DBNotes: A Post-It System for Relational Databases based on Provenance*

Laura Chiticariu, Wang-Chiew Tan, Gaurav Vijayvargiya
{laura,wctan,gaurav}@cs.ucsc.edu

Department of Computer Science, UC Santa Cruz
1156 High Street, Santa Cruz, CA 95060

November 17, 2004

Abstract

We demonstrate DBNotes, a Post-It note system for relational data. In DBNotes, it is possible to attach zero or more notes to every value in a relation. When a relation is queried, DBNotes propagates the attached notes to the result of the query. The method by which a note is propagated is based on provenance (aka lineage). As a consequence, if every value in a relation is attached with a note that describes its address, the notes associated with a value in the result of a query show the provenance of the value.

In this demonstration, we show how one can use the query language pSQL of DBNotes to propagate and query annotations in different ways. We also demonstrate how one can use DBNotes to provide a high-level explanation of the provenance and flow of a value that may be the result of many query transformation steps. A high-level explanation shows a possible journey taken by the value through various databases and query transformation steps. DBNotes can also provide a detailed explanation at each transformation step to precisely explain why a value has moved from a source to a target database.

1 Introduction

We demonstrate DBNotes, a Post-It note system for relational databases where every piece of data may be associated with zero or more notes (or annotations). These annotations are transparently propagated along as data is being transformed. The method by which annotations are propagated is based on provenance: the annotations associated with a piece of data d in the result of a transformation consist of the annotations associated with each piece of data in the source where d is copied from. One immediate application of this system is to use annotations to systematically trace the provenance and flow of data. If every piece of source data is attached with an annotation that describes its address (i.e., origins), then the annotations of a piece of data in the result of a transformation describe its provenance. Hence, one can easily determine the provenance of data through a sequence of transformation steps simply by examining the annotations. Annotations can also be used to store additional information about data. Since a database schema is often proprietary, the ability to insert new information about data without having to change the underlying schema is a useful feature. For example, an error report could be attached to an erroneous piece of data, and this error report will be propagated to other databases along transformations, thus notifying other users of the error. Overall, the annotations on the result of a transformation can also provide an estimate on the quality of the resulting database.

Summary of DBNotes Demonstration Features We demonstrate four main features of DBNotes.

1. First, we demonstrate pSQL, an extension of a fragment of SQL that allows one to specify how annotations should propagate through an SQL query. We demonstrate three types of propagation schemes that are supported

*Supported in part by an NSF CAREER Award IIS-0347065 and an NSF grant IIS-0430994.

by pSQL. Annotations can be propagated according to where data is copied from (the default scheme), according to where data is copied from in *all* equivalent queries (the default-all scheme), or according to the user specification (the custom scheme).

2. The second feature of DBNotes that we demonstrate is the ability to use pSQL to query both data and its annotations. With this feature, one can pose queries to retrieve, for example, all tuples that are derived from a particular source or all tuples with an attached error report.

3. The third feature of DBNotes that we demonstrate is the ability to provide a detailed explanation on the provenance of an annotation, an attribute value or a tuple in the result of a query transformation. For example, we explain the provenance of a tuple by demonstrating a proof that shows how the query transformation produces the output tuple. We show a binding for each variable in the SQL query and demonstrate that under these bindings, the conditions in the `WHERE` clause are satisfied and the desired output tuple is produced according to the `SELECT` clause of the SQL query.

4. The last feature of DBNotes we demonstrate is its capability to visually describe the journey (i.e., the provenance and flow) taken by a piece of data through various databases. For example, when a user asks about the journey of a piece of data d in a database D, DBNotes displays the sequence of databases and transformation steps that derive d in D. It also displays the sequence of databases and transformation steps that depend on d in D. At the visual interface, the user also has the option of "zooming in" on each transformation step to see a detailed explanation of each transformation step.

