On the Provenance of Non-Answers to Queries over Extracted Data[*]

Jiansheng Huang, Ting Chen, AnHai Doan, and Jeffrey F. Naughton
University of Wisconsin at Madison, USA.
jhuang@rocketfuelinc.com, {tchen, anhai, naughton}@cs.wisc.edu

ABSTRACT

In information extraction, uncertainty is ubiquitous. For this reason, it is useful to provide users querying extracted data with explanations for the answers they receive. Providing the provenance for tuples in a query result partially addresses this problem, in that provenance can explain why a tuple is in the result of a query. However, in some cases explaining why a tuple is not in the result may be just as helpful. In this work we focus on providing provenance-style explanations for non-answers and develop a mechanism for providing this new type of provenance. Our experience with an information extraction prototype suggests that our approach can provide effective provenance information that can help a user resolve their doubts over non-answers to a query.

1. INTRODUCTION

Search engines can help a user answer a question by locating information sources based on keywords, but not by answering the question directly. As a simple example, a search engine can lead one to a page giving weather information about a city, but cannot directly answer the question "what is the average low temperature in Santa Barbara in May?" Information extraction (IE) addresses this issue by bridging the gap between structured queries and unstructured data. A sample of systems supporting IE include GATE [1], MALLET [2], MinorThird [4], ExDB [10], DBLife [19], SQOUT [24], Avatar [27], and IWP [33]. The number of large-scale IE applications is growing. Unfortunately, information extraction is still an imprecise art, and errors invariably creep into the structured data that is extracted from the unstructured sources. Helping IE users and developers detect and understand such errors is imperative.

Two common techniques advocated for managing such errors are probabilistic databases [31] and provenance [9]. For example, Avatar and ExDB use probabilistic databases, while DBLife uses provenance, and Trio [34] aims to combine both. While these techniques have been helpful, neither completely solves the problems arising from errors in data extraction. Using probabilistic databases is problematic in situations where it is difficult to assign probabilities to the data items. More importantly, while probabilities can help answer questions about how confident one is about a tuple in an answer to a query, they do nothing to explain why a tuple is or is not an answer to a query. The provenance work to date solves half of this problem, in that it can justify why a tuple appears as an answer to a query, but it has no mechanisms for explaining why another tuple is not in the result.

The desire for an explanation of non-answers to queries arises naturally when querying extracted data. For example, a user of DBLife may be surprised to find out that the system believes that person x was not on the program committee of conference y. In fact x may have actually been on the program committee of y, but this fact does not appear in the extracted data, perhaps due to bugs in extractors, or inaccuracies in sources, or incomplete coverage of sources. In such a case, it is important to help developers debug the system and to help users understand why they got the result they did. If, on the other hand, the "fact" really shouldn't be in the result, it is important to explain to the user why this is the case so that they can gain confidence in the non-answer. In both cases, the provenance of non-answers can help, and that is the focus of this paper.

At first thought, the "provenance of non-answers" may seem like a silly idea, because the "non-answers" to a query can be a large or even infinite set, and even for a single non-answer, there may be a large number of possible reasons why it is not an answer. However, in practice this does not mean that no useful information can be returned to a user. The basic idea underlying our work is to start with a given query and data instance and to pose questions like "Could this non-answer become an answer, and if so, how? That is, what modifications to the existing database (for example, tuple insertions or updates) would be required for the non-answer to change to an answer?" In this paper, we propose a mechanism for answering questions like this systematically.

One key idea in our approach is to allow proxy tuples of variables to "stand in" for the potentially infinite set of tuples that could be inserted in the database (this is reminiscent of the use of variables in a C-table [25]). With the addition of these proxy tuples, and the consideration of modifications to existing tuples, it is possible to "derive" non-

[*]Funded by National Science Foundation Award SCI-0515491, NSF Career IIS-0347903, an Alfred Sloan fellowship, and an IBM Faculty Award.

PVLDB '08, August 23-28, 2008, Auckland, New Zealand
Copyright 2008 VLDB Endowment, ACM 978-1-60558-305-1/08/08

answers, in that one can show that with certain modifications to the database a non-answer would become an answer. Then, informally, we define the *provenance of a non-answer* to be the set of tuples (including proxy tuples) from which it could be potentially derived, possibly supplemented with information about their contributing data sources. We can compute this provenance by augmenting the database with these proxy tuples and executing a "provenance query."

The notion of proxy tuples and potential updates allows us to consider large or infinite sets of non-answer tuples. The next critical issue is how to prevent users from being swamped with too much information, as the provenance of all non-answers that can be generated through arbitrary updates is typically too large to be manageable. Our approach to mitigate this information overload is to provide a focussing capability that limits the class of modifications to the database that are considered when computing the provenance of non-answers.

One class of restrictions on the set of updates considered is integrity constraints. We do not have to consider potential derivations that require updates that would violate integrity constraints. While this is useful, another, perhaps more useful class of restrictions deals with "trust." That is, often a user "trusts" some tables far more than others. For example, it may be that some table has been loaded from a trusted data source, while another is extracted by a new, unproven extractor. In this case it may be helpful to the user to see whether a non-answer might become an answer if the trusted table remained as is and only the table loaded by the unproven extractor were modified. The same is true on an attribute basis — some attributes may be trusted more than others. Also, even if no table and/or attribute is trusted more than any other, the user or developer may be able to gain an insight by asking hypotheticals such as "if I take this table to be correct, can you tell me if a non-answer could become an answer by modifications to this other table?" By varying which tables and attributes are "trusted" the user can explore the space of provenance for non-answers in a systematic way.

Our main contributions in this paper are the following: (1) we propose a conceptual framework for provenance of non-answers with the capability of exploiting both constraints and user trust; (2) we provide an algorithm for computing the provenance of a potential answer or of all potential answers for a query in SQL; (3) we evaluate our techniques with debugging scenarios in an information extraction prototype.

In the rest of the paper, we start with a survey of related work in Section 2. We then cover our assumptions and terminology in Section 3. In Section 4 we present our framework and algorithm for computing the provenance of non-answers. We evaluate our techniques in Section 5. In Section 6 we wrap up the paper.

2. RELATED WORK

Information extraction from text has received much attention in the database, AI, Web, and KDD communities (for recent tutorials, see [15, 20, 28]). Research in this area has addressed a wide range of topics, including improving extraction accuracy (e.g., with novel techniques such as HMM and CRF [15]), minimizing extraction time [11, 26], extraction at the Web scale [10, 23], extraction over template-based data (i.e., wrapper construction) [16], developing extraction architecture [14, 29], managing extraction uncertainty [21], and developing declarative extraction language [30]. Despite this broad range of topics, surprisingly very little has been done on the problem of computing provenance in the context of information extraction. As far as we know, ours is the first work that examines the problem of generating provenance-style explanations for non-answers in information extraction.

There is a large and growing body of research dealing with data provenance ([32] presents a recent survey.) Basic ideas studied in practice include lazy evaluation of an "inverted" query over what is in the database (e.g., [9, 17, 35]) and eager propagation of provenance information (alternatively known as annotations) during the evaluation of a query (e.g., [6, 12]). A general theory on provenance can be found in a recent work on provenance semirings [22].

The semantics of a potential answer in our work is dependent on future updates to a database and therefore is different from the semantics of a possible answer in a ULDB [5], where a possible answer is produced from a possible instance that is stored in the database. Nor should a potential answer be confused with a probabilistic answer in a probabilistic database (e.g., [8]), where an answer is produced from probabilistic data stored in the database.

