Package Specification for R Collection

Dachuan Yu

DOCOMO USA Labs
3240 Hillview Avenue, Palo Alto, CA 94304
yu@docomolabs-usa.com

November 18, 2010

ABSTRACT

TABLE OF CONTENTS

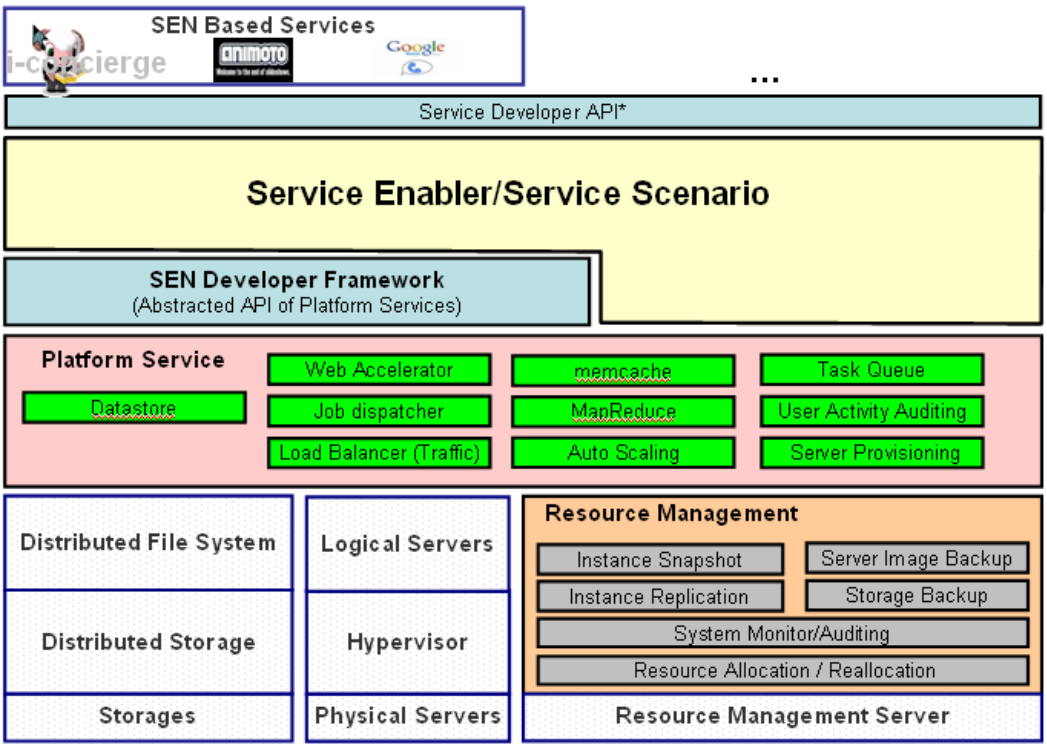

Figure 1: SEN architecture

1 Introduction

This document specifies a proof-of-concept prototype of a collection package for the R programming language. In this prototype, we focus on demonstrating the feasibility of various ideas and defer several practical concerns to future work.

2 SEN & Developer Framework

2.1 Background on SEN

Explain SEN to set up the context. Point out the need for a developer framework

2.2 SEN Developer Framework

Web application frameworks help developers build Web applications. They provide core functionality common to most Web applications, including session management, data persistence, templating, security, and caching. Developers save a significant amount of time. However, most frameworks are not designed with the Cloud in mind, especially in terms of large-scale distributed computing and scalability.

To better assist application development for the Cloud , SEN developer framework (DF)

focuses on the development of service enablers on top of platform services in a Cloud architecture. At the current stage, we are investigating the use of a collection-based framework to bridge the gap between service enablers and platform services.

In particular, platform services support the complete life cycle of building and delivering web applications and services. Service enablers provide common facilities on aspects specific to DOCOMO services. The primary goal of the collection-based framework is to produce higher-level programming abstractions for carrying out tasks over the cloud. This allows programmers to focus on their specific application domains without being distracted by common implementation details that carry out distributed computing tasks.