Related Work A study on the problem of tracing the provenance of data that is the result of a query applied on a relational database was first approached by [5]. The type of provenance studied by [5] is called *why-provenance* according to [3]. This type of provenance explains why a tuple is in the result of a query transformation. In [3], a notion of *where-provenance* is also characterized. The where-provenance of an attribute value tells one exactly where that value is copied from. DBNotes propagates annotations based on where-provenance. In both [3, 5], a "reverse" query is generated in order to answer provenance. This approach is *lazy* in the sense that it requires a reverse query to be generated and evaluated only when the provenance of an output tuple is sought for. In contrast, DBNotes computes provenance *eagerly* by forwarding annotations as data is transformed. A high-level description of how a piece of data has moved along transformations can be constructed by examining the annotations in each database. A reverse query is also generated and evaluated by DBNotes when a detailed explanation on the provenance of an annotation, attribute value or tuple in a database is sought for. A lineage tracing system for data warehouses [4] has been implemented. In this system, the lineage of a tuple in the result of a complex relational view is explained based on relational operators. Velegrakis et al [9] independently developed a system which allows one to query the provenance of data across transformations at the schema level. In contrast, DBNotes traces the provenance of a value across multiple transformations at the schema, as well as the data level. STAG [10] is a system that allows for querying of annotations over digital documents using an SQL-like query language. However, unlike DBNotes, STAG does not propagate annotations. The idea of propagating annotations along transformations is in fact not new and has been proposed in various forms in existing literature. We refer the interested reader to [2] for more discussion on related work.

2 Demonstration Overview

We make use of the following example restaurant database for our demonstration.

2.1 Our Example Database

Our example restaurant database contains information about restaurants owned by Hollywood celebrities in the greater Los Angeles area. We have collected information from four different sources [1, 6, 7, 8] and the corresponding schema for each source is shown below:

Blue_Pages(Name, Address, Owner, Cuisine, Cost)
LATimes(Res_Name, Location, Owner, Cost, Type_of_food)
Restaurant_Reviews(Name, Owned_by, Food, Address, Expensive)
Seeing_Stars(Name, Address, Getting_there, Owner, Cost, Cuisine)

Every attribute value of every tuple may be associated with zero or more annotations. For example, the tuple t_1

t_1: (Madre's {LATimes:Good}, Pasadena, Jennifer Lopez, Expensive, Cuban {May be the worst Cuban food in CA})

in relation LATimes has one annotation attached to the value "Madre's" and one annotation attached to the value "Cuban". The annotations are shown in braces. The rest of the attribute values do not have any annotations. Annotations are not part of the database schema and the method by which annotations are stored and propagated is transparent to the user.

2.2 Propagating Annotations

We first demonstrate a feature of pSQL, an extension of a fragment of SQL, that allows one to specify how annotations should propagate through an SQL query. For example, the pSQL query below integrates information about restaurants from the first two sources and propagates annotations in the default way (i.e., according to where data is copied from). For Q_1, this means that every tuple in Blue_Pages and LATimes as well as the associated annotations are emitted to the result.

Q_1:
```
SELECT DISTINCT   Name, Address, Owner, Cuisine, Cost
FROM              Blue_Pages
PROPAGATE         DEFAULT
UNION
SELECT DISTINCT   Res_Name AS Name, Location AS Address,
                  Owner, Type_of_Food AS Cuisine, Cost
FROM              LATimes
PROPAGATE         DEFAULT
```

If a tuple occurs in both Blue_Pages and LATimes, only one of the tuples is emitted and the corresponding annotations are unioned together. For example, if t_1 is a tuple in LATimes and t_2 (shown below) is a tuple in Blue_Pages, then the tuple t shown below appears in the output of Q_1.

t_2: (Madre's {Blue_Pages:Bad}, Pasadena, Jennifer Lopez, Cuban {Bad Food}, Expensive {Expensive Restaurant})
t: (Madre's {Blue_Pages:Bad, LATimes:Good}, Pasadena, Jennifer Lopez, Cuban {Bad Food, May be the worst Cuban food in CA}, Expensive {Expensive Restaurant})

In addition to the default propagation scheme, we shall also demonstrate the default-all and custom propagation schemes supported by pSQL. The custom scheme is useful, for example, when one wants to retrieve annotations from one source over another. In the default-all scheme, the set of annotations attached to a piece of data in the output is invariant to the way a user formulates her query. We refer the interested user to [2] for more details regarding these two schemes.

