Program Slicing and Data Provenance

James Cheney University of Edinburgh

Abstract

Provenance is information that aids understanding and troubleshooting database queries by explaining the results in terms of the input. Slicing is a program analysis technique for debugging and understanding programs that has been studied since the early 1980s, in which program results are explained in terms of parts of the program that contributed to the results. This paper will briefly review ideas and techniques from program slicing and show how they might be useful for improving our understanding of provenance in databases.

1 Introduction

The result of a query could be considered "incorrect" in a number of ways: the input data might be erroneous; the query might contain incorrect data values; or the query itself might be misleading or subject to misinterpretation. For example, consider the queries Q_1, Q_2, Q_3 :

Q_1	SELECT Name, Height FROM People WHERE Name = 'James'
Q_2	SELECT Name, '200' AS Height FROM People WHERE Name = 'James'
Q_3	SELECT P.Name, Q.Weight AS Height
	FROM People P, People Q
	WHERE P.Name = 'James' AND Q.Name = 'Bob'

Suppose that each of these queries returns the same record (Name:James, Height:200) when run against some database *DB*, having a table with schema People(Name, Height, Weight). We might interpret this result as saying that the person James has height 200cm; this happens to be incorrect if 'James' refers to the author of this article. However, in the first case, the error is in the *original data*; in the second case, the error is in the *query*; and in the third case, the error is the mismatch between the user's *interpretation* of the query result and what the query actually says. Of course, there are many other possible sources of error or misinterpretation, such as units of measure (e.g. centimeters versus inches) which we will not consider here.

An expert user who is familiar with the semantics of the query language and who has access to the database can, with some effort, trace erroneous query results to the underlying data in the input, and perhaps "clean" or repair the errors. A lot of recent research has been undertaken to automate the expensive process of correcting errors (or reconciling inconsistencies) in databases, often called *data cleaning* [8]. Automatic data cleaning works best when there is a clear, formal definition of "correct" or "consistent" data; in practice, correctness is

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Copyright 2007 IEEE. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

often taken to be consistency with keys, functional dependencies, or other database constraints. However, it is usually left to the user to determine which constraints characterize "clean" data.

The other problems of misformulation or misinterpretation of a query are more difficult to detect and correct. This problem is compounded by barriers between end-users and databases in typical systems. For example, in a typical Web application, queries are generated by middleware based on user input from a form, so the user who must interpret the results of the query is often not the author of the query, and may not have direct access to either the query or database. Thus, from such a user's point of view, the database (and the overall system) is a "black box" that accepts form input and produces results, which are presented as bare assertions without any supporting explanation or *evidence* that could be used to decide whether the results are trustworthy or not or whether the query accurately reflects the user's interpretation of the results.

There are, of course, many possible ways to bridge this gap. Previous work on *provenance* in databases (see, for example, [9] for an overview) has sought to provide such explanations, for example to answer questions about a query result such as "Why was this record part of the result?" or "Where in the input database did this value come from?" In this article, we consider provenance to be any information that explains how the results were obtained from the underlying database. However, this informal definition begs the questions: just what *is* an explanation, and what makes one explanation preferable to another?

A number of answers have been proposed in previous work on provenance. For example, in approaches such as Cui, Widom and Wiener's *lineage* [6] and Buneman, Khanna and Tan's *why-provenance*, an explanation (called "witness" in [4]) for the presence of a record t in the output of a query Q run on database DB is a subset DB' of the records in DB such that $t \in Q(DB')$. Moreover, there is a "best" explanation DB' is obtained by combining all of the *minimal* explanations. A related approach called *where-provenance* [4] records the source locations in the input from which output data were copied. Most of these definitions are sensitive to the syntax of the query, thus the provenance may be altered by query rewriting. Minimal why-provenance is insensitive to query rewriting, but it appears difficult to extend beyond monotone (SELECT-FROM-WHERE-UNION) queries. In particular, features such as negation, grouping, and aggregation are problematic for these techniques.

However, databases are certainly not the only setting in which it is important to be able to explain the behavior of a large system. This is a central issue in software engineering, security, and many other areas. Therefore it may be worthwhile to consider whether ideas or techniques in these other branches of computer science can be transferred to the database and data provenance settings.

Program slicing is a well-explored technique in software engineering. Intuitively, program slicing attempts to provide a concise explanation of a bug or anomalous program behavior, in the form of a fragment of the program that shows only those parts "relevant" to the bug or anomaly. There seems to be a compelling analogy between program slicing and data provenance, since most approaches to the latter propose to explain part of the result of a query using a "relevant" part of the input database. In this article, we explore this analogy and discuss a form of provenance based on ideas from program slicing and related concepts such as *dependency analysis*.

