Have Your Cake and Eat it, Too: Data Provenance for Turing-Complete SQL Queries

Tobias Müller

(Advisor: Torsten Grust)
University of Tübingen
Tübingen, Germany

to.mueller@uni-tuebingen.de

ABSTRACT

We report on our work about the computation of *data provenance* for *feature-rich SQL*. Among further constructs, our prototype supports correlated subqueries, aggregations, recursive queries and window functions. Our analysis approach completely sidesteps relational algebra and instead requires a translation of the input query into an imperative-style program. Provided that the target language is Turing-complete, any SQL query can be covered. We employ a new variant of program analysis which consists of a dynamic and a static part. This two-step approach enables us to dodge limitations that a Turing-complete computation model entails for program analyses otherwise. The derived data provenance directly reflects the data provenance of the original SQL query.

1. INTRODUCTION

Data provenance [3,4] is metadata — primarily about the origin of a certain data piece. Everyday examples for desirable provenance information are the *From:* header field in an email or citations in academic papers. In these two cases, the provenance is trivial and does not need any clever algorithms for its computation (at least: should not).

However, in the context of *real-world relational database systems* there is a deficiency regarding the provenance computation for *contemporary implementations* of *SQL*. SQL, being the standard of relational query languages, has support for advanced language constructs like recursive queries or window functions. Further, nesting of queries is possible, for example, through (correlated) subqueries. These features make writing queries convenient but also make the data provenance of query results non-trivial in the general case. Concrete scenarios in which data provenance for SQL has proved being relevant are the view update/maintenance problem [4], data warehouses [4] and debugging purposes [5]. The analysis approach we are going to describe is capable of computing the data provenance for any non-updating SQL query.

1.1 Provenance Model

We adopt a basic distinction of *Where-* and *Why*-provenance as originally introduced by Buneman and Tan [1]:

- Where-provenance: where has a certain data piece originated? Exactly which table cells were copied or transformed to yield an output cell?
- Why-provenance: why is a certain data piece in the result? Which input table cells were inspected to decide about the existence or contents of an output cell?

1.2 Basic Example

Figure 1 shows an intentionally simple SQL query and corresponding example tables. Mouse pointer ❶ represents an inquiry for the data provenance of table cell t_4: $C_6H_{12}O_6$.

According to the SQL query, two input columns are accessed: `compound` is used to decide if a tuple gets filtered or not. If a tuple qualifies, its value sitting in `formula` is copied over into the result table. Our provenance analysis accordingly finds the result being why-dependent on tuple t_2: glucose and being where-dependent on t_2: $C_6H_{12}O_6$.

In the following sections we revisit this example and illustrate how this outcome actually is computed using our program analysis.

1.3 Advanced Example

The provenance analysis of the query found in Figure 2(b) is a unique feature of our approach: to the best knowledge of the author, only our analysis approach can deal with recursive SQL queries.

The query syntax-checks molecular formulae. Technically, the finite state machine depicted in Figure 3 is executed. The encoded FSM and input formulae can be found in Figure 2(a).

We inspect result cell O_7^{3-} (mouse pointer ❷). The according highlights within the `compounds` table (citrate

compounds

	<u>compound</u>	formula
t_1	citrate	$C_6H_5O_7^{3-}$
t_2	glucose	$C_6H_{12}O_6$
t_3	hydronium	H_3O^+

(a) Database instance.

```
SELECT formula
  FROM compounds
 WHERE compound = 'glucose'
```

output

	formula
t_4	$C_6H_{12}O_6$ ❶

(b) Query and result.

Figure 1: Basic query example and provenance markers.

Proceedings of the VLDB 2016 PhD Workshop, September 9, 2016. New Delhi, India.

and $C_6H_5O_7^{3-}$) show the same pattern as in the basic example of Section 1.2.

More interesting markers can be found within table `fsm`. The highlighted cells inside t_8 and t_9 indicate which state changes were triggered while parsing the first letters of the formula.