The problem addressed here is also related to problems of view updates [18], database repairs [13] and reverse query processing [7]. Traditionally, the view update problem deals with finding updates that are side-effect free and unambiguous. The provenance of non-answers problem does not require such restrictions. Thus, we can report the provenance of non-answers to queries that, when considered as views, are not updateable. The database repair problem deals with a problem orthogonal to ours in that it considers how to answer queries over an inconsistent database, while our problem attempts to find updates to a consistent database (with respect to integrity constraints) that changes a non-answer to a query into an answer to the query. Reverse query processing is related to our work in that both try to deduce potential or possible database instances that could produce a result tuple to a query, and both have considered integrity constraints. The key difference between the two approaches is that our work considers the current database instance when computing hypothetical updates that could generate answers, while reverse query processing generates a database instance solely based on an intended result and a query.

3. PRELIMINARIES

In this work, we assume that we have an IE system that stores extracted data in a collection of data tables in a relational database. A table is called a *trusted table* if it is assumed to be correct and complete, so we do not have to consider updates or insertions to it when computing the provenance of non-answers. An attribute is called a *trusted attribute* if its values in existing tuples are correct and therefore updates to them can be ignored. (Note, however, that new values can appear in trusted attributes when new tuples are inserted.) A user can choose to trust tables or individual attributes that appear in a database.

Since the tuples in the tables are generated by running extractors over documents, it is useful to record, for each tuple, the extractor and document that generated that tuple. One way to do so would be to store this information in the tuple itself. It is even more helpful if, in the context of explaining the absence of information, we can decide, for a hypothetical tuple, which data source and extractor would generate that tuple if it did indeed exist. For some tasks this is indeed possible (for example, if we know that official information about a CS department must come from that department's web pages.)

In such cases, a system implementer may choose to maintain another internal table S_i, which we call a *data source table*, for each data table R_i in order to keep track of data source information including document locations (URLs) and extractors. We emphasize that such data source tables are optional in our work. Without them, our techniques can still report what tuples must be added to or modified in the database for a given non-answer to become an answer; we just lose the ability to tie these data to specific data sources. If a data source table exists for a data table, we assume that the data source for each tuple in a data table is determined through a foreign key (fk_i) and primary key (pk_i) relationship between the data table (R_i) and the data source table (S_i). In this case we call the foreign key or the primary key the "data source key." In this paper, we assume that attributes in a data source key are always trusted.

Given our assumptions on trusted attributes and trusted tables, our definitions will use an *allowable update/insertion* to refer to an update/insertion that is applied to untrusted tables and attributes, and that satisfies any given integrity constraints.

In this paper a query is confined to an SPJ expression with conjunctive predicates. Although our techniques can be applied to more sophisticated SQL queries by looking at their underlying SPJ expressions, in general more future work is required to handle queries involving aggregates or subqueries. Finally, for simplicity, we do not deal with general satisfiability here and assume that a query is satisfiable (that is, it does not contain unsatisfiable predicates like R.a = 2 AND R.a = 3).

Running Example

We now introduce a running example. Suppose an IE application extracts ranking information for some CS PhD programs from the CRA ranking source, and job openings from the department information sources of some schools. Furthermore, suppose that **openings** is a table storing a school name, the school state, and a field indicating whether or not the school has a job opening. In this example we assume that among these attributes, only the job-opening indicator is extracted from data sources. The school name and state are filled in from trusted sources. Also, **ranking** is another table storing the rank for each school, and we assume that both attributes of the **ranking** table are extracted. Table 1 and Table 2 are sample instances of **openings** and **ranking**.

Now suppose a student has a question about which schools in the state of California are within the top 4 and have job openings. The student can answer this question by issuing the following query:

```
SELECT o.SCHOOL, r.RANK FROM openings o, ranking r
WHERE o.SCHOOL = r.SCHOOL AND o.STATE = ‘ca’ AND
      o.OPENING = ‘yes’ AND r.RANK <= 4;
```

Given our example data, the query returns Stanford and its rank in the result. Executing this query is much more convenient than answering it by hand through consulting the many relevant data sources. However, this convenience comes at the price of uncertainty in the query result. Looking at the result, a user might start wondering: Why is Berkeley not in the result? Is it because it is not one of the top 4? Or is it because it does not have any job openings? Or is Berkeley not in the state of California? With our approach to the provenance of non-answers, we can provide the user with automated facilities to help answer this kind of question.

SCHOOL	STATE	OPENING
stanford	ca	yes
mit	ma	no
cmu	pa	yes

Table 1: `openings`

SCHOOL	RANK
stanford	1
mit	2
berkeley	3
cmu	4

Table 2: `ranking`

For example, if a user wants to know why Berkeley is a non-answer to the query, we will be able to report provenance information that immediately clarifies the following: (1) Berkeley is a potential answer; (2) Berkeley is ranked No. 3; (3) if a potential tuple (berkeley, ca, yes) is inserted into the `openings` table, Berkeley will become an answer (4) the CS department homepage about jobs at Berkeley and the job extractor pair can be looked into to see why this potential tuple was not generated.

4. FROM ANSWERS TO NON-ANSWERS

In this section we present definitions for the provenance of answers and non-answers. Non-answers are split into potential answers (those non-answers that could become answers through some allowable modification of the database) and never-answers (those non-answers that can never become answers with allowed updates.) As we mentioned before, a crucial usability problem is that we should provide a user with focused provenance. We will show how to use constraints and user trust to achieve this purpose.

4.1 Defining Provenance

Given a database D and a relation R, in our definitions we use $R(D)$ to denote the current relation instance for R in D.

4.1.1 Answers

For completeness, we begin by defining the provenance of an answer — the provenance of an answer tuple is the (multi-)set of tuples it is derived from, supplemented with information about the data source for each tuple. In the following, by "derivation" of a tuple, we mean a set of database tuples and how they are used in the evaluation of a query to produce an answer.

DEFINITION 1. Let Q be a query that mentions R_1, ..., R_n and D be the current database. If t is an answer to Q, then there exists at least one derivation of t in the database D. Let $t_i \in R_i(D)$ $(1 \leq i \leq n)$ stand for a set of base tuples that yield a derivation of t. The *provenance* of t consists of t_i (and s_i if R_i's data source table S_i exists, where $s_i \in S_i$ and $s_i.pk_i = t_i.fk_i$) for all i over $1 \leq i \leq n$, for any derivation of t. □

4.1.2 Potential Answers

We now look at non-answers that could become answers. We will define such non-answers as potential answers. Our definition will consider two types of updates: (1) insertion of a new tuple into a base relation, (2) modification of an attribute value in a tuple of a base relation. Deletions are not considered because they would not help produce a potential answer for SPJ queries. Our intuition is that a potential answer should be able to become an answer with a sequence of type-1 and type-2 updates. If there are no allowable type-1 and type-2 updates that could turn a non-answer into an answer, then the non-answer is a never-answer. The following is our definition for a potential answer and the provenance of a potential answer. In the definition, a potential database D' means a database that can be resulted from a sequence of type-1 and type-2 updates to the current database D, and $\overrightarrow{null}_i$ stands for the null proxy tuple for R_i where every attribute has a null value that can be thought of as a variable that appears only once in the DB.

DEFINITION 2. Let Q be a query that mentions R_1, ..., R_n and D be the current database that satisfies the constraints. Let t be a non-answer to Q over D. t is a *potential answer* if there exists a sequence of allowable type-1 and type-2 updates to D that yields a potential database D' that satisfies the constraints, over which t is an answer to Q.

If t is a potential answer, there exists at least one derivation for t in a potential database D'. Let $t'_i \in R_i(D')$ $(1 \leq i \leq n)$ be a set of base tuples that yield a derivation of t and let $t_i \in R_i(D)$ $(1 \leq i \leq n)$ be the original tuples in D that have been updated to t'_i $(1 \leq i \leq n)$. In particular, if t'_i is an inserted tuple, then we set $t_i = \overrightarrow{null}_i$. The *provenance* of t consists of t_i and t'_i (and s_i if R_i's data source table S_i exists, where $s_i \in S_i$ and $s_i.pk_i = t'_i.fk_i$), for all i over $1 \leq i \leq n$, for any potential derivation of t. □

The following is an example of a potential answer and the provenance of the potential answer.

