Distributed Storage and Lists

Stefan Theußl

December 14, 2011

Abstract

Distributed lists are list-type objects where elements (i.e., arbitrary R objects) are stored in serialized form on a distributed storage. The latter is often used in high performance computing environments to process large quantities of data. First proposed by Google, data located in such an environment is most efficiently processed using the MapReduce programming model. The R package **DSL** provides an environment for creating and handling of distributed lists. The package allows to make use of different types of storage backends, in particular the Hadoop Distributed File System. Furthermore, it offers functionality to operate on such lists efficiently using the MapReduce programming model.

1 Introduction

Distributed lists are list-type objects using a *distributed storage* to store their elements. Typically, distributed lists are advantageous in environments where large quantities of data need to be processed at once since all data is stored out of the main memory which is often limited. Usually, a "distributed file system" (DFS) can serve as a container to hold the data on a distributed storage. Such a container can hold arbitrary objects by serializing them to files.

A recurrent function when computing on lists in R (R Development Core Team, 2011) is `lapply()` and variants thereof. Conceptually, this is similar to a "Map" function from functional programming where a given (R) function is applied to each element of a vector (or in this case a list). Furthermore, another typical type of function often applied to lists is a function which combines contained elements. In functional programming this is called "Reduce" but variants thereof also exists in other areas (e.g., in the MPI standard, see `http://www.mpi-forum.org/docs/mpi22-report/node103.htm#Node103`).

First proposed by Google the Map and Reduce functions are often sufficient to express many tasks for analyzing large data sets. They implement a framework which follows closely the MapReduce programming model (see Dean and Ghemawat, 2004, and `http://en.wikipedia.org/wiki/MapReduce`). Note however, that as pointed out e.g., in Lämmel (2007) Map and Reduce operations in the MapReduce programming model do not necessarily follow the definition from functional programming. It rather aims to support computation (i.e., map

and reduction operations) on large data sets on clusters of workstations in a distributed manner. Provided each mapping operation is independent of the others, all maps can be performed in parallel. Hadoop (`http://hadoop.apache.org/`) is an open source variant of this framework.

Package **DSL** is an extension package for R for creating and handling list-type objects whose elements are stored using a distributed storage backend. For operating on such distributed lists efficiently the package offers methods and functions from the MapReduce programming model. In particular, **DSL** allows to make use of the Hadoop Distributed File System (HDFS, see Borthakur, 2010) and Hadoop Streaming (MapReduce) for storing and distributed processing of data. In Section 2, we describe the underlying data structures, and the MapReduce functionality. Examples are discussed in Section 3. Section 4 concludes the paper.

2 Design and Implementation

2.1 Data Structures

2.1.1 Distributed Storage

The S3 class `"DStorage"` defines a virtual storage where files are kept on a file system which possibly spans over several workstations. Data is distributed automatically among these nodes when using such a file system. Objects of class `"DStorage"` "know" how to use the corresponding file system by supplied accessor and modifier methods. The following file systems are supported to be used as distributed storage (DS):

`"LFS"`**:** the local file system. This type uses functions and methods from the packages **base** and **utils** delivered with the R distribution to handle files.

`"HDFS"`**:** the Hadoop distributed file system. Functions and Methods from package **hive** (Theußl and Feinerer, 2011) are used to interact with the HDFS.

Essentially, such a class needs methods for reading and writing to the distributed storage (DS). Note however that files are typically organized according to a published standard. Thus, one should not write or modify arbitrary files or directories on such a file system. To account for this, class `"DStorage"` specifies a directory `base_dir` which can be modified freely but avoids that read/write operations can escape from that directory. The following (**DSL**-internal) methods are available for objects of class `"DStorage"`.

- `DS_dir_create()`
- `DS_get()`
- `DS_list_directory()`
- `DS_put()`

- `DS_read_lines()`
- `DS_unlink()`
- `DS_write_lines()`

Depending on the type of storage suitable functions from different packages will be used to interact with the corresponding file system. Whereas `DS_dir_create()`, `DS_list_directory()`, `DS_read_lines()`, `DS_unlink()`, and `DS_write_lines()` mimic the behavior of corresponding functions of package **base** (`dir.create()`, `dir()`, `readLines`, `unlink()`, and `writeLines()`, respectively), functions `DS_get()` and `DS_put()` can be used to read/write R objects from/to disk.

The main reason of having such a virtual storage class in R is that it allows for easy extension of memory space in the R working environment. E.g., this storage can be used to store arbitrary (serialized) R objects. These objects are only loaded to the current working environment (i.e., into RAM) when they are needed for computation. However, it is in most cases not a good idea to place many small files on such a file system due to efficiency reasons. Putting several serialized R objects into files of a certain maximum size (e.g., line by line as key/value pairs) circumvents this issue. Indeed, frameworks like Hadoop benefit from such a setup (see Section *Data Organization* in Borthakur, 2010). Thus, a constructor function must take the following arguments:

`type:` the file system type,

`base_dir:` the directory under which chunks of serialized objects are to be stored,

`chunksize:` the maximal size of a single chunk.

