

R on Databricks Compendium

Zac Davies

2024-06-20


Table of contents

What is this?	4
I Mlflow	5
1 Log R Models to Unity Catalog	6
1.1 Unity Catalog Model Requirements	6
1.2 Working Through the Solution	8
1.2.1 Patching <code>mlflow_log_model</code>	9
1.2.2 Logging Model with a Signature	10
1.2.3 Registration to Unity Catalog	10
1.3 Fixing <code>mlflow</code>	12
2 Load R Models from Unity Catalog	13
2.1 How?	13
2.2 Why is this so simple?	13
2.2.1 The details	13
II Package Management	15
3 Faster Package Installs	16
3.1 Setting Repo within Notebook	16
3.2 Cluster Environment Variable & Init Script	17
3.3 Setting Repo for Cluster Library	18
4 Persisting Packages	19
4.1 Where Packages are Installed	19
4.2 Persisting a Package	20
4.3 Adjusting <code>.libPaths()</code>	21
4.3.1 Ordering	21
4.3.2 Helpful Functions	22
4.3.3 Avoiding Repetition	23
4.4 Organising Packages	24
5 {renv}	25

III	Data Engineering	26
6	{dbplyr} and {odbc}	27
7	{SparkR} to {sparklyr}	28
7.1	Introduction	28
7.1.1	Overview of {SparkR} and {sparklyr}	28
7.2	Environment setup	28
7.2.1	Installation	28
7.2.2	Connecting to Spark	29
7.3	Reading & Writing Data	29
7.3.1	TL;DR	29
7.3.2	Loading Data	30
7.3.3	Creating Views	30
7.3.4	Writing Data	31
7.3.5	Reading Data	32
7.4	Processing Data	33
7.4.1	Select, Filter	33
7.4.2	Adding Columns	34
7.4.3	Grouping & Aggregation	34
7.4.4	Joins	34
7.5	User Defined Functions (UDFs)	35
7.6	Machine learning	39
7.6.1	Linear regression	39
7.6.2	K-means clustering	39
7.7	Performance and optimization	40
7.7.1	Collecting	40
7.7.2	In-Memory Partitioning	41
7.7.3	Caching	42
IV	Miscellaneous	43
8	{htmlwidgets} in notebooks	44
9	OAuth {odbc}	46
9.1	U2M Example	46

What is this?

 Under Development

 This is not intended to be an exhaustive guide, it's currently a place for me to document and collate useful information regarding R on Databricks.

There aren't many definitive examples of how to use R and Databricks together - hopefully the content here will serve as a useful resource.

Part I

Mlflow

1 Log R Models to Unity Catalog

Currently {mlflow} doesn't support directly logging R models to Unity Catalog. This section will cover why, and then how to overcome each roadblock.

1.1 Unity Catalog Model Requirements

For models to be logged into Unity Catalog they **must** have a [model signature](#). The Model signature defines the schema for model inputs/outputs.

Typically when using python this would be inferred via [model input examples](#). Input examples are optional but strongly recommended.

The documentation discusses [signature enforcement](#), currently this isn't implemented for R. Therefore you can decide if the signature is a dummy value for the sake of moving forward, or correct to clearly communicate the behaviour of the model.

! Important

It's important to clarify that for python the signature is enforced at time of inference *not* when registering the model to Unity Catalog.

The signature correctness is not validated when registering the model, it just has to be syntactically valid.

So, let's look at the [existing code](#) to log models in the `crate` flavour:

```
mlflow_save_model.crate <- function(model, path, model_spec=list(), ...) {  
  if (dir.exists(path)) unlink(path, recursive = TRUE) ①  
  dir.create(path)  
  
  serialized <- serialize(model, NULL) ②  
  
  saveRDS(  
    serialized, ③  
    file.path(path, "crate.bin")  
  )  
}
```

```

model_spec$flavors <- append(model_spec$flavors, list(
  crate = list(
    version = "0.1.0",
    model = "crate.bin"
  )
))
mlflow_write_model_spec(path, model_spec)
model_spec
}

```

- ① Create the directory to save the model if it doesn't exist, if it does, empty it
- ② Serialise the model, which is an object of class `crate` (from `{carrier}` package)
- ③ Save the serialised model via `saveRDS` to the directory as `crate.bin`
- ④ Define the model specification, this contains metadata required ensure reproducibility. In this case it's only specifying a version and what file the model can be found within.

The missing puzzle piece is the definition of a signature. Instead of explicitly adding code to the crate flavour itself, we'll take advantage of the `model_spec` parameter.

That means we can focus on `mlflow::mlflow_log_model` directly, we'd need to adjust the code as follows:

```

mlflow_log_model <- function(model, artifact_path, signature, ...) {
  temp_path <- fs::path_temp(artifact_path)

  model_spec <- mlflow_save_model(
    model, path = temp_path,
    model_spec = list(
      utc_time_created = mlflow_timestamp(),
      run_id = mlflow_get_active_run_id_or_start_run(),
      artifact_path = artifact_path,
      flavors = list(),
      signature = signature
    ),
    ...)

  res <- mlflow_log_artifact(path = temp_path, artifact_path = artifact_path)

  tryCatch({
    mlflow::mlflow_record_logged_model(model_spec)
  },

```

```

error = function(e) {
  warning(
    paste("Logging model metadata to the tracking server has failed, possibly due to older
          server version. The model artifacts have been logged successfully.",
          "In addition to exporting model artifacts, MLflow clients 1.7.0 and above",
          "attempt to record model metadata to the tracking store. If logging to a",
          "mlflow server via REST, consider upgrading the server version to MLflow",
          "1.7.0 or above.", sep=" ")
  )
})
res
}

```

- ① Add a new parameter `signature`
- ② Propagate `signature` to the `model_spec` parameter when invoking `mlflow::mlflow_save_model`

Benefit of this method is that all model flavors will inherit the capability to log a signature.