3 Overview of Solution

3.1 Collection-Oriented Programming

The main idea is to manipulate a collection of data as a whole. Typical collection-oriented languages include Fortran 90, APL, CM Lisp, Paralation Lisp, SETL, etc. Typical collection operations include map, reduce, filter, member test, pack, select, permute, etc.

We would like to apply collection-oriented programming to Cloud computing. The key is to effectively implement the collection model and primitives on a parallel architecture.

Given the collection-based model, programmers work with collections without having to invoke platform services directly. They manipulate a collection of data as a whole. Tasks are distributed and carried out transparently. In addition, common programming patterns such as MapReduce can be implemented as libraries.

Our collection package contains the following elements:

- A collection mode for R that handles large amount of data.
- Collection primitives for transparent distributed computing on the cloud, e.g., Map, reduce, filter, member test, pack, select, permute, etc.
- Data can be Imported from common data sources, e.g., datastore, database, files.

4 Usage of Collection Package

4.1 Deployment Scenario

We consider the following deployment scenario. We provide rightscript for fixed deployment on RightScale. R programmer launches our rightscript for access to an R environment with the collection package running on the Cloud. R programmer writes a program and uses collection mode and primitives when appropriate. R programmer runs the program.

4.2 Hello World

Double and preproc are user defined functions. Load, select, and map are collection routines. Explain at a high level and defer details to later.

4.3 Simulator Building

We now explain how to build a simulator using the collection library in R.

```
#! /path/to/Rscript

library(collection)

args <- commandArgs(TRUE)
dbInfo <- args[1]
queryInfo <- args[2]

double <- function(elem) { elem*2 }
preproc <- function(elem) { true }

myCl <- cl::load(dbInfo, queryInfo)
myCl1 <- cl::select(preproc, myCl)
myCl2 <- cl::map(double, myCl1)

myStatus <- cl::store(dbInfo)

q(status=myStatus)
```

Figure 2: SEN architecture

4.3.1 Simulator Definition

4.3.2 Notations

Currying Currying is the technique of transforming a function that takes multiple arguments (or an n-tuple of arguments) in such a way that it can be called as a chain of functions each with a single argument. The practical motivation for currying is that very often the functions obtained by supplying some but not all of the arguments to a curried function (often called partial application) are useful.

For example, Given a function `F <- function(x,y,z){...}`, we usually apply F to 3 parameters. However, we could also do partial function application with 1 or 2 parameters This can be achieved by defining auxiliary functions for every static appearance of partial function application in the code as follows:

```
Fa <- function(y,z){F(a)}
Fab <- function(z){Fa(b)}
```

In later sections, we will use the following shorthand for easy of exposition:

- `(F a)` means `Fa` above
- `(F a b)` means `Fab` above

We will also use the following form of variables (pattern matching) for ease of reading:

- `(F, T)` to represent a pair variable
- `[S]` to represent a collection variable with element type the same as `S`
- `[(F,T)]` to represent a collection variable, where each element is a pair

4.3.3 Step 1 - Single User, Single Path

We use `S` to refer to a set of shape files for a given geographic area. In practice, it may contain road, rail road, and cit boundary information. We do not use a collection to represent shape files.

We use `U` to refer to all metadata about a given user, e.g., type of user, mode of transportation, etc. We will use a collection to represent multiple users in a later step

We use `Gen1` to generate start/end randomly for `U` on `S`. It is implemented as a single-machine algorithm without using collection.

```
Gen <- function(S, U){...}
```

We use `Traj1` to generate trajectory between `F` and `T` for `U` on `S`. It is implemented as a single-machine algorithm without using collection.

```
Traj1 <- function(S, U, (F, T)){...}
```

Example use of Gen1 and Traj1 are as follows:

```
(F, T) <- Gen1(S, U)
traj <- Traj1(S, U, (F, T))
```

4.3.4 Step 2 - Single User, Multiple Paths

We use a `Gen2` function to generate a sequence of start/end randomly for `U` on `S`. The result is put into a collection.

```
myCl <- cl :: new(name)
for(i = 1; i < 100, i + +){
    (F, T) <- Gen1(S, U)
    cl :: add(myCl, (F, T))
}
myCL
```

