# Making Provenance Work for You

*Barb, Emery, Aaron, Margo, Elizabeth, Thomas, Matt, Luis, Joe, Orenna, Khanh, others?*

**Abstract** To be useful, scientific results must be reproducible and trustworthy. Data provenance—the history of data and how it was computed—underlies reproducibility of and trust in data analyses. Our work focuses on collecting data provenance from R scripts and providing tools that use the provenance to increase the reproducibility of and trust in analyses done in R. Specifically, our "ProvTools" use data provenance to: document the computing environment and inputs and outputs of a script's execution; support script debugging and exploration; and explain differences in behavior across repeated execution of the same script. Use of ProvTools can help both the original author and later users of a script reproduce and trust its results.

Here is a comment – Barb

## Introduction

In today's data-driven world, an increasing number of people are finding themselves needing to perform data analyses in the course of their work. Often these people have not been educated in programming [**Aaron:** *We train animals, we educate people* –Maya Angelou] and may think of programming as a means to an end, where they are interested in the results but find the programming itself to be tedious. Working with data in this way is often exploratory. A script may be written to produce a plot that allows a visual understanding of the data, which might then lead to a realization that the data needs to be cleaned to remove bad values, and statistical tests need to be performed to determine the strength or trends of relationships. Examining these results may raise more questions and lead to more code. This type of exploratory programming can easily lead to scripts that grow over time to include useful code, but also code that is no longer relevant, with the result being a collection of code that is difficult to understand, debug, or modify.

On top of this, computing environments change quickly. New versions of operating systems, libraries and programming languages are constantly coming out. In an ideal world [**Aaron:** only FORTRAN's, heh heh] everything would be backwards-compatible, but in reality, what worked last week might stop working next week. It can be difficult to determine what went wrong, especially if programming is an occasional activity.

In this paper, we introduce ProvTools: an evolving set of R packages that use data provenance to help users save workable copies of their scripts and data, debug their scripts, understand how their output data was derived, discover what has changed when a script stops working, and facilitate reproducibility.

[**Barb:** I am reluctant to call them ProvTools. That is the name of the github organization(?) that we are not maintaining.]

[**Emery:** Agreed. Maybe "Provenance Tools for R"? This is the term we use on our GitHub web page.]

[**Aaron:** ok, but it'd be nice to have a catchier phrase than "Provenance Tools for R". Isn't ProvTools **our** github org? Why aren't we maintaining it?]

[**Emery:** Our GitHub org is "End-to-end-provenance". ProvTools was set up separately by Matt.]

[**Aaron:** Matt is part of the project, right? Why don't we subsume it/continue to use it? ]

[**Aaron:** it's not just scientists who need these tools...] [**Margo:** Agree; in fact I think the emphasis should be on people for whom the programming/code is incidental – the code is just a means to an end. Also, I think we should broaden the challenge, since we are focusing on all the things you can do with provenance that aren't reproducibility – no one has adopted provenance for reproducibility; based on the abstract, the goal of this paper is to help research programmers [a term used by Philip Guo] do their job more efficiently.] So, what are those challenges: 1) scripts grow by accretion and over time the programmer loses an overall view of everything in the script, 2) debugging is time consuming and tedious, 3) parameter tweaking takes a long time, 4) software changes and it's never clear whether those changes affect you, 5) others?] [**Barb:** I have rewritten the intro to address these comments. Better?] [**Aaron:** yes]

[**Emery:** To reap the advantages of provenance, beyond the simple case of working with one's own scripts and data, there needs to be community buy-in for collecting provenance and community standards for how to encode it. So we might add a plug to that effect, perhaps at the end of the paper.]

[**Emery:** Some general observations:

(1) It seems to me the one incontrovertible use of provenance is in trying to replicate a past result.

Without provenance, we may be reduced to time-consuming and potentially unsuccessful trial and error. With provenance, we have a good chance of replicating the original environment, though that chance may fade over time. If provenance was collected, provSummarizeR extracts the essential information in a useful format and provExplainR can help spot differences between executions.

(2) R has pretty good native support for debugging with its ability to highlight and run selected lines of code and to inspect variable values at the command line. Time-traveling debugging with provDebugR also seems promising, especially in cases where there is a significant cost to rerunning a script. We should promote it here as a promising idea, though of course time will tell whether it catches on.

(3) Simplifying code with provClean is also a novel and promising application, though to be really useful I think it needs to be extended, as Barb has suggested, beyond identifying the code needed for a single result to identifying the code needed for a set of results.

(4) When we make the pitch for adopting provenance and encoding standards, we might suggest starting with a community standard for encoding the information returned by provSummarizeR. If there were a such a standard, repositories like ours might begin to use it.]

[**Aaron:** one way to think about this would be to routinely have a `prov` directory within a github repo, to accompany the more frequent `bin, doc, results, source`]

## Motivating Example

[**Barb:** Aaron, can you work on this section? I think you can make a more believable example than I can.]

[**Aaron:** probably. let's talk about what kind of example you're thinking about at our next conference call. Also, what's the difference between a "motivating example" and a "use case"? I think I understand from Margo's comments below on Use cases, but I'm not certain.]

Kim has been studying blah blah and has a growing suspicion that there is a relationship between x and y. She has found a reliable source of data and has developed an R script to analyze the data. While discussing this with her colleague Fred over lunch, he tells her about another good source of data for her study. [**Barb:** Obviously, need something other than blah blah and x and y.] [**Aaron:** we could put it in a Bayesian context, where Kim is using an uninformative (flat) prior and Fred points her at another dataset that could sharpen (inform) her prior.]

Excited by this new opportunity, she downloads the data and tries to load them [**Aaron:** datum, **noun, singular**; data **noun, plural**] into her script. Unfortunately, the data are not stored in the same way, and contain extra information she does not need [**Aaron:** this equates/conflates data with dataset. Intentional? Standard usage? From the prov perspective, they are both objects...and that is pointed out in the "What is data provenance" section, below]. With some effort, she is able to modify her script so that it can read both datasets and combine them in such a way that her script can analyze them.