1.2 Working Through the Solution

To keep things simple we'll be logging a “model” (a function which divides by two).

```

half <- function(x) x / 2

half(1:10)

```

```
[1] 0.5 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0
```

Without any changes, a simplified example of logging to `{mlflow}` would look like:

```

library(carrier)
library(mlflow)

with(mlflow_start_run(), {
  # typically you'd do more modelling related activities here
  model <- carrier::crate(~half(.x))
  mlflow_log_model(model, "model")
})

```

- ① As discussed earlier, this is where things start to go awry with respect to Unity Catalog

1.2.1 Patching mlflow_log_model

Note

Technically, patching `mlflow_log_model` isn't the only way to achieve this fix - you could modify the yaml after it's written.

I won't be showing that method as It's just as tedious and can change depending on the model flavour (with respect to where artifacts may reside), patching is more robust.

```
mlflow_log_model <- function(model, artifact_path, signature = NULL, ...) { ①

  format_signature <- function(signature) { ②
    lapply(signature, function(x) {
      jsonlite::toJSON(x, auto_unbox = TRUE)
    })
  }

  temp_path <- fs::path_temp(artifact_path)

  model_spec <- mlflow_save_model(model, path = temp_path, model_spec = list(
    utc_time_created = mlflow::mlflow_timestamp(),
    run_id = mlflow::mlflow_get_active_run_id_or_start_run(),
    artifact_path = artifact_path,
    flavors = list(),
    signature = format_signature(signature) ③
  ), ...)

  res <- mlflow_log_artifact(path = temp_path, artifact_path = artifact_path)

  tryCatch({
    mlflow::mlflow_record_logged_model(model_spec)
  },
  error = function(e) {
    warning(
      paste("Logging model metadata to the tracking server has failed, possibly due to older",
            "server version. The model artifacts have been logged successfully.",
            "In addition to exporting model artifacts, MLflow clients 1.7.0 and above",
            "attempt to record model metadata to the tracking store. If logging to a",
            "mlflow server via REST, consider upgrading the server version to MLflow",
            "1.7.0 or above.", sep=" ")
    )
  })
}
```

```

    res
  }

# overriding the function in the existing mlflow namespace
assignInNamespace("mlflow_log_model", mlflow_log_model, ns = "mlflow")

```

- ① `signature` has been added to function parameters, it's defaulting to `NULL` so that existing code won't break
- ② Adding `format_signature` function so don't need to write JSON by hand, adding this within function for simplicity
- ③ `signature` is propagated to `mlflow_save_model`'s `model_spec` parameter which will write a valid signature

1.2.2 Logging Model with a Signature

```

with(mlflow_start_run(), {
  # typically you'd do more modelling related activities here
  model <- carrier::crate(~half(.x))
  signature <- list(
    inputs = list(list(type = "double", name = "x")),
    outputs = list(list(type = "double"))
  )
  mlflow_log_model(model, "model", signature = signature)
})

```

- ① Explicitly defining a `signature`, a list that contains `input` and `outputs`, each are lists of lists respectively
- ② Passing defined signature to the now patched `mlflow_log_model` function

1.2.3 Registration to Unity Catalog

Now that the prerequisite of adding a model signature has been satisfied there is one last hurdle to overcome, registering to Unity Catalog.

The hurdle is due to `{mlflow}` not having been updated yet to support registration to Unity Catalog directly. The easiest way to overcome this is to simply register the run via python.

For example:

```
import mlflow
mlflow.set_registry_uri("databricks-uc")

catalog = "main"
schema = "default"
model_name = "my_model"
run_uri = "runs:/<run_id>/model"

mlflow.register_model(run_uri, f"{catalog}.{schema}.{model_name}")
```

①

① You'll need to either get the `run_uri` programmatically or copy it manually

To do this with R you'll need to make a series of requests to Unity Catalog endpoints for registering model, the specific steps are:

1. (Optional) Create a new model in Unity Catalog

- POST request on `/api/2.0/mlflow/unity-catalog/registered-models/create`
 - name: 3 tiered namespace (e.g. `main.default.my_model`)

2. Create a version for the model

- POST request on `/api/2.0/mlflow/unity-catalog/model-versions/create`
 - name: 3 tiered namespace (e.g. `main.default.my_model`)
 - source: URI indicating the location of the model artifacts
 - run_id: `run_id` from tracking server that generated the model
 - run_tracking_server_id: Workspace ID of run that generated the model
- This will return `storage_location` and `version`

3. Copy model artifacts

- Need to copy the artifacts to `storage_location` from step (2)

4. Finalise the model version

- POST request on `/api/2.0/mlflow/unity-catalog/model-versions/finalize`
 - name: 3 tiered namespace (e.g. `main.default.my_model`)
 - version: `version` returned from step (2)

It's *considerably* easier to just use Python to register the model at this time.

1.3 Fixing mlflow

Ideally this page wouldn't exist and `{mlflow}` would support Unity Catalog. Hopefully some-time soon I find the time to make a pull request myself - until then this serves as a guide.

2 Load R Models from Unity Catalog

{mlflow} doesn't support directly logging R models to Unity Catalog (without jumping through a few hoops). However, you can easily load models from Unity Catalog.