EXAMPLE 1. (berkeley, 3) is a non-answer to the query in our running example, but there exists an insertion of `openings′`(berkeley, ca, yes) which, together with the existing tuple `ranking`(berkeley, 3), yields a derivation of (berkeley, 3). Therefore (berkeley, 3) is potential answer; its provenance includes `openings`(null, null, null), `openings′`(berkeley, ca, yes), `ranking`(berkeley, 3).

If there are no data source tables, then a user gets provenance in the form of t_i and t'_i tuples, which tell the user that for their non-answer to be an answer, t_i would have to be modified to be t'_i (if t_i is an all-null proxy tuple, this means a new tuple t'_i would have to be inserted.) This may already be useful to a user. If a data source table is available, we can give even more information, and can tie things back to the extraction itself. In such a case the user can determine that for their non-answer to be an answer, t_i would have to be modified to be t'_i, and ($s_i.url$, $s_i.extractor$) would have had to create this new t'_i.

As we mentioned earlier, given a potential answer, if we do not consider trust in or other constraints on allowable updates, then we have the problem that any combination of base tuples from the relations referenced by the query can be modified and yield a potential derivation for the potential answer. Let us look at an example.

EXAMPLE 2. The previous example shows that (berkeley, 3) is a potential answer to the example query. Without trust in or constraints on the data, we claim that any combination of tuples from the `openings` and `ranking` tables can be updated and yield a potential derivation for (berkeley, 3). Take `openings`(mit, ma, no) and `ranking`(mit, 2) as an example. A type-2 update can change `openings`(mit, ma, no) into `openings′`(berkeley, ca, yes) and another type-2 update can change `ranking`(mit, 2) into `ranking′`(berkeley, 3). The updated tuples yield a derivation for (berkeley, 3). Obviously this applies to other combinations of tuples as well.

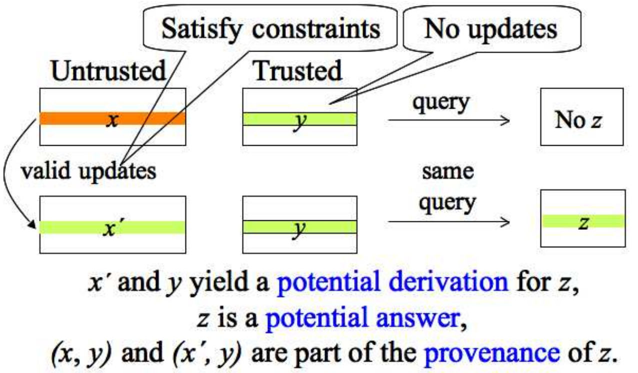

Figure 1: How trust and constraints restrict a potential derivation for a non-answer.

Another issue is that without considering trust in or constraints on data, many potential answers exist that will make little sense.

EXAMPLE 3. (cmu, 4) is a non-answer to the query in our running example. But there exists an update from `openings`(cmu, pa, yes) to `openings′`(cmu, ca, yes) which, together with the tuple `ranking`(cmu, 4), yields a derivation for (cmu, 4). Therefore (cmu, 4) is potential answer, which will surprise those who know that CMU is not in California.

Such a problem can be avoided by using trust and constraints (Figure 1). Regarding trust, if a table is trusted to be complete, then no type-1 updates to the table are allowed. (Otherwise we say that a table is *appendable*.) If a table is trusted to be correct, no type-2 updates to the table are allowed. Similarly, if an attribute of a table is trusted to be correct, no type-2 updates to the attribute are allowed. With trust, a non-answer is a potential answer if there exist type-1 and type-2 updates to untrusted data that yield a potential derivation of the non-answer.

The following is an example of avoiding the problems earlier by using trust in data.

EXAMPLE 4. If the school attribute and the state attribute of the `openings` table are trusted, then `openings`(mit, ma, no) and `ranking`(mit, 2) cannot be updated to yield a potential derivation for (berkeley, 3). If we also trust the completeness of the `openings` table, then (cmu, 4) is not a potential answer because the state for "cmu" cannot be changed to "ca".

4.1.3 Never-Answers

A never-answer to a query is a non-answer that can never become an answer in any potential database given the constraints on and trust in the data.

EXAMPLE 5. If we trust the `ranking` table, (edgewood, 1) is a never-answer. This is because no updates to the `openings` table can yield a potential derivation for (edgewood, 1), given that the `ranking` table is trusted.

4.2 Computing Provenance

The previous section has defined a semantics for the provenance of non-answers. In this section we consider the problem of computing the provenance of potential answers to a given SQL query. Our goal here is to use SQL for this computation, so that this computation can be supported by commercial RDBMS without modification. We begin with computing the provenance of answers.

4.2.1 Answers

By definition, the base tuples that yield a derivation of an answer must appear in the database. Furthermore, if available, the data source information for each base tuple can be added by joining to the corresponding data source table through the data source key. We can compute the provenance of answers to Q by generating a query from Q as in the following: (1) for $1 \leq i \leq n$, if S_i exists, add S_i to the list of tables in Q; (2) for $1 \leq i \leq n$, if S_i exists, add a join between R_i and S_i on $R_i.fk_i = S_i.pk_i$, (3) for each R_i, project out its attributes and for each S_i, project out the $url, extractor$ attributes.

In this provenance query, base tuples that can yield a derivation are reported in one horizontal "provenance tuple". If desired, this horizontal tuple could be "split out" into its constituent base tuples by simple post-processing.

4.2.2 Potential Answers

The base tuples that can be updated to yield a potential derivation for a potential answer must either appear in the database or be null proxy tuples. Also, the values of trusted attributes in each base tuple must satisfy the selection predicates (from the user query) on the attributes unless the tuple is a null proxy tuple; and the values of two trusted attributes from two base tuples must satisfy any join predicates on the attributes unless one of them is a null from a null proxy tuple. The predicates considered here include those that are implicit from transitive rules.

With the above idea, we have designed Algorithm 2 (listed in the Appendix) for computing the provenance of potential answers. It consists of the following basic steps:

1. complete the user query with predicates implicit from constraints and transitivity rules;

2. build the return attributes for the provenance query;

3. build predicates for the provenance query by retaining all predicates on trusted tables or trusted attributes;

4. if a data source table is available, generate trusted join predicates on data source keys for determining data sources of base tuples returned from the provenance query;

5. augment untrusted tables with null proxy tuples and evaluate the provenance query by applying the trusted predicates to tables mentioned in the user query and their corresponding data source tables.

`Trust and Constraints`	`Join between` R_1 `and` R_2
None	$R_1 \times (R_2 \cup \{\overrightarrow{null_2}\})$
Trust $R_2.c_2$	$(R_1 \bowtie R_2) \cup (R_1 \times \{\overrightarrow{null_2}\})$
Trust $R_2.c_2$ & $R_2.c_2$ unique	R_1 left outer join R_2
Trust $R_2.c_2$ & R_2's completeness	$R_1 \bowtie R_2$

Table 3: Trust options and constraints on an attribute c_2 of an untrusted table R_2 that joins with an attribute c_1 of a trusted table R_1 and their impact on the join expression between R_1 and R_2 for computing the provenance of potential answers. Here $R_1 \times R_2$ means a cross product between R_1 and R_2, $R_1 \bowtie R_2$ means a natural join between R_1 and R_2.

The algorithm considers domain constraints, unique constraints (including keys and unique indexes), and foreign key constraints that reference data source tables.

We now elaborate on how we build the return attributes for a provenance query. In addition to projecting out attributes of R_i's and S_i's, we project out attributes that represent a potential tuple for a given untrusted table R_i. If an untrusted attribute of a potential tuple for R_i is equivalent to a constant or a trusted attribute based on predicates in a user query, we project out the equivalent constant or the trusted attribute in the place of the untrusted attribute; Otherwise we project out a variable symbol with the meaning that the attribute cannot be uniquely determined for each return tuple of the provenance query. An attribute of a potential tuple is placed to the right side of the corresponding attribute of a base tuple separated by an arrow (e.g., $v \rightarrow v'$ — this arrow is a syntactic sugar to indicate that v must be changed to v' for a potential derivation).