In the rest of this article, we provide some background discussion of dependency analysis and program slicing (Section 2), show how similar ideas can be used to develop a fine-grained notion of dependency provenance (Section 3), and conclude by discussing some research questions for future work (Section 4). We focus on high-level exposition rather than technical details which can be found in a recent paper [5].

2 Program slicing background

Consider the straight-line program fragment shown in Figure 1(a). If we execute this program in a context where initially x = 1, y = 2, z = 3, w = 4, then the final w-value of the program will be w = 23. If we were expecting a different result value for w, such as 17, then we might like to know what parts of the program are responsible. To diagnose the problem, it would be helpful to highlight a subset of the statements which were relevant to the final result of w, and ignore the other statements. Informally, a *slice* is a subset of the statements of the program.

x = y + 2 * z; y = z + 3 * w;	x = y + 2*z; y = z + 3*w;	x = y + 2*z;	x = y + 2*z; y = z + 3*w;	x = y + 2*z; y = z + 3*w;
z = w - 4 * x; w = x + y;	 w = x + y;		z = w - 4 * x; $w = x + y;$	 w = x + y;
y = z + 3 * w;	···		y = z + 3 * w;	···
z = w - 4 * x;				z = w - 4 * x;
(a) Program	(b) <i>w</i> -slice	(c) <i>x</i> -slice	(d) y-slice	(e) <i>z</i> -slice

Figure 1: Straight-line program and slices with respect to w, x, y, and z

if (x == 0) {	if (x == 0) {	if (x == 0) {	if (x == 0) {
y = z + w;	y = z + w;		y = z + w;
x = 10;			
w = y + 1;	w = y + 1;		w = y + 1;
} else {	} else {	} else {	} else {
y = x + w;			
x = x - 1;			
w = 5;	w = 5;	w = 5;	
}	}	}	}
(a) Program	(b) Static <i>w</i> -slice	(c) Dynamic <i>w</i> -slices	s for $x = 0, x \neq 0$

Figure 2: Conditional program with static and dynamic slices with respect to w

that are relevant to some part of the output. Figure 1(b) shows a slice of the program with respect to w; we have replaced the statements that do not "contribute" to the final value of w with ellipses. Similarly, Figure 1(c)–(e) depict slices with respect to x, y, and z.

Conditional expressions make the slicing problem slightly more interesting. For example, consider Figure 2(a). Since conditionals introduce the possibility of having code in the program that is not executed during a particular run, we distinguish between static and dynamic slices; the former cannot take into account the values actually encountered at run time. A *static slice* for this program with respect to w includes statements in both branches because we do not know which branch will be taken; see Figure 2(b). In a dynamic slice, we may omit all of the code in the branch that is not taken; for example, depending on whether the initial value of x is zero or nonzero, the dynamic slice for w would be as shown in the left or right of Figure 2(c), respectively.

It is, of course, trivial to find at least one program slice: the program itself. However, the goal of slicing is to aid understanding a large and complex program by identifying a small, and hopefully easy-to-understand, subset of program points. As with most interesting program properties, computing minimal slices (whether static or dynamic) is undecidable; it is intractable even if we restrict to programs with conditionals and assignment but without while-loops or recursion. Thus, in practice, program slicing techniques attempt to conservatively approximate the minimal slice.

Slicing captures an intuitive debugging process used by experienced programmers [12]. Since its introduction by Weiser [11], both static and dynamic program slicing have been investigated extensively [10]. Subsequent research has identified *dependence* as a key concept in slicing and a number of related program analysis techniques [1]. In program analysis, dependence information describes how parts of a program, such as variables or control flow points, affect other parts. This information is valuable because it can be used to predict how the program will behave statically before execution or to understand how the program actually behaved after execution. Dependences are often classified into *data dependences*, or dependences on data from which an object was computed, and *control dependences*, or dependences on data that affected the flow of control leading to the computation of an object.

While the majority of research on slicing has considered imperative (C) or object-oriented paradigms, slicing techniques have also been adapted to declarative (functional or logic) programming paradigms which are closely related to database query languages such as SQL.

3 A slicing approach to provenance

In databases, it is usually the *data* that is large and poorly understood, while the query is relatively small. Previous work on data provenance has often defined provenance as a set of "parts" of the input (e.g. fields or records) that "explains" a part of the output. There is a compelling analogy between program slicing, which uses part of a program as a concise "explanation" for part of the output, and data provenance, which uses part of the database to explain part of the output. This analogy suggests that we may be able to transfer ideas and techniques for program slicing into the database and data provenance setting. We explore this idea in the rest of the article.