The semantics of pSQL queries are described in detail in [2]. Briefly, for each tuple in the cartesian product of relations in the `FROM` clause, we check that the conditions in the `WHERE` clause are satisfied and then propagate the qualifying annotations to the output as specified in the `PROPAGATE` clause. Finally, we take the annotation union of all emitted tuples. This means that duplicate output tuples are merged and the annotations associated to the same output value are unioned together.

2.3 Querying Annotations

We also demonstrate a feature of pSQL which allows one to query over data and annotations through the following use cases. Suppose we have an integrated view, called Restaurants, of all four sources that is given by executing a query

written against Q_1 and the two sources Restaurant_Reviews and Seeing_Stars and the resulting schema is the same as that of Q_1. An important feature of pSQL that allows annotations to be queried is through the use of the keyword ANNOT(attribute-name) which elevates annotations of attribute-name to first-class citizens. Some examples of how one could pose queries against data and annotations are shown below in queries Q_2 to Q_4.

Q_2:
```
SELECT DISTINCT  *
FROM             Restaurants r,
                 ANNOT(r.Name) a
WHERE            r.cost = 'Expensive' AND
                 a LIKE '%Bad%'
PROPAGATE        DEFAULT
```

The query Q_2 retrieves all expensive restaurants with a bad rating. The *annotation variable* a ranges over elements in the set of annotations associated with each Name value of each tuple in Restaurants. This query demonstrates how one can query over data and annotations.

Q_3:
```
SELECT DISTINCT  r.Name AS Name
FROM             Restaurants r,
                 ANNOT(r.Name) a1,
                 ANNOT(r.Name) a2
WHERE            a1 LIKE '%LATimes:Good%' AND
                 a2 LIKE '%Blue_Pages:Bad%'
PROPAGATE        DEFAULT
```

The query Q_3 retrieves all restaurants that have a good rating from LATimes and a bad rating from Blue_Pages. This query demonstrates how one can query about source information provided that annotations carry information about the sources.

The semantics of queries Q_2 and Q_3 are as follows. For each tuple in the cartesian product of relations and annotations sets in the FROM clause, we check that the conditions in the WHERE clause are satisfied and then propagate the qualifying annotations to the output as specified in the PROPAGATE clause. Finally, we take the annotation union of all emitted tuples.

Q_4:
```
SELECT DISTINCT  r.Name AS Name,
                 COUNT(a) AS Bad_Ratings
FROM             Restaurant r, ANNOT(r.Name) a
WHERE            a LIKE '%Bad%'
PROPAGATE        a TO Name
GROUP BY         Name
HAVING           COUNT(a) > 2
```

The query Q_4 (with a custom propagation scheme) retrieves bad restaurants where a restaurant is considered as bad if it has more than two bad ratings. This query demonstrates how one can count the number of annotations. The semantics of query Q_4 are as follows. We take the cartesian product of relations and annotation sets in the FROM clause, discard the tuples that do not satisfy the conditions in the WHERE clause (i.e., the annotation bound to a does not match the pattern '%Bad%'), then propagate the remaining annotations as specified in the PROPAGATE clause. (Note that only the "bad" annotations propagate according to these semantics.) We then group the remaining tuples by the Name value. For each group, we count the annotations associated with the Name value and keep only the groups with more than two annotations. Finally, we take the annotation union of the remaining tuples.

