

# OPEN DATA SCIENCE CONFERENCE

Burlingame | November 2nd 2017

Nov 02  
2:00 PM  
Room T2

Modeling big data with R, sparklyr, and Apache Spark

BIG DATARINTERMEDIATE

Dr. John Mount

Consulting Algorithmist/Researcher/Principal at Win-Vector LLC and  
Co-author of Practical Data Science with R



# 3 levels of “big data”

1. Need to build summaries or simple predictive models from a large amount of data.

- Can sample data and run in memory.

2. Need to apply a model or transformation to a lot of data.

- Can distribute work to a cluster.

3. Need to compute something involving all the relations between the data.

- Need big data systems and algorithms.

# Not teaching OPS/ENG

- We are loaning you single cluster accounts that are non-durable
  - temp tables disappear when you disconnect
- This means we (hopefully) don't end up running “a consulting clinic for installation bugs” (thanks Garrett Grolemond).
- It also means we are not discussing issues such as lifetime of data inside Spark and h2o, starting and stopping the cluster, pinning items into memory, or organization of storage.
- It isn't that OPS isn't important, it is just too big to be in this workshop (and varies depending on where you end up working with big data).



Configuration conflict





# Big data systems have a cost

- Big data systems usually trade high throughput for some combination of:
  - High latency
  - Limited state
  - Limited communication patterns
  - Inconvenience
- In fact they can prohibit some superior algorithms



# “Scalability! But at what COST?”

## McSherry, Isard, Murray

- Define COST as cluster size (in machines) needed to be as fast as a good in-ram (single machine, multiple CPU) implementation.
- Typical COST of “best of breed” systems in article:
  - 10
  - 100
  - 256
  - infinity

# Redefine “big data”

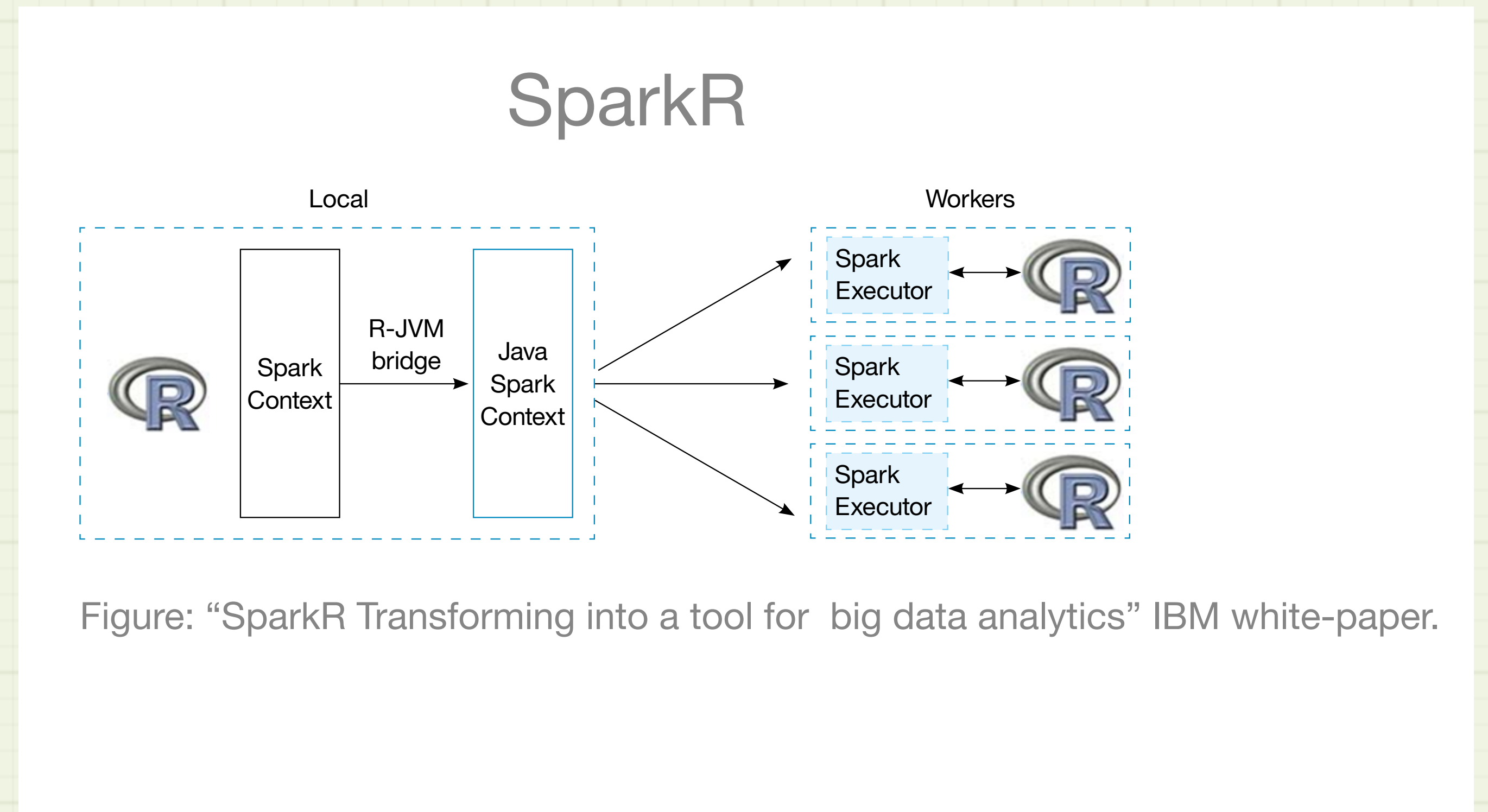
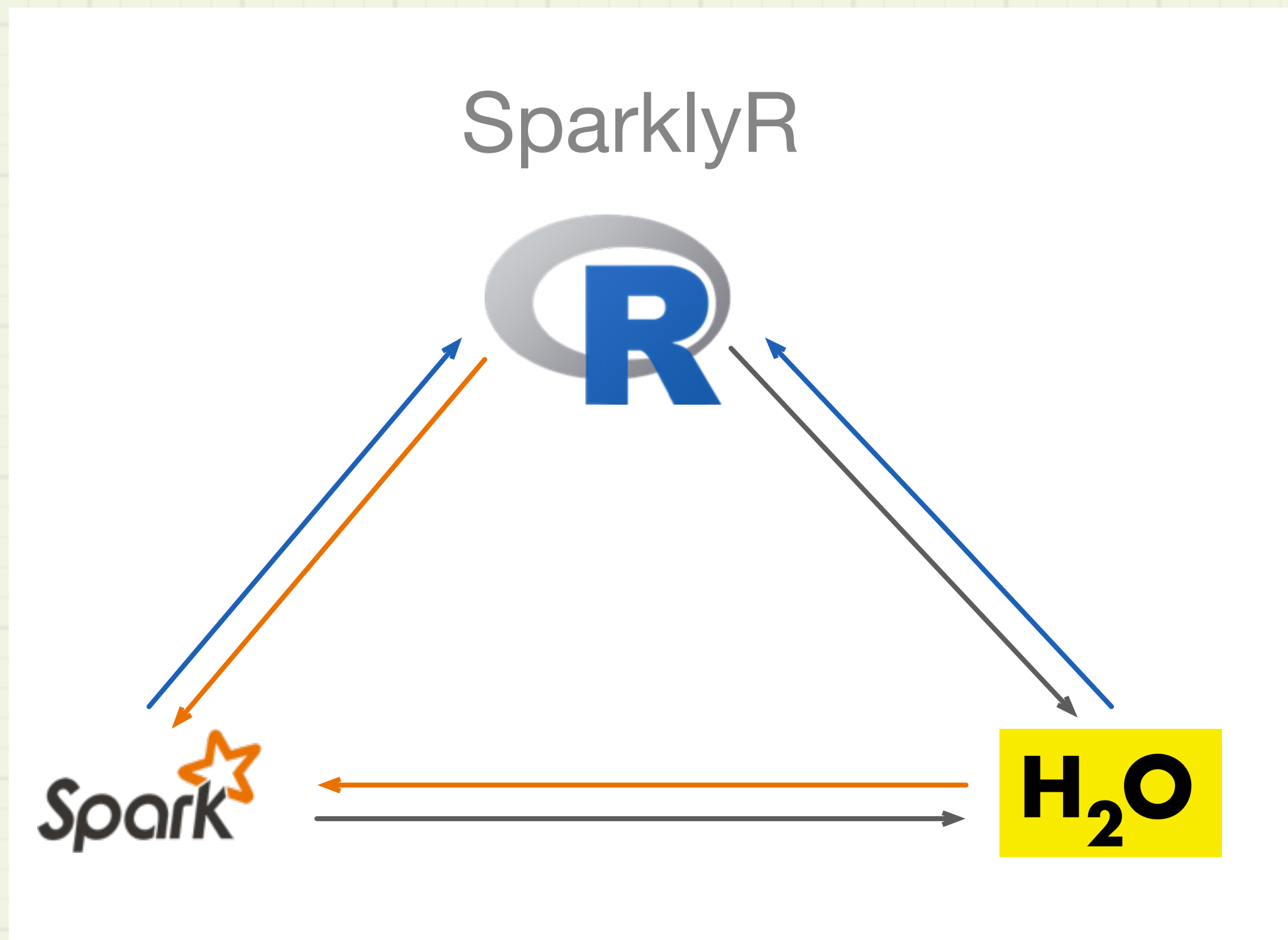
## Big Data > 1/3 RAM