Things seem to be going well. She continues to work on her analysis and modifies her script to produce a plot showing the relationship. With the first dataset there seemed to be a strong relationship between x and y, but with the addition of the second dataset, it's not clear there is a relationship at all. What has gone wrong? [**Aaron:** nothing has "gone wrong" unless Kim was so wedded to her preconceived notion about the hypothesis implied by the relationship in her data, or so convinced that her self-worth is tied up in a *P* value that she is not open to new information. ]

Kim had the foresight to collect data provenance as she was working, so she starts up provDebugR to examine the calculations the script had done, thankful that the provenance-based debugger allows her to examine the data values after the time-consuming downloading and cleaning steps instead of needing to rerun her script from the beginning to see these intermediate values. Doing this, she realizes that the values in the second dataset are much lower than those of the first dataset. She goes back to the website from which she downloaded the second dataset to get a better understanding of how they collected their data. As she examines their methodology, she finds the problem. They used different units than in the first dataset. Returning to her script, she adds the instructions to do the necessary conversion and now the second dataset corroborates the results of the first dataset. [**Aaron:** lucky her. in the real world, most second datasets fail to support the first. Hence the 's "reproducibility crisis"].

[**Aaron:** more seriously, though. The point of our tools is that they collect the prov silently so that Kim doesn't need to have had the foresight to collect the provenance. She just needs to use the tools, which have been collecting prov for her.]

With the positive results in hand, Kim writes a paper, being careful to save the script, data, and data provenance so that she will be certain to be able to address any questions the reviewers may have. After several months, Kim hears from the reviewers. The reviews are quite positive, however, the

reviewers have requested that the plots be presented differently.

Kim pulls up the saved script and begins making the requested changes. Unfortunately, things do not go smoothly. With the passage of time, Kim has forgotten how the script works and wishes she had done a better job of commenting and cleaning up the script. Now, she gets an inscrutable error just trying to rerun the code. She is still using the same data and the same script. Why doesn't it work? [**Aaron:** but she had the foresight to collect prov (above) so why didn't it help her here? Seems forced. If she were using provtools that collect the prov silently, this would be moot]

Kim uses provExplainR to compare the data provenance from when she submitted the paper to the data provenance of the current failed execution. In doing this she discovers that she is running a different version of R and that she has downloaded a new version of the spiffy-stats package [**Aaron:** hardly spiffy. RJ would not appreciate this, given that you haven't specified "spiffy-stats" library earlier]. Could one of these be the cause of her problems? Looking more closely at the error, she sees that it is coming from a call to a function in the spiffy-stats package. Using provDebugR's StackOverflow feature, she quickly learns how other users have resolved the error she is getting. Fortunately, this turns out to be easy to fix and Kim gets her script running again.

In this example, we have shown data provenance as providing a foundation that tools can use to help a programmer understand the behavior of their scripts. In the remainder of the paper, we elaborate on what we mean by data provenance and the tools that we provide to collect and use provenance for R scripts.

## What is data provenance?

[**Barb:** This section has been rewritten.]

Data provenance is the history of a data item ("datum") or a dataset ("data"), describing how it came into existence. In the context of databases, data provenance is generally used to understand results of queries to a database. In the context of file systems, provenance generally refers to the processes that were used to create or [**Aaron:** "or" includes "and", hence "and/or" is redundant] modify a file. In contrast to these, our focus is on *language-level provenance*, how data are created and manipulated by a programming language during the execution of a script or program. Our particular focus is on the R language and software system because of its popularity among data analysts. For brevity, when we say "provenance" or "data provenance" in this paper, we mean language-level provenance.

We associate three types of information with provenance: environment information, coarse-grained information, and fine-grained information. *Environment information* includes information about the computing environment in which the script was executed. This includes information such as the operating system version, the R version, and the versions of the R libraries used, as each of these may play a role in understanding the details of how a script behaves. *Coarse-grained information* includes the source code of the script(s), the data input to the script, the data output by the script, and plots produced by the script. *Fine-grained information* includes an execution trace. Specifically, for each line of the script that is executed, it includes the data used on that line as well as the data computed by that line. These input and output values are chained together to allow for a detailed lineage of an output value to be retrieved.

[**Emery:** Or maybe "summary provenance" and "detailed provenance"?]

Consider this simple example that loads in the cars dataset and plots mpg against cylinders.

```
1  # Load the mtcars data set that comes with R
2  data (mtcars)
3
4  # All the cars
5  allCars.df <- mtcars
6
7  # Create separate data frames for each number of cylinders
8  cars4Cyl.df <- allCars.df[allCars.df$cyl == 4, ]
9  cars6Cyl.df <- allCars.df[allCars.df$cyl == 6, ]
10 cars8Cyl.df <- allCars.df[allCars.df$cyl == 8, ]
11
12 # Create a table with the average mpg for each # cylinders
13 cylinders = c(4, 6, 8)
14 mpg = c(mean(cars4Cyl.df$mpg), mean(cars6Cyl.df$mpg), mean(cars8Cyl.df$mpg))
15 cyl.vs.mpg.df <- data.frame (cylinders, mpg)
16
```

```
17 # Plot it
18 plot(cylinders, mpg)
```

The environment information includes information such as that the script was executed on a 64-bit x86 Mac, running R 3.6.0. The coarse-grained information identifies the location of the mtcars dataset in the file system, and saves a copy of the script, the original dataset input, and the plot produced. The fine-grained information records the input and output data for each line of code, linking these together so that we can see how the values computed in one statement are used in a later statement. For example, the fine-grained information shows that the value assigned to cars4Cyl.df on line 8 is used on line 14 when calculating mpg. Further, with our tools, we could determine the forward lineage of cars4Cyl.df, or how cars4Cyl.df is used, to see which lines of code depend on this value:

```
8  cars4Cyl.df <- allCars.df[allCars.df$cyl == 4, ]
14 mpg = c(mean(cars4Cyl.df$mpg), mean(cars6Cyl.df$mpg), mean(cars8Cyl.df$mpg))
15 cyl.vs.mpg.df <- data.frame (cylinders, mpg)
18 plot(cylinders, mpg)
```

Alternatively, we could determine the backward lineage of cars4Cyl.df to see how its value was computed:

```
2  data (mtcars)
5  allCars.df <- mtcars
8  cars4Cyl.df <- allCars.df[allCars.df$cyl == 4, ]
```

[**Barb:** This is as far as I have gotten with updates.]