`Gen2` can be used as follows:

```
[(F, T)] <- Gen2(S, U)
```

We now define a `Traj2` function as follows:

```
Traj2 <- function(S, U){
    [(F, T)] <- Gen2(S, U)
    [traj] < −map((Traj1 S U), [(F, T)])
    reduce(concat, [traj])
}
```

`Traj2` can be used as `traj` $< -$`Traj2(S, U)`. The function body of `Traj2` calls `Gen2` to get a sequence of start/end `[(F, T)]` for `U` on `S`. It then uses `map` to apply `(Traj1 S U)` to `[(F, T)]` element-wise to get segments of the trajectory. Finally, the results are pieced together using `reduce` by combining all segments.

4.3.5 Step 3 - Multiple Users, Multiple Paths

We use `[U]` to refer to a collection of user metadata. Each element of `[U]` provides information for a single user, e.g., loaded from a configuration file.

We define `Traj3` as follows:

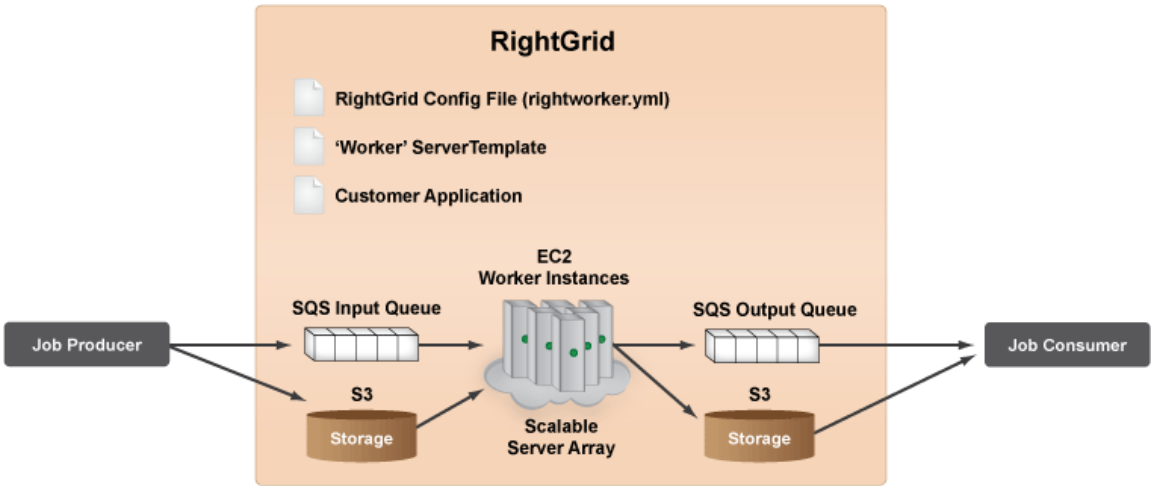

Figure 3: RightGrid architecture

```
Traj3 <- function(S, [U]){
    map((Traj2 S), [U])
}
```

`Traj 3` can be used as $[\texttt{traj}] < -\texttt{Traj3}(\texttt{S}, \texttt{U})$. the function body of `Traj3` calls `map` to apply `(Traj2 S)` to `[U]` element-wise to get trajectories of every user.

5 Package Implementation

5.1 RightGrid

We use RightGrid to simplify our implementation.

The RightGrid batch processing system leverages Amazons EC2, SQS, and S3 web services to process large numbers of jobs in a scalable and cost efficient manner. It can also be used to process long running tasks that are spawned off of a web site.

RightGrid is a framework to help programmers access Amazon Web Services (AWS) with minimal effort. RightGrid takes care of all the scheduling, load management, and data transport so that the job processing can operate on local data, on an EC2 instance. RightGrid also monitors overall progress of jobs and provides access to log files and any associated error information.

Figure 3 shows the architecture of RightGrid. Here are the basic entities involved:

- Work Unit - the meta-data that describes the task. It contains the path to the input data and any other the relevant information.
- Job Producer - submits work units to an SQS input queue and stores data associated with the work units in a bucket on S3. The job producer can be a web server that sends long-running tasks, or it could be a back-end system that sends large numbers of tasks to be performed on a specific dataset.