# *What is Spark?*

- Open-source Apache computing engine
- Bigger-than-memory data, low-latency distributed computing
- Can integrate with the Hadoop ecosystem
- Built-in machine learning



# Architecture choices



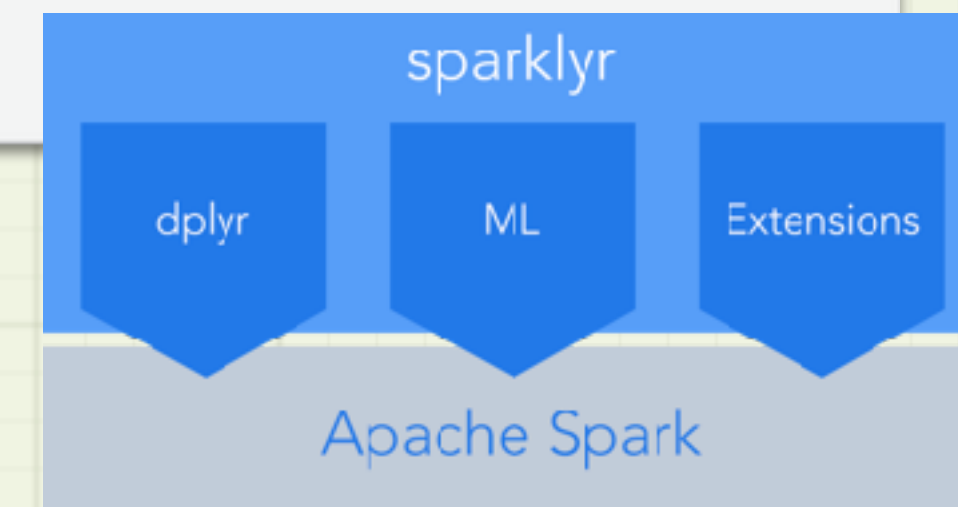
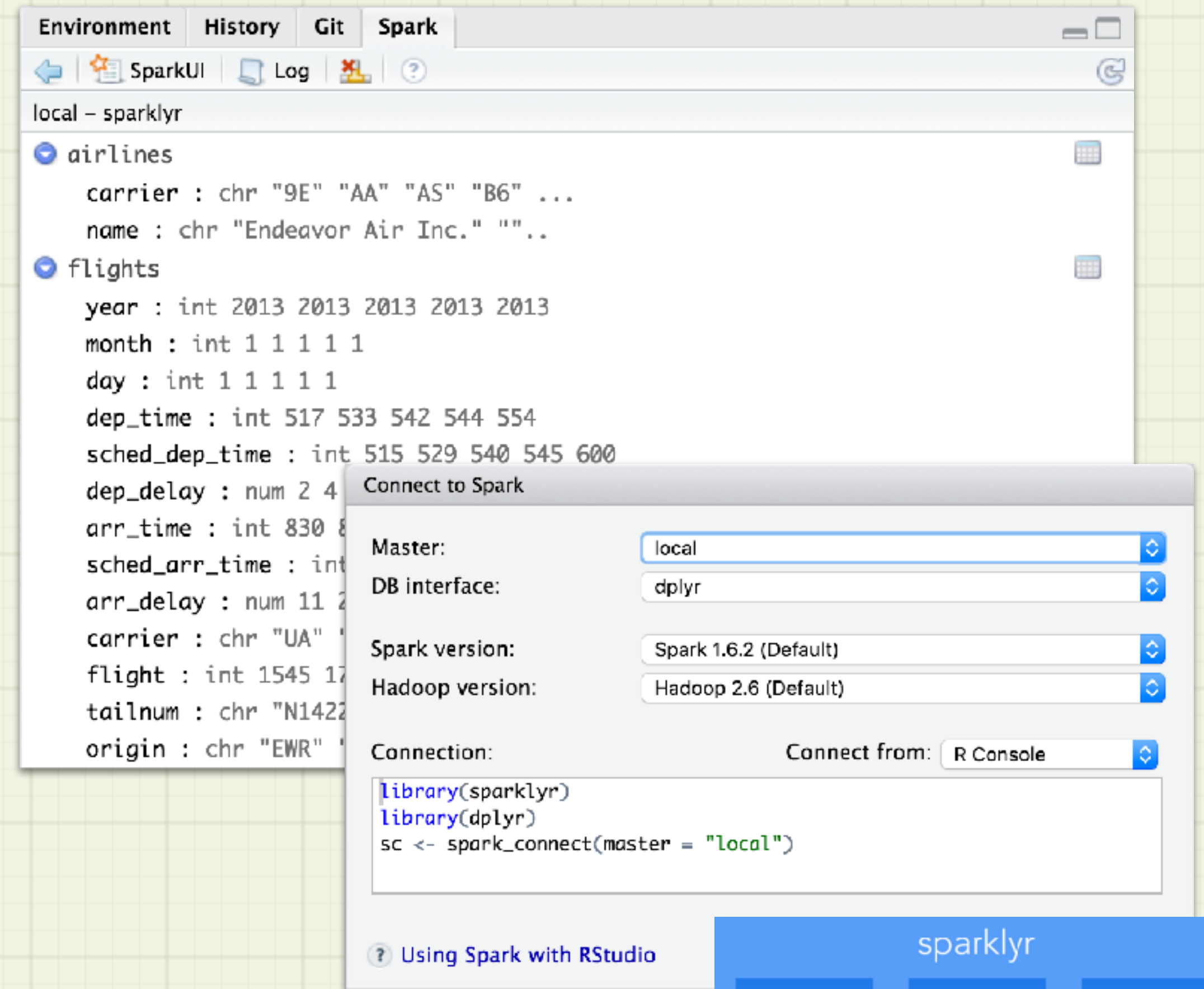


# Comparison

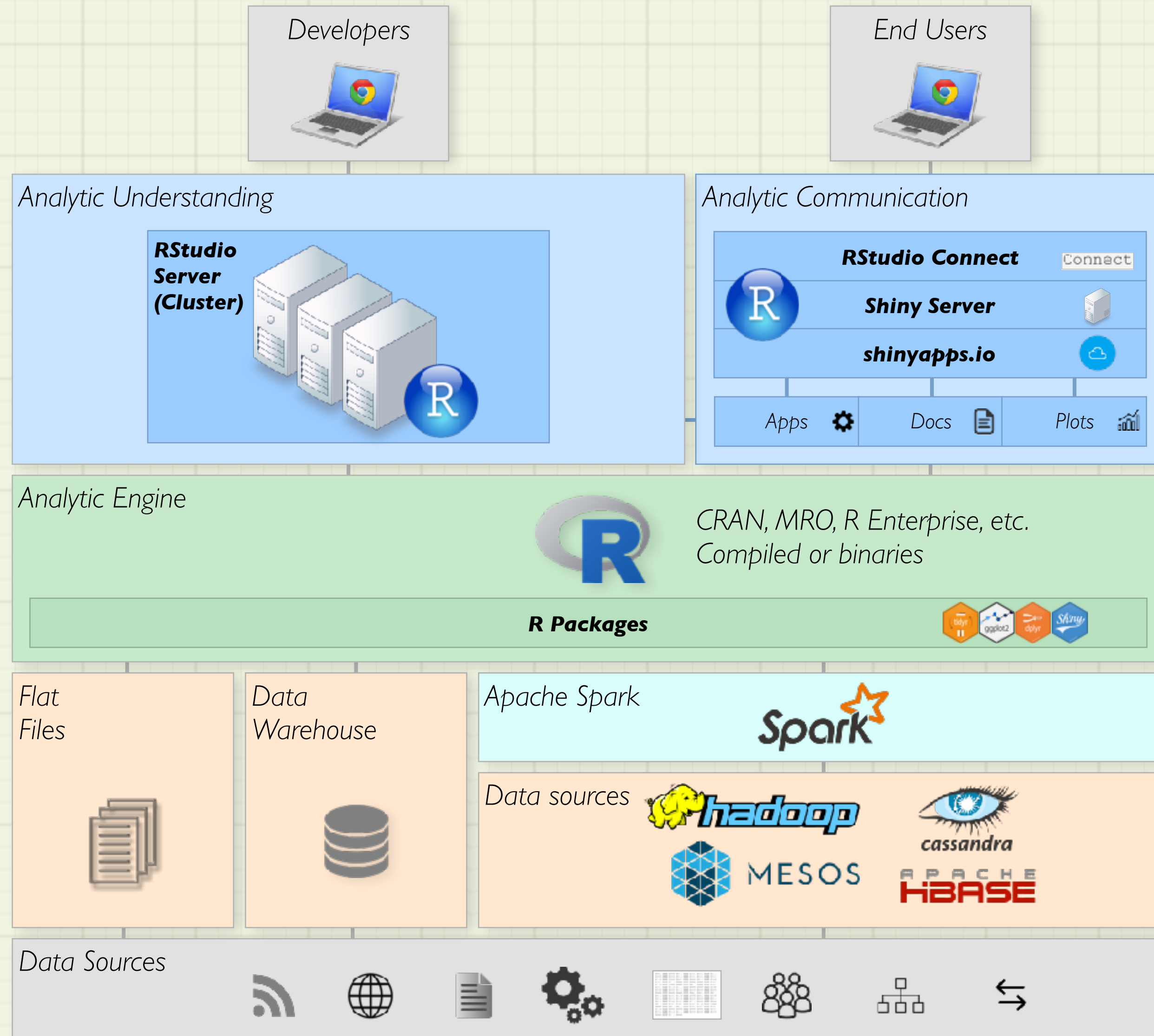
- SparklyR / h2o
  - Command remote native Spark using adapted `dplyr` notation to allow workflows *as if* you were working in R.
  - Use native SparkML, and h2o methods to do the work.
- SparkR
  - Use R to command copies of distributed R across a cluster.
    - Allows R user defined functions.
  - Not currently a `dplyr` backend.
- Which is “better”?
  - It depends on your data, infrastructure, tasks, and legacy code.
  - Also depends *a lot* on the current state of each adapter, and both are under rapid development.
  - Expect to see both going forward.
- We are going to concentrate on SparklyR.