## Use cases

Collecting fine-grained data provenance is a challenging task, but it has little benefit if the provenance is never used. In this section, we describe various ways that provenance can be used to help the scientist or data analyst with their job. [**Margo:** Use cases are typically specific things a user wants to do, not broad categories ; the walkthrough example should allow us to describe things that a user wants to do with the script in question and then generate the categories of the types of things users do. Alternately, you can say that these are categories and then translate them to use cases.]

[**Aaron:** I assume that file-level provenance also has little benefit if it is never used. Will we have use cases for both file level and fine-grained provenance?]

[**Margo:** I recommend constructing a simple example that we can use to demonstrate all the use cases – it should be something real-ish, but small enough to present both the script and meaningful provenance snippets.]

[**Emery:** I like the use case of trying to verify a published result. Suppose we come across an interesting paper and the author has archived his or her data, R script, and provenance. We want to verify the author's results, so we first use provSummarizeR to make sure we have access to all of the original inputs and are aware of any differences in computing environment. We run the script and get a different result. So then we use provExplainR to explore the differences more closely. Perhaps the version of R or of one of the R libraries is different. Or maybe the input files are not the same or the data retrieved from a URL has changed. If provExlainR identifies differences in the provenance graphs, we use provDebugR to look more closely at intermediate data values and steps executed at the point where the graphs begin to diverge. We could also use provViz to see how the execution pathways might diverge at that point.]

### Documentation

[**Margo:** I think this is conflating two things: documentation which should precisely mirror the definition used in the rest of the world and program comprehension, which is a term used to describe understanding what a program is doing at a somewhat deeper level.] [**Aaron:** this subsection comes sort of out of nowhere. Could use a better transition from something. Maybe it belongs in the "what is data provenance" section, above] Fundamentally, data provenance is documentation. What data were used as input? What script was used to process the input? What version of R and user-contributed packages were used to interpret the script? On what computing platform? Just as with the provenance of a piece of art, the purpose of documentation at this level is to build trust. You wouldn't buy a used car without checking its Carfax report. Data provenance is essentially the Carfax report for

data. The provSummarizeR tool described below provides this type of report. [**Thomas:** Bad cultural expectation. I don't think anyone outside of the US knows Carfax. This sort of things should be avoided.] [**Aaron:** yep]

### Debugging

[**Aaron:** ditto here. I'm not following the organization of the subsections under the Use cases section. However, this one seems like if we included a use-case it could fit well here] Given fine-grained provenance, it is possible to examine the values of variables in a script at different points during execution. This can help a data analyst track down errors in the script's execution.

In addition, fine-grained provenance allows the data analyst to ask the questions "how did $x$ get its value?" and "what was computed based on the value of $x$?" In this way, the analyst can focus in on the portions of the script that are relevant to a specific computation. [**Aaron:** is this referring to ProvClean?] [**Emery:** provDebugR and provViz]

We provide the provViz tool described below to allow a graphical examination of the fine-grained provenance. The provDebugR tool, under development, provides a more standard command-line debugger interface to the same information.

### Reproducibility

A reproducible script is one that can be re-executed and get the same results. Unfortunately, this is surprisingly difficult. With the passage of time, new libraries may be installed which break the script. If you share your script with a colleague, it might not behave the same. The provExplainR tool, described below, can be used to understand what has changed. It compares the provenance collected by two executions of a script and identifies differences in the computing environment, input data, and scripts to help the data analyst understand why a script is no longer working.

Scripts are often developed in an exploratory fashion. When complete, the data analyst may realize that only some of the plots being produced are useful. The provClean tool, under development, uses provenance to determine which code was used to produce the desired output and retain just that code.

[**Emery:** We might expand on this with the example of someone trying to reproduce a result published by someone else (see above). This is (or should be) a fairly common scenario. How would provenance and our tools make this easier? Why might it fail?]

## Related Work [Margo: It's quite possible this section should go at the end after we've described ProvTools]

### Provenance collection

There are many systems that collect provenance and several excellent survey papers on provenance systems (Freire et al., 2008; Herschel et al., 2018; Pimentel et al., 2019). Here we cite only the most similar systems. [**Margo:** If we are going to start with "most similar" then I think we probably want to start with the other R provenance capture (especially for this audience), then talk about noWorkflow and then possibly, reprozip.]

Provenance collection is a common feature of workflow tools such as Kepler (Altintas et al., 2006), Taverna (Zhao et al., 2008), Vistrails (Scheidegger et al., 2008), [**Margo:** These are all over a decade old; is there nothing more recent?] and many others. Workflow systems capture the provenance at the level of workflow steps, where each step itself is typically the application of a computational tool. This provides a course-grained provenance, which is useful when developing large systems by composing existing software tools, as is done with workflow tools.

The provenance collected by `rdtLite` is fine-grained provenance, where the provenance details are recorded at the level of programming statements or functions. Another tool that collects such fine-grained provenance is noWorkflow (Murta et al., 2014) which collects provenance for Python.

There has been previous work in collecting provenance for R. Much of this work collects provenance at the level of files. rctrack (Liu and Pounds, 2014) uses R's trace function to record information about files read and written and the computing environment. It saves copies of data files and scripts with the goal of being able to reproduce a computation. Similarly, recordr (Peter Slaughter, 2018) records information about files read and written and the computing environment. It can also save copies of those files.

adapr (Gelfond et al., 2018) stores hash values of data files with the R code in a github repository. They assume the data themselves are stored elsewhere. Their goal is to be able to confirm that data

match the data used by the code. If the data are modified, the modification will be observable, but the original data cannot be restored by adapr.