A variable for an untrusted attribute in our report means that its value can be anything that satisfies query or constraint predicates (e.g., a range predicate), but cannot be uniquely determined for each return tuple. For simplicity, we do not report such predicates along with a variable for our discussions in this paper, although it is fairly straightforward to have them reported if so desired.

We now elaborate on how trust and constraints impact the type of join between two tables for computing the provenance of potential answers. As we will show in our experimental evaluation, the type of a join used in a provenance query between two tables will affect the number of provenance tuples returned in a significant way. The trust and constraints we consider here include trusted attributes, trusted completeness of a table, and unique constraints (including keys and unique indexes).

Let us assume that we have an equi-join predicate between an attribute c_1 of a trusted table R_1 and an attribute c_2 of an untrusted table R_2. There are four possible types of joins between R_1 and R_2 for computing the provenance of potential answers depending on what trust and constraints we have on c_2 of R_2:

- If there is no trust in $R_2.c_2$, then a value of $R_2.c_2$ in any existing tuple of R_2 can be potentially updated to join with any tuple of R_1. Furthermore, a new tuple can also be inserted into R_2 to join with any tuple of R_1. This effectively means a cross product between R_1

o.SCHOOL	o.STATE	o.OPENING	os.URL	os.EXTRACTOR	r.SCHOOL	r.RANK	rs.URL	...
null→berkeley	null→ca	null→yes	http://cs.berkeley...	job-extractor	berkeley	3	http://cra.org...	...

Table 4: `Provenance of an example non-answer: (berkeley, 3)`

and $R_2 \cup \{\overrightarrow{null_2}\}$ for the provenance query. The algorithm generates a cross product between two tables by ignoring a join predicate that references at least one untrusted attribute.

- If $R_2.c_2$ is trusted, then any existing tuple of R_2 either joins with some tuple in R_1 or will never be able join with anything in R_1 (notice that $R_1.c_1$ is trusted). But a new tuple inserted into R_2 may be able to join with any tuple of R_1. In this case, we want a natural join between R_1 and R_2 unioned with a cross product between R_1 and $\{\overrightarrow{null_2}\}$ for the provenance query. To achieve this, the algorithm preserves the join predicate between R_1 and R_2 and creates a disjunction between the join predicate and a new predicate $R_2.* = \overrightarrow{null_2}$ ($R_2.*$ represents all attributes of R_2).

- If $R_2.c_2$ is trusted and unique, then any tuple in R_1 that already joins with an existing tuple in R_2 will not be able to join with any tuple inserted in the future to R_2. However, a tuple in R_1 that does not join with any existing tuple in R_2 will be able to join with a potential tuple inserted into R_2. This is effectively a left outer join between R_1 and R_2. For this case, the algorithm specifies a left outer join clause between R_1 and R_2.

- Lastly, if $R_2.c_2$ is trusted and R_2's completeness is also trusted, then any potential derivation of a potential answer has to satisfy the natural join between R_1 and R_2. The algorithm achieves this by preserving a join predicate on attributes that are both trusted.

The range of trust and constraint options and their impact on the join expression between R_1 and R_2 for computing potential answers is summarized in Table 3. Similar conclusions can be reached if we remove the assumptions on R_1 because R_1 and R_2 are symmetric.

Our algorithm can either compute the provenance of a potential answer (if one is specified) or the provenance of all potential answers (if one is not.) To compute the provenance of a potential answer, we pass in an array of attribute value pairs specifying the potential answer.

4.2.3 An Example

We illustrate the algorithm with our running example by computing the provenance of a potential answer (berkeley, 3) to our example query. We now add two data source tables: `openings_sources (school,state,url,extractor)`, which stores one source for each school in the `openings` table; and `ranking_source (url,extractor)`, which stores a ranking source for the `ranking` table. The `openings` table has a foreign key on the school and state attributes that references `openings_sources`'s primary key on the same attributes.

The school name and state attributes of the `openings` table are part of the data source key, and are trusted by assumption. But we do not trust the opening attribute. For this example, we assume that we trust `ranking` and that there is a unique constraint on the school name attribute of the `openings` table.

To compute the provenance of (berkeley, 3), Algorithm 2 appends the predicate for specifying (berkeley, 3) to the query and builds up a provenance query (shown below). Because there is a join predicate between the `openings` table and the `ranking` table and a unique constraint on the school name attribute of the `openings` table (the left table in the query), Subroutine 1 specifies a right outer join between `openings` and `ranking` in the provenance query.

```
SELECT o.SCHOOL ||‘->’||‘berkeley’,
   o.STATE ||‘->’||‘ca’, o.OPENING ||‘->’||‘yes’,
   os.URL, os.EXTRACTOR, /*from trusted tables*/
   r.SCHOOL, r.RANK, /* from trusted tables*/
   rs.URL, rs.EXTRACTOR /*from trusted tables*/
FROM openings o RIGHT OUTER JOIN ranking r
     ON o.SCHOOL = r.SCHOOL,
     openings_sources os, ranking_source rs
WHERE r.RANK <= 4 AND r.RANK = 3 AND
      os.SCHOOL = r.SCHOOL AND
      os.SCHOOL = ‘berkeley’ AND os.STATE = ‘ca’;
```

The query produces a provenance tuple similar to the one in Table 4. The provenance explains that Berkeley is not an answer because there is no tuple in `openings` in the current database that joins with `ranking`(berkeley,3), but it is a potential answer in that if a new tuple `openings`′(berkeley, ca, yes) is inserted, then it would join with `ranking`(berkeley,3) to yield a derivation for (berkeley,3). If the data source table is available, furthermore the user can look into the associated data source and extractor to try to determine why the tuple `openings`′(berkeley, ca, yes) was not generated in the extraction process.

4.2.4 Never-Answers

Our algorithm can detect if a non-answer is a never-answer. If a non-answer is a potential answer, the algorithm will return a non-empty provenance report for it. Therefore if the algorithm returns an empty provenance report for a non-answer, it must be a never-answer. Constraints, trusted data, and the query specification can be examined to understand why a non-answer is a never-answer to a query.

EXAMPLE 6. If we use our algorithm to compute the provenance of MIT, it will return an empty result. Therefore MIT is a never-answer. By examining the constraints, we find that the `openings` table has a foreign key on the school and state attributes that references `openings_sources`'s primary key on the same attributes. This means that the school and state attributes of the `openings` table are trusted by our assumption, so we could never see a tuple with MIT in CA.

4.3 Example: Anatomy of Potential Answers

Our techniques not only help explaining a specific potential answer, but also help figuring out all potential answers. Table 5 shows the provenance of all potential answers that can be reported by our techniques for the example query and data set. We begin by explaining some aspects of Table 5.

o.SCHOOL	STATE	OPENING	r.SCHOOL	**RANK**
stanford	ca	yes	stanford	1
(stanford	ca	yes)?	stanford	1
(berkeley	ca	yes)?	berkeley	3
stanford	ca	yes	mit →s...	2→Y
stanford	ca	yes	ber... →s...	3→Y
stanford	ca	yes	cmu→s...	4→Y
stanford	ca	yes	(stanford	Y)?
(X	ca	yes)?	(X	Y)?
(X	ca	yes)?	s...→X	1→Y
(X	ca	yes)?	mit→X	2→Y
(X	ca	yes)?	ber...→X	3→Y
(X	ca	yes)?	cmu →X	4→Y

Table 5: Provenance of answers and potential answers. The groups from top to bottom: (1) answers, (2) potential answers when a user trusts `ranking`, (3) potential answers when a user trusts `openings` or neither table, (4) additional potential answers when a user trusts neither table. Answer attributes are in bold font.