# sparklyr

- **R package.** New, open-source package from RStudio
- **dplyr.** Complete dplyr back-end for Spark
- **IDE.** Integrated with the RStudio IDE
- **Extensible.** Extensible foundation for Spark and R



# Typical setup



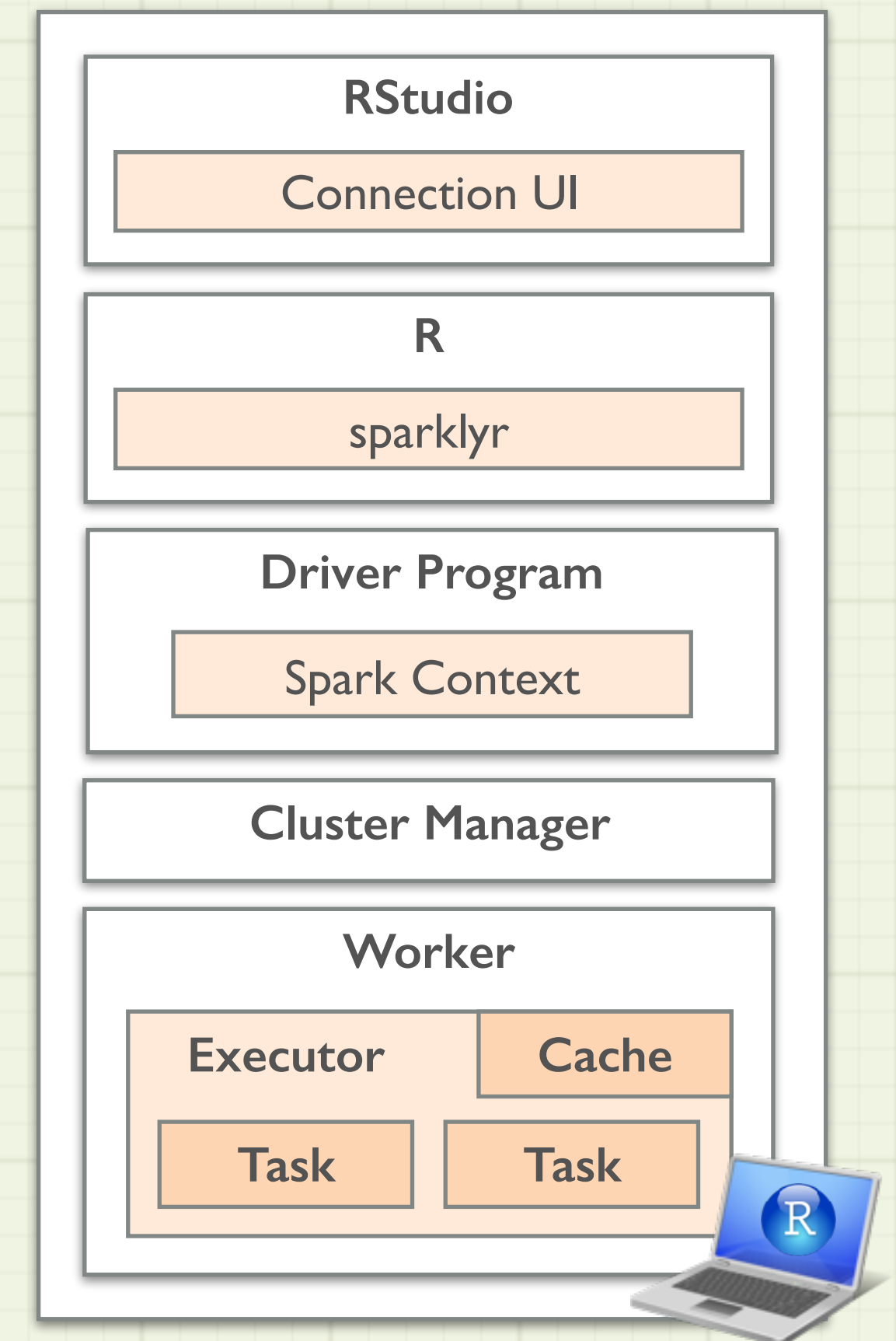
# Local Mode

```
library(sparklyr)
library(dplyr)

spark_install()

sc <- spark_connect("local")

my_tbl <- copy_to(sc, iris)
```