Blue_Pages:

r:

Name	Address	Owner	Cuisine	Cost
Madre's	Pasadena	Jennifer Lopez	Cuban	Expensive

Frequents:

f:

Name	Restaurant
Ben Affleck	Madre's

a:

ANNOT(r.Name)
Blue_Pages:Bad

The conditions in the WHERE clause of Q_5 are satisfied as follows:

1. The condition f.Restaurant = r.Name is satisfied because f.Restaurant = "Madre's" = r.Name.
2. The condition r.Cost = 'Expensive' is satisfied because r.Cost = "Expensive";
3. The condition a LIKE '%Blue_Pages:Bad%' is satisfied because the annotation "Blue_Pages:Bad" currently bound to the annotation variable a matches the pattern '%Blue_Pages:Bad%'.

According to the SELECT clause of Q_5:

1. Since f.Name is "Ben Affleck" and f.Name is copied to Celebrity in the output, the Celebrity value of the tuple produced by the current binding is "Ben Affleck".
2. Since r.Name is "Madre's" and r.Name is copied to Restaurant in the output, the Restaurant value of the tuple produced by the current binding is "Madre's".

Figure 1: The bindings for variables r, f and a in Q_5 and an explanation for why "Madre's" is in the result according to these bindings.

2.4 Explaining Data Provenance

The third feature of DBNotes that we demonstrate is its ability to provide detailed explanations on the provenance of an element (i.e., an annotation, an attribute value or a tuple) in the result of a query transformation. We achieve this in a declarative fashion, by demonstrating a proof that shows how the transformation produced that element.

The query Q_5 below retrieves all celebrities that frequent expensive restaurants which received a bad rating from Blue_Pages. (We assume our example database also contains a relation Frequents(Name, Restaurant) that records information about famous stars and the restaurants they visit. This information is obtained from [8].)

```
Q5:
SELECT DISTINCT   f.Name AS Celebrity, r.Name AS Restaurant
FROM              Frequents f, Blue_Pages r,
                  ANNOT(r.Name) a
WHERE             f.Restaurant = r.Name AND
                  r.Cost = 'Expensive' AND
                  a LIKE '%Blue_Pages:Bad%'
PROPAGATE         DEFAULT
```

Let t_3 be a tuple (shown below) in Frequents which states that Ben Affleck is a regular of Madre's. Suppose t_2 (from Section 2.2) is a tuple in Blue_Pages, then the following tuple t' appears in the output of Q_5:

t_3: (Ben Affleck {Dated Jennifer Lopez}, Madre's)
t': (Ben Affleck {Dated Jennifer Lopez}, Madre's {Blue_Pages:Bad})

When asked to provide a detailed explanation of the provenance of the attribute value "Madre's" in the result of Q_5, DBNotes will first generate a reverse query. The purpose of this query is to extract the relevant source tuples that produced the attribute value "Madre's". DBNotes will show, for each tuple and annotation variable of Q_5, a binding to source tuples and respectively, source annotations, and demonstrate that under these bindings, the conditions in the WHERE clause are satisfied and the desired output element is produced according to the SELECT clause of Q_5. In this case, DBNotes shows the bindings for r, f and a and detailed explanation as shown in Figure 1.