While these R provenance systems collect valuable information useful to allow archiving of provenance with data, they do not support fine-grained provenance. [**Margo:** This suggests that fine-grain provenance is somehow useful/important; thus, the use cases above need to demand fine-grain provenance in specific ways so that in this section, one can refer back and explain how the related work being discussed can't do X, and X is an important and useful thing to do.] In contrast, CXXR (Silles and Runnalls, 2010; Runnalls and Silles, 2012) computes fine-grained provenance using a modified R interpreter where the read-eval-print loop is modified to collect provenance. The collected provenance is available interactively but is not stored persistently. This type of provenance can be helpful for debugging but does not support archiving the provenance.

In contrast to these, rdtLite saves information persistently about file inputs and outputs that is useful for archival purposes and also saves fine-grained provenance useful for debugging.

### Reproducibility

Closely related to provenance collection is recording environmental information to support reproducibility. Here, we discuss some systems whose primary focus is reproducibility.

The packrat package (packrat) manages R packages on a per-project basis. In this way, different projects might use different versions of R packages. By keeping track of library versions, it is easier to reproduce results on a different computer.

encapsulator (Pasquier et al., 2018) creates a virtual machine containing curated code that includes the subset of the source code needed to generate a particular artifact, such as a plot, along with R packages and other system dependencies and all input data. The virtual machine can then be easily moved to another computer and re-executed. [**Aaron:** is this being maintained?]

The cacher package (Peng, 2008) bundles data and scripts so that they can be distributed together to other users to reproduce the computation. Additionally, data values are cached in a database during execution after each expression is evaluated. Later, these cached values can be used in place of re-running expensive computations.

The archivist package (Biecek and Kosiński, 2017) stores R objects that can be shared and searched for. When an object is archived, it also stores environment information, such as the versions of R libraries in use. For objects with dependent data, such as plots created with ggplot2, the dependent data is also saved. Additionally, the archivist package defines a new operator %a% that works similarly to the pipe operator from magrittr, except that it also stores the sequence of operations with the archived object. The lineage of objects created with this operator can then be queried.

While these packages support reproducibility and data sharing, except for archivist's %a% operator, they do not support lineage queries. They record environmental information and, in some cases, intermediate data values, but not provenance in general. The encapsulator application is an example of building reproducibility support on top of a general provenance package.

[**Margo:** I feel like this whole section should be condensed into a more synthesized single paragraph that says, "There exist tools for helping with reproducibility, but they all lack ... provTools addresses this shortcoming."]

## The rdtLite package suite

The rdtLite package collects provenance. Some other packages use the collected provenance to satisfy a variety of use cases. Another group of packages facilitate the construction of new tools using the provenance.

The packages that provide user-level functionality are provSummarizer, provDebugR, provViz, provExplainR, and provClean. provSummarizer provides a human-readable summary of the provenance. provDebugR allows debugging of R code going backward and forward through the code without rerunning. provClean can reduce a complicated script to the minimal code necessary to produce a particular output. provViz allows examination of the provenance via a graphical representation of the execution trace.

There are two packages intended for tool developers: provParseR and provGraphR. provParseR reads in the provenance and provides an API for convenient access to parts of the provenance. provGraphR provides an API to answer lineage questions about the collected provenance.

In this section, we describe each of these packages, beginning with provenance collection.

[**Margo:** Since this is just an overview, I wonder if it might better be captured in a table; then we

just say, "Table 1 lists all the tools that comprise the <Name> suite." Second, calling this the rdtLite is, I think, confusing – you've mentioned ProvTools above, rdtLite not at all. If the package is provtools, then that's what this section should be; it then relies on Rdt and Rdtlite as its capture mechanism]

### Collecting provenance

We have written two packages that collect provenance from R: rdt (formerly known as RDataTracker (Lerner and Boose, 2014b), (Lerner et al., 2018)) and rdtLite. rdtLite is available on CRAN and is the package discussed here. rdt is available on github (https://github.com/End-to-end-provenance/rdt). rdtLite is more stable but collects less detailed provenance than rdt does. The additional provenance collected by rdt includes fine-grained provenance internal to functions and internal to control constructs. Both tools output their provenance using the same PROV-JSON format. **[Thomas:** Is the above pertinent to the understanding of the paper?]

### rdtLite

rdtLite can be used to run a script that is stored in an R file, or it can be used to collect provenance as commands are run interactively in the console. To run a script, the `prov.run` function is used:

```
library (rdtLite)
prov.run ("script.R")
```

To run interactively, the commands for which provenance should be collected are surrounded by a call to `prov.init` and `prov.quit`:

```
library (rdtLite)
prov.init ()
data <- read.csv ("mydata.csv")
plot (data$x, data$y)
prov.quit ()
```

The provenance that rdtLite collects is information about each file or URL read by the script, each file written by the script, and each plot created by the script. In addition, an execution trace of the top-level R statements is also saved. This trace identifies the statement executed. It also records any variables set or used by the statement. When a variable is set, the type of the value, including its container (like vector, data frame, etc.), its dimensions, and class (like character, numeric, etc.) re recorded. If the container is a vector of length 1, the data value is stored in the JSON file as well. For larger containers, the value can be saved in a separate snapshot file. The user can control how much data to save using the `snapshot.size` parameter in `prov.init` and `prov.run`. The default is to not save snapshots.

The provenance is stored in an extension of the PROV-JSON standard[1] that supports detailed information about fine-grained provenance [2], such as a list of libraries used, a mapping from functions called to the libraries from which they come, script line numbers, and data values and their types.

The JSON file is stored with a provenance directory that also contains copies of input and output files and the R scripts executed. By default, the provenance data is stored in the R session temporary directory, but the user can change this location either at the time that `prov.run` or `prov.init` is called, or by setting the `prov.dir` option, for example, in the .Rprofile file.

If an R script calls R's `source` function to execute another R script, calls to `source` should be replaced with calls to rdtLite's `prov.source` function. This allows provenance collection to continue within the sourced script. If `prov.source` is not used, the call to the `source` function appears as a single statement in the execution trace, and uses of variables or updates to variables within the sourced script are not recorded.

