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
type 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
necessarily 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 $\mathbf{1}$ represents
an inquiry for the data provenance of table cell $t_4: \text{C}_6\text{H}_{12}\text{O}_6$.
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 consequently finds the result being why-dependent on tuple $t_2: \text{glucose}$ and being where-dependent on $t_4: \text{C}_6\text{H}_{12}\text{O}_6$.
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 $G_7^-$ (mouse pointer $\mathbf{2}$). The
accoding highlights within the compounds table (citrate)

SELECT formula FROM compounds
WHERE compound = ’glucose’

$\begin{array}{|c|c|}
\hline
\text{compound} & \text{formula} \\
\hline
\text{citrate} & \text{C}_6\text{H}_{12}\text{O}_6 \\
\text{glucose} & \text{C}_6\text{H}_{12}\text{O}_6 \\
\text{hydronium} & \text{H}_3\text{O}^+ \\
\hline
\end{array}$

(a) Database instance. (b) Query and result.

Figure 1: Basic query example and provenance markers.
and (\(\text{C}_6\text{H}_12\text{O}_7\)) 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.

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 hand-crafted 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. 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.

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 addressed 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.

(a) Database instance.

(b) Recursive SQL query driving the FSM.

(c) Parsing trace.

Figure 2: Advanced query example and provenance markers.

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).

Figure 4: Overview of the two-step analysis.
data =
    ["compound": "citrate", "formula": "C₆H₅O₇⁻", "compound": "glucose", "formula": "C₆H₁₂O₆"],
    ["compound": "hydronium", "formula": "H₃O⁺"]);

res = [];
foreach row in data do
    c = row["compound"];
    if c == "glucose" then
        <true, "compound", 1, true, "compound", false, 0, false>
    else
        skip
    end
end

// res: [{"formula", "C₆H₅O₇⁻"}]

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 \( \text{not: result} \) 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:

- \( \text{log}_{cf} \) (control flow): which/how often does a certain code branch get executed by if and foreach?
- \( \text{log}_{ix} \) (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 \text{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 \( \text{log}_{cf} \) 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 \( \text{log}_{ix} \). 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.

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.

During analysis, these ids are created by the according statement with the form of the original program computing the same result. The log contains.

\[ \text{P := } (c, e) \]
\[ \text{c := } \{ \text{pid}_1, ..., \text{pid}_n \} \]
\[ \text{γ(P)} := c \]
\[ \text{pid} \in \{ 1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3 \} \]
\[ \text{e := } \{ l_1 \mapsto P_1, ..., l_n \mapsto P_n \} \]
\[ \text{γ(pids)} := \{ \text{pid} : \text{pid} \notin \text{pids} \} \]

Figure 6: Log contents.

Figure 7: Data structures used in provenance computation.
The strongest group of related work builds upon prove-
cative 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.

4. CONCLUSIONS

The approach presented in this article pushes the bound-
aries of the provenance analysis for SQL queries. Our proto-
type can analyse queries with advanced but timely SQL lan-
guage 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


3.4 Related Work

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