2.1 How?

```
# loading 'prod' alias of `zacdav`.`default`.`my_r_model`  
model <- mlflow_load_model("models:/zacdav.default.my_r_model@prod")
```

Yep, very straightforward, it just works as-is.

2.2 Why is this so simple?

A natural question is “*why did I have to go through all the pain to register the model, but loading just works?*”

The quick answer is that {mlflow} doesn't have internal consistency with how it operates, some methods use direct API calls (e.g. registration) and where as others defer to the [mlflow CLI](#) via a system call (e.g. loading).

2.2.1 The details

As of version 2.20.1 we can split {mlflow} into those that directly invokes mlflow's REST API's via R, or the mlflow CLI.

Table 2.1: Which functions use REST or CLI clients within {mlflow}

	Invokes REST Client	Invokes CLI Client
Not relevant to Unity Catalog (e.g. experiment tracking related)	<code>mlflow_create_experiment()</code> <code>mlflow_search_experiments()</code> <code>mlflow_set_experiment_tag()</code> <code>mlflow_get_experiment()</code> <code>mlflow_delete_experiment()</code> <code>mlflow_delete_experiment()</code> <code>mlflow_rename_experiment()</code> <code>mlflow_transition_model_version_stage()</code> <code>mlflow_record_logged_model()</code>	
Relevant to Unity Catalog (e.g. model registry)	<code>mlflow_set_model_version_tag()</code> <code>mlflow_search_registered_models()</code> <code>mlflow_create_registered_model()</code> <code>mlflow_get_registered_model()</code> <code>mlflow_rename_registered_model()</code> <code>mlflow_update_registered_model()</code> <code>mlflow_delete_registered_model()</code> <code>mlflow_get_latest_versions()</code> <code>mlflow_create_model_version()</code> <code>mlflow_get_model_version()</code> <code>mlflow_update_model_version()</code> <code>mlflow_log_model()</code> *	<code>mlflow_download_artifacts_from_uri()</code> <code>mlflow_download_artifacts()</code> <code>mlflow_log_artifact()</code> <code>mlflow_load_model()</code>

`mlflow_load_model()` calls `mlflow_download_artifacts_from_uri()`, which in turn uses the CLI directly. The CLI was updated to support Unity Catalog, hence why it ‘*just works*’ and the functions inherit Unity Catalog compatibility without code changes.

You may notice `mlflow_log_model()` has an asterix, that is because it uniquely calls CLI client (`mlflow_log_artifact()`) and REST client (`mlflow_record_logged_model()`), I’ve placed it in the REST column for now.

This is part of the problem discussed previously in Section 1.2.3. Unfortunately the CLI doesn’t expose the model registry methods required to support registering models to Unity Catalog, these need to be implemented as part of the REST client in {mlflow}.

Part II

Package Management

3 Faster Package Installs

You may have noticed that when installing packages in the notebook it can take a while. It could be minutes, hours in extreme cases, to install the suite of packages your project requires. This is especially tedious if you need to do this every time a job runs, or each morning when your cluster is started.

Clusters are ephemeral and by default have no persistent storage, therefore installed packages will not be available on restart.

By default Databricks installs packages from [CRAN](#). CRAN does not provide pre-compiled binaries for Linux (Databricks clusters' underlying virtual machines are Linux, Ubuntu specifically).

[Posit](#) to save the day! [Posit provides a public package manager](#) that has all packages from CRAN (and [Bioconductor](#)!). There is a [helpful wizard](#) to get started.

With our new found knowledge we can make installing R packages within Databricks significantly faster. There are multiple ways to solve this, each differing slightly, but fundamentally the same.

3.1 Setting Repo within Notebook

The quickest method is to follow the [wizard](#) and adjust the `repos` option:

```
# set the user agent string otherwise pre-compiled binarys aren't used
# e.g. selecting Ubuntu 22.04 in wizard
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["platform"])),
  repos = "https://packagemanager.posit.co/cran/__linux__/jammy/latest"
)
```

① `HTTPUserAgent` is [required when using R 3.6 or later](#)

This works well but not all versions of the [Databricks Runtime](#) use the same version of Ubuntu.

It's easier to detect the Ubuntu [release code name](#) dynamically:


```
release <- system("lsb_release -c --short", intern = T) ①

# set the user agent string otherwise pre-compiled binarys aren't used
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["platform"])),
  repos = paste0("https://packagemanager.posit.co/cran/__linux__/", release, "/latest")
)
```

① `system` is used to run the command to retrieve the release code name

The downside of this method is that it requires every notebook to adjust the `repos` and `HTTPUserAgent` options.

3.2 Cluster Environment Variable & Init Script

Databricks clusters allow specification of [environment variables](#), there is a specific variable (`DATABRICKS_DEFAULT_R_REPOS`) that can be set to adjust the default repository for the entire cluster.

You can again refer to the [wizard](#), the environment variables section of cluster should be:

```
DATABRICKS_DEFAULT_R_REPOS=<posit-package-manager-url-goes-here>
```

Unfortunately this isn't as dynamic as the first option and you still need to set the `HTTPUserAgent` in `Rprofile.site` via an [init script](#).

The init script will be:

```
#!/bin/bash
# Append changes to Rprofile.site
cat <<EOF >> "/etc/R/Rprofile.site"
options(
  HTTPUserAgent = sprintf("R/%s R (%s)", getRversion(), paste(getRversion(), R.version["platform"])),
)
EOF
```

! Important

Due to how Databricks starts up the R shell for notebook sessions it's not straightforward to adjust the `repos` option in an init script alone.

`DATABRICKS_DEFAULT_R_REPOS` is referenced as part of the startup process *after*

`Rprofile.site` is executed and will override any earlier attempt to adjust `repos`. Therefore you'll need to use both the init script and the environment variable configuration.