# *Spark SQL*

Use dplyr syntax to translate R code into Spark SQL (HiveQL)

## DPLYR

```
my_tbl %>%  
  filter(Petal_Width < 0.3) %>%  
  select(Petal_Length, Petal_Width) %>%  
  arrange(Petal_Length)
```

## SPARK SQL

```
select Petal_Length, Petal_Width  
from iris  
where Petal_Width < 0.3  
order by Petal_Length
```

# *Nota bene*

- Remote data services (intentionally) look a lot like local data.
- But there are differences.



# Examples

```
library("sparklyr")  
library("dplyr")  
sc <- spark_connect(master = "local",  
                     version = "2.0.0",  
                     hadoop_version="2.7")  
iris_tbl <- copy_to(sc, iris, "iris",  
                   overwrite= TRUE)
```

# iris\_tbl is a handle, not data

```
> str(iris)
```

```
'data.frame': 150 obs. of  5 variables:
 $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
 $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
 $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
 $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
 $ Species      : Factor w/ 3 levels "setosa","versicolor",...: 1 1 1 1 1 1 1 1 1 1 ...
```

```
> str(iris_tbl)
```

```
List of 2
 $ src:List of 1
  ..$ con:List of 11
   .. ..$ master      : chr "local[4]"
   .. ..$ method      : chr "shell"
   .. ..$ app_name     : chr "sparklyr"
   .. ..$ config       :List of 5
   .. .. ..$ sparklyr.cores.local : int 4
...
```





# fix: dplyr::glimpse()

```
> glimpse(iris_tbl)
```

```
Observations: 150
```

```
Variables: 5
```

```
$ Sepal_Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, 5.4, 4.8, 4.8, 4.3, 5.8, 5.7, ...  
$ Sepal_Width  <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, 3.7, 3.4, 3.0, 3.0, 4.0, 4.4, ...  
$ Petal_Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, 1.5, 1.6, 1.4, 1.1, 1.2, 1.5, ...  
$ Petal_Width  <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, 0.2, 0.2, 0.1, 0.1, 0.2, 0.4, ...  
$ Species      <chr> "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", "setosa", ...
```