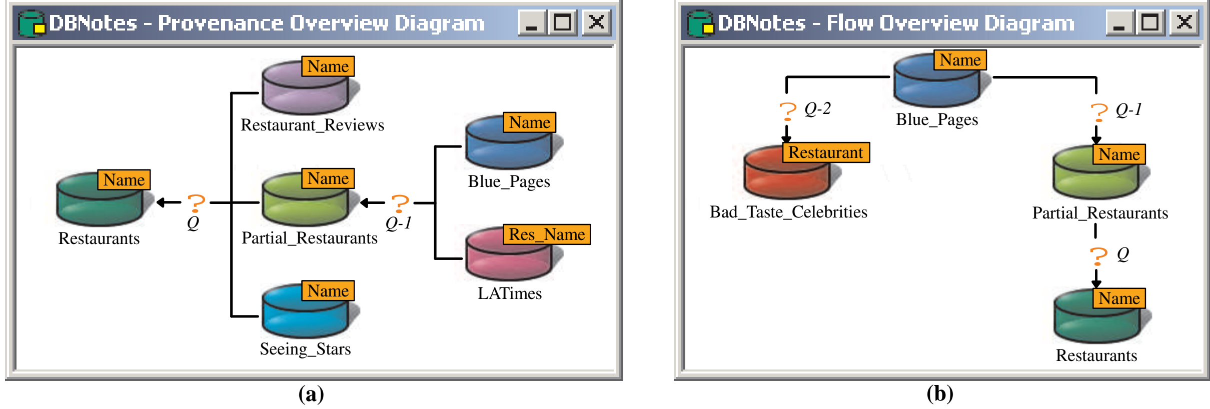

Figure 2: (a) A visual of the provenance of the value "Madre's" in the relation Restaurants, and (b) a visual of the flow of the value "Madre's" from the relation Blue_Pages.

2.5 Tracing the Provenance and Flow of Data

The last feature of DBNotes that we demonstrate is its capability to provide a visual of the "journey" taken by a piece of data through various databases and transformation steps, where the term "journey" refers to both the provenance and flow of that piece of data. When asked to show the provenance and flow of a piece of data d in a database D, DBNotes shows the sequence of databases and transformation steps that derived d in D as well as the sequence of databases and transformation steps that rely on d in D.

If every value is annotated with its address, then the annotations associated with a value in the result of a query describe the provenance of that value since annotations are, by default, propagated based on provenance. We call such annotations, *provenance annotations*. In fact, DBNotes automatically maintains provenance annotations. Whenever a user writes a query Q over a database D, DBNotes creates a new database based on $Q(D)$ that modifies each annotation by adding the address of each value to each existing annotation associated with that value. To exemplify, suppose the result of query Q_1 is stored in a relation called Partial_Restaurants ("PR" in short). DBNotes automatically attaches the address of "Madre's" in Partial_Restaurants to the associated annotations. We obtain the following annotations for "Madre's" in the relation Partial_Restaurants:

{(PR, t, Name).(Q_1)(LATimes, t_1, Res_Name):"LATimes:Good",
(PR, t, Name).(Q_1)(Blue_Pages, t_2, Name):"Blue_Pages:Bad"}.

The first annotation states that the value at the address (PR, t, Name) (i.e., the Name value of tuple t in Partial_Restaurants) is copied from the value at the address (LATimes, t_1, Res_Name) through query Q_1 and this value is the source value with an associated annotation "LATimes:Good". On the whole, the two annotations tell us that the value "Madre's" at address (PR, t, Name) was copied from the corresponding values at addresses in LATimes and Blue_Pages via the transformation Q_1. It is from these provenance annotations that DBNotes could determine the sequence of databases and query transformations that derived a value. For the provenance of the value "Madre's" in the integrated view Restaurants, DBNotes displays the diagram shown in Figure 2(a) (assuming query Q integrates information from Partial_Restaurants, Restaurant_Reviews and Seeing_Stars and the latter two also contain information about the restaurant "Madre's").

In addition to provenance, DBNotes is also capable of displaying a visual of the flow of a value in a database. The flow of a value, however, cannot be eagerly computed, as we are unaware of what future transformations might depend on the value. Instead, DBNotes computes the flow of a value through a Transformations catalog which records information about source and target databases along with the transformations between them. As an example, when DBNotes is asked to compute the flow of the value "Madre's" (at address (Blue_Pages, t_2, Name)) in Blue_Pages of Figure 2(a), it begins by examining the Transformations catalog in search for databases that depend on Blue_Pages. Assuming that the result of Q_5 is stored in a relation called Bad_Taste_Celebrities, the two databases that are discovered