3.3 Setting Repo for Cluster Library

i Note

Similar to setting `DATABRICKS_DEFAULT_R_REPOS` this requires the `HTTPUserAgent` also to be set and it's unlikely to be helpful other than for its purpose of installing a package to make it available for all cluster users.

[Cluster libraries](#) can install R packages and support specification of the repository.

4 Persisting Packages

! Important

Evaluate if [faster package installs](#) is able to solve any installation pain-points before investigating persisting packages, faster installs is easier to set-up and manage.

Databricks clusters are ephemeral and therefore any installed packages will not be available on restart. If the cluster has cluster libraries defined then those libraries are installed after the cluster is started - this can be time consuming when there are multiple packages.

The article on [faster package installs](#) details how to reduce the time it takes to install each package. Faster installs are great, but sometimes it's preferable to not install at all and persist the packages required, similar to how you'd use R locally.

4.1 Where Packages are Installed

When installing packages with `install.packages` the default behaviour is that they'll be installed to the first element of `.libPaths()`.

`.libPaths()` returns the paths of "R library trees", directories that R packages can reside. When you load a package it will be loaded from the first location it is found as dictated by `.libPaths()`.

When working within a Databricks notebook `.libPaths()` will return 6 values by default, in order they are:

Path	Details
<code>/local_disk0/.ephemeral_nfs/envs/rEnv-<session-id></code>	The first location is always a notebook specific directory, this is what allows each notebook session to have different libraries installed .

Path	Details
/databricks/spark/R/lib	Only {SparkR} is found here
/local_disk0/.ephemeral_nfs/cluster_libraries/r	Cluster libraries - you could also install packages here explicitly to share amongst all users (e.g. <code>lib</code> parameter of <code>install.packages</code>)
/usr/local/lib/R/site-library	Packages built into the Databricks Runtime
/usr/lib/R/site-library	Empty
/usr/lib/R/library	Base R packages

It's important to understand that the order defines the default behaviour as it's possible to add or remove values in `.libPaths()`. You'll almost certainly be adding values, there's little reason to remove values.

i Note

All following examples will use [Unity Catalog Volumes](#). [DBFS](#) can be used but it's not recommended.

4.2 Persisting a Package

The recommended approach is to first install the library(s) you want to persist on a cluster via a notebook.

For example, let's persist `{leaflet}` to a volume:

```
install.packages("leaflet") ①
# determine where the package was installed
pkg_location <- find.package("leaflet") ②
# move package to volume
new_pkg_location <- "/Volumes/<catalog>/<schema>/<volume>/my_packages" ③
file.copy(from = pkg_location, to = new_pkg_location, recursive = TRUE) ④
```

- ① Installing `{leaflet}`
- ② Return path to package files, from what was explained before we know this will be a sub-directory of `.libPaths()` first path
- ③ Define the path to volume where package will be persisted, make sure to adjust as needed
- ④ Copy the folder contents recursively to the volume

At this point the package is persisted, but if you restart the cluster or detach and reattach and try to load `{leaflet}` it will fail to load.

The last step is to adjust `.libPaths()` to include the volume path. You could make it the first value by:

```
# adjust .libPaths
.libPaths(c(new_pkg_location, .libPaths()))
```

①

- ① I recommend against making it the first value, will detail why in [Ordering](#)

4.3 Adjusting `.libPaths()`

4.3.1 Ordering

Given that `.libPaths()` can return 6 values in a notebook you might wonder if there a “best” position to add your new volume path(s) to, that will depend on how you want packages to behave.

A safe default is to add a path *after* the cluster libraries location (currently 3rd), this will make it appear as if the Databricks Runtime has been extended to include packages in the volume path(s).

Alternatively you could add it after the first path and all users will still have the notebook scope package behaviour by default but cluster libraries may not load if they appear in the earlier paths under a different version.

It will be up to you to decide what works best.

! Important

I don't recommend pre-pending `.libPaths()` with volume paths as packages will attempt to install to the first value and you cannot directly install packages to a volume path (due to volumes being backed onto cloud storage). This is why the example for persisting copies after installation.

An example of adjusting `.libPaths()` looks like:

```
volume_pkgs <- "/Volumes/<catalog>/<schema>/<volume>/my_packages"
.libPaths(new = append(.libPaths(), volume_pkgs, after = 3))
```

4.3.2 Helpful Functions

The examples can be used to build a set of functions to make this easier.

Copying a Package

```
copy_package <- function(name, destination) {
  package_loc <- find.package(name)
  file.copy(from = package_loc, to = destination, recursive = TRUE)
}

# e.g. move {ggplot2} to volume
copy_package("ggplot2", "/Volumes/<catalog>/<schema>/<volume>/my_packages")
```

Alter .libPaths()

```
add_lib_paths <- function(path, after, version = FALSE) {
  if (version) {
    rver <- getRversion()
    lib_path <- file.path(path, rver)
  } else {
    lib_path <- file.path(path)
  }

  # ensure directory exists
  if (!file.exists(lib_path)) {
    dir.create(lib_path, recursive = TRUE)
  }

  lib_path <- normalizePath(lib_path, "/")

  message("primary package path is now ", lib_path)
  .libPaths(new = append(.libPaths(), lib_path, after = after))
  lib_path
}
```

- ① Allows specifying `version` as `TRUE` or `FALSE` to suffix the supplied `path` with the current R version

4.3.3 Avoiding Repetition

To avoid manually adjusting `.libPaths()` every notebook you can craft an [init script](#) or set [environment variables](#), depending on the desired outcome.