E.g., a DS of `type "LFS"` using the system-wide or a user-defined *temporary directory* as the base directory (`base_dir`) and a chunk size of 10MB can be instanciated using the function `DStorage()`:

```
> ds <- DStorage( type = "LFS", base_dir = tempdir(),
+                 chunksize = 10 * 1024^2 )
```

Further methods to class `"DStorage"` are a corresponding `print()` and a `summary()` method.

```
> ds

DStorage.
- Type: LFS
- Base directory on storage: /tmp/Rtmp5vb4Th
- Current chunk size [bytes]: 10485760

> summary(ds)
```

```
DStorage.
- Type: LFS
- Base directory on storage: /tmp/Rtmp5vb4Th
- Current chunk size [bytes]: 10485760
- Registered methods:
  dir_create, fetch_last_line, get, list_directory
  put, read_lines, unlink, write_lines
```

2.1.2 Distributed Lists

Distributed lists are defined in R by the S3 class `"DList"`. Objects of this class behave similar to standard R lists but use a distributed storage of class `"DStorage"` to store their elements. Distributed lists can be easily constructed using the function `DList()` or can be coerced using the generic function `as.DList()`. Available methods support coercion of R lists and character vectors representing paths to data repositories as well as coercion of `"DList"` objects to lists.

```
> dl <- DList( letters = letters, numbers = 0:9  )
> l <- as.list( letters )
> names(l) <- LETTERS
> dl2 <- as.DList(l)
> identical( as.list(dl2), l )

[1] TRUE

> dl3 <- as.DList( system.file("examples", package = "DSL") )
```

Note that the above example uses a default storage type, namely `"LFS"` using a *temporary directory* generated with `tempdir()` as the base directory. In order to set a user defined storage the `DStorage` argument to the `DList()` constructor is used.

```
> dl <- DList( letters = letters, numbers = 0:9, DStorage = ds )
```

Conceptually we want a distributed list to support a set of intuitive operations, like accessing each element (stored somewhere on a DFS) in a direct way, displaying the distributed list and each individual element, obtaining information about basic properties (e.g., the length of the list), or applying some operation on a range of elements. These requirements are formalized via a set of interfaces which must be implemented by the `"DList"` class:

Display Since elements of the list are not directly available the `print` and `summary` methods should provide other useful information about the distributed list (like the number of list elements).

Length The `length()` function must return the number of list elements.

Names Named list must be supported.

Subset The `[[` operator must be implemented so that individual elements of the distributed list can be retrieved.

MapReduce Map and Reduce operations as well as variants of `lapply` (which are conceptually similar to Map) can be used to express most of the computation on `"DList"` objects.

```
> dl

A DList with 2 elements

> summary(dl)

        Length Class  Mode
letters 26     -none- character
numbers 10     -none- numeric

> names( dl2 )

 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N"
[15] "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

> length(dl3)

[1] 2

> dl3[[1]]

[1] "/home/theussl/lib/R/2.14/DSL/examples/file01"
```

MapReduce is discussed in more detail in the next section.

2.2 Methods on Distributed Lists

The MapReduce programming model as defined by Dean and Ghemawat (2004) is as follows. The computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user expresses the computation as two functions: Map and Reduce. The Map function takes an input pair and produces a set of intermediate key/value pairs. The Reduce function accepts an intermediate key and a set of values for that key (possibly grouped by the MapReduce library). It merges these values together to form a possibly smaller set of values. Typically, just zero or one output value is produced per reduce invocation. Furthermore, data is usually stored on a (distributed) file system which is recognized by the MapReduce library. This allows such a framework to handle lists of values (here objects of class `"DList"`) that are too large to fit in main memory (i.e., RAM).

`DGather`: this collective operation is similar to an MPI_GATHER (`http://www.mpi-forum.org/docs/mpi22-report/node95.htm#Node95`). However, instead of collecting results from processes running in parallel, `DGather()` collects the contents of chunks holding the elements of a `"DList"`. By default a named list of length the number of chunks is to be returned. Its elements are character vectors of values from key/value pairs stored in chunks read line by line from the corresponding chunk. Alternatively, `DGather()` can be used to retrieve the keys only.

`DLapply`: is an (l)apply-type function which is used to iteratively *apply* a function to a set of input values. In case of `DLapply()` input values are elements of `"DList"` objects (i.e., the value of a key/value pair). A distributed list of the same length is to be returned.

`DMap`: is similar to `DLapply()` above but always takes both the key and the value from a key/value pair as input. Thus, keys can also be modified. Indeed, the returned object can differ in length from the original as opposed to when using `DLapply`.