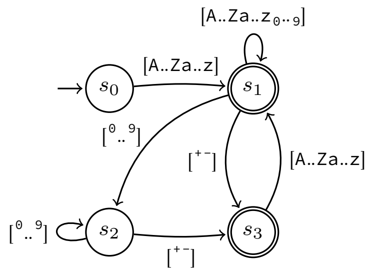

Figure 3: FSM.

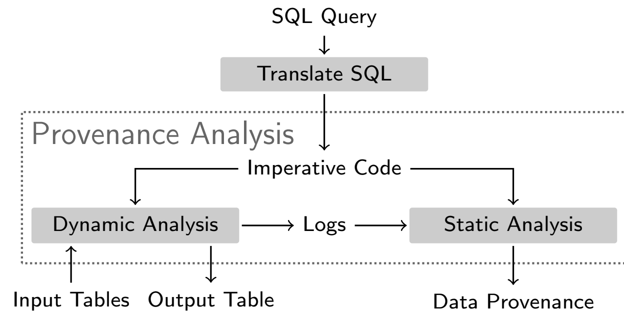

Figure 4: Overview of the two-step analysis.

1.4 Analysis Overview

Figure 4 provides a graphical overview of our analysis approach. The actual provenance analysis happens within the dotted box. It requires the SQL query to be translated into imperative program code. For our prototype, we use a handcrafted SQL compiler. Contemporary database systems like *HyPer* [9] perform such translation internally.

The provenance analysis itself consists of two steps. At first, a *dynamic analysis* takes place which includes code instrumentation and execution. This step actually computes the same query result as a regular query processor would do.[1] As a side effect, two light-weight execution logs are written. They describe the execution flow during runtime and are a key element of this approach.

In our second step, a *static analysis* is carried out exploiting the runtime knowledge encoded within the logs. Our static analysis does absolutely no data processing. The data provenance is derived from program code and logs only. It is inspired by *Program Slicing* [2, 10].

In Section 3, all elements of our provenance analysis will be explained in deeper detail.

2. SQL COMPILATION

Figure 5 shows a simplified yet executable translation of the basic SQL query in Figure 1(b). Ignore the logging statements until the subsequent section.

The target language is kept minimal to just fit our needs: it can compute query results but has no support for I/O operations, for example. Due to space limitations and as the presented code fragment consists of well-known language elements we do not give a formal definition.

[1]As part of our future work, we seek to modify an existing database system and let it run the dynamic analysis simultaneously with query execution.

You find the table `compounds` of Figure 1(a) represented as a data structure (list of dictionaries). The algorithm iterates over the input table (line 3) and if a tuple has qualified (line 5), its formula is appended to the result (line 7).

Please note that we combined input data (i.e. database instance) and the computation algorithm into one program. In the regular case, both of them are kept separate (refer to Figure 4).

3. PROVENANCE ANALYSIS

Before we get to the details of our approach, we shed some light on the theoretical limits of program analysis and the arising dilemma. The theorem of Rice is a result of computational theory. Cast informally, the theorem states that in the Turing-complete computation model only trivial questions about the behavior of a program can be answered. A sample trivial question would be: *how many lines has the program?* However, non-trivial properties of a program (such as data provenance) can only be adressed if the program actually is executed.

This gives rise to the following dilemma: to embrace a rich SQL dialect, we want to be Turing-complete (i.e., compute anything). Regarding program analysis, however, we want to avoid Turing completeness and its implications formulated in the theorem of Rice. **The approach illustrated next allows us to have the cake and eat it, too. It allows us to stay in the Turing-complete computation model during runtime and to switch into a weaker computation model for provenance analysis.**

compounds

	compound	formula
t_5	citrate	$C_6H_5O_7^{3-}$
t_6	glucose	$C_6H_{12}O_6$
t_7	hydronium	H_3O^+

fsm

	source	labels	target	final
t_8	0	A..Za..z	1	false
t_9	1	A..Za..z0..9	1	true
t_{10}	1	0..9	2	true
t_{11}	1	+-	3	true
t_{12}	2	0..9	2	false
t_{13}	2	+-	3	false
t_{14}	3	A..Za..z	1	true