Caution

In practice this interferes with how Databricks sets up the environment, validate any changes thoroughly before rolling out to users.

4.3.3.1 Init Script

Note

This example appends to the existing `Renviron.site` file to ensure any settings defined as part of runtime are preserved.

The last two lines of the script are setting `R_LIBS_SITE` and `R_LIBS_USER`. Changing these lines can give you granular control over order for anything after the 1st value of `.libPaths()` as it's injected when the notebook session starts.

```
#!/bin/bash
volume_pkgs=/Volumes/<catalog>/<schema>/<volume>/my_packages
cat <<EOF >> "/etc/R/Renviron.site"
R_LIBS_USER=%U:/databricks/spark/R/lib:/local_disk0/.ephemeral_nfs/cluster_libraries/r:$volume_pkgs
EOF
```

- ① Define the path(s) to add to `R_LIBS_USER`
- ② Append line to `/etc/R/Renviron.site` with location after cluster libraries, you can rearrange the paths as long as they remain : separated

4.3.3.2 Environment Variables

Caution

How the Databricks Runtime defines and uses the R environment variables is something that may change and should be tested carefully, especially if upgrading runtime versions.

There are particular environment variables (`R_LIBS`, `R_LIBS_USER`, `R_LIBS_SITE`) that can be set to initialise the library search path (`.libPaths()`).

`R_LIBS` and `R_LIBS_USER` are defined as part of start-up processes in Databricks Runtime and they'll be overridden, it's easier to adjust via an [Init Script](#).

`R_LIBS_SITE` can be set via an [environment variable](#) but is referenced by `/etc/R/Renviron.site` and will provides limited control over where the path will appear in the `.libPaths()` order (it will appear 5th, after the packages included in the Databricks runtime) unless using an init script to alter `/etc/R/Renviron.site` directly.

4.4 Organising Packages

When going down this route of persisting packages you should consider how this is organised and managed long term to avoid making things messy.

Some practices you can consider include:

- Maintaining directories of packages per project, team, or user
- Ensuring directories are specific to an R version (and potentially even Databricks Runtime version)
- Coupling the use of persistence with `{renv}`

5 {renv}

 Under Development

Part III

Data Engineering

6 {dbplyr} and {odbc}

 Under Development

7 {SparkR} to {sparklyr}

7.1 Introduction

Beginning with Spark 4.x, [{SparkR} will be deprecated](#). Going forward, `{sparklyr}` will be the recommended R package for working with Apache Spark. This guide is intended to help users understand the differences between `{SparkR}` and `{sparklyr}` across Spark APIs, and aid in code migration from one to the other. It combines basic concepts with specific function mappings where appropriate.

7.1.1 Overview of {SparkR} and {sparklyr}

`{SparkR}` and `{sparklyr}` are both R packages designed to work with Apache Spark, but differ significantly in design, syntax, and integration with the broader R ecosystem.

`{SparkR}` is developed as part of Apache Spark itself, and its design mirrors Spark's core APIs. This makes it straightforward for those familiar with Spark's other language interfaces - Scala and Python. However, this may be less intuitive for R users accustomed to the [tidyverse](#).

In contrast, `{sparklyr}` is developed and maintained by [Posit PBC](#) with a focus on providing a more R-friendly experience. It leverages `{dplyr}` syntax, which is highly familiar to users of the `{tidyverse}`, enabling them to interact with Spark DataFrames using R-native verbs like `select()`, `filter()`, and `mutate()`. This makes `{sparklyr}` easier to learn for R users, especially those who are not familiar with Spark's native API.

7.2 Environment setup

7.2.1 Installation

If working inside of the Databricks Workspace, no installation is required - you can simply load `{sparklyr}` with `library(sparklyr)`. To install `{sparklyr}` on a machine outside of Databricks, [follow these steps](#).

7.2.2 Connecting to Spark

When working inside of the Databricks workspace, you can connect to Spark with `{sparklyr}` with the following code:

```
library(sparklyr)
sc <- spark_connect(method = "databricks")
```

When connecting to Databricks remotely via [Databricks Connect](#), a slightly different method is used:

```
sc <- spark_connect(method = "databricks_connect")
```

For more details and an extended tutorial on Databricks Connect with `{sparklyr}`, see the [official documentation](#).

7.3 Reading & Writing Data

In contrast to generic `read.df()` and `write.df()` [functions in {SparkR}](#), `{sparklyr}` has a family of `spark_read_*` and `spark_write_*` [functions](#) to load and save data. There are also unique functions to create Spark DataFrames or Spark SQL [temporary views](#) from R data frames in memory.

7.3.1 TL;DR

Table 7.1: Recommended function mapping

{SparkR}	{sparklyr}
<code>createDataFrame()</code>	<code>copy_to()</code>
<code>createOrReplaceTempView()</code>	Use <code>invoke()</code> with method directly
<code>saveAsTable()</code>	<code>spark_write_table()</code>
<code>write.df()</code>	<code>spark_write_<format>()</code>
<code>tableToDF()</code>	<code>tbl()</code> (or <code>spark_read_table()</code> when it's fixed)
<code>read.df()</code>	<code>spark_read_<format>()</code>

7.3.2 Loading Data

To convert a R data frame to a Spark DataFrame, or to create a temporary view out of a DataFrame to apply SQL to it:

SparkR

```
# create SparkDataFrame from R data frame
mtcars_df <- createDataFrame(mtcars)
```

sparklyr

```
# create SparkDataFrame and name temporary view 'mtcars_tmp'
mtcars_tbl <- copy_to(
  sc,
  df = mtcars,
  name = "mtcars_tmp",
  overwrite = TRUE,
  memory = FALSE
)
```

①

②

- ① `copy_to()` will create a temporary view of the data with the given `name`, you can use `name` to reference data if using SQL directly (e.g. `sdf_sql()`).
- ② Default behaviour of `copy_to()` will set `memory` as `TRUE` which caches the table. This helps when reading the data multiple times - sometimes its worth setting to `FALSE` if data is read as one-off.