Recall the queries Q_1, Q_2, Q_3 from the introduction. Suppose we run each of them on the input database consisting of the table People shown in Figure 3. This database contains just three entries. When run against this table, queries Q_1-Q_3 produce produces exactly one record, namely (James, 200).

We now might like to know: What parts of the input does the Height field in this record depend on? There are many possible answers, depending on how we interpret the term "depend". One natural notion is to consider the how a change to each part of the input affects the output. We say that a part of the output *depends on* a part of the input if changing the input part *may* result in a change to the output part. Thus, as in program slicing, we need to consider not just what actually did happen but also what might have happened: how would the output change if the input were slightly different?

We consider three kinds of dependences: dependences of output *relations*, *records*, or *fields* on field values in the input. Consider a query Q and input database I and record $s \in I$ with field B. We say that the output relation *depends on* s.B if changing the value of s.B may cause the output to change in any way. We say that a record $r \in Q(DB)$ *depends on* s.B if changing the value of s.B may delete in r from the output. Finally, we say that the field value r.A in the output *depends on* s.B in the input if there is some way to change the value of s.A that either deletes r from the output or changes the value of r.A. The *dependency provenance* of r.A is the set of all input fields s.B on which r.A depends on. Since the dependency provenance of a part of the input is a subset of fields of the input, we can think of it as being a *data slice* of the input in which irrelevant parts not in the dependency provenance are elided.

We want to emphasize that this is only an informal definition but that it can be made precise and generalized; however, here we will only illustrate the idea through examples. Recall the example from the introduction. Figures 4(a–c) show data slices of the input data u_1 .Height for queries Q_1-Q_3 . For Q_1 , the dependency provenance of u_1 .Height consists of t_1 .Name and t_1 .Height. The value of u_1 .Height was copied from t_1 .Height, and the output also depends on t_1 .Name, because changing this value would make u_1 disappear from the output. For Q_2 , however, as shown in Figure 4(b), u_1 does not depend on t_1 .Height; the value 200 was provided by the query, not copied from the input. It does still depend on t_1 .Name field for the same reason as Q_1 . For Q_3 , as shown in Figure 4(c), t_1 .Height does not depend on t_1 .Height in the input, but it *does* depend on t_3 .Name and t_3 .Weight.

Dependency provenance is clearly similar in some respects to previous approaches such as why-provenance, where-provenance and lineage. In particular, where-provenance (that is, the input field from which an output field was "copied") appears to be included in the dependency provenance. Moreover, for conjunctive queries like the above, the lineage (that is, the input records that "contributed" in some way to an output record) appears to

]	People					
id	Name	Height	Weight		id	Name	Height
t_1	James	200	190	\implies	Iu	-	U
t_2	Alice	160	150		u_1	James	200
t_3	Bob	204	200				

Figure 3: Input data and result of running queries Q_1 , Q_2 , and Q_3

ſ	id	Name	Height	Weight	id	Name	Height	Weight	id	Name	Height	Weight
	Iu	Iname	Height	weight	Iu	Iname	meight	weight	t_1	James		
	t_1	James	200		t_1	James			01			
-	-			I	-				t_3	Bob	• • •	200
L												
((a) Q_1 (b) Q_2					(\cdot))					
						(C) (23					

Figure 4: Data slices with respect to u_1 . Height and queries Q_1 , Q_2 , and Q_3

include all of the records mentioned in the dependency provenance. Finally, why-provenance seems very closely related, but a direct comparison is difficult because the original paper [4] used a semi-structured, deterministic tree model quite different from the relational model we use here. We are glossing over many details here; characterizing the precise relationship between these approaches (and other recent proposals for data provenance in queries [7] and updates [3, 2]) is beyond the scope of this article.

Now we consider a second example, a query Q_4 with grouping and aggregation:

SELECT Name, AVERAGE(Salary) FROM Employees WHERE Year >= 2005 GROUP BY Name

This query returns the names and average salaries since 2005 of all employees; a sample input database and result is shown in Figure 5. Note that Alice has no entries since 2004 so does not appear in the result.

In the previous example, we considered only dependences of output fields on input fields; the relation and record dependences are not very interesting for this example. Relation and record dependences become more important for queries such as Q_4 involving grouping and aggregation.