(a) Database instance.

```
WITH RECURSIVE
run(compound, step, state, formula) AS (
  SELECT compound, 0, 0, formula
  FROM   compounds
 UNION ALL
  SELECT this.compound, this.step + 1 AS step,
         edge.target AS state, right(this.formula, -1) AS formula
  FROM   run AS this, fsm AS edge
  WHERE  length(this.formula) > 0
  AND    this.state = edge.source
  AND    strpos(edge.labels, left(this.formula, 1)) > 0
)
SELECT r.step, r.state, r.formula
FROM   run AS r
WHERE  r.compound = 'citrate'
```

(b) Recursive SQL query driving the FSM.

output

step	state	formula
0	0	$C_6H_5O_7^{3-}$
1	1	$_6H_5O_7^{3-}$
2	1	$H_5O_7^{3-}$
3	1	$_5O_7^{3-}$
4	1	O_7^{3-}
5	1	$_7^{3-}$
6	1	3-
7	2	-
8	3	

(c) Parsing trace.

Figure 2: Advanced query example and provenance markers.

```
1  data =
     [{"compound": "citrate", "formula": "C6H5O7^3-"},
      {"compound": "glucose", "formula": "C6H12O6"},
      {"compound": "hydronium", "formula": "H3O^+"}];
2  res = [];
3  foreach row in data do
                                          put(log_cf, true)
                                          put(log_ix, idxOf(row, data))
4    c = row["compound"];
                                          put(log_ix, "compound")
5    if c == "glucose" then
                                          put(log_cf, true)
6      t = {"formula": row["formula"]};
                                          put(log_ix, "formula")
7      append(res, t)
                                          put(log_ix, idxOf(t, output))
8    else
                                          put(log_cf, false)
9      skip
10   fi
11 od
                                          put(log_cf, false)
12 // res: [{"formula", "C6H12O6"}]
```

Figure 5: The translated and instrumented SQL query.

3.1 Two-Step Program Analysis

To make this switch possible we run consecutive dynamic and static analyses (compare Figure 4).

During dynamic analysis, the behavior (*not:* result) of certain program statements is recorded in logs. For example, an **if**-statement can branch into the **then**- or the **else**-block. We record this (binary) decision. During static analysis, this makes the behavior of an **if**-statement predetermined. The **if** does no longer actively contribute to the computation and can be replaced by the according **then**- or **else**-branch.

When applying this *record & replace* discipline for a relevant subset of a program's statements, we get an equivalent form of the original program computing the same result. But now, the computation model has been simplified and is open for running an exhaustive program analysis. In the remainder of this section we explain the two analysis steps in detail.

3.2 Dynamic Analysis

As motivated above, we aim to record the behavior of program statements during runtime. The following two logs are appended to:

- log_{cf} (control flow): which/how often does a certain code branch get executed by **if** and **foreach**?
- log_{ix} (indices): at which locations are elements inside lists/dictionaries accessed?

During runtime, these properties are available and can easily be recorded. We use the technique of code instrumentation to create the two logs.

For an instrumented example, see Figure 5. The instrumentation instructions are placed on the righthand side of the listing. The first argument of the *put*()-function is the type of log we want to append to. Its second argument is the actual value being logged. Figure 6 lists the according logs. These are written (and read) sequentially and do not need any further meta-data, keeping the logs small.

The logged data items are to be interpreted in the context of the (uninstrumented) source code. For example, the first entry of log_{cf} corresponds to the first control flow decision in the program at line 3. The **foreach** loop opened there can either execute its body (another time) or terminate and continue at the statement after line 11. We encode these decisions using Boolean values. The first **true** found in the log indicates that the body has been executed. The last **false** indicates that the **foreach** loop has exited. Similarly, an **if**-statement can decide between **then** (yields **true**) or **else** (yields **false**).

List/dictionary element accesses get logged in log_{ix}. Note that **foreach** and append implicitly use numeric indices to read/write from/into lists and need to be included. The `idxOf()` function retrieves the ordinal position of a list element.

log_{cf}	log_{ix}
⟨ **true**,	⟨ 0,
false,	"compound",
true,	1,
true,	"compound",
true,	"formula",
false,	0,
false ⟩	2,
	"compound" ⟩

Figure 6: Log contents.

