

# Caching Code Chunks in Dynamic Documents

## The `weaver` package

Seth Falcon<sup>1</sup>

Fred Hutchinson Cancer Research Center, 1100 Fairview Avenue North, Seattle, WA 98109, USA, e-mail: [sfalcon@fhcrc.org](mailto:sfalcon@fhcrc.org)

Received: date / Revised version: date

**Abstract** Authoring dynamic documents can become tedious for authors when a document contains one or more time consuming code chunks and each edit requires reprocessing all of the document. We introduce the `weaver` package that allows computationally expensive code chunks to be cached in order to speed up the edit/process/review cycle for dynamic documents authored using the Sweave framework.

**Key words** Dynamic Documents – Reproducible Research – Caching

## 1 Introduction

Dynamic documents provide a powerful framework that benefits both authors and readers of statistical reports. Authors benefit from the ability to update reports with new code and data. Readers can obtain the raw document and reproduce the computations on their own. As described by [1], a *dynamic document* is a report describing an analysis that consists of two types of content: text chunks and code chunks. The text chunks describe the analysis and possibly the code. The code chunks contain programming language source code and are intended to be executed in order to produce the computational results of the analysis.

Processing a raw dynamic document involves removing the code chunks, evaluating them, and replacing them by their value. The result of such processing is a new document that can be further processed to produce a static document in a desired format such as PDF, HTML, or ODF [3].

Sweave [4] is a dynamic document processing tool written in R [5] which supports text chunks marked up using  $\LaTeX$  and code chunks written in R. A noweb [6] like syntax is used to provide a simple way to delineate the two

types of chunks in the raw document. Since the introduction of the `Sweave` function in R's `utils` package,  $\text{\LaTeX}$ /R Sweave documents have seen wide use. All packages in the Bioconductor project contain a *vignette*, an Sweave document that demonstrates how to use the software. This has proved to be an effective documentation strategy and illustrates the benefits of dynamic documents. Readers enjoy a hands-on example and software developers get documentation that updates itself and provides a modest form of quality control (vignettes must build without errors before a package is released).

A frustration that authors often encounter with dynamic document implementations is the time it takes to see the result of edits made to the raw document. Even minor edits to text or code require re-computation of all code chunks and re-rendering of the resulting intermediate file before viewing the result. This can be time consuming, especially if some of the code chunks contain computationally expensive calculations.

One way to circumvent the need for re-evaluating every code chunk is to store the value of code chunks in a file-based cache. On subsequent processing runs one determines which code statements have been altered since the last time the document was processed and only those statements that have been altered, or that depend on the values of statements that have been altered, are re-computed. Statements that have not been altered since the last run have their value retrieved from the cache.

A caching mechanism could be added to any dynamic document processing system. Here we discuss the `weaver` package which provides a cache implementation for the Sweave framework. The `Sweave` function in R's `utils` package provides an extensible processor for Sweave documents. By default, the `Sweave` function uses the `RweaveLatex` driver to process documents. The `weaver` package provides an alternative driver, `weaver`, that supports the caching mechanism described here.

The `weaver` package uses functions from the `codetools` package to analyze the code in each statement and estimate its dependencies. Since the caching mechanism does not capture most side-effects, the author must indicate whether caching should be used for each code chunk in the document. In the next section the caching mechanism implemented in `weaver` is described in detail. Section 3 describes how to use `weaver` to author Sweave documents and Section 4 discusses shortcomings of the current system and directions for future development.

## 2 `weaver`'s Caching Mechanism

The author of an Sweave document specifies which code chunks should be cached by setting an option in the code chunk's header (details are provided in Section 3). Code chunks that have been marked for caching are called *cached code chunks*. The caching mechanism identifies each expression within a cached code chunk by calculating its md5 [7] digest.

For each expression, the result of the computation and an estimate of its dependencies is stored in a separate cache file named by the digest of the

expression. During subsequent processing runs, the cache file is read and a decision is made as to whether or not to use the cached result based on the dependency data.

In addition to cache files stored on disk, the caching mechanism relies upon an in-memory table `sym2hash` that stores a mapping for each symbol defined by a cached code chunk to the digest of the expression that last defined it. A symbol is *tracked* if it is the result of evaluating an expression in a cached code chunk. All tracked symbols are stored in the `sym2hash` table which is initialized for each processing run. The table also has an `updated` flag for each symbol which indicates whether the symbol was recomputed during the current run.