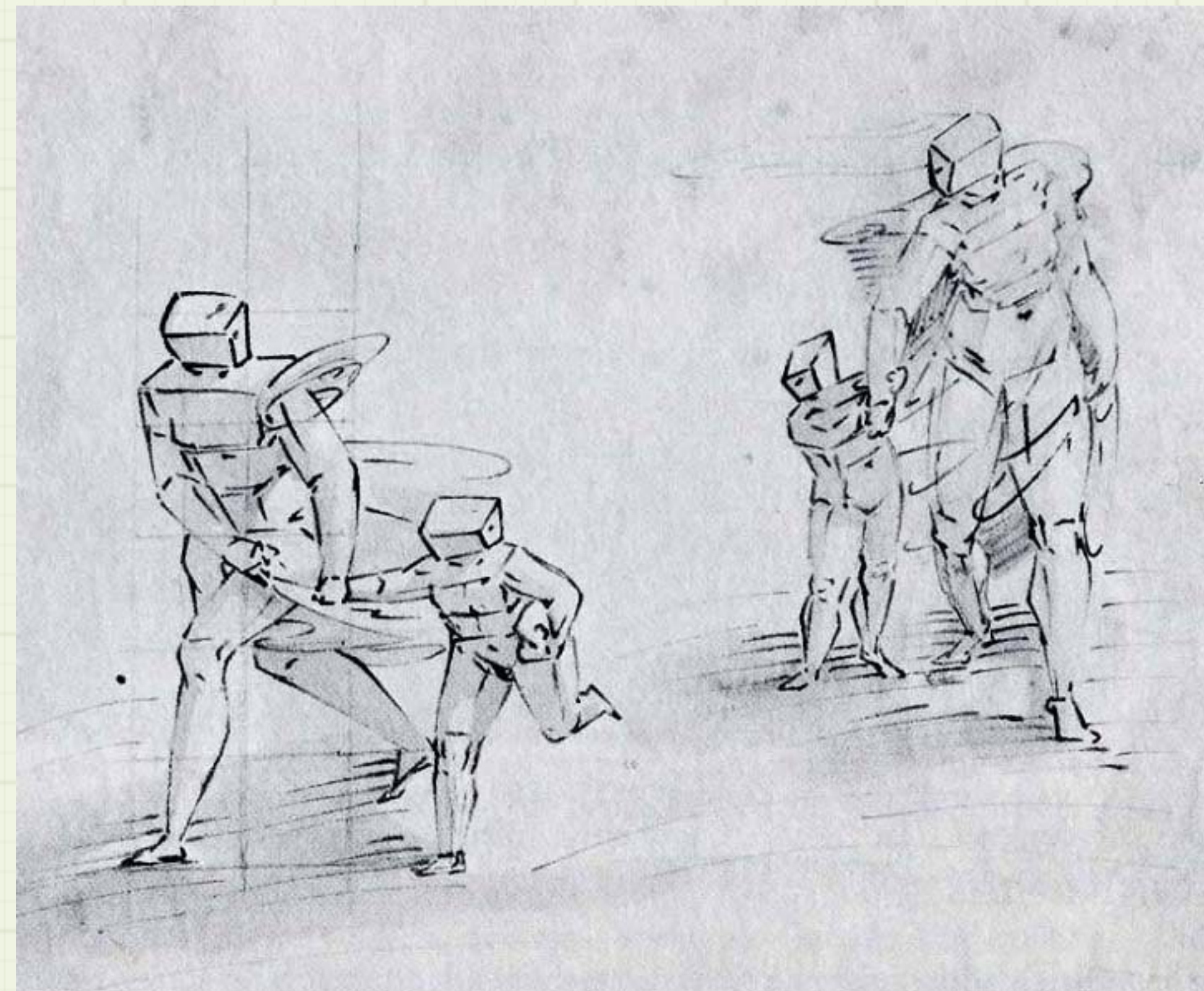


# Mnemonic

In standard R operations are wrestling in with your data



With handles you send operations out to your data



Handles (def: abstract reference to a resource) necessarily have reference semantics. Though we can make objects in the data pool express value semantics.



# again: *handles*

```
> summary(iris)
```

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
Min. :4.300	Min. :2.000	Min. :1.000	Min. :0.100	setosa :50
1st Qu.:5.100	1st Qu.:2.800	1st Qu.:1.600	1st Qu.:0.300	versicolor:50
Median :5.800	Median :3.000	Median :4.350	Median :1.300	virginica :50
Mean :5.843	Mean :3.057	Mean :3.758	Mean :1.199	
3rd Qu.:6.400	3rd Qu.:3.300	3rd Qu.:5.100	3rd Qu.:1.800	
Max. :7.900	Max. :4.400	Max. :6.900	Max. :2.500	

```
> summary(iris_tbl)
```

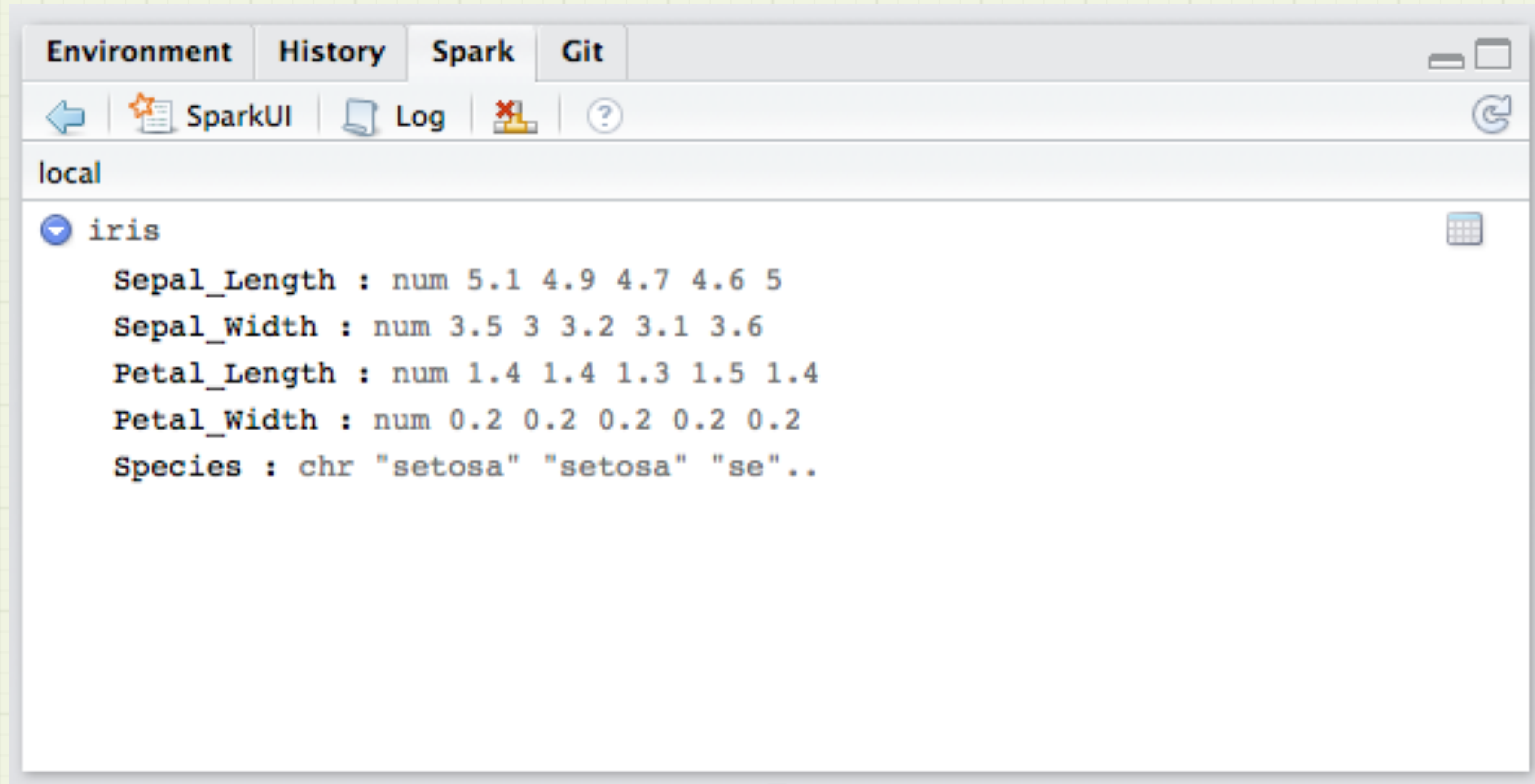
	Length	Class	Mode
src 1		src_spark	list
ops 3		op_base_remote	list

```
> broom::glance(iris_tbl)
```

```
Error: glance doesn't know how to deal with data  
of class tbl_sparktbl_sqltbl_lazytbl
```