For clarity, we use the convention that all variables start with capital letters and all constants use either numbers or lower-case letters. To keep the table compact, we have omitted the data source information and also used "$(A \mid B)$?" in place of "null→A, null→B" to indicate a potential tuple to be inserted. Here A or B can be constants or variables.

Let us look at some examples. The first row in the table (stanford, ca, yes, stanford, 1) represents the answer (stanford, 1). The second row ((stanford, ca, yes)?, stanford, 1) represents a potential answer (stanford, 1) that would become an answer if a potential tuple `openings`′(stanford, ca, yes) were inserted into the database. This simply means that if the data were modified, there would be alternate derivations of (stanford, 1) as an answer. In other words, the fact that something is an answer does not preclude it from being a potential answer as well. The fourth row (stanford, ca, yes, mit → stanford, 2 → Y) represents a potential answer (stanford, Y) that would become an answer if `ranking`(mit, 2) were updated to `ranking`′(stanford, Y) for some $Y \leq 4$. The eighth row ((X, ca, yes)?, (X, Y)?) represents a potential answer (X,Y) for some school X and rank Y, which would become an answer if a potential tuple `openings`′(X, ca, yes) were inserted and a potential tuple `ranking`′(X, Y) were inserted for some $Y \leq 4$.

In the absence of any specified trust, more potential answers appear in the third and fourth groups. The third group shows that only Stanford is in a potential answer. The fourth group shows five potential answers that depend on a potential tuple being inserted into the `openings` table. Given that there is a foreign key constraint on the `openings.SCHOOL` and `openings.STATE` attributes that references the `openings_sources` table, we can find out what schools are located in CA by querying the `openings_sources` table, which contains all the possible values of X in those potential answers. By querying the `openings_sources` table, we see that Berkeley is located in CA, and therefore can appear in a potential answer. As a different example, we can also learn that MIT is not located in CA, and therefore cannot be a value for X. So, MIT cannot appear in a potential answer.

o.SCHOOL	STATE	OPENING	r.SCHOOL	**RANK**
stanford	ca	yes	stanford	1
(berkeley	ca	yes)?	berkeley	3
(X	ca	yes)?	(X	Y)?
(berkeley	ca	yes)?	berkeley	3→Y

Table 6: Provenance of answers and potential answers when there are unique constraints on both `openings.SCHOOL` and `ranking.SCHOOL` and `ranking.SCHOOL` is trusted. The groups from top to bottom: (1) answers, (2) potential answers when a user trusts `ranking`, (3) potential answers when a user trusts neither table.

We now show how trust and constraints help reduce the number of potential answers. If we trust `ranking.SCHOOL`, and there is a unique constraint on both `openings.SCHOOL` and `ranking.SCHOOL`, then many provenance tuples in Table 5 will be eliminated. For example, the previous provenance tuple (stanford, ca, yes, mit, 2) will not be returned because `ranking`(mit, 2) is not allowed to be updated to `ranking`′(stanford, Y), given that `ranking.SCHOOL` is trusted. As another example, the previous provenance tuple ((stanford, ca, yes)?, stanford, 1) will not be returned because the expected new tuple `openings`′(stanford, ca, yes) would violate the unique constraint on the `openings.SCHOOL` attribute. With these trust and constraints, the provenance of potential answers are shown in Table 6, which is much smaller than Table 5.

5. AN EXPERIMENT

To gain an experience in the application of our techniques, we applied them in some debugging scenarios in an information extraction prototype. Our experiment focuses on demonstrating the utility of the provenance of a specific non-answer as well as the effect of trust and constraints on the size of the reported provenance. We then scale up the database size and evaluate the performance of provenance queries in comparison to a user query. We also report the number of provenance tuples in the scaled-up database to give some idea about how manageable the provenance of non-answers will be in a larger database.

5.1 Usage Scenarios

We started out by implementing a prototype for extracting ranks of the top 100 some CS programs from a web ranking document provided by the Computer Research Association [3] and job openings from the web documents of the CS departments. We then used the prototype to fetch web documents into our database repository and extract structures from those fetched documents to generate a data set.

Our database consists of four tables, which are similar to the tables in our running example: `openings`, `ranking`, `openings_sources`, `ranking_source`. The only difference is that we have a data set that is extracted from web documents fetched from the Web. Over this extracted data set, we can now ask "what schools in California are within the top 25 and have job openings" with the following query:

```
SELECT o.SCHOOL FROM openings o, ranking r
WHERE o.STATE = ‘ca’ AND o.OPENING = ‘yes’ AND
      o.SCHOOL = r.SCHOOL AND r.RANK <= 25;
```

Given the extracted data set, the query returns: Stanford, Berkeley, Caltech, UCLA, USC. We now demonstrate how our provenance techniques perform in helping us explain non-answers. We use the following question scenarios.

- Why is UC San Diego (UCSD) not an answer?
- What are all potential answers?

Before we start, let us clarify some assumptions here. The school name and state attributes of the `openings` table form a foreign key that references the primary key consisting of the same attributes in `openings_sources`. The school name and state attributes are part of the data source key, and they are trusted attributes. We do not need a similar data source key for the `ranking` table since there is only one source in `ranking_source`. For this experiment, we assume that the completeness of `openings` is trusted.

We now examine why UCSD is not an answer. Our techniques generate the following provenance query for UCSD. This provenance query has a cross product between `openings` and `ranking`.

```
SELECT o.SCHOOL, o.STATE, o.OPENING ||‘->’||‘yes’,
  os.URL, os.EXTRACTOR, r.SCHOOL ||‘->’|| o.SCHOOL,
  r.RANK ||‘->’||‘X’, rs.URL, rs.EXTRACTOR
FROM openings o, openings_sources os,
     (SELECT * FROM ranking UNION
      SELECT NULL, NULL FROM dual) r,
     ranking_source rs
WHERE o.STATE = ‘ca’ AND
      o.SCHOOL = ‘Univ of California-San Diego’ AND
      o.SCHOOL = os.SCHOOL AND o.STATE = os.STATE;
```

This provenance query returns 109 result tuples. The result tells us that UCSD is a potential answer and there are 109 provenance tuples, each of which contains base tuples that could be updated to yield potential derivations of UCSD. If we look at one provenance tuple, we find that UCSD is not an answer because its `OPENING` attribute has a value of ‘no’ and that if the value is updated to ‘yes’, then UCSD could become an answer. This piqued our interest because we had thought that UCSD did indeed have an opening.

To try to determine what was going on, we looked at the associated data source information in our provenance report and found that indeed UCSD did have an opening on its web page. Upon further examination we determined that the document fetcher in our prototype had a buffer overflow problem when reading a long line, which caused the original document to be truncated. After we fixed this problem and ran the extractor again, and the `OPENING` attribute was updated to ‘yes’ and UCSD became an answer to the test query. This is an unexpected example of our provenance reporting techniques actually helping us debug our prototype application.

We now evaluate how additional trust and constraints help us reduce the number of provenance tuples. Without additional trust or constraints, our current report includes 109 provenance tuples. This makes sense because if `ranking` is not trusted, then each tuple in it (108 of them) and a new tuple (represented by a null proxy tuple) could potentially be updated or inserted so that they would join with the UCSD tuple in the `openings` table.

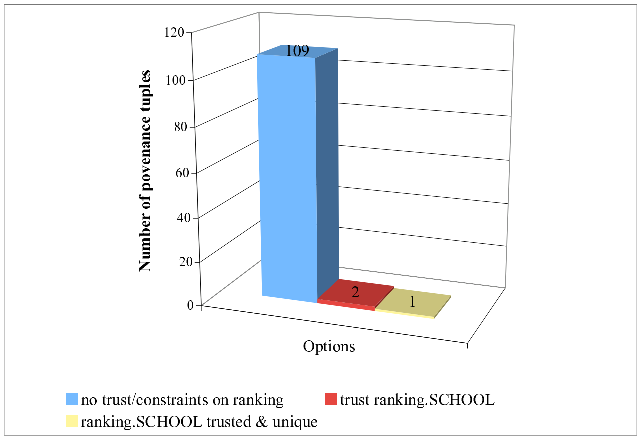

Figure 2: A comparison of the numbers of provenance tuples for UCSD when different trust and constraints options (in addition to the ones assumed for all cases) are used for computing the provenance of UCSD.