To explain the caching mechanism, we will trace what happens to the expressions in the hypothetical cached code chunk displayed in Figure 1. Note that the function `getDigest` shown below is for demonstration purposes only and not included in the `weaver` package.

```
<<c1, cache=TRUE>>=  
## All expressions will be cached,  
## but side effects will be lost.  
z <- 1  
x <- rnorm(5) + z  
y <- x * 10  
@
```

**Fig. 1** An example code chunk with the `cache` option set to `TRUE`. All expressions in this code chunk will be cached when the document is processed with the `weaver` Sweave driver.

Each expression is processed in turn. Assuming this is the first time the document has been processed, the steps taken for each expression are:

1. Compute the digest.
2. Determine the dependencies.
3. Evaluate the expression in a temporary environment.
4. Write a cache file containing the temporary environment and the dependency information.
5. For each symbol defined in the temporary environment, add a mapping to the `sym2hash` table mapping the symbol to the digest for the current expression. Also set the `updated` flag for the symbol in the table.

The first expression is trivial. The digest is computed and no dependencies are found other than `z` itself. The current implementation adds this otherwise unnecessary dependency in order to catch a dependency on the left hand side of expressions like `z[1] <- 3`.

```
> getDigest(quote(z <- 1))
```

```
[1] "7add28fda4f4c47c0db0d5c8d44946be"
> findDeps(quote(z <- 1))
[1] "<-" "z"
```

The output of `findDeps` is a vector containing the symbols that appear in the expression. Any symbol that is tracked (appears in `sym2hash`) is a dependency. A cache file with name matching the digest is written to disk and an entry mapping `z` to digest `7add2...` is added to the `sym2hash` table (described below).

The second expression is worth examining in more detail since it has a dependency on `z`. For each dependency, we determine if the dependency is tracked. For the second expression, we will find an entry for `z` in `sym2hash`. This mapping is copied into the `DEPS` field in the cache file.

```
> getDigest(quote(x <- rnorm(5) + z))
[1] "6c6a1b4a1de07372d9c21d31ef851447"
> findDeps(quote(x <- rnorm(5) + z))
[1] "+"      "<-"      "rnorm"  "z"      "x"
```

The second expression is evaluated in a temporary environment and saved to a cache file. An entry for `x` is added to the `sym2hash` table mapping to this expression's digest. The third expression follows similarly.

For subsequent processing runs, the steps taken for each expression are slightly different:

1. Compute the digest.
2. If no matching cache file is found, perform steps as in the first processing run.
3. Read dependencies from the cache file and compare the stored digest of dependencies to current values in the `sym2hash` table. if there is a digest mismatch for one of the dependencies or one of the dependencies has its updated flag set, recompute and write a new cache file.
4. Load values from the cache.

To illustrate the steps, consider a second processing run in which the author has changed the first expression to set the value of `z` to two (`z <- 2`). The first expression is new so it will be evaluated. For the second expression, we will compute its digest and find the cache file with a matching name from the previous run. In the `DEPS` field of the cache file will be the mapping of dependency `z` to the digest computed on the first run. This will not match the current value associated with `z` in the `sym2hash` table and indicates that the current expression must be recomputed. After recomputing and re-caching according to the steps described above, the mapping for `x` is updated in the `sym2hash` table and the updated flag is set to true. For the third expression, we will find the dependence on `x`, but in this case the

digests will match. However, since the updated flag is set for `x`'s entry in the `sym2hash` table, we know that the current expression must be recomputed.

The cache is not currently document specific. All documents in the same directory will share a single cache directory. This can be useful for documents with related computations, but may not be desired. A planned enhancement will allow the user to specify the name of the cache directory.

### 3 Using weaver

To use `weaver` with a given Sweave document, one must indicate which code chunks should be cached. This is done by adding a chunk-level option, `cache=TRUE`, to the code chunk header. Since the caching mechanism does not handle side-effects, such as loading data or printing, it is rarely desirable to cache all code chunks in a document. Code chunks that do not have a `cache` option specified, or have `cache=FALSE`, are not cached. Figure 1 shows an example code chunk with `cache=TRUE`. Each expression in this code chunk will be cached.

Because the default driver for the `Sweave` function, `RweaveLatex`, ignores options it does not know about, `weaver` documents can be processed without modification. This provides an easy means to process a document and ensure that no cached data is used.