3.3 Static Analysis

Our static analysis does an abstract (value-less) interpretation of the uninstrumented source code. Instead of computing values, all input values are replaced by unique numeric identifiers. These *pids* are propagated during program interpretation and successively create a variable environment containing the data provenance information. Based on the basic query example of Figure 1 we present a simplified subset of our provenance derivation algorithm.

Figure 8 shows provenance inference rules denoted in operational semantics. The top STATEMENTS rule is the entry point for the interpretation. It takes the first statement s out of all statements ss to be interpreted. In general, interpretation of statements is triggered by the $\Rrightarrow$ symbol and leads to an update of the current variable environment Γ. The CF symbol represents the current data provenance for the control flow. The idea behind this is that reaching a certain code section depends on a number of branching decisions carried out by **if**/**else** statements. The dependencies for these decisions are collected in CF and propagated during program interpretation.

The numeric ids which represent a data provenance relationship are defined in Figure 7. There are the two kinds pid_e and pid_y which stand for *Where*- and *Why*-provenance, respectively. During analysis, these ids are created by the *new*()-function (for an example, see the LIT-STR rule). Initially, all *pids* are of the *Where*-type because any pid_e represents a certain value and a location of origin. During interpretation, they may be converted into *Why*-type using function $\Upsilon()$.

The main data structure is P. It can represent any value of any type of our programming language. Its second component e is used for container types (i.e., lists/dictionaries) to store contained elements. The first component c is used for both, containers as well as atomic values (e.g., strings). It represents the provenance for that value itself. The logs log_{cf} and log_{ix} are read by the inference rules. See rules IF-TRUE and IF-FALSE, for example. The *popf*()-function reads and removes the first element of the according log.

The inference rules presented in Figure 8 are suitable to compute the data provenance of the basic query and finally yield the environment shown in Figure 9. As the

$$
\begin{array}{ll}
P := \langle c, e \rangle & l := \textit{any identifier} \\
c := \{pid_1, ..., pid_n\} & \gamma(P) := c \\
pid \in \{1_e, 1_y, 2_e, 2_y, 3_e, 3_y, ...\} & \epsilon(P) := e \\
e := \{l_1 \mapsto P_1, ..., l_n \mapsto P_n\} & \Upsilon(pids) := \{pid_y : pid_{e|y} \in pids\}
\end{array}
$$

Figure 7: Data structures used in provenance computation.

Statements
$$\frac{CF;\Gamma \vdash s \Rrightarrow \Gamma_1 \quad CF;\Gamma_1 \vdash ss \Rrightarrow \Gamma_2}{CF;\Gamma \vdash s\ \texttt{;}\ ss \Rrightarrow \Gamma_2}$$

PutVar
$$\frac{CF;\Gamma \vdash e \Rrightarrow P \quad \Gamma_{res} = \Gamma + \{v \mapsto P\}}{CF;\Gamma \vdash v\ \texttt{=}\ e \Rrightarrow \Gamma_{res}}$$

Skip
$$\frac{}{CF;\Gamma \vdash \texttt{skip} \Rrightarrow \Gamma}$$

If-True
$$\frac{\begin{array}{c} popf(\mathsf{log}_{cf}) \quad CF;\Gamma \vdash e \Rrightarrow P_e \\ CF_{if} = \Upsilon(CF \cup \gamma(P_e)) \quad CF_{if};\Gamma \vdash ss_1 \Rrightarrow \Gamma_{res} \end{array}}{CF;\Gamma \vdash \texttt{if}\ e\ \texttt{then}\ ss_1\ \texttt{else}\ ss_2\ \texttt{fi} \Rrightarrow \Gamma_{res}}$$

If-False
$$\frac{\neg popf(\mathsf{log}_{cf}) \quad \ldots^{2}}{CF;\Gamma \vdash \texttt{if}\ \ldots\ \texttt{fi} \Rrightarrow \Gamma_{res}}$$

Foreach-False
$$\frac{\neg popf(\mathsf{log}_{cf})}{CF;\Gamma \vdash \texttt{foreach}\ \ldots\ \texttt{od} \Rrightarrow \Gamma}$$