7.3.3 Creating Views

SparkR

```
# create temporary view
createOrReplaceTempView(mtcars_df, "mtcars_tmp_view")
```

sparklyr

```
# direct equivalent from SparkR requires `invoke`
# usually redundant given `copy_to` already creates a temp view
spark_dataframe(mtcars_tbl) |>
  invoke("createOrReplaceTempView", "mtcars_tmp_view")
```

7.3.4 Writing Data

SparkR

```
# save SparkDataFrame to Unity Catalog
saveAsTable(
  mtcars_df,
  tableName = "<catalog>.<schema>.<table>",
  mode = "overwrite"
)

# save DataFrame using delta format to local filesystem
write.df(
  mtcars_df,
  path = "file:./<path/to/save/delta/mtcars>",
  source = "delta",
  mode = "overwrite"
)
```

①

① `write.df()` supports other formats via `source` parameter

sparklyr

```
# save tbl_spark to Unity Catalog
spark_write_table(
  mtcars_tbl,
  name = "<catalog>.<schema>.<table>",
  mode = "overwrite"
)

# save tbl_spark using delta format to local filesystem
spark_write_delta(
  mtcars_tbl,
  path = "file:./<path/to/save/delta/mtcars>",
  mode = "overwrite"
)

# Using {DBI}
library(DBI)
dbWriteTable(
  sc,
  value = mtcars_tbl,
  name = "<catalog>.<schema>.<table>",
```

```
    overwrite = TRUE
  )
```

7.3.5 Reading Data

SparkR

```
# load Unity Catalog table as SparkDataFrame
tableToDF("<catalog>.<schema>.<table>")

# load csv file into SparkDataFrame
read.df(
  path = "file:<path/to/read/csv/data.csv>",
  source = "csv",
  header = TRUE,
  inferSchema = TRUE
)

# load delta from local filesystem as SparkDataFrame
read.df(
  path = "file:<path/to/read/delta/mtcars>",
  source = "delta"
)

# load data from a table using SQL
# recommended to use `tableToDF`
sql("SELECT * FROM <catalog>.<schema>.<table>")
```

sparklyr

```
# currently has an issue if using Unity Catalog
# recommend using `tbl` (example below)
spark_read_table(sc, "<catalog>.<schema>.<table>", memory = FALSE)

# load table from Unity Catalog with {dplyr}
tbl(sc, "<catalog>.<schema>.<table>")

# or using `in_catalog`
tbl(sc, in_catalog("<catalog>", "<schema>", "<table>"))

# load csv from local filesystem as tbl_spark
```



```

spark_read_csv(
  sc,
  name = "mtcars_csv",
  path = "file:/<path/to/delta/mtcars>",
  header = TRUE,
  infer_schema = TRUE
)

# load delta from local filesystem as tbl_spark
spark_read_delta(
  sc,
  name = "mtcars_delta",
  path = "file:/tmp/test/sparklyr1"
)

# using SQL
sdf_sql(sc, "SELECT * FROM <catalog>.<schema>.<table>")

```

7.4 Processing Data

7.4.1 Select, Filter

SparkR

```

# select specific columns
select(mtcars_df, "mpg", "cyl", "hp")

# filter rows where mpg > 20
filter(mtcars_df, mtcars_df$mpg > 20)

```

sparklyr

```

# select specific columns
mtcars_tbl |>
  select(mpg, cyl, hp)

# filter rows where mpg > 20
mtcars_tbl |>
  filter(mpg > 20)

```

7.4.2 Adding Columns

SparkR

```
# add a new column 'power_to_weight' (hp divided by wt)
withColumn(mtcars_df, "power_to_weight", mtcars_df$hp / mtcars_df$wt)
```

sparklyr

```
# add a new column 'power_to_weight' (hp divided by wt)
mtcars_tbl |>
  mutate(power_to_weight = hp / wt)
```

7.4.3 Grouping & Aggregation

SparkR

```
# calculate average mpg and hp by number of cylinders
mtcars_df |>
  groupBy("cyl") |>
  summarize(
    avg_mpg = avg(mtcars_df$mpg),
    avg_hp = avg(mtcars_df$hp)
  )
```

sparklyr

```
# calculate average mpg and hp by number of cylinders
mtcars_tbl |>
  group_by(cyl) |>
  summarize(
    avg_mpg = mean(mpg),
    avg_hp = mean(hp)
  )
```

7.4.4 Joins

Suppose we have another dataset with cylinder labels that we want to join to mtcars.