# fix: RStudio Spark browser





# optional (experimental/expensive) fix:

## `replyr::replyr_summary()`

```
> library("replyr")  
> replyr_summary(iris_tbl, countUniqueNonNum= TRUE)
```

	column	index	class	nrows	nna	nunique	min	max	mean	sd	lexmin	lexmax
1	Sepal_Length	1	numeric	150	0	NA	4.3	7.9	5.843333	0.8280661	<NA>	<NA>
2	Sepal_Width	2	numeric	150	0	NA	2.0	4.4	3.057333	0.4358663	<NA>	<NA>
3	Petal_Length	3	numeric	150	0	NA	1.0	6.9	3.758000	1.7652982	<NA>	<NA>
4	Petal_Width	4	numeric	150	0	NA	0.1	2.5	1.199333	0.7622377	<NA>	<NA>
5	Species	5	character	150	0	3	NA	NA	NA	NA	setosa	virginica



# Remote data is not “in the tidyverse”

```
> library("tidyr")  
> iris %>% nest(-Species)
```

```
# A tibble: 3 × 2
```

Species	data
<fctr>	<list>
1 setosa	<tibble [50 × 4]>
2 versicolor	<tibble [50 × 4]>
3 virginica	<tibble [50 × 4]>

```
> iris_tbl %>% nest(-Species)
```

```
Error in UseMethod("nest_") :  
no applicable method for 'nest_' applied to an object  
of class "c('tbl_spark', 'tbl_sql', 'tbl_lazy', 'tbl')"
```



# To use the “The Split-Apply-Combine Strategy for Data Analysis”

- You must use `dplyr::group_by()`
- And restrict yourself to operators that are “group aware.”

```
> iris_tbl %>%  
  group_by(Species) %>%  
  summarize_all(funs(typical=mean))
```

Source: query [3 x 5]

Database: spark connection master=local[4] app=sparklyr local=TRUE

	Species	Sepal_Length_typical	Sepal_Width_typical	Petal_Length_typical	Petal_Width_typical
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	versicolor	5.936	2.770	4.260	1.326
2	virginica	6.588	2.974	5.552	2.026
3	setosa	5.006	3.428	1.462	0.246

# How is all this implemented?

- `dplyr` uses `sparklyr` to translate each `dplyr`-verb into SparkSQL.
- `dplyr` collects a compound query representing arbitrarily long sequences of operations on the remote (Spark) data.
- The query is lazy, and only run if and when the data is actually used in an eager calculation (such as printing).



# Query examples

```
> iris_tbl %>%  
  group_by(Species) %>%  
  summarize_all(funs(typical=mean)) %>%  
  show_query
```

```
<SQL>  
SELECT `Species`, AVG(`Sepal_Length`) AS `Sepal_Length_typical`, AVG(`Sepal_Width`) AS  
`Sepal_Width_typical`, AVG(`Petal_Length`) AS `Petal_Length_typical`, AVG(`Petal_Width`) AS  
`Petal_Width_typical`  
FROM `iris`  
GROUP BY `Species`
```

```
> iris_tbl %>%  
  head %>%  
  show_query
```

```
<SQL>  
SELECT *  
FROM `iris`  
LIMIT 6
```

# Being eager

- `dplyr::collapse()`
  - Try and simplify the the accumulated query.
- `dplyr::compute()`
  - Force computation, materialize result of query into a new table.
- `dplyr::collect()`
  - Force computation, materialize result of query into a local `tbl`.

# Lazy eval: nothing done until something triggers compute()

```
res <- iris_tbl %>%  
  group_by(zspecies) %>%      #OOOPS!  
  summarize_all(funs(typical=mean))  
# no error!
```

```
print(res)
```

Error:

org.apache.spark.sql.AnalysisException:  
cannot resolve '`zspecies`' given

```
res <- iris_tbl %>%  
  group_by(zspecies) %>%  
  summarize_all(funs(typical=mean)) %>%  
  compute()
```

Error:

org.apache.spark.sql.AnalysisException:  
cannot resolve '`zspecies`' given

```
res <- iris_tbl %>%  
  group_by(Species) %>%  
  summarize_all(funs(typical=mean)) %>%  
  show_query
```

<SQL>

SELECT `Species`, AVG(`Sepal\_Length`) ...

```
res <- iris_tbl %>%  
  group_by(Species) %>%  
  summarize_all(funs(typical=mean)) %>%  
  compute(name= 'summarizedIris') %>%  
  show_query
```

<SQL>

SELECT \*  
FROM `summarizedIris`



# Be cautious with remote data

- Semantics may be different than R conventions.
- May have limited ability to represent NA/NULL.
- Will not be able to represent factors other than as strings.
- May have different integer and floating point arithmetic rounding and rules.
- May have different column names and quoting/escaping conventions.
- Do not accept arbitrary R user define functions.
  - Will substitute many common R functions by name.
- May not have easy to access ranking and other window functions.





# The biggest difference

- Most remote data sources do not:
  - Guarantee row order
  - Support row-names
- These are deliberately *not* included as relational concepts.
- R local data frames guarantee row order
  - Many calculations depend on this
  - These calculations will not be correct on remote data sources.