Foreach-True
$$\frac{\begin{array}{c} popf(\mathsf{log}_{cf}) \quad CF;\Gamma \vdash e \Rrightarrow P_e \quad P_{el} = \epsilon(P_e)[popf(\mathsf{log}_{ix})] \\ \Gamma_{for} = \Gamma + \{v \mapsto \langle \gamma(P_e) \cup \gamma(P_{el}), \epsilon(P_{el})\rangle\} \\ CF;\Gamma_{for} \vdash ss\ \texttt{;}\ \texttt{foreach}\ v\ \texttt{in}\ e\ \texttt{do}\ ss\ \texttt{od} \Rrightarrow \Gamma_{res} \end{array}}{CF;\Gamma \vdash \texttt{foreach}\ v\ \texttt{in}\ e\ \texttt{do}\ ss\ \texttt{od} \Rrightarrow \Gamma_{res}}$$

Append
$$\frac{\begin{array}{c} CF;\Gamma \vdash e \Rrightarrow P_e \quad P_v = \Gamma[v] \\ P = \langle \gamma(P_v), \epsilon(P_v) + \{popf(\mathsf{log}_{ix}) \mapsto P_e\}\rangle \quad \Gamma_{res} = \Gamma + \{v \mapsto P\} \end{array}}{CF;\Gamma \vdash \texttt{append}(v,e) \Rrightarrow \Gamma_{res}}$$

GetVar
$$\frac{P = \Gamma[v] \quad P_{res} = \langle \gamma(P) \cup CF, \epsilon(P)\rangle}{CF;\Gamma \vdash v \Rrightarrow P_{res}}$$

Lit-Str
$$\frac{P_{res} = \langle \{new()\} \cup CF, \varnothing\rangle}{CF;\Gamma \vdash c \Rrightarrow P_{res}}$$

GetVar-Idx
$$\frac{\begin{array}{c} P = \epsilon(\Gamma[v])[popf(\mathsf{log}_{ix})] \quad CF;\Gamma \vdash e \Rrightarrow P_e \\ P_{res} = \langle \gamma(P) \cup CF \cup \gamma(\Gamma[v]) \cup \Upsilon(\gamma(P_e)), \epsilon(P)\rangle \end{array}}{CF;\Gamma \vdash v\texttt{[}e\texttt{]} \Rrightarrow P_{res}}$$

Lit-Dict
$$\frac{\left|CF;\Gamma \vdash e_i \Rrightarrow P_i\right|_{i=0\ldots n} \quad P_{res} = \langle \{new()\} \cup CF, \{\left|\ell_i \mapsto P_i\right|_{i=0\ldots n}\}\rangle}{CF;\Gamma \vdash \texttt{\{}\ell_0\texttt{:}e_0,\ldots,\ell_n\texttt{:}e_n\texttt{\}} \Rrightarrow P_{res}}$$

Lit-List
$$\frac{\left|CF;\Gamma \vdash e_i \Rrightarrow P_i\right|_{i=0\ldots n} \quad P_{res} = \langle \{new()\} \cup CF, \{\left|i \mapsto P_i\right|_{i=0\ldots n}\}\rangle}{CF;\Gamma \vdash \texttt{[}e_0,\ldots,e_n\texttt{]} \Rrightarrow P_{res}}$$

BinOp
$$\frac{CF;\Gamma \vdash e_1 \Rrightarrow P_1 \quad CF;\Gamma \vdash e_2 \Rrightarrow P_2 \quad P_{res} = \langle \gamma(P_1) \cup \gamma(P_2), \varnothing\rangle}{CF;\Gamma \vdash e_1 \circledast e_2 \Rrightarrow P_{res}}$$

Figure 8: Inference rules for data provenance.

main result, we find four provenance relationships located in `res[0]["formula"]`. The highlighted pids 5_e (relates to t_2: $C_6H_{12}O_6$) and 4_y (relates to t_2: glucose) constitute the data provenance visualized in Figure 1. 6_y and 10_y do not correspond to table cells and may be ignored. We already presented a visualization prototype in a recent demo paper [8].