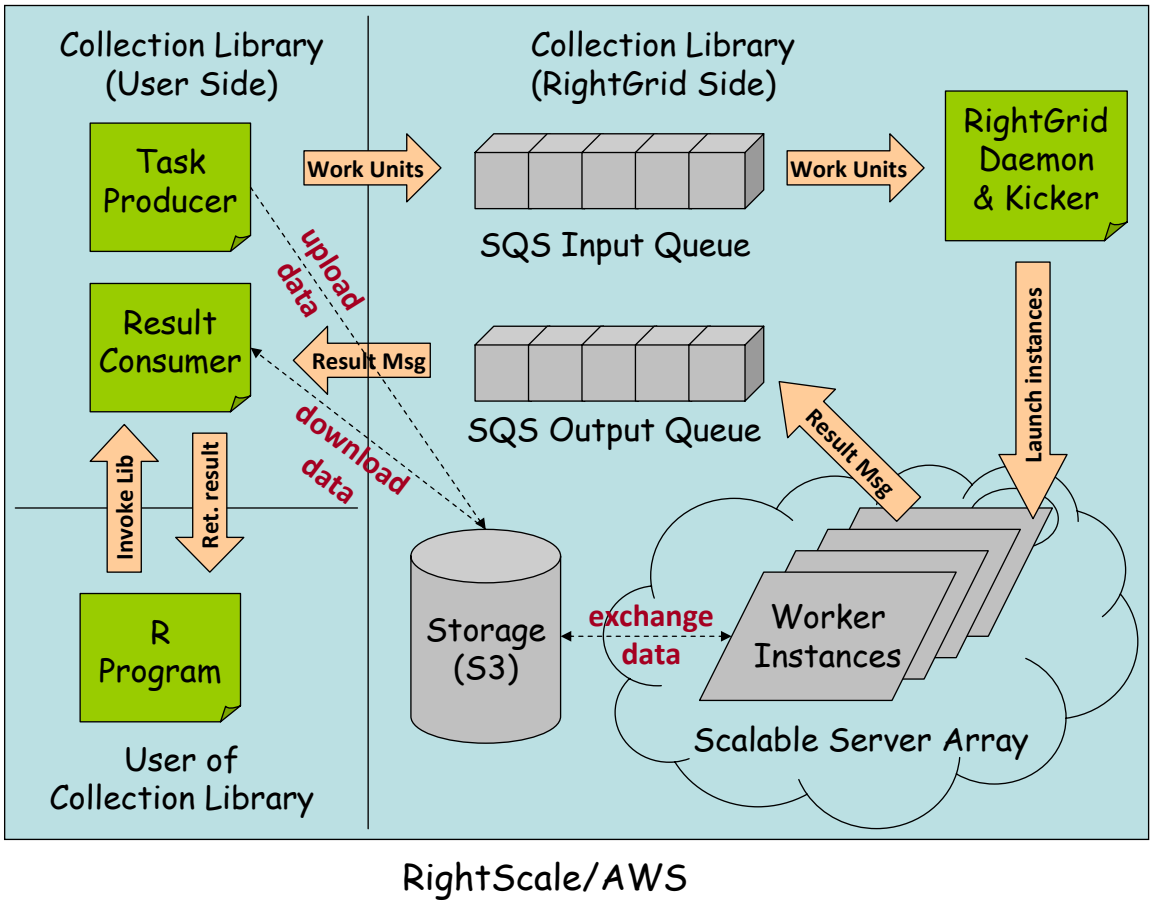

Figure 4: Architecture for the implementation of colletion library

- Job Consumer - parses the result messages. If necessary, it can also update a central database accordingly.
- RightGrid Configuration File (rightworker.yml) - the configuration file that defines the key variables of RightGrid.
- "Worker" ServerTemplate - the ServerTemplate that is used to create each worker instance in the scalable server array.
- Customer Application - defines how each work unit should be processed by a worker instance.

We refer readers to RightScale support portal (http://support.rightscale.com/03-Tutorials/02-AWS/03-Grid_Edition) for more details.