SparkR

```
# create another SparkDataFrame with cylinder labels
cylinders <- data.frame(
  cyl = c(4, 6, 8),
  cyl_label = c("Four", "Six", "Eight")
)
cylinders_df <- createDataFrame(cylinders)

# join mtcars_df with cylinders_df
join(
  x = mtcars_df,
  y = cylinders_df,
  mtcars_df$cyl == cylinders_df$cyl,
  joinType = "inner"
)
```

sparklyr

```
# create another SparkDataFrame with cylinder labels
cylinders <- data.frame(
  cyl = c(4, 6, 8),
  cyl_label = c("Four", "Six", "Eight")
)
cylinders_tbl <- copy_to(sc, cylinders, "cylinders", overwrite = TRUE)

# join mtcars_tbl with cylinders_tbl
mtcars_tbl |>
  inner_join(cylinders_tbl, by = join_by(cyl))
```

7.5 User Defined Functions (UDFs)

Suppose we want to categorize horsepower into ‘High’ or ‘Low’ based on a threshold

Note

This is an arbitrary example; in practice we would recommend `case_when()` combined with `mutate()`.

```
# define custom function
categorize_hp <- function(df) {
  df$hp_category <- ifelse(df$hp > 150, "High", "Low")
}
```

```
df
}
```

SparkR

UDFs in {SparkR} require an output schema, which we define first.

```
# define the schema for the output DataFrame
schema <- structType(
  structField("mpg", "double"),
  structField("cyl", "double"),
  structField("disp", "double"),
  structField("hp", "double"),
  structField("drat", "double"),
  structField("wt", "double"),
  structField("qsec", "double"),
  structField("vs", "double"),
  structField("am", "double"),
  structField("gear", "double"),
  structField("carb", "double"),
  structField("hp_category", "string")
)
```

To apply this function to each partition of a Spark DataFrame, we use `dapply()`.

```
# apply function across partitions using dapply
dapply(
  mtcars_df,
  categorize_hp,
  schema
)
```

To apply the same function to each group of a Spark DataFrame, we use `gapply()`. Note that the schema is still required.

```
# apply function across groups
gapply(
  mtcars_df,
  cols = "hp",
  func = categorize_hp,
  schema = schema
)
```

sparklyr

💡 Tip

Highly recommended [‘Distributing R Computations’](#) guide in `{sparklyr}` docs, it goes into much more detail on `spark_apply()`.

i Note

`spark_apply()` will do its best to derive the column names and schema of the output via sampling 10 rows, this can add overhead that can be omitted by specifying the `columns` parameter.

```
# ensure that {arrow} is loaded, otherwise may encounter cryptic errors
library(arrow)
```

```
# apply the function over data
# by default applies to each partition
mtcars_tbl |>
  spark_apply(f = categorize_hp)

# apply the function over data
# Using `group_by` to apply data over groups
mtcars_tbl |>
  spark_apply(
    f = summary,
    group_by = "hp"
  )
```

①

- ① In this example `group_by` isn't changing the resulting output as the functions behaviour is applied to rows independently. Other functions that operate on a set of rows would behave differently (e.g. `summary()`).

`SparkR::spark.lapply()` is unique in that it applies to lists in R, as opposed to DataFrames. There is no exact equivalent in `{sparklyr}`, but using `spark_apply()` with a DataFrame with unique IDs and grouping it by ID will behave similarly in many cases, or, more creative functions that operate on a row-wise basis.

SparkR

```
# define a list of integers
numbers <- list(1, 2, 3, 4, 5)
```

```
# define a function to apply
square <- function(x) {
  x * x
}

# apply the function over list using spark
spark.lapply(numbers, square)
```

sparklyr

```
# create a spark DataFrame of given length
sdf <- sdf_len(sc, 5, repartition = 1)

# apply function to each partition of data.frame
spark_apply(sdf, f = nrow) ①

# apply function to each row (option 1)
spark_apply(sdf, f = nrow, group_by = "id") ②

# apply function to each row (option 2)
row_func <- function(df) { ③
  df |>
    dplyr::rowwise() |>
    dplyr::mutate(x = id * 2)
}
spark_apply(sdf, f = row_func)
```

- ① `spark_apply()` defaults to processing data based on number of partitions, in this case it will return a single row as due to `repartition = 1`.
- ② To force behaviour like `spark.lapply()` you can create a `DataFrame` with `N` rows and force grouping with `group_by` set to a unique row identifier (in this case it's the `id` column automatically generated by `sdf_len()`). This will return `N` rows.
- ③ This requires writing a function that operates across rows of a `data.frame`, in some occasions this *may* be faster relative to (2). Specifying `group_by` is optional for this example. This example does not require `rowwise()`, but is just to illustrate one method to force computations to be for every row. Your function should take care to import required packages, etc.

7.6 Machine learning

Full examples for each package can be found in the official reference for `{SparkR}` and `{sparklyr}`, respectively.

If not using Spark MLlib it is recommended to use UDFs to train with the library of your choice (e.g. `{xgboost}`).

7.6.1 Linear regression

SparkR

```
# select features
training_df <- select(mtcars_df, "mpg", "hp", "wt")

# fit the model using Generalized Linear Model (GLM)
linear_model <- spark.glm(training_df, mpg ~ hp + wt, family = "gaussian")

# view model summary
summary(linear_model)
```

sparklyr

```
# select features
training_tbl <- mtcars_tbl |>
  select(mpg, hp, wt)

# fit the model using Generalized Linear Model
linear_model <- training_tbl |>
  ml_linear_regression(response = "mpg", features = c("hp", "wt"))

# view model summary
summary(linear_model)
```

7.6.2 K-means clustering

SparkR

```
# apply KMeans clustering with 3 clusters using mpg and hp as features
kmeans_model <- spark.kmeans(mtcars_df, mpg ~ hp, k = 3)

# get cluster predictions
predict(kmeans_model, mtcars_df)
```

①

① Predicting on input data to keep example simple

sparklyr

```
# use mpg and hp as features
features_tbl <- mtcars_tbl |>
  select(mpg, hp)

# assemble features into a vector column
features_vector_tbl <- features_tbl |>
  ft_vector_assembler(
    input_cols = c("mpg", "hp"),
    output_col = "features"
  )

# apply K-Means clustering
kmeans_model <- features_vector_tbl |>
  ml_kmeans(features_col = "features", k = 3)

# get cluster predictions
ml_predict(kmeans_model, features_vector_tbl)
```

①

① Predicting on input data to keep example simple

7.7 Performance and optimization

7.7.1 Collecting

Both {SparkR} and {sparklyr} use the same function name, `collect()`, to convert Spark DataFrames to R data frames. In general, only collect small amounts of data back to R data frames or the Spark driver will run out of memory, crashing your script (and you want to use Spark to accelerate workloads as much as possible!).