If we trust the school name attribute of the `ranking` table, our techniques generate the following new provenance query. It has an extra predicate that joins `openings` and `ranking`:

```
SELECT o.SCHOOL, o.STATE, o.OPENING ||‘->’||‘yes’,
  os.URL, os.EXTRACTOR, r.SCHOOL ||‘->’|| o.SCHOOL,
  r.RANK || ‘->’ || ‘X’, rs.URL, rs.EXTRACTOR
FROM openings o, openings_sources os,
     (SELECT * FROM ranking UNION
      SELECT NULL, NULL FROM dual) r,
     ranking_source rs
WHERE o.STATE = ‘ca’ AND
      o.SCHOOL = ‘Univ of California-San Diego’ AND
      (o.SCHOOL = r.SCHOOL or r.SCHOOL IS NULL) AND
      o.SCHOOL = os.SCHOOL AND o.STATE = os.STATE;
```

The new query returns 2 provenance tuples. One provenance tuple contains the `ranking` tuple for UCSD, while the other contains the null proxy tuple for `ranking`.

If, in addition to trusting `ranking.SCHOOL`, there is a unique constraint on the attribute, then our techniques generate the following query, which now uses a left outer join between `openings` and `ranking`:

```
SELECT o.SCHOOL, o.STATE, o.OPENING ||‘->’||‘yes’,
  os.URL, os.EXTRACTOR, r.SCHOOL ||‘->’|| o.SCHOOL,
  r.RANK || ‘->’ || ‘X’, rs.URL, rs.EXTRACTOR
FROM openings o LEFT OUTER JOIN ranking r
     on o.SCHOOL = r.SCHOOL,
     openings_sources os, ranking_source rs
WHERE o.STATE = ‘ca’ AND
      o.SCHOOL = ‘Univ of California-San Diego’ AND
      o.SCHOOL = os.SCHOOL AND o.STATE = os.STATE;
```

This query returns one provenance tuple, which is the result of a join between the UCSD tuple in `openings` and the UCSD tuple in `ranking`. We summarize the difference in Figure 2 in terms of the numbers of provenance tuples for UCSD, when different trusting options are used.

To answer the second question, we issue a provenance query for all potential answers. If there is no additional

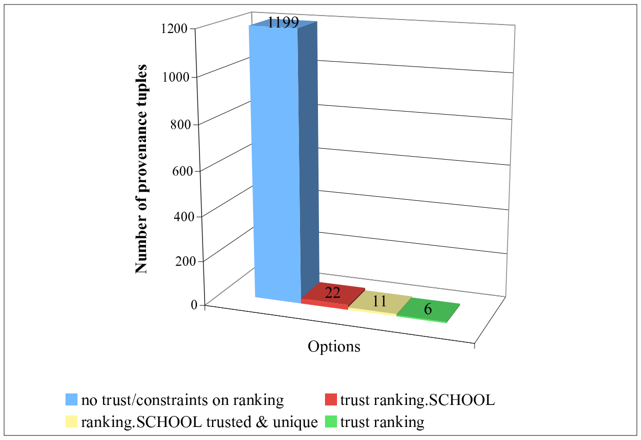

Figure 3: A comparison of the numbers of provenance tuples when different trust and constraints options are used for computing the provenance of all potential answers.

trust in or constraints on `ranking`, the provenance query returns 1199 provenance tuples for all potential answers. If we look at the `openings.SCHOOL` attribute, we find the following potential answers in addition to the answer schools: UC-Irvine, UC-Santa Cruz, UC-Davis, Naval Postgraduate School, UCSD, UC-Santa Barbara. This is an example of how our techniques help us find all potential answers.

We now evaluate how trust or constraints help us reduce the number of provenance tuples and potential answers. If we trust the `ranking.SCHOOL` attribute, the provenance query returns 22 provenance tuples. The provenance tuples still show that all the 6 schools above are potential answers. This is an example of how trust in data helps reduce the number of provenance tuples, but not necessarily the number of distinct potential answers.

If, in addition to trusting the `ranking.SCHOOL` attribute, there is a unique constraint on `ranking.SCHOOL`, then the provenance query returns 11 provenance tuples. Here again, the above 6 schools are also potential answers.

If we trust the `ranking` table, the provenance query returns 6 provenance tuples, which show that only UCSD is a potential answer in addition to the answers schools. This is an example of how trust in data helps reduce the number of distinct potential answers.

In Figure 3, we summarize the impact of trust and constraints on the number of provenance tuples for potential answers.

5.2 Performance and Scalability

The extraction prototype we experimented contains around 100 schools. In order to evaluate the performance of provenance queries as well as the size of provenance in a larger database, we scaled up the prototype database by a factor of 100. Specifically, we took each school tuple in both `openings` and `ranking`, and replicated it 100 times, but gave each replicated tuple a new fictional school name while keeping all other attributes the same. In addition, we replaced the rank for each replicated tuple in `ranking` with a distinct rank. We left everything else in the prototype database as it is (i.e., no additional secondary structures such as indexes were created).

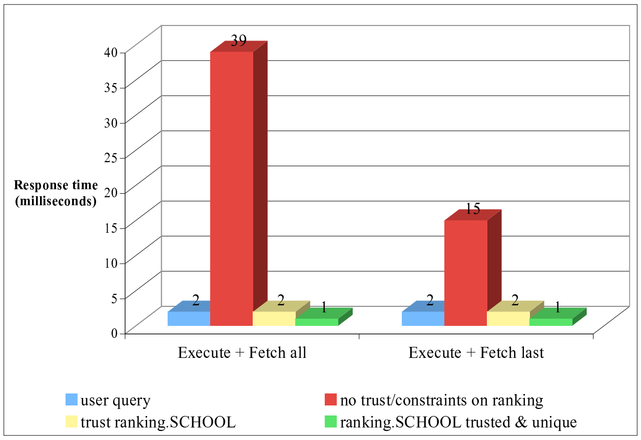

Figure 4: A comparison of the response times for the user query and the provenance queries (corresponding to different trust options) for computing the provenance of UCSD. The numbers labeled "execute + fetch all" include the time for executing a query and fetch all result tuples. The numbers labeled "execute + fetch last" include the time for executing a query and fetch the last result tuple.

5.2.1 *Performance*

We evaluated the performance of the user query as well as the provenance queries. We ran our experiments on a computer with two Intel i686 2GHz CPUs and 2GB of memory. The operating system was Centos Linux (version 2.6.9-55.0.12.ELsmp). We used the Oracle 10g Express server as our database system and its out-of-the-box settings. We ran each query 11 times and measured its response time averaged over the last 10 warm runs. The measurement script was implemented in PL/SQL. The response times of the user query and the queries for computing the provenance of UCSD are shown in Figure 4, and the response times of the user query and the queries for computing the provenance of all potential answers are shown in Figure 5. The response time for the user query is included in both figures for comparison.

Figure 4 contains two sets of response times. The "execute + fetch all" numbers include the time for executing a query and fetching all result tuples, while the "execute + fetch last" numbers include the time for executing a query and fetching only the last result tuple. For evaluating the "execute + fetch last" numbers, a query is fully evaluated and all result tuples are produced internally. The purpose of reporting both response times is to differentiate between the time to compute the result of a query and the time to iterate through the result of the query.

As shown in Figure 4, when there is no trust in or constraints on the `ranking` table, the query for computing provenance of UCSD is an order of magnitude slower than the user query. It turns out that the slowdown is partly caused by the cost of fetching all result rows (10801 of them), which are much more than the result rows (5 of them) of the user query. This can be seen from the figure because the response time measured for executing the same query and fetching only the last tuple is much smaller.