# dplyr::mutate example

```
> iris_tbl %>%  
  summarize(mx = mean(Sepal_Length))
```

Source: query [1 x 1]

Database: spark connection master=local[4] app=sparklyr local=TRUE

```
      mx  
    <dbl>  
1 5.843333
```

```
> iris %>%  
  summarize(mx = median(Sepal.Length))
```

```
      mx  
1 5.8
```

```
> iris_tbl %>%  
  summarize(mx = median(Sepal_Length))
```

Error: org.apache.spark.sql.AnalysisException: Undefined function: 'MEDIAN'. This function is neither a registered temporary function nor a permanent function registered in the database 'default'.; line 1 pos 7



# Spark union example

```
> d1 <- data_frame(x= 1, y= 'a')
> d2 <- data_frame(y= 'b', x= 2)
```

```
> bind_rows(d1, d2)
```

```
# A tibble: 2 × 2
```

	x	y
	<dbl>	<chr>
1	1	a
2	2	b

```
> union(d1, d2)
```

```
# A tibble: 2 × 2
```

	x	y
	<dbl>	<chr>
1	2	b
2	1	a

```
> d1s <- copy_to(sc, d1)
```

```
> d2s <- copy_to(sc, d2)
```

```
> bind_rows(d1s, d2s)
```

```
Error: incompatible sizes (1 != 2)
```

```
> union(d1s, d2s)
```

```
Source:   query [2 x 2]
```

```
Database: spark connection
```

```
master=local[4] app=sparklyr local=TRUE
```

	x	y
	<chr>	<chr>
1	b	2.0
2	1.0	a

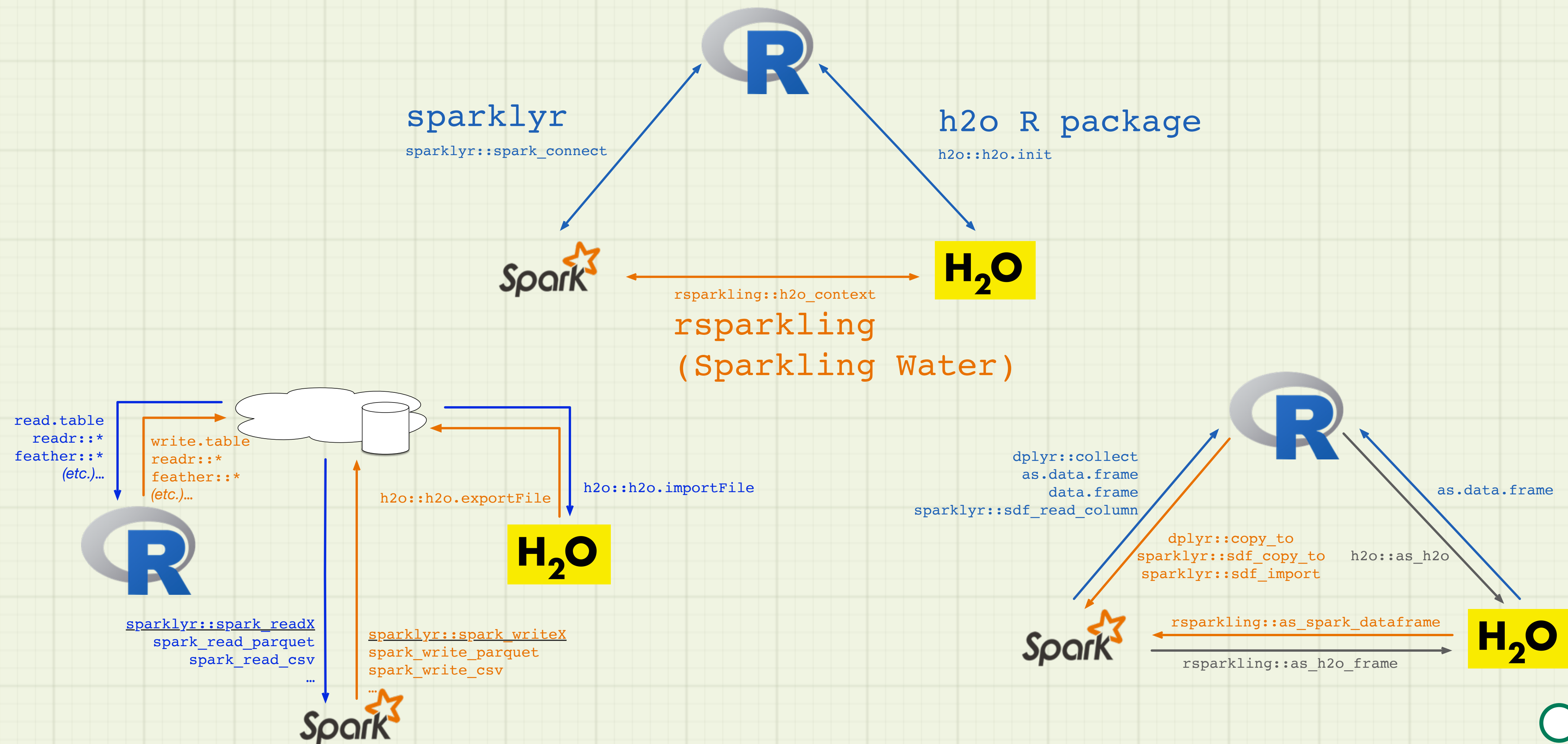


# Some commands to remember

Command	What it does
<code>DBI::dbListTables(sc)</code>	Lists data items in Spark
<code>dplyr::tbl(sc, 'iris')</code>	Build a handle pointing to the Spark object with the given name
<code>dplyr::copy_to(sc, iris, "iris")</code>	Copy data from R to Spark and choose name of result
<code>dplyr::collect(iris_tbl)</code>	Copy data from Spark to R
<code>dplyr::db_drop_table(sc, 'iris')</code>	Drop object by name



# Please keep the “three island slides” as a handy reference



# Quality of implementation varies by service provider

Service provider	R package	Quality of the experience
SQLite	RSQLite	excellent
PostgreSQL	RPostgreSQL	excellent
Spark2.x.x	SparklyR	excellent
Spark1.6.2	SparklyR	passable
MySQL	RMySQL	low

# Also consider using Spark-specific commands

- **sdf\_\*** : Use spark data frame commands to manipulate data frames.
  - Examples: sdf\_partition, sdf\_predict, sdf\_sample
- **ft\_\*** : Use feature transforms to manipulate features.
  - Examples: ft\_bucketizer, ft\_index\_to\_string
- **ml\_\*** : Use machine learning algorithms to train models.
  - Examples: ml\_kmeans, ml\_logistic\_regression, ml\_pca