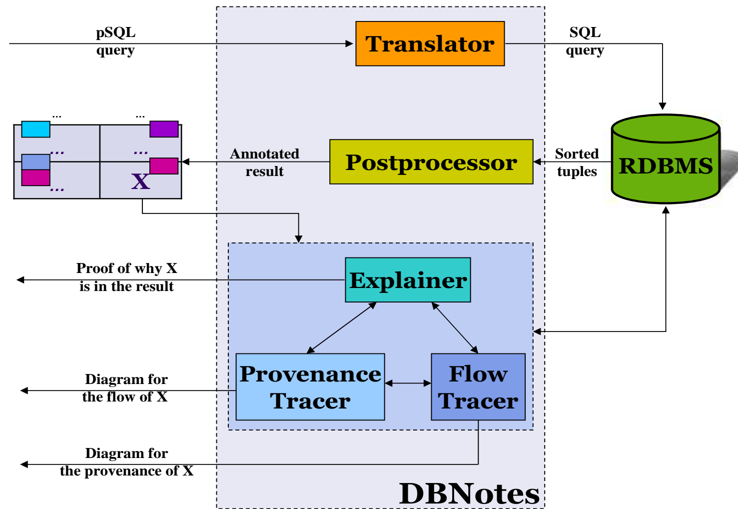

Figure 3: Architecture of DBNotes.

in this phase are Partial_Restaurants and Bad_Taste_Celebrities. In the next phase, DBNotes scans both these relations and, by examining the annotations, discovers that values "Madre's" at addresses (Partial_Restaurants, t, Name) and (Bad_Taste_Celebrities, t', Restaurant) were copied from the value "Madre's" at the address (Blue_Pages, t_2, Name) via the transformations Q_1 and Q_5 respectively. In another iteration, the Transformations catalog is re-examined in search for relations that are derived from Partial_Restaurants or Bad_Taste_Celebrities. In this iteration, DBNotes discovers that a value in Restaurants is copied from (Partial_Restaurants, t, Name). The value "Madre's" in Bad_Taste_Celebrities and Restaurants did not flow elsewhere and hence the resulting figure in Figure 2(b) is displayed.

DBNotes also offers the option of "zooming in" on each transformation step, in order to see a detailed explanation (similar to what was described in Section 2.4) of how a value was produced in the output of that transformation step. For example, if a user wants to understand the reasons "Madre's" appears in the result of Q_5, DBNotes displays the detailed explanation shown in Figure 1.

3 System Architecture

The architecture of our system is illustrated in Figure 3. The explainer module is described in Section 2.4. The provenance and the flow tracer modules are described in Section 2.5. The purpose of the Translator and Postprocessor modules is to compute the result of pSQL queries. The translator module translates a pSQL query into an SQL query against DBNotes's underlying storage scheme for annotations. The postprocessor module merges together annotations associated with identical addresses. We refer the interested reader to [2] for more details about these two modules. DBNotes is currently implemented with Java v1.4.2 on top of Oracle 9i.

References

[1] Restaurant Reviews - Los Angeles Southern California. http://losangeles.about.com/cs/restaurantreviews/.

[2] D. Bhagwat, L. Chiticariu, W. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 900–911, Canada, 2004.

[3] P. Buneman, S. Khanna, and W. Tan. Why and Where: A Characterization of Data Provenance. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 316–330, London, United Kingdom, 2001.

[4] Y. Cui and J. Widom. Lineage Tracing in a Data Warehousing System. In *Proceedings of the International Conference on Data Engineering (ICDE)*, pages 683–684, San Diego, California, 2000.

[5] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems (TODS)*, 25(2):179–227, 2000.

[6] Los Angeles Times. http://www.latimes.com.

[7] Online Blue Pages for Restaurants. http://www.restaurants.com.

[8] Seeing Stars in Hollywood. http://www.seeing-stars.com.

[9] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and Querying Data Transformations. In *Proceedings of the International Conference on Data Engineering (ICDE)*, Tokyo, Japan, 2005.

[10] S. Yamini and A. Gupta. Spatiotemporal Annotation Graph (STAG): A Data Model for Composite Digital Objects. In *Proceedings of the International Conference on Data Engineering (ICDE) - to appear*, Tokyo, Japan, 2005.