`DReduce`: this collective operation takes a set of (intermediate) key/value pairs and combines values with the same associated key using a given directive (the reduce function). By default values are concatinated using the `c()` operator.

```
> dl <- DList( line1 = "This is the first line.",
+              line2 = "Now, the second line." )
> res <- DLapply( dl, function(x) unlist(strsplit(x, " ")) )
> as.list( res )

$line1
[1] "This"  "is"    "the"   "first" "line."

$line2
[1] "Now,"   "the"    "second" "line."

> foo <- function( keypair )
+     list( key = paste("next_", keypair$key, sep = ""), value =
+          gsub("first", "mapped", keypair$value) )
> dlm <- DMap( x = dl, MAP = foo)
> ## retrieve keys
> unlist(DGather(dlm, keys = TRUE, names = FALSE))

[1] "next_line1" "next_line2"

> ## retrieve values
> as.list( dlm )
```

```
$line1
[1] "This is the mapped line."

$line2
[1] "Now, the second line."
```

Further methods on `"DList"` objects are prefixed with `DL_`. Currently, only methods for interacting with the underlying `"DStorage"` are available.

`DL_storage:` accesses the storage of `"DList"` objects. Returns objects of class `"DStorage"`.

`DL_storage<-:` replaces the storage in `"DList"` objects. Data is automatically transferred to the new storage.

```
> l <- list( line1 = "This is the first line.",
+            line2 = "Now, the second line." )
> dl <- as.DList( l )
> DL_storage(dl)

DStorage.
- Type: LFS
- Base directory on storage: /tmp/Rtmp5vb4Th
- Current chunk size [bytes]: 10485760

> ds <- DStorage("HDFS", tempdir())
> DL_storage(dl) <- ds
> as.list(dl)

$line1
[1] "This is the first line."

$line2
[1] "Now, the second line."
```

3 Examples

3.1 Word Count

This examples demonstrates how `Dmap()` and `DReduce()` can be used to count words based on text files located somewhere on a given file system. The following two files contained in the example directory of the package will be used.

```
> ## simple wordcount based on two files:
> dir(system.file("examples", package = "DSL"))

[1] "file01" "file02"
```

We use a temporary directory as the base directory of a new "DStorage" object. By setting the maximum chunk size to 1 Byte we force the name of each file being placed in a separate chunk. Then we store the absolute path to the text files as elements of a "DList" object.

```
> ## first force 1 chunk per file (set max chunk size to 1 byte):
> ds <- DStorage("LFS", tempdir(), chunksize = 1L)
> ## make "DList", i.e., read file contents and store in chunks
> dl <- as.DList( system.file("examples", package = "DSL"),
+                 DStorage = ds )
```

Data is read into chunks (one per original file) by using a simple call of DMap() on the distributed list.

```
> ## read files
> dl <- DMap(dl, function( keypair ){
+     list( key = keypair$key,
+           value = tryCatch(readLines(keypair$value),
+                            error = function(x) NA) )
+ })
```

The contents of the files is split into words using the following call.

```
> ## split into terms
> splitwords <- function( keypair ){
+     keys <- unlist(strsplit(keypair$value, " "))
+     mapply( function(key, value) list( key = key, value = value),
+             keys, rep(1L, length(keys)),
+             SIMPLIFY = FALSE, USE.NAMES = FALSE )
+ }
> res <- DMap( dl, splitwords )
> as.list(res)

$Hello
[1] 1

$World
[1] 1

$Bye
[1] 1

$World
[1] 1

$Hello
[1] 1
```

```
$DSL
[1] 1

$Bye
[1] 1

$DSL
[1] 1
```

Eventually, collected intermediate results are summed.

```
> ## now aggregate by term
> res <- DReduce( res, sum )
> as.list( res )

$Hello
[1] 2

$World
[1] 2

$DSL
[1] 2

$Bye
[1] 2
```

3.2 Temperature

TODO: Example from Hadoop book. around 30 GB of raw data.

4 Conclusion and Outlook

Package **DSL** was designed to allow for handling of large data sets not fitting into main memory. The main data structure is the class `"DList"` which is a list-type object storing its elements on a virtual storage of class `"DStorage"`. The package currently provides basic data structures for creating and handling `"DList"` and `"DStorage"` objects, and facilities for computing on these, including map and reduction methods based on the MapReduce paradigm.

Possible future extensions include:

- `"DStorage"` interface to NoSQL database systems,
- better integration of the **parallel** package. Currently only the *multicore* version of `lapply` is used for `"LFS"` type `"DStorage"`.

Acknowledgments

We are grateful to Christian Buchta for providing efficient C code for collecting partial results in `DReduce()`.

References

D. Borthakur. HDFS architecture. *Document on Hadoop Wiki*, 2010. URL `http://hadoop.apache.org/common/docs/r0.20.2/hdfs_design.html`.

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the Sixth Symposium on Operating System Design and Implementation*, pages 137–150, 2004. URL `http://labs.google.com/papers/mapreduce.html`.

R. Lämmel. Google's MapReduce programming model—revisited. *Science of Computer Programming*, 68(3):208–237, 2007.

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. URL `http://www.R-project.org`. ISBN 3-900051-07-0.

S. Theußl and I. Feinerer. **hive***: Hadoop InteractiVE*, 2011. URL `http://CRAN.R-project.org/package=hive`. R package version 0.1-13.