Upon completion of a script called with `prov.run`, or after a call to `prov.quit`, the user will have a provenance directory whose name is `prov_script`, where "script" is the name of the script file, or `prov_console` in the case of an interactive session. Inside the directory, there will be:

- `prov.json` - the JSON file containing the fine-grained provenance
- `data` folder containing copies of input and output files, URLs, plots created, and snapshot files.

[1] http://www.w3.org/Submission/prov-json
[2] https://github.com/End-to-end-provenance/ExtendedProvJson

- `scripts` folder containing a copy of the main script and any scripts executed by calling `prov.source`.

While the collected provenance can be archived, we do not expect a user to use this collected provenance directly, but rather to use the tools described in the next section to provide different types of functionality.

### Using provenance

Applications that use provenance exist in separate packages from the provenance collector. These tools would work equally well with the provenance collected by other tools that produce the same JSON format.

### provSummarizeR

The provSummarizeR package provides functions that produce a textual summary of the collected provenance.

The provenance summary consists of: [**Margo:** I think this would be much nicer with an annotated display of the output. Ah, you have that below – if you annotate that output, then you dont need to itemize it here. These examples would also be better if they referred back to the example that threads throughout the paper. If Arju were wondering about X, provSummarizr would tell them that Y (highlighted in the output)]

- a description of the execution environment,
- a list of the libraries used and their versions,
- a list of any scripts source'd,
- a list of the input and output files, including their timestamp and hash value
- a list of the errors and warnings encountered, including the message and the line on which the error occurred

For example, consider this simple R script:

```
data <- read.csv ("mydata.csv")
plot (data$x, data$y)
```

The summary of the provenance generated by this short script is shown below.

```
PROVENANCE SUMMARY for script.R

ENVIRONMENT:
Executed at 2018-12-07T14.29.59EST
Script last modified at 2018-12-03T13.49.29EST
Executed with R version 3.5.1 (2018-07-02)
Executed on x86_64 running darwin15.6.0
Provenance was collected with rdtLite 1.0.3
Provenance is stored in ~/tmp/prov/prov_script
Hash algorithm is md5

LIBRARIES:
base 3.5.1
datasets 3.5.1
ggplot2 3.0.0
graphics 3.5.1
grDevices 3.5.1
methods 3.5.1
provSummarizeR 1.0
stats 3.5.1
utils 3.5.1

SOURCED SCRIPTS:
None
```

```
INPUTS:
File : mydata.csv
   2018-12-03 13:50:14
   f777f89d9b76dfda566ea85d24a391e1

OUTPUTS:
File : dev.off.4.pdf
   2018-12-07 14:30:00
   c90d4198fd8e59f7f92a2e69ed5c3701

ERRORS:
None
```

There are three functions provided to the user to generate the summary:

```
prov.summarize(save = FALSE, create.zip = FALSE)
prov.summarize.file(prov.file, save = FALSE, create.zip = FALSE)
prov.summarize.run(r.script, save = FALSE, create.zip = FALSE, ...)
```

`prov.summarize` produces a summary for the last provenance collected in the current R session. `prov.summarize.file` is passed the name of a JSON file containing provenance and produces a summary from it. `prov.summarize.run` is passed the name of a file containing an R script. It runs the script, collects its provenance, and produces a summary.

If the optional parameter save is true, the summary will be saved to a file in the provenance directory in addition to being displayed on the screen. If the optional parameter create.zip is set to true, the provenance directory will be packaged into a zip file.

`provSummarizeR` is available on CRAN.

### provDebugR

The `provDebugR` package provides debugging support by using the provenance to help the user understand the state of their script at any point during execution.

For example, consider this simple, but buggy script:

```
w <- 4:6
x <- 1:3
y <- 1:10
z <- w + y
y <- c('a', 'b', 'c')
xyz <- data.frame (x, y, z)
```

Running this script produces a warning and an error:

```
Error in data.frame(x, y, z) :
  arguments imply differing number of rows: 3, 10
In addition: Warning message:
In w + y : longer object length is not a multiple of shorter object length
```

Of course, with a short script like this, a user could simply step through the script one line at a time and examine the results, but for the purposes of demonstrating the debugger imagine that this code is buried within a large script. The lines of code might not be consecutive as shown here and it may even be difficult to determine what lines caused the reported errors.

The debugger provides some functions that are particularly helpful for understanding warning and error messages. For example, if the user needs help understanding where a warning came from, the debug.warning.trace function is useful. Executing with no parameters lists all the warnings. Then, calling debug.warning.trace with the number of a warning will show the lines of code that led up to the warning.

```
> debug.warning.trace()
Possible results:

1 In  w + y :  longer object length is not a multiple of shorter object length
```

```
> debug.error.trace(stack.overflow = TRUE)
Your Error: Error in data.frame(x, y, z): arguments imply differing number of
rows: 3, 10

[1] "What does the error &quot;arguments imply differing number of rows: x,
y&quot; mean?"
[2] "ggplot gives &quot;arguments imply differing number of rows&quot; error in
geom_point while it isn&#39;t true - how to debug?"
[3] "qdap check_spelling Error in checkForRemoteErrors(val) : one node
produced an error: arguments imply differing number of rows"
[4] "Creating and appending to data frame in R (Error: arguments imply
differing number of rows: 0, 1)"
[5] "SparkR collect() and head() error for Spark DataFrame: arguments imply
differing number of rows"
[6] "strata() from &quot;sampling&quot; returns an error: arguments imply
differing number of rows"

Choose a numeric value that matches your error the best or q to quit:
1
```

**Figure 1:** Using `debug.error.trace`

```
Pass the corresponding numeric value to the function for info on that warning
> debug.warning.trace(1)
[[1]]
  script line       code
1      1    1   w <- 4:6
2      1    3  y <- 1:10
3      1    4 z <- w + y
```

Similarly, the user can get information about what led up to an error using debug.error.trace.

```
> debug.error.trace()
Your Error: Error in data.frame(x, y, z): arguments imply differing number of rows: 3, 10

  script line                      code
1      1    1                  w <- 4:6
2      1    2                  x <- 1:3
3      1    3                 y <- 1:10
4      1    4                z <- w + y
5      1    5     y <- c("a", "b", "c")
6      1    6 xyz <- data.frame(x, y, z)
```

The `debug.error.trace` function has an optional logical parameter, `stack.overflow`. If `TRUE` is passed in for its value, `debug.error.trace` uses the stackexchange API to search Stack Overflow for help with the error message. To use the Stack Overflow API, the error message is first modified to remove references to quoted strings, as those represent things like variable names that are specific to the script. It then uses Stack Exchange's API to search StackOverflow looking for posts that match the query, and sorts them by the number of votes they have received. It then lists the questions asked in the top 6 posts and asks the user to select one. It then opens a browser showing chosen Stack Overflow post.