To prevent out of memory errors, {SparkR} has built-in optimizations in Databricks Runtime that help collect data or execute user-defined functions (which also require collecting data

to workers). To ensure smooth performance with `{sparklyr}` for collecting data and UDFs, make sure to load the `{arrow}` package in your scripts.

```
# when on Databricks DBR 14.3 or higher {arrow} is pre-installed
library(arrow)
```

If you encounter issues with collecting *large* datasets with `{sparklyr}` the methods documented [here](#) may assist, however, hitting this is typically an indicator that you should defer more work to Spark.

7.7.2 In-Memory Partitioning

SparkR

```
# repartition the SparkDataFrame based on 'cyl' column
repartition(mtcars_df, col = mtcars_df$cyl)

# repartition the SparkDataFrame to number of partitions
repartition(mtcars_df, numPartitions = 10)

# coalesce the SparkDataFrame to number of partitions
coalesce(mtcars_df, numPartitions = 1)

# get number of partitions
getNumPartitions(mtcars_df)
```

sparklyr

```
# repartition the tbl_spark based on 'cyl' column
sdf_repartition(mtcars_tbl, partition_by = "cyl")

# repartition the tbl_spark to number of partitions
sdf_repartition(mtcars_tbl, partitions = 10)

# coalesce the tbl_spark to number of partitions
sdf_coalesce(mtcars_tbl, partitions = 1)

# get number of partitions
sdf_num_partitions(mtcars_tbl)
```

7.7.3 Caching

SparkR

```
# cache the SparkDataFrame in memory  
cache(mtcars_df)
```

sparklyr

```
# cache the tbl_spark in memory  
tbl_cache(sc, name = "mtcars_tmp")
```

Part IV

Miscellaneous

8 {htmlwidgets} in notebooks

When you try to view a {htmlwidget} based visualisation (e.g. {leaflet}) in a Databricks notebook you'll find there is no rendered output by default.

The Databricks [documentation](#) details how to get this working but requires specification of the workspace URL explicitly and writes out files to DBFS's FileStore without cleaning up after itself.

The new method avoids those steps and is drastically simplified and easier to use, just run the below function in a Databricks notebook:

```
enable_htmlwidgets <- function(height = 450) {  
  
  # new option to control default widget height, default is 450px  
  options(db_htmlwidget_height = height) ①  
  
  system("apt-get update && apt-get --yes install pandoc", intern = T) ②  
  if (!base::require("htmlwidgets")) { ③  
    utils::install.packages("htmlwidgets")  
  }  
  
  # new method will fetch height based on new option, or default to 450px  
  new_method <- function(x, ...) { ④  
    x$height <- getOption("db_htmlwidget_height", 450)  
    file <- tempfile(fileext = ".html")  
    htmlwidgets::saveWidget(x, file = file)  
    contents <- as.character(rvest::read_html(file))  
    displayHTML(contents)  
  }  
  
  utils::assignInNamespace("print.htmlwidget", new_method, ns = "htmlwidgets") ⑤  
  invisible(list(default_height = height, print = new_method))  
}
```

- ① The height of the htmlwidget output is controlled via an option (db_htmlwidget_height), this allows the height to be adjusted without re-running the function

- ② Installing `pandoc` as it's required to use `htmlwidgets::saveWidget`
- ③ Ensure that `{htmlwidgets}` is installed
- ④ Function that writes the widget to a temporary file as a self-contained `html` file and then reads the contents and presents via `displayHTML`
- ⑤ Override the `htmlwidgets::print.htmlwidget` method

9 OAuth {odbc}

When using {odbc} to connect to Databricks clusters and SQL warehouses you'll likely have used a personal access token (PAT). It's not uncommon for workspace administrators to [disable the use of PATs](#).

If you are unable to create a PAT you are still able to connect to Databricks but you'll need to use OAuth (either [M2M](#) or [U2M](#)).

User-to-machine (U2M) is typically what you'd want to use. Good news, [the Databricks ODBC driver supports both since 2.7.5](#).

9.1 U2M Example

Note

OAuth U2M or OAuth 2.0 browser-based authentication works only with applications that run locally. It does not work with server-based or cloud-based applications.

When running this code you should be prompted to login to the workspace or you'll see a window that says "success". You can close the window and continue working in R.

```
library(odbc)
library(DBI)

con <- DBI::dbConnect(
  drv = odbc::databricks(),
  httpPath = "/sql/1.0/warehouses/<warehouse-id>",
  workspace = "<workspace-name>.cloud.databricks.com",
  authMech = 11,
  auth_flow = 2
)
```

- ① {odbc} recently added `odbc::databricks()` to simplify connecting to Databricks ([requires version >=1.4.0](#))
- ② The `httpPath` can be found in the 'Connection Details' tab of a SQL warehouse

- ③ **workspace** refers to the workspace URL, also found in ‘Connection Details’ tab as ‘Server hostname’
- ④ The [docs](#) mention setting **AuthMech** to 11 and **Auth_Flow** to 2