching [1] and information extraction [7] settings. Thus they are not easily extensible to other operators. Our goal, in contrast, is to support incremental execution for a broad class of operators, whose semantics is unknown to the system.

Transaction concurrency control in relational databases has been well studied in the literature [21, 30, 31]. To the best of our knowledge, our work is the first attempt in exploiting the DAG structure of IE/II programs to provide efficient concurrency control outside RDBMS.

7. CONCLUSION AND FUTURE WORK

Despite recent advances in improving the accuracy and efficiency of IE/II programs, writing these programs remains a difficult problem, largely because automatic IE/II are inherently imprecise, and there is no easy way for human users to provide feedback into such programs. To address this problem, we proposed an end-to-end framework that allows developers to quickly write declarative IE/II programs with the capabilities of incorporating human feedback. In addition to the framework, we also provided optimization techniques to improve the efficiency and concurrency of program execution. Experiments with DBLife demonstrated the utility of the framework.

Our work has raised more research problems than those solved. For example, what are the strength and weakness of different user interfaces in supporting human interactions? How can we extend the framework to support views over multiple tables? Supporting multiple-table views requires us to revisit the semantics of program execution. This is because user updates on such views may potentially update multiple tables in an execution graph, and incorporating updates will be much more complicated. Furthermore, in this work, we assume that users are either reliable or there is a control mechanism that can filter and aggregate unreliable user feedback into reliable feedback. As a next step, we plan to investigate issues raised from unreliable user feedback, and develop such mechanisms. Finally, we plan to conduct user studies to evaluate the usability of the framework.

Acknowledgments: We thank the anonymous reviewers for invaluable comments. This work is supported by NSF

Career grant IIS-0347943, a DARPA seedling grant, and grants from the Microsoft Jim Gray Systems Lab, IBM, Yahoo, and Google.

8. REFERENCES

[1] P. A. Bernstein, S. Melnik, and J. E. Churchill. Incremental schema matching. In *VLDB-06*.

[2] J. A. Blakeley, P.-A. Larson, and F. W. Tompa. Efficiently updating materialized views. *SIGMOD Record*, 15(2), 1986.

[3] P. Bohannon, S. Merugu, C. Yu, V. Agarwal, P. DeRose, A. Iyer, A. Jain, V. Kakade, M. Muralidharan, R. Ramakrishnan, and W. Shen. Purple SOX extraction management system. *SIGMOD Record*, 37(4), 2008.

[4] P. Buneman, S. Khanna, and W. C. Tan. Why and where: A characterization of data provenance. In *ICDT-01*.

[5] P. Buneman and W. C. Tan. Provenance in databases. In *SIGMOD-07*.

[6] X. Chai, B.-Q. Vuong, A. Doan, and J. F. Naughton. Efficiently incorporating user feedback into information extraction and integration programs. Technical report. [Online] Available: `http://www.cs.wisc.edu/~xchai/papers/hlog_report.pdf`.

[7] F. Chen, A. Doan, J. Yang, and R. Ramakrishnan. Efficient information extraction over evolving text data. In *ICDE-08*.

[8] F. Chen, B. J. Gao, A. Doan, J. Yang, and R. Ramakrishnan. Optimizing complex extraction programs over evolving text data. In *SIGMOD-09*.

[9] L. Chiticariu, P. G. Kolaitis, and L. Popa. Interactive generation of integrated schemas. In *SIGMOD-08*.

[10] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB-01*.

[11] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *ACL-02*.

[12] P. DeRose, X. Chai, B. Gao, W. Shen, A. Doan, P. Bohannon, and J. Zhu. Building community wikipedias: A human-machine approach. In *ICDE-08*.

[13] P. DeRose, W. Shen, F. Chen, A. Doan, and R. Ramakrishnan. Building structured web data portals: A top-down, compositional, and incremental approach. In *VLDB-07*.

[14] P. DeRose, W. Shen, F. Chen, Y. Lee, D. Burdick, A. Doan, and R. Ramakrishnan. DBLife: A community information management platform for the database research community. In *CIDR-07*.

[15] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD-01*.

[16] A. Doan, J. F. Naughton, R. Ramakrishnan, A. Baid, X. Chai, F. Chen, T. Chen, E. Chu, P. DeRose, B. Gao, C. Gokhale, J. Huang, W. Shen, and B.-Q. Vuong. Information extraction challenges in managing unstructured data. *SIGMOD Record*, 37(4), 2008.

[17] A. Doan, R. Ramakrishnan, F. Chen, P. DeRose, Y. Lee, R. McCann, M. Sayyadian, and W. Shen. Community information management. *IEEE Data Eng. Bull.*, 29(1), 2006.

[18] D. Ferrucci and A. Lally. UIMA: An architectural approach to unstructured information processing in the corporate research environment. *Nat. Lang. Eng.*, 10(3-4), 2004.

[19] M. Franklin, A. Halevy, and D. Maier. From databases to dataspaces: A new abstraction for information management. *SIGMOD Record*, 34(4), 2005.

[20] G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The lixto data extraction project - back and forth between theory and practice. In *PODS-04*.

[21] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks and degrees of consistency in a shared data base. In *IFIP-76*.

[22] T. J. Green, G. Karvounarakis, Z. G. Ives, and V. Tannen. Update exchange with mappings and provenance. In *VLDB-07*.

[23] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. *SIGMOD Record*, 24(2), 1995.

[24] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Eng. Bulletin*, 18(2), 1995.

[25] J. Huang, T. Chen, A. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. *PVLDB*, 1(1), 2008.

[26] P. G. Ipeirotis, E. Agichtein, P. Jain, and L. Gravano. To search or to crawl?: Towards a query optimizer for text-centric tasks. In *SIGMOD-06*.

[27] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD-08*.

[28] G. Kasneci, M. Ramanath, F. Suchanek, and G. Weikum. The YAGO-NAGA approach to knowledge discovery. *SIGMOD Record*, 37(4), 2008.

[29] Y. Katsis, A. Deutsch, and Y. Papakonstantinou. Interactive source registration in community-oriented information integration. In *VLDB-08*.

[30] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2), 1981.

[31] P. L. Lehman and S. B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4), 1981.

[32] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An algebraic approach to rule-based information extraction. In *ICDE-08*.

[33] S. Sarawagi. Information extraction. *FnT Databases*, 1(3), 2008.

[34] W. Shen, P. DeRose, R. McCann, A. Doan, and R. Ramakrishnan. Toward best-effort information extraction. In *SIGMOD-08*.

[35] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative information extraction using datalog with embedded extraction predicates. In *VLDB-07*.

[36] W. Wu, C. T. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *SIGMOD-04*.