Figure 1 shows a sample dialog using `debug.error.trace`. Selecting 1 results in the user's browser going to the page displayed in Figure 2. By scrolling down through answers to this question (not shown here), the user will hopefully get helpful advice that will enable them to solve their problem.

Tracing how a result is computed is also available to understand how a variable got its value:

```
> debug.lineage("z")
$z
  script line       code
```

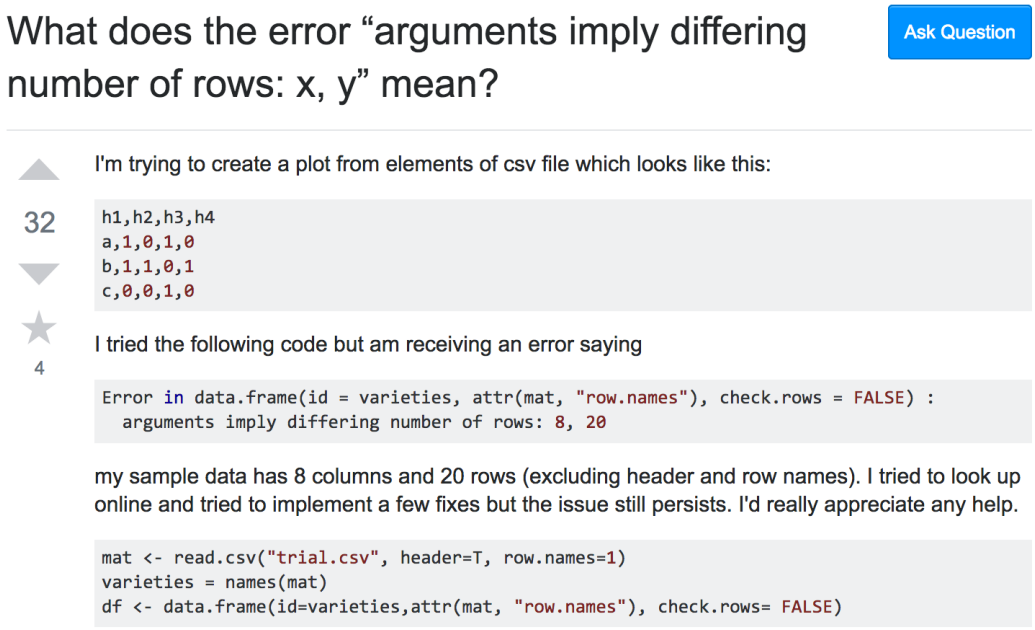


**Figure 2:** Stack Overflow Page to Resolve an Error

```
1      1    1   w <- 4:6
2      1    3  y <- 1:10
3      1    4 z <- w + y
```

This shows the lines of code that led to computing the final value of z. Notice that line 2, the assignment to x is not included since x is neither directly nor indirectly involved in computing the value of z.

A common cause of programming errors in R is caused by automatic type conversions. Consider this simple script:

```
x <- 1
y <- 1:10
x <- x + y
if (x == 2) {
  print ("x is 2")
} else {
  print ("x is not 2")
}
```

Running this script produces the following output:

```
[1] "x is 1"
Warning message:
In if (x == 1) { :
  the condition has length > 1 and only the first element will be used
```

The programmer may be surprised to get this warning message, as the assignment back to x may have been a mistake. Since R is a dynamically-typed language, there is no error at the time of the assignment, but only later when the value is used. An experienced R programmer may realize that this message probably means that x is a vector containing more than a single value, which is not what the programmer expected. The `debug.variable.type` function can be used to confirm this hypothesis and locate where the bad assignment occurred:

```
> debug.variable.type("x")
$x
  script line       scope container dim    type
1      1    1 R_GlobalEnv    vector   1 numeric
2      1    3 R_GlobalEnv    vector  10 numeric
```

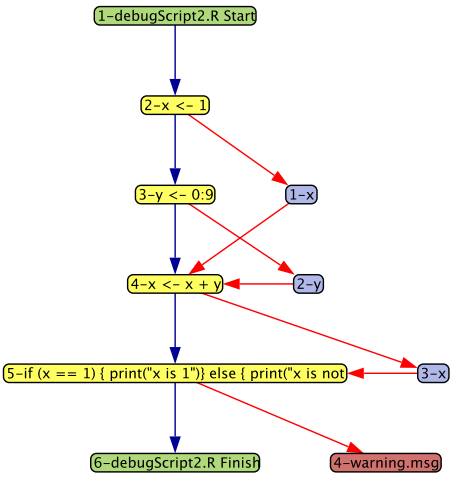


**Figure 3:** A Provenance Graph as Displayed using `provViz`

The `debug.variable.type` function finds all assignments to `x`, identifying where that assignment happened along with typing information, which includes the container (such as vector, list, data frame), the dimensions of the container, and the class of the values within the container. From this output, the programmer can see that on line 1, `x` was assigned a single numeric value and on line 3, its value was changed to be a vector containing 10 numerics. Looking back at the source code, the programmer may realize that the assignment should have been to `y` rather than to `x`.

`provDebugR` is available at https://github.com/End-to-end-provenance/provDebugR.

### provViz

The provenance collected by rdtLite is stored as a graph. There are two types of nodes: data nodes and procedure nodes. Data nodes represent things like variables, files and URLs. Procedure nodes represent executed R statements. An edge from a data node to a procedure node represents a use of the data. For example, this may be on the right hand side of an assignment statement, or passed as an argument to a function. An edge from a procedure node to a data node indicates that the procedure produced the data, for example, by assigning to a variable or writing to a file. An edge between two procedure nodes indicates the order in which the R statements were executed.