When the `ranking.SCHOOL` attribute is trusted, the query

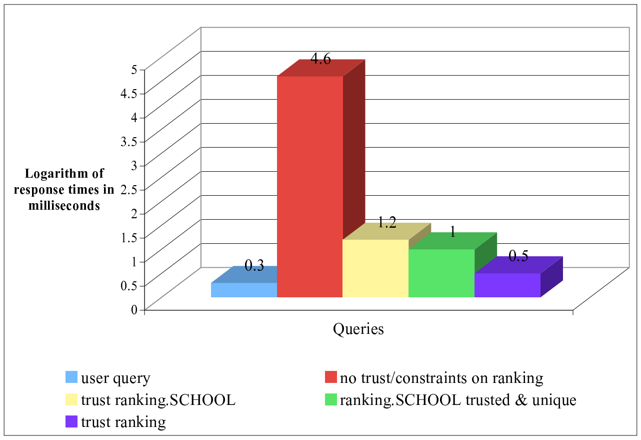

Figure 5: A comparison of the response times for the user query and the queries (corresponding to different trust and constraints) for computing the provenance of all potential answers. The numbers include the time for executing a query and fetch all result tuples.

for computing the provenance of UCSD takes approximately the same amount of time as the user query. This encouraging result shows that the performance of provenance queries can be as good as that of a user query if the join attributes are trusted.

Figure 5 shows that the costs of computing the provenance of all potential answers are expensive compared to the user query. When there is no trust in or constraints on the `ranking` table, the cost of computing the provenance of all potential answers is four orders of magnitude higher. When `ranking.SCHOOL` is trusted, the cost of computing the provenance of all potential answers is about one order of magnitude higher. When the `ranking` is trusted, the cost of computing the provenance of all potential answers is very close to that of the user query.

5.2.2 Number of Provenance Tuples

We now report the number of provenance tuples in the scaled-up database. Figure 6 shows that the number of provenance tuples for UCSD also increases by a factor of 100 if there is no trust in or constraint on the `ranking` table, but that there is no increase in the numbers if `ranking.SCHOOL` is trusted.

Figure 7 shows that the number of provenance tuples for all potential answers increases by four orders of magnitude if there is no trust in or constraint on the `ranking` table. This is because both the `openings` table and the `ranking` table have increased by a factor of 100 in the scaled-up database and the provenance query (when there is no trust in or constraint on `ranking`) contains a cross product between the two tables. If the `ranking.SCHOOL` attribute is trusted, the number of provenance tuples for all potential answers increases by two orders of magnitude, which is the same as the scale-up factor of the database. If the `ranking` table is trusted, the number of provenance tuples for all potential answers does not increase. This is because when we scale up the `ranking` table, all schools synthetically generated have distinct ranks that are higher than 108, therefore they do not contribute to the provenance of potential answers when we trust the `ranking` table.

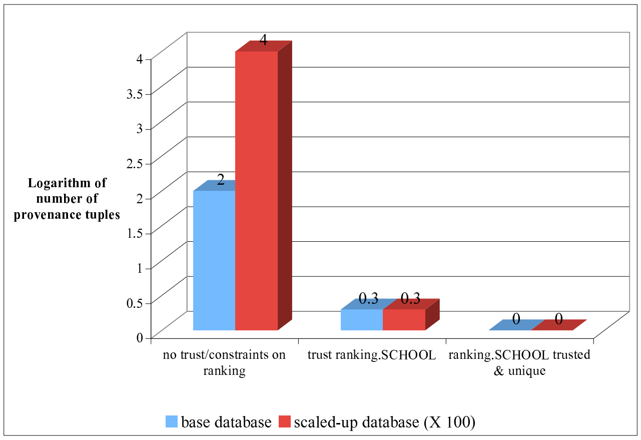

Figure 6: A comparison of the numbers of provenance tuples for UCSD in the base database and in the scaled-up database.

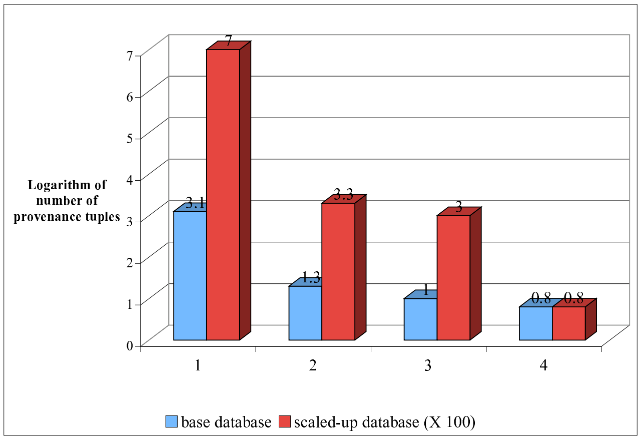

Figure 7: A comparison of the numbers of provenance tuples for all potential answers in the base database and in the scaled-up database when (1) no trust/constraints are on `ranking`, (2) `ranking.SCHOOL` is trusted, (3) `ranking.SCHOOL` is trusted and unique, (4) `ranking` is trusted.

A qualitative analysis of the number of provenance tuples can show some insight on how well it scales in general. Consider a basic query which joins two tables R_1 and R_2 on $R_1.c_1 = R_2.c_2$, and project out $R_1.c_1$ as the query's return attribute. The query may or may not have selection predicates on other attributes of R_1 and R_2. Assume R_1 is trusted and n is the number of tuples in R_2. Given the question "why is x not an answer?" for a potential answer x, we ask how many provenance tuples our techniques will report. Let m_x and n_x be the average number of tuples in R_1 and R_2 that have x as a value for the attributes $R_1.c_1$ and $R_2.c_2$.

If there are no trust in or constraints on $R_2.c_2$, any existing values of $R_2.c_2$ can be updated to the value x. Furthermore, new tuples with the value x can be appended R_2. Therefore our techniques will produce $m_x * (n + 1)$ provenance tuples.

trust or constraints	# provenance tuples
no trust in or constraint on R_2	$m_x * (n+1)$
$R_2.c_2$ trusted	$m_x * (n_x+1)$
$R_2.c_2$ trusted and unique	m_x

Table 7: Trust and constraints on R_2, and their impact on the number of provenance tuples for x, with the assumption that R_1 is trusted. Here n is the number of tuples in R_2, m_x and n_x are the average numbers of tuples in R_1 and R_2 that have x as a value for the attributes $R_1.c_1$ and $R_2.c_2$.

If $R_2.c_2$ is trusted, existing values of $R_2.c_2$ cannot be updated. Therefore only existing tuples of R_2 that already have the value x for c_2 or a new tuple can potentially join with tuples in R_1 that have x as the value of $R_1.c_1$. In this case our techniques will produce $m_x * (n_x + 1)$ provenance tuples.

If $R_2.c_2$ is trusted and unique, then there can be only one tuple from R_2 that can have the value x for c_2. Therefore our techniques will produce m_x provenance tuples.

These results, summarized in Table 7, align with the experimental results for the number of provenance tuples of a specific potential answer. The analysis can be extended to cases when R_1 is not trusted and for the number of provenance tuples of all potential answers.

To summarize, our evaluation has shown that (1) the provenance of a target non-answer can help a user understand whether it is a potential answer or a never-answer, and why; (2) our techniques can also find all potential answers for a query given what is trusted or enforced; (3) adding constraints or trusting part of a database is critical for reducing the number of provenance tuples for potential answers; (4) the performance and scalability study shows that with certain trust and constraints, our techniques are still able to produce a focused provenance report with a reasonable performance when a database is scaled up to contain more data.

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a conceptual framework for reporting the provenance of potential answers. In this framework, trust and constraints are important for providing focused provenance of potential answers. We have designed an algorithm that exploits trust, domain constraints and unique constraints to systematically compute the provenance of potential answers for SPJ queries. An experimental evaluation using a simple but real IE prototype has shown that our techniques are useful for quickly understanding non-answers and sometimes even for correcting a false non-answer (one that should have been an answer).