To process a document using `weaver`, load the `weaver` package and then use `weaver()` as the `driver` argument to `Sweave`. The `weaver` package is available in Bioconductor's software repository<sup>1</sup>. If you haven't installed `weaver`, you can do so using Bioconductor's `biocLite` installation tool. Here is an example:

```
> ## to install weaver
> source("http://bioconductor.org/biocLite.R")
> biocLite("weaver")

> ## to use weaver for document 'myFile.Rnw'
> library("weaver")
> Sweave("myFile.Rnw", driver=weaver())
```

#### 3.1 What not to cache

The caching mechanism evaluates each expression in a cached code chunk in a temporary environment and saves all objects that result from the evaluation. Therefore, most side-effects are not captured by the caching mechanism<sup>2</sup>. Functions that print, define S4 classes or methods, or set global

---

<sup>1</sup> <http://bioconductor.org/packages/release/bioc/>

<sup>2</sup> Side-effects that create objects in the evaluation environment will be captured. Unfortunately, the `data` function cannot be used in cached code chunks as the data it loads is put into the global environment, not the evaluation environment by default.

options should not be placed in chunks that have the `cache` option set to `TRUE`. Since printing is a side-effect, cached chunks should be treated the same as chunks with Sweave's `results=hide` option set.

Plotting functions are a special case since the side effect of writing a PDF or PostScript file is persistent. Plotting expressions can be cached and will be recomputed when dependencies are changed. However, if the graphics file is deleted, the system will not (yet) detect this.

Special attention is also needed when using pseudo-random number generating functions such as `rnorm`. The state of the random number generator (RNG) is not captured. If there are cached and uncached expressions in the document that change the state of the RNG, then the random streams will be different between a cached and an uncached run.

As practical advice, we strongly recommend flushing and rebuilding the cache after significant code changes, and certainly before reviewing a final draft. The cache database is stored in a directory named `r_env_cache` in the current working directory. Removing this directory is the best way to be certain that the following run will not use any cached data. A log file is produced in the current working directory named `weaver_debug_log.txt`. Reviewing it can be useful in determining what the weaver system thinks the dependencies of a given expression are.

Repeated expressions within a given chunk will only be computed once with subsequent instances being pulled from the cache. This may not be desired if one is calling random number generating functions, for example. To get around this, one can place the calls in different code chunks or change the expressions so they are not identical.

## 4 Discussion

The weaver package has received much positive feedback since its release. For many common uses, it works well. For example, in working on the `GOstats` [2] vignette the processing time is reduced from 3:37 (Sweave) to 0:29 (weaver, after an initial run taking 4:28). So there is a modest cost for caching results, but processing time after minor edits is much faster. Caching very large data may be slower, in some cases, than recomputing.

The weaver package implements a rather simple strategy for determining when to recompute a given expression given the data stored in the cache file. Extensions to this mechanism could address some of the limitations of the current system. For example, the system could capture side effects from `data` and `plot` by maintaining a list of common functions that produce side-effects. Additional code analysis would reveal which function is being called and special action could be taken for functions in the list.

Another weakness of the current dependency resolution approach is that only tracked dependencies are handled. That is, if an expression depends on variables defined in a non-cached code chunk, the dependency will not be tracked and the cache will not be updated when the non-cached code

changes. An improvement would be to issue a warning to the user when this occurs.

Other features that are planned for future versions of `weaver` include: an option to force re-computation (and re-caching) of all chunks, and an option to turn off dependency tracking and force use of the cache.

## 5 Conclusion

Caching is likely to be an important tool for authoring dynamic documents. We have shown it is possible and that there are considerable benefits. The `weaver` package allows authors of Sweave documents to enjoy substantially shorter processing times when working on their documents by preventing repeated evaluation of the same expressions. However, it is difficult for caching to be entirely consistent with the standard evaluation paradigm. Dynamic documents that mix coding languages will increase this challenge significantly. Authors must be vigilant; removing the cache directory and reprocessing regularly is strongly recommended.

## References

1. R. Gentleman and D. Temple Lang. Statistical analyses and reproducible research. Technical Report 2, Bioconductor Project Working Papers, 2004. <http://www.bepress.com/bioconductor/paper2>.
2. Robert Gentleman. Using GO for statistical analyses. In Jaromir Antoch, editor, *Compstat 2004 – Proceedings in Computational Statistics*, pages 171–180, Heidelberg, 2004. Physika Verlag, Heidelberg, Germany.
3. Max Kuhn. *odfWeave: Sweave processing of Open Document Format (ODF) files*, 2007. R package version 0.4.8.
4. Friedrich Leisch. Sweave: Dynamic generation of statistical reports using literate data analysis. In Wolfgang Härdle and Bernd Rönz, editors, *Compstat 2002 — Proceedings in Computational Statistics*, pages 575–580. Physika Verlag, Heidelberg, Germany, 2002. ISBN 3-7908-1517-9.
5. R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.
6. Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.
7. R. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992. Status: INFORMATIONAL.