5.2 Architecture for Collection Implementation

We make use of RightGrid for fast prototyping. The arhictecture of collection implementatin is given in Figure 4.

Under our deployment scenario, all components of the architecture are deployed on RightScale. As part of the library implementation, every collection routine is implemented to have three components:

- Job producer - creates tasks based on the primitive and a partition of the collection, stores computation and data in S3, and puts task descriptor in the input queue.
- Worker - applies the given computation to elements of a partition, exchanging data with storage as needed
- Job consumer - checks the output queue for completed tasks and download daata from storage as needed.

Here is what happens at runtime when a collection routine is invoked:

- R program invokes library routine (written in R).
- The corresponding task producer and result consumer is used to instantiate and launch RightGrid.
- Based on the tasks in the input queue, RightGrid Daemon and kicker will launch worker instances (written in R) as needed, and the worker instances put computation results into storage and output queue.

5.3 Collection Routines

5.3.1 Core Routines

`loadCL`	A function to load data from datastore into a collection.

Description A function to load data from datastore into a collection. A collection object will be created and returned.

Usage `loadCl(name,keys)`

Arguments

`name`	The name of the datastore (e.g., S3 bucket name)
`keys`	A list of keys to objects in the datastore [[[**to fix representation**]]]

Details `loadCl` is used when data is in datastore. The name of the datastore and the keys of objects to be loaded must be specified. The efficient representation of keys is to be decided.

Value A collection object is returned. It can be manipulated by other routines in the collection library. Its internal representation is discussed in a later section. Note that collection user does not need to know the representation.

Examples `myCl <- loadCl(bucket_name, keys)`

`newCl`	A function to create an empty collection.

Description A function to create an empty collection. A collection object will be created and returned.

Usage `newCL()`

Arguments
none

Details `newCL` is used to create a collection from scratch. It is usually followed by other collection routines to grow elements of the collection.

Value A collection object is returned.

Examples
```
myCl <- newCl()
myCl.update(1, 123)
```

`myCl.update`	A function to update an entry of a collection.

Description A function to update an entry of a collection. The collection object will be updated

Usage `myCl.update(index, value)`

Arguments

`index`	the index of the collection element to be updated
`value`	the value to be used to update the element

Details `update` is used to change a particular element of a collection in the same way that a vector element is updated. We may use operator overloading to provide better syntax.

Value N/A

Examples
```
myCL <- newCL()
myCl.update(1, 123)
```

`map`	A function to apply a given function to a collection element-wise.

Description A function to apply a given function to a collection element-wise. A new collection object will be created and returned.

Usage `map(fun, myCl)`

Arguments

`fun`	the function to be applied to all elements
`myCl`	the collection being manipulated

Details `map` automatically carries out the computation on the Cloud without user intervention. We may use operator overloading to provide better syntax.

Value A collection object is returned, whose elements are results of the function application on the orignal collection.

Examples

```
double <- function(elem){elem * 2}
myCL1 <- map(double, myCl)
```

`reduce` A function to combine elements of a collection using a given function.

Description A function to combine elements of a collection using a given function. A new collection object will be created and returned.

Usage `reduce(fun, myCl)`

Arguments

`fun`	the function to be applied to combine all elements of `myCl` in an unspecified order
`myCl`	the collection being manipulated

Details `reduce` automatically carries out the computation on the Cloud without user intervention. The combination can happy in any order. The function `fun` is expected to be associative and commutative so that the result of `reduce` is predictable. We may use operator overloading to provide better syntax.

Value A collection object is returned, whose elements are results of combining the elements of the original collection with the function.

Examples

```
double <- function(elem){elem * 2}
plus <- function(x, y){x + y}
myCl1 <- map(double, myCl)
myCl2 <- reduce(plus, myCl1)
```

5.3.2 Major Routines

`importDB`	A function to import data from database into a collection.

Description A function to import data from database into a collection. A collection object will be created and returned.

Usage `importDB(dbInfo, queryInfo)`

Arguments

`dbInfo`	A list consisting of access information to the database. [[[**to fix representation**]]]
`queryInfo`	Query to be performed on the database.