3.4 Related Work

The strongest group of related work builds upon provenance propagation through query transformation on the algebraic layer. For example, there is the *Provenance Semirings* approach [7] as well as the *PERM* system [6]. In more recent work, both of them were extended to support aggregations and subqueries, respectively.

[2]Analogous to If-True: ss_2 is interpreted.

$$\begin{array}{l} \texttt{data}: \langle \{10_e\}, \{\texttt{0} \mapsto \langle \{3_e\}, \texttt{"compound"} \mapsto \langle \{1_e\}, \varnothing\rangle, \\ \qquad\qquad\qquad\qquad \texttt{"formula"} \mapsto \langle \{2_e\}, \varnothing\rangle\rangle, \\ \qquad\quad \texttt{1} \mapsto \langle \{6_e\}, \texttt{"compound"} \mapsto \langle \{4_e\}, \varnothing\rangle, \\ \qquad\qquad\qquad\qquad \texttt{"formula"} \mapsto \langle \{5_e\}, \varnothing\rangle\rangle, \\ \qquad\quad \texttt{2} \mapsto \langle \{9_e\}, \texttt{"compound"} \mapsto \langle \{7_e\}, \varnothing\rangle, \\ \qquad\qquad\qquad\qquad \texttt{"formula"} \mapsto \langle \{8_e\}, \varnothing\rangle\rangle\}\rangle \\[1ex] \texttt{row}: \langle \{9_e, 10_e\}, \{\texttt{"compound"} \mapsto \langle \{7_e\}, \varnothing\rangle, \\ \qquad\qquad\qquad \texttt{"formula"} \mapsto \langle \{8_e\}, \varnothing\rangle\}\rangle \\[1ex] \texttt{c}: \langle \{7_e, 9_e, 10_y\}, \varnothing\rangle \\[1ex] \texttt{t}: \langle \{4_y, 6_y, 10_y\}, \{\texttt{"formula"} \mapsto \langle \{5_e, 4_y, 6_y, 10_y\}, \varnothing\rangle\}\rangle \\[1ex] \texttt{res}: \langle \varnothing, \texttt{0} \mapsto \langle \{4_y, 6_y, 10_y\}, \\ \qquad\qquad \{\texttt{"formula"} \mapsto \langle \{5_e, 4_y, 6_y, 10_y\}, \varnothing\rangle\}\rangle\rangle \end{array}$$

Figure 9: Resulting environment Γ after static analysis. Pids of non-input-values ($> 10_{e|y}$) got dropped.

The aforementioned algebraic approaches are all limited in their expressiveness and extending the number of supported algebraic operators is non-trivial.

4. CONCLUSIONS

The approach presented in this article pushes the boundaries of the provenance analysis for SQL queries. Our prototype can analyse queries with advanced but timely SQL language features. Due to Turing-completeness, this approach can deal with any (non-updating) query translated into imperative code.

It is part of our future work to run this approach in the environment of a decent DBMS. In parallel, we pursue the derivation of *How*-provenance [3], i.e. get each one of the computed provenance relations associated to the SQL clauses accountable for its existence.

5. REFERENCES

[1] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *Proc. ICDT*, 2001.

[2] J. Cheney. Program Slicing and Data Provenance. *IEEE Data Engineering Bulletin*, 30(4):22–28, 2007.

[3] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases*, 1(4), 2007.

[4] Y. Cui, J. Widom, and J. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM TODS*, 25(2), 2000.

[5] B. Dietrich, T. Müller, and T. Grust. The best bang for your bu(ck)g. In *Proc. EDBT*, 2016.

[6] B. Glavic and G. Alonso. Perm: Processing Provenance and Data on the Same Data Model Through Query Rewriting. In *Proc. ICDE*, 2009.

[7] T. Green, G. Karvounarakis, and V. Tannen. Provenance Semirings. In *Proc. PODS*, 2007.

[8] T. Müller and T. Grust. Provenance for SQL Based on Abstract Interpretation: Value-less, but Worthwhile. In *Proc. VLDB*, Hawaii, USA, 2015.

[9] T. Neumann. Efficiently Compiling Efficient Query Plans for Modern Hardware. In *Proc. VLDB*, 2011.

[10] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4), 1984.