The `provViz` package provides functionality that allows the user to view the graph, and explore it to examine intermediate data values, input and output files, and perform lineage queries. Figure 3 shows what a provenance graph looks like when displayed using `provViz`. The node colors indicate the type of node. Data nodes representing variables are displayed in purple. Red data nodes represent warnings and errors. Yellow nodes represent R statements. Green nodes come in pairs and represent the start and end of a group of R statements. Green nodes can be clicked on to reduce the collection of surrounded R statements into a single node, which is useful to make larger graphs more manageable.

The `provViz` package provides three functions:

- `prov.visualize` displays the graph associated with the last provenance collected in the current R session.
- `prov.visualize.file` reads the provenance from a file and displays its graph.
- `prov.visualize.run` runs an R script, collecting its provenance, and displays the graph on completion of the script.

These functions connect to a Java program called DDG Explorer (Lerner and Boose, 2014a), which does the actual work of creating and managing the display. In addition to viewing the graph, DDG Explorer provides the following functionality:

- Viewing of input and output data files
- Viewing of cached copies of data downloaded from a URL
- Viewing of plots created
- Viewing of intermediate data values
- Viewing a subgraph showing the forward or backward lineage of a data node

- Viewing the source code for a node or the entire script
- Comparing R scripts
- Comparing provenance graphs
- Searching for nodes by name and type
- Sorting procedure nodes based on execution time

provViz is available on CRAN.

### provExplainR

The provExplainR tool reads two provenance directories and generates differences between two versions including the environment in which the scripts were executed, versions of attached libraries, and differences in the main and sourced scripts.

To view differences between two provenance directories:

```
prov.explain (
    dir1 = "prov_MyScript_2019-08-06T15.59.18EDT",
    dir2 = "prov_MyScript_2019-08-21T16.25.58EDT")
```

Here is an example of what the comparison result looks like. provExplainR first looks at name and content of the main and sourced scripts, then versions of attached libraries, environment attributes (like architecture, operating systems, script timestamp, etc.), and versions of provenance tool rdt/rdtLite

```
You entered:
dir1 = prov_MyScript_2019-08-06T15.59.18EDT
dir2 = prov_MyScript_2019-08-21T16.25.58EDT

SCRIPT CHANGES:
The content of the main script MyScript.R has changed
### dir1 main script MyScript.R was last modified at: 2019-08-06T15.40.34EDT
### dir2 main script MyScript.R was last modified at: 2019-08-21T15.54.37EDT

Sourced scripts in dir2 but not in dir1:
### HelperScript.R, which was last modified at: 2019-08-21T15.27.44EDT

LIBRARY CHANGES:
Library version differences:
      name dir1.version dir2.version
     dplyr        0.8.1        0.8.3
 rmarkdown         1.13         1.14


Libraries in dir2 but not in dir1:
         name version
      cluster   2.0.8
 provExplainR   0.1.0


Libraries in dir1 but not in dir2:
No such libraries were found

ENVIRONMENT CHANGES:
Value differences:
Attribute: total elapsed time
### dir1 value: 7.68
### dir2 value: 6.917

Attribute: provenance directory
### dir1 value: /Users/khanhl.ngo/HarvardForest/prov_MyScript_2019-08-06T15.59.18EDT
### dir2 value: /Users/khanhl.ngo/HarvardForest/prov_MyScript_2019-08-21T16.25.58EDT

Attribute: provenance collection time
```

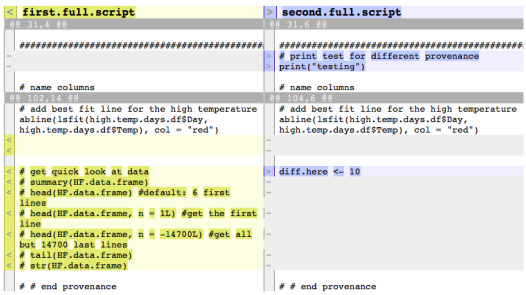


**Figure 4:** Comparing scripts using `provExplainR`

```
### dir1 value: 2019-08-06T15.59.18EDT
### dir2 value: 2019-08-21T16.25.58EDT


PROVENANCE TOOL CHANGES:
Tool differences:
No differences have been detected
```

To view the difference between two scripts in the first and second provenance directories:

```
prov.diff.script (
  first.script = "MyScript.R",
  dir1 = "prov_MyScript_2019-08-06T15.59.18EDT",
  dir2 = "prov_MyScript_2019-08-21T16.25.58EDT")
```

`provExplainR` uses the `diffobj` package to display the changes in a script as shown in Figure 4.

### Developing new provenance-based tools

In addition to end user tools as described above, we have also made available packages intended for programmers interested in developing their own tools incorporating provenance information. In this section, we describe those tools.

### provParseR

The `provParseR` package parses the JSON provenance and provides a convenient API to access portions of the provenance.

To get started the tool developer would call `prov.parse`:

```
prov.parse(prov.input, isFile = TRUE)
```

The `prov.input` parameter is a string that can either be the path to a JSON file containing provenance, or it can be a provenance string itself. The second parameter, `isFile` is used to disambiguate these cases. The default assumption is that `prov.input` is the path to a file. This function returns an object whose class is `ProvInfo`.

The remaining functions provided by `provParseR` are getters that are passed a `ProvInfo` object and return information, typically a data frame containing that portion of the provenance.

For example, `get.input.files` returns a data frame containing a subset of the data nodes that correspond to files or URLs that are read by the script. The data frame that is returned includes the following information:

- id - a unique id
- name - the file name or URL
- value - the path to a saved copy of the file or URL contents
- type - one of File or URL
- hash - the hash value of the file

- location - the absolute path to the file

Another function provided is `get.environment`, which returns a data frame including information about the execution environment, such as the architecture and operating system on which the script was executed, the version of R, and the modification and execution times of the script.

Two functions provide information about the R libraries used. The `get.libs` function returns the name and version of each library. The `get.func.lib` function returns the name of each function called from a library and which library it is from.