Details `importDB` is used when data is in database. The database and query must be specified; their representations are to be decided.

Value A collection object is returned, which contains data obtained from the query. It can be manipulated by other routines in the collection library. Its internal representation is discussed in a later section. Note that collection user does not need to know the representation.

Examples `myCl` <− `importDB(myDB, myQuery)`

`exportDB`	A function to export data from collection into a database.

Description A function to export data from collection into a database.

Usage `exportDB(myCl, dbInfo, queryInfo)`

Arguments

`myCl`	the collection of data to be stored
`dbInfo`	A list consisting of access information to the database. [[[**to fix representation**]]]
`queryInfo`	Query to be performed on the database.

Details `exportDB` is used to export data into database. The database and query must be specified; their representations are to be decided.

Value `null`

Examples `exportDB(myCl, myDB, myQuery)`

getElem	A function to obtain a piece of data from a collection by index.

Description A function to obtain a piece of data from a collection by index.

Usage `getElem(myCl, index)`

Arguments

`myCl`	the collection to read data from
`index`	index of the data in the collection

Details `getElem` is used to read a single element of a collection by index. It can be used in the same matter as vector indexing.

Value The value of the data identified by the index in the collection is returned.

Examples `getElem(myCl, index)`

dropCL	A function to drop the datastore table that stores the collection data.

Description A function to drop the datastore table that stores the collection data.

Usage `dropCL(myCl)`

Arguments

`myCl`	the collection to remove

Details `dropCL` is used to reclaim storage when a collection and its data are no longer in use.

Value `null`

Examples `dropCL(myCl)`

select	A function to obtain elements of a collection based on a given predicate.

Description A function to obtain elements of a collection based on a given predicate.

Usage `select(myCl, pred)`

Arguments

`myCl`	the collection to obtain elements from
`pred`	the predicate for deciding the whether an element would be selected

Details `select` is used to select certain elements out of a collection.

Value It returns a new collection containing those elements satisfying the predicate.

Examples `even <- function(elem){elemmod2 = 0}`
$select(myCl, even)$

`pack`	A function to obtain elements of a collection based on a boolean collection.

Description A function to obtain elements of a collection based on a boolean collection.

Usage `pack(myCl, bCl)`

Arguments

`myCl`	the collection to obtain elements from
`bCl`	a boolean collection where elements are either 0 or 1

Details `pack` is used to select certain elements out of a collection.

Value If element `i` of `bCl` is 1, then element `i` in `myCl` will be selected; all selected elements are packed into the result collection. It returns a new collection containing those elements.

Examples `bCl <- ...`
`pack(myCl, bCl)`

5.3.3 Extensional Routines

`importFile`	A function to import data from a file into a collection.

Description A function to import data from a file into a collection. A collection object will be created and returned.

Usage `importFile(filename, keys)`

Arguments

`filename`	path and name of the file to import from; the file is assumed in a table format [[[**TBD**]]]
`keys`	A list of keys to identify data in the file

Details `importFile` is used when data is in a file. The name of the file and the keys to identify data must be specified. The file format is to be decided.

Value A collection object is returned, which contains data obtained from the file. It can be manipulated by other routines in the collection library. Its internal representation is discussed in a later section. Note that collection user does not need to know the representation.

Examples `myCl <- importFile(filename, keys)`

`exportFile`	A function to export data from collection into a file.

Description A function to export data from collection into a file.

Usage `exportFile(myCl, filename)`

Arguments

`myCl`	the collection of data to be stored
`fileName`	path and name to the file to be created

Details `exportFile` is used to export data into a file. The names of the collection and the file must be specified.

Value A boolean value will be returned to signal whether the exporting is successful.

Examples `exportFile(myCl, filename)`

`permute`	A function to rearrange order of elements based on an index collection.

Description A function to rearrange order of elements based on an index collection.

Usage `permute(myCl, iCl)`

Arguments

`myCl`	the collection of elements to be rearranged
`iCl`	an index collection where elements refer to indices of `myCl`

Details `permute` takes two conformable ordered collections: the data collection and the index collection. The latter is a collecgtion whose elemnts are a permutation of the indices (no index is repeated and all are present) of the first collection.