There is a great deal of room for future work. It is our hope that our work will encourage other researchers to study issues related to the provenance of non-answers to queries. Possible areas for future study include alternative definitions of the provenance of non-answers, other algorithms for computing this and alternative definitions for provenance of non-answers, and the utility of this kind of provenance for users and developers through studies with deployed information extraction applications.

7. REFERENCES

[1] *GATE*. http://gate.ac.uk/ie/annie.html.

[2] *MALLET*. http://mallet.cs.umass.edu.

[3] *Computer Research Association*. http://www.cra.org/.

[4] *MinorThird*. http://minorthird.sourceforge.net.

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

[6] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. In *VLDB*, 2004.

[7] C. Binnig, D. Kossmann, E. Lo. Reverse Query Processing. In *ICDE*, 2007.

[8] J. Boulos, N. Dalvi, B. Mandhani, S. Mathur, C. Re, and D. Suciu. Mystiq: a system for finding more answers by using probabilities. In *SIGMOD*, 2005.

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

[10] M. J. Cafarella, C. Re, D. Suciu, and O. Etzioni. Structured querying of web text data: A technical challenge. In *CIDR*, 2007.

[11] A. Chandel, P. C. Nagesh, and S. Sarawagi. Efficient batch top-k search for dictionary-based entity recognition. In *ICDE*, 2006.

[12] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *SIGMOD*, 2005.

[13] J. Chomicki. Consistent Query Answering: Five Easy Pieces. In *ICDT*, 2007.

[14] E. Chu, A. Baid, T. Chen, A. Doan, and J. F. Naughton. A relational approach to incrementally extracting and querying structure in unstructured data. In *VLDB*, 2007.

[15] W. Cohen and A. McCallum. Information extraction from the web. In *KDD*, 2003.

[16] V. Crescenzi, G. Mecca, and P. Merialdo. Roadrunner: Towards automatic data extraction from large web sites. In *VLDB*, 2001.

[17] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *VLDB*, 2001.

[18] U. Dayal and P. A. Bernstein. On the Updatability of Relational Views. In *VLDB*, 1978.

[19] P. DeRose, W. Shen, F. Chen, A. Doan, R. Ramakrishnan Building Structured Web Community Portals: A Top-Down, Compositional, and Incremental Approach In *VLDB*, 2007.

[20] A. Doan, R. Ramakrishnan, and S. Vaithyanathan. Managing information extraction: state of the art and research directions. In *SIGMOD*, 2006.

[21] M. Garofalakis and D. Suciu. Special issue on probabilistic data management. In *IEEE Data Engineering Bulletin*, 2006.

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

[23] M. Gubanov and P. A. Bernstein. Structural text search and comparison using automatically extracted schema. In *WebDB*, 2006.

[24] A. Jain, A. Doan, L. Gravano Optimizing SQL Queries over Text Databases In *ICDE*, 2008.

[25] T. Imielinski and W. Lipski. Incomplete information

in relational databases. *J. ACM*, 31(4), 1984.

[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*, 2006.

[27] T. S. Jayram, R. Krishnamurthy, S. Raghavan, S. Vaithyanathan, and H. Zhu. Avatar information extraction system. *IEEE Data Eng. Bull.*, 29(1), 2006.

[28] S. Sarawagi. Automation in information extraction and data integration. In *VLDB*, 2002.

[29] W. Shen, P. DeRose, R. McCann, A. Doan, R. Ramakrishnan Toward Best-effort Information Extraction In *SIGMOD*, 2008.

[30] W. Shen, A. Doan, J. Naughton, R. Ramakrishnan Declarative Information Extraction Using Datalog with Embedded Extraction Predicates In *VLDB*, 2007.

[31] D. Suciu. Managing imprecisions with probabilistic databases. In *Twente Data Management*, 2006.

[32] W. C. Tan. Research problems in data provenance. *IEEE Data Eng. Bull.*, 27(4), 2004.

[33] D. Weld, F. Wu, E. Adar, S. Amershi, J. Fogarty, R. Hoffmann, K. Patel, M. Skinner Intelligence in Wikipedia In *AAAI*, 2008.

[34] J. Widom. Trio: A system for integrated management of data, accuracy, and lineage. In *CIDR*, 2005.

[35] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *ICDE*, 1997.

8. APPENDIX

The presentation of the algorithm makes the following assumptions: (1) when appending a predicate to another predicate, the AND operator is used; (2) the steps for building predicates on $S_i.pk_i$ apply individually to all attributes in a key; (3) $\prod$ is used as a shorthand for joining a list of tables. For clarity, some steps are grouped in Subroutine 1 and Subroutine 3. We assume that the subroutines have access to the variables in Algorithm 2.

Subroutine 1 Check constraints to see if outer joins should be used (Details discussed in Section 4.2.2 and Table 3).

```
for each join predicate in $\_pred$ do
    if both join attributes are unique and both tables are appendable then
        Add a full outer join clause between the two tables in $\_query$;
    else if left attribute is unique and left table is appendable then
        Add a right outer join between the two tables in $\_query$;
    else if right attribute is unique and right table is appendable then
        Add a left outer join clause between the two tables in $\_query$;
    end if
end for
```

Algorithm 2 Given a query, trusted tables, trusted attributes, and a non-answer, return provenance of the non-answer if it is a potential answer or of all potential answers if a non-answer is not specified.

```
Require: $Q$; trusted tables (optional);
         trusted attributes (optional);
         a non-answer (optional).
Ensure: Provenance of potential answers reported.
$\_rv = \emptyset$; // return vector for provenance query
$\_pred =$ ''; // predicates for provenance query
$\_join = \emptyset$; // join tables for provenance query
$\_query =$ ''; // provenance query
Append domain constraints and predicates for specifying the non-answer to $Q$;
Replace $Q$'s predicates with the transitive closure;
Build $\_rv$; // Details discussed in Section 4.2.2
Build $\_pred$; // Details in Subroutine 3
// Build the list of join tables.
for each table $R_i$ mentioned in $Q$ do
    if $R_i$ is not trusted and appendable then
        Add $R_i \cup \{\overrightarrow{null_i}\}$ and $S_i$ to $\_join$;
    else
        Add $R_i$ and $S_i$ to $\_join$;
    end if
end for
$query = (\pi_{\_rv}\sigma_{\_pred}(\prod_{R\in\_join} R))$;
Check constraints to see if outer joins should be used in $\_query$; // Details in Subroutine 1
Reduce $\_query$; // optional step for query display: remove redundant clauses
Evaluate $\_query$;
Return tuples that do not include conflicting updates;
```

Subroutine 3 Build predicates for the provenance query.

```
Copy out predicates on trusted tables in $Q$ to $\_pred$;
for each untrusted table $R_i$ do
    Copy out predicates on $R_i$'s trusted attributes that do not reference other untrusted attributes and append them to $\_pred$;
end for
// Add predicates for data source tables.
for each table $R_i$ mentioned in $Q$ do
    Append $R_i.fk_i = S_i.pk_i$ to $\_pred$;
    // Apply transitive rule for predicates on data source keys
    if $R_i$ is not trusted then
        Copy out $Q$'s predicates on $R_i.fk_i$ that do not reference other untrusted attributes to $\_temp$;
        Replace $R_i.fk_i$ with $S_i.pk_i$ in $\_temp$;
        Append $\_temp$ to $\_pred$;
    end if
end for
// Filter in null proxy tuples for untrusted tables
for each predicate $\_temp$ in $\_pred$ do
    for each appendable table $R_i$ mentioned in $\_temp$ do
        Replace $\_temp$ with ($\_temp\ OR\ R_i.* = \overrightarrow{null_i}$); // $R_i.*$ stands for all attributes of $R_i$
    end for
end for
```