Other functions provide information about the R statements executed and the edges between nodes. See the package's help page for a complete list of the functions and what they do.

`provParseR` is available on CRAN.

### provGraphR

The `provGraphR` package provides the ability to make lineage queries over the provenance. To get started, the tool developer should call `create.graph`:

```
create.graph(prov.input = NULL, isFile = TRUE)
```

As with `prov.parse`, the default behavior is for `prov.input` to be the path to a JSON provenance file and for `isFile` to be TRUE. Alternatively, `prov.input` can be a string containing JSON provenance if `FALSE` is passed for `isFile`. The `create.graph` function returns an adjacency matrix representation of the graph.

The adjacency matrix can be passed to the `get.lineage` function to do lineage queries.

```
get.lineage(adj.graph, node.id, forward = F)
```

The second parameter to the function is node.id. Each node in the graph has a unique id. Using the functions provided by the parser, such as `get.input.files`, `get.output.files`, `get.variables.set`, and `get.variables.used`, the tool developer can find the id of the file or a variable to get the lineage of.

The `get.lineage` function can compute either backward or forward provenance. Backward provenance is used to understand how a variable was computed, while forward provenance is used to understand how a variable is used in later computation. These functions can similarly provide information about how input data is used, or how the values stored in an output file or a plot are computed. In any case, the return value is a vector of node ids identifying the nodes involved in the lineage. The lineage is transitive, so that backward provenance traces back to input files or constants, while forward lineage traces to output. This function is the key to the various trace and lineage functionality provided in `provDebugR`.

`provGraphR` is available on CRAN.

## Conclusions

Data provenance contains a wealth of information. While initially thought of as documentation of the history of the data, in order to bolster trust in the data, it has many uses beyond that. In particular, fine-grained provenance offers rich opportunities to develop tools that provide further insight into a script's behavior and offer hopes for improving reproducibility of scripts.

## Bibliography

I. Altintas, O. Barney, and E. Jaeger-Frank. Provenance collection support in the Kepler scientific workflow system. In *Proceedings of the International Provenance and Annotation Workshop*, pages 118–132, Chicago, May 2006. Springer-Verlag. [p5]

P. Biecek and M. Kosiński. archivist: An R package for managing, recording and restoring data analysis results. *Journal of Statistical Software, Articles*, 82(11):1–28, 2017. [p6]

J. Freire, D. Koop, E. Santos, and C. T. Silva. Provenance for computational tasks: A survey. *Computing in Science and Engineering*, 10(3):11–21, May/June 2008. [p5]

J. Gelfond, M. Goros, B. Hernandez, and A. Bokov. A system for an accountable data analysis process in R. *The R Journal*, 10(1):6–21, July 2018. [p5]

M. Herschel, R. Diestelkämper, and H. B. Lahmar. A survey on provenance: What for? what form? what from? *VLDB Journal*, 2018. [p5]

B. Lerner, E. Boose, and L. Perez. Using introspection to collect provenance in R. *Informatics*, 5(12), 2018. URL http://www.mdpi.com/2227-9709/5/1/12/htm. [p7]

B. S. Lerner and E. R. Boose. Poster: Rdatatracker and ddg explorer — capture, visualization and querying of provenance form r scripts. In *Proceedings of the International Provenance and Annotation Workshop*, Cologne, Germany, June 2014a. [p12]

B. S. Lerner and E. R. Boose. RDataTracker: Collecting provenance in an interactive scripting environment. In *Proceedings of 6th USENIX Workshop on the Theory and Practice of Provenance (TaPP '14)*, Cologne, Germany, June 2014b. [p7]

Z. Liu and S. Pounds. An R package that automatically collects and archives details for reproducible computing. *BMC Bioinformatics*, 15(138), 2014. [p5]

L. Murta, V. Braganholo, F. Chirigati, D. Koop, and J. Freire. noworkflow: Capturing and analyzing provenance of scripts. In *Proceedings of IPAW 2014*, Cologne, Germany, June 2014. [p5]

packrat. packrat, 2019. URL https://rstudio.github.io/packrat/. [p6]

T. Pasquier, M. K. Lau, X. Han, E. Fong, B. S. Lerner, E. Boose, M. Crosas, A. Ellison, and M. Seltzer. Sharing and preserving computational analyses for posterity with `encapsulator`. *Computing in Science and Engineering*, 20(4):111–124, July 2018. doi: 10.1109/MCSE.2018.042781334. [p6]

R. Peng. Caching and distributing statistical analyses in R. *Journal of Statistical Software, Articles*, 26(7): 1–24, 2008. [p6]

C. J. L. P. Peter Slaughter, Matthew B. Jones. recordr, 2018. URL https://github.com/NCEAS/recordr. [p5]

J. F. Pimentel, J. Freire, L. Murta, and V. Braganholo. A survey on collecting, managing, and analyzing provenance from scripts. *ACM Comput. Surv.*, 52(3), June 2019. [p5]

A. Runnalls and C. Silles. Provenance tracking in r. In *Proceedings of the 4th International Conference on Provenance and Annotation of Data and Processes*, IPAW'12, pages 237–239, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-34221-9. doi: 10.1007/978-3-642-34222-6_25. [p6]

C. Scheidegger, D. Koop, E. Santos, H. Vo, S. Callahan, J. Freire, and C. Silva. Tackling the provenance challenge one layer at a time. *Concurrency and Computation: Practice and Experience*, 20(5):473–483, 2008. [p5]

C. A. Silles and A. R. Runnalls. Provenance-awareness in R. In *Proceedings of the 3rd International Conference on Provenance and Annotation of Data and Processes*, pages 64–72, 2010. [p6]

J. Zhao, C. Goble, R. Stevens, and D. Turi. Mining Taverna's semantic web of provenance. *Concurrency and Computation: Practice and Experience*, 20(5):463–472, 2008. [p5]

*Author One*
*Affiliation*
*Address*
*Country*
author1@work

*Author Two*
*Affiliation*
*Address*
*Country*
author2@work

*Author Three*
*Affiliation*
*Address*
*Country*
author3@work