Value The result of the operation is a collection in which the index collection specifies where the corresponding elements of the data collection goes in the result collection.

Examples `permute(myCl, iCl)`

`comp` Comprehension creates a new collection by applying a function to all elements of an exist

---

Description Comprehension creates a new collection by applying a function to all elements of an existing collection that satisfy some property.

Usage `comp(myCl, pred, fun)`

Arguments

`myCl`	a collection-valued expression
`pred`	a boolean predicate to be applied to elements of `myCl`
`fun`	a function to be applied to elements of `myCl` that satisfies `pred`

Details `comp` [[[**TBD**]]]

Value The result of the operation is a collection of all `fun(x)` with `x` chosen from all the elements in `myCl` that satisfy `pred`.

Examples

```
double <- function(elem){elem * 2}
pred <- function(elem){elemmod2 = 0}
myCl1 <- comp(myCl, pred, fun)
```

`member` A function to test if an element is a member of a collection.

Description A function to test if an element is a member of a collection.

Usage `member(elem, myCl)`

Arguments

`elem`	a value to be checked
`myCl`	a collection to be checked

Details This is a member test function.

Value The result of the operation is a boolean value indicating whether the element exists in the collection.

Examples
```
elem <- 2
found <- member(elem, myCl)
```

intersection	A function to get elements that exist in both collections.

Description A function to get elements that exist in both collections.

Usage
```
intersection(myCl1, myCl2)
```

Arguments

myCl1	a collection
myCl2	a collection

Details

Value The result of the operation is a collection containing the intersection of the two arguments.

Examples
```
result <- intersection(myCl1, myCl2)
```

union	A function to get elements that exist in either collection.

Description A function to get elements that exist in either collection.

Usage
```
union(myCl1, myCl2)
```

Arguments

myCl1	a collection
myCl2	a collection

Details

Value The result of the operation is a collection containing the union of the two arguments.

Examples
```
result <- union(myCl1, myCl2)
```

5.4 Other Implementation Details

Collectin representation.

Overall, a directory containing DESCRIPTION file and subdirectories

- R subdirectory: library routine implementation in regular R code
- INDEX file: one-line description for each routine
- Data subdirectory: data to be loaded via lazy loading or data(), based on DESCRIPTION; supported types are code, tables, and images
- Load hooks: functions for initialization and cleanup (e.g., set options, load shared objects); this includes First.lib and Last.lib. For packages with name spaces: .onLoad, .onAttach, onUnload, .Last.lib.
- Compression may be used if recorded in DESCRIPTION file
- Others include demo, exec, inst, man, po, src, tests, NAMESPACE, configure, cleanup, LICENSE, etc

Other steps before releasing package include documentation, profiling and possibly optimization, debugging, etc

Maybe show an example on NAMESPACE file and load hooks.

6 Future work

operator overloading

User libraries

SVN

Optimization

Drupal

We would like to address some of the following technical issues in due course:

- Setup related tasks to be mostly done in load(collectionslibrary) call. This would involve setting up queues, s3 bucket setup, worker instances etc.
- As far as possible none of calls involve creation of ec2 resources tasks. i.e. queues, buckets etc.
- There may be issues considering the time related to transfer of data to and from S3. I think the transfer rate is not very fast unless amazons separate network is used.
- Also a suggestion would be to do parallel transfer, i.e. multiple data connections.
- Keep in mind that S3 objects have a max size.

- Larger goal should be run Rcollections from local machine.
- Creation/Deletions of instances involves time. For every call this would be a hit. Reusing instances should be the goal.
- Amazon instance image of R needs to be built with collections library. S3 to/from EC2 data transfer varies from small to large instances. Not sure of the exact rates but for some instances are kind of 20/25 mb/s I think.
- We have a limit of 50 instances and out of which only 25-30 may be free. Plus if you max out the use of instances, other teams might not be able to launch.
- Need to separate S3 buckets based on user or id, so that if multiple users are running the program, a segregation boundary is maintained.

7 Conclusion