Figures 6(a-c) show the data slices for the whole output, record u_1 (and u_1 .Salary), and field u_1 .Name, respectively. The whole output depends on everything in the input except for Alice's salary fields; changing them cannot affect the output, but other changes may. The dependency provenance of u_1 is shown in Figure 6(b). The presence of record u_1 clearly depends on all of the data in t_2 and t_3 ; changing any of these fields may affect the average, which would replace u_1 with some other record (James, avg'). Record u_1 also depends on t_1 .Year and t_6 .Name. The reason is that changing '2004' to '2008' in t_1 or changing 'Bob' to 'James' in t_6 would affect the average associated with James in the output. Coincidentally, the provenance of u_1 .Salary turns out to be the same as the provenance of u_1 , and the reasoning is similar. Finally, in Figure 6(c), we see that u_1 .Name does not (directly) depend on anything in the input. Of course, the presence of u_1 does depends on several parts of the input, so u_1 .Name depends "indirectly" on these parts as well, but there is no single field in the input that we can change that will change u_1 .Name in the result.

Employees							
id	Name	Salary	Year				
t_1	James	1000	2004				
t_2	James	1100	2005				
t_3	James	1200	2006				
t_4	Alice	1900	2003				
t_5	Alice	2000	2004				
t_6	Bob	1000	2006				

id

 t_1

 t_2

 t_3

 t_6

id Name Salary	
iu Mallie Salary	
\rightarrow u_1 James 1150	
u_2 Bob 1000	

Figure 5: Input data and result of running query Q_4

Salary

. . .

1100

1200

. . .

Year 2004

2005

2006

. . .

id

Name

(c) For u_1 .Name

Salary

Year

Name

James

James

Bob

. . .

 \rightarrow

id	Name	Salary	Year
t_1	James	1000	2004
t_2	James	1100	2005
t_3	James	1200	2006
t_4	Alice	• • •	2003
t_5	Alice	• • •	2004
t_6	Bob	1000	2006

(a) For whole output

(b) For u_1 and u_1 .Salary



4 Conclusions

We believe that the key question any approach to provenance must answer is what the provenance information *explains* about a query result in the context of the input data and query semantics that is not conveyed by the query result value itself. Previous approaches, such as why-provenance, where-provenance, and lineage have been based on intuitive notions of explanations such as identifying the source data that "influenced" or "contributed to" a part of the output or from which a part of the output was "copied". However, corresponding semantic correctness properties relating these forms of provenance to the actual semantics of a query have proven elusive or hard to generalize beyond monotone queries.

We have outlined one approach, dependency provenance, which is based on well-understood techniques from programming languages such as dependency analysis and program slicing. We believe this approach captures intuitions similar to those motivating other provenance techniques, but may be easier to generalize to the full range of features found in databases, including grouping, aggregation and stored procedures. However, this work is still relatively speculative and more research is needed to determine the feasibility of computing (or conservatively approximating) dependency provenance in practice and scale. Nevertheless, there appears to be a deep connection between program slicing and data provenance that we may be able to exploit by transferring ideas, tools, and techniques from programming languages research.

References

- [1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *POPL*, pages 147–160, New York, NY, USA, 1999. ACM Press.
- [2] Peter Buneman, Adriane P. Chapman, and James Cheney. Provenance management in curated databases. In *Proceedings of the 2006 SIGMOD Conference on Management of Data*, pages 539–550, Chicago, IL, 2006. ACM Press.
- [3] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. In *ICDT 2007*, number 4353 in Lecture Notes in Computer Science, pages 209–223. Springer, 2007.
- [4] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Why and where: A characterization of data provenance. In *Proc. 2001 International Conference on Database Theory*, number 1973 in LNCS, pages 316–330. Springer-Verlag, 2001.
- [5] James Cheney, Amal Ahmed, and Umut A. Acar. Provenance as dependency analysis. In M. Arenas and M. I. Schwartzbach, editors, *Proceedings of the 11th International Symposium on Database Programming Languages (DBPL 2007)*, number 4797 in LNCS, pages 139–153. Springer-Verlag, 2007.
- [6] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.
- [7] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, pages 31–40, New York, NY, USA, 2007. ACM Press.
- [8] Erhard Rahm and Hong-Hai Do. Data cleaning: Problems and current approaches. *IEEE Bulletin of the Technical Committee on Data Engineering*, 23(4), December 2000.
- [9] Wang-Chiew Tan. Provenance in databases: Past, current, future. This issue.
- [10] F. Tip. A survey of program slicing techniques. Journal of programming languages, 3:121-189, 1995.
- [11] Mark Weiser. Program slicing. In ICSE, pages 439-449, Piscataway, NJ, USA, 1981. IEEE Press.
- [12] Mark Weiser. Programmers use slices when debugging. Commun. ACM, 25(7):446–452, 1982.