

# Using Andromeda

Martijn J. Schuemie

2020-05-01

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Permanence . . . . .	1
1.2	Technology . . . . .	1
<b>2</b>	<b>Creating an Andromeda object</b>	<b>1</b>
2.1	Closing Andromeda objects . . . . .	2
2.2	Temporary file location . . . . .	2
<b>3</b>	<b>Querying data from an Andromeda object</b>	<b>3</b>
3.1	Simple meta-data . . . . .	3
3.2	Using SQL . . . . .	3
<b>4</b>	<b>Batch operations</b>	<b>4</b>
4.1	Safe mode . . . . .	4
<b>5</b>	<b>Saving and loading Andromeda objects</b>	<b>5</b>

## 1 Introduction

The **Andromeda** package provides the ability to work with data objects in R that are too large to fit in memory. Instead, these objects are stored on disk. This is slower than working from memory, but may be the only viable option. **Andromeda** aims to replace the now orphaned **ff** package.

### 1.1 Permanence

To mimic the behavior of in-memory objects, when working with data in **Andromeda** the data is stored in a temporary location on the disk. You can modify the data as you can see fit, and when needed can save the data to a permanent location. Later this data can be loaded to a temporary location again and be read and modified, while keeping the saved data as is.

### 1.2 Technology

**Andromeda** heavily relies on **RSQLite**, an R wrapper around **SQLite**. **SQLite** is a low-weight but very powerful single-user SQL database that can run from a single file on the local file system. Although **SQLite** and therefore **Andromeda** can be queried using SQL, **Andromeda** favors using **dbplyr**, a **dplyr** implementation, to work with the data.

## 2 Creating an Andromeda object

Creating a empty **Andromeda** object is easy:

```
library(Andromeda)
andr <- andromeda()
```

We can add new tables to the **Andromeda** environment like this:

```
andr$cars <- cars
andr
```

```
#> # Andromeda object
#> # Physical location: C:\Users\MSCHUEMI\AppData\Local\Temp\1\RtmpYbaLTr\filebf82b18cb1.sqlite
#>
#> Tables:
#> $cars (speed, dist)
```

We could have achieved the same by adding the table when creating the **Andromeda** object:

```
andr <- andromeda(cars = cars)
```

Of course, we probably want to add data to the **Andromeda** environment that is much larger than can fit in memory. One way to achieve this is by iteratively adding more and more data to the same table. As an example here we simply add the same data to the existing table:

```
appendToTable(andr$cars, cars)
```

The data to append should have the same columns as the existing data, or else an error will be thrown.

Data can be copied from one **Andromeda** to another:

```
andr2 <- andromeda()
andr2$cars <- andr$cars
```

For very large tables this may be slow. A faster option might be to copy the entire **Andromeda** object:

```
andr3 <- copyAndromeda(andr)
```

## 2.1 Closing Andromeda objects

Every **Andromeda** object will have a corresponding data file in a temporary location on your local file system. This file will be automatically deleted when the **Andromeda** object is no longer used. It is best practice not to rely on R to decide when to do this, but explicitly cause the file to be cleaned up by calling the **close** statement:

```
close(andr)
close(andr2)
close(andr3)
```

Once an **Andromeda** is closed the underlying file is deleted, and it can no longer be used. You can check whether an **Andromeda** object is still valid:

```
isValidAndromeda(andr)
```

```
#> [1] FALSE
```

## 2.2 Temporary file location

By default **Andromeda** uses the default temporary file location of your operating system to store the **Andromeda** objects while you are working on them. You can override the location by setting the **andromedaTempFolder** option:

```
options(andromedaTempFolder = "c:/andromedaTemp")
```

This only applies to **Andromeda** objects that are created from that point onward. Prior objects will stay where they are.

### 3 Querying data from an Andromeda object

**Andromeda** relies on **dbplyr**, a **dplyr** implementation, for querying the data. A key aspect of **dbplyr** is lazy execution. This means that we can define a query, but the query will not be executed until we explicitly request so using the `collect()` statement. For example, we may want to know the number of cars that can go faster than 10:

```
andr <- andromeda(cars = cars)
andr$cars %>%
  filter(speed > 10) %>%
  count() %>%
  collect()
```

```
#> # A tibble: 1 x 1
#>       n
#>   <int>
#> 1     41
```

Here we first filter the table to those rows where `speed > 10`, after which we count the number of remaining records. This query is not executed until we call `collect()`. We can also have **Andromeda** call `collect` for us, by assigning the query to a table:

```
andr$fastCars <- andr$cars %>%
  filter(speed > 10)
```

This way the query result does not have to pass through memory, but instead is directly stored in **Andromeda**.

#### 3.1 Simple meta-data

If we wish to know the tables that exist in an **Andromeda** object, we can call the generic `names` function. Similarly, we can call `colnames` to get the names of the columns in a table. We can call `nrow` and `ncol` to get the number of columns. Note that these functions also work on query results, but be warned that `nrow` and `ncol` require actually executing the query, so may be slow:

```
names(andr)
```

```
#> [1] "cars"      "fastCars"
```

```
colnames(andr$cars)
```

```
#> [1] "speed" "dist"
```

```
nrow(andr$cars)
```

```
#> [1] 50
```

```
ncol(andr$cars %>% select(speed))
```

```
#> [1] 1
```

#### 3.2 Using SQL

In the end, an **Andromeda** is still an **RSQLite** database, and can be queried using SQL:

```
RSQLite::dbGetQuery(andr, "SELECT * FROM cars LIMIT 5;")
```

```
#>   speed dist
#> 1     4    2
#> 2     4   10
#> 3     7    4
#> 4     7   22
#> 5     8   16
```

However, for consistency it is recommended to use the `dplyr` functions instead.

## 4 Batch operations

Often we'll need to perform some action against the data that is not supported by `dplyr`. For example, we may want to write data to a CSV file, or perform some complex computation. Since we cannot assume an entire table (or query result) will fit in memory, we must assume we should do this batch-wise. For this the **Andromeda** package provides two functions: `batchApply` and `groupApply`. The former executes a function on batches of predefined length. We can specify the batch size by setting the `batchSize` argument, which defaults to 100,000. Here is a silly example, where take the number of rows in a batch, and multiply it by some number:

```
doSomething <- function(batch, multiplier) {
  return(nrow(batch) * multiplier)
}
result <- batchApply(andr$cars, doSomething, multiplier = 2, batchSize = 10)
result <- unlist(result)
result
```

```
#> [1] 20 20 20 20 20
```

Alternatively, using `groupApply` we can execute a function on groups of rows, defined by the value in some variable in the table. In this example, we first filter to fast cars, and then perform the same meaningless computation as in the previous example, this time on groups of rows defined by having the same speed:

```
doSomething <- function(batch, multiplier) {
  return(nrow(batch) * multiplier)
}
result <- groupApply(andr$cars %>% filter(speed > 10),
  doSomething,
  groupVariable = "speed",
  multiplier = 2)
result <- unlist(result)
result
```

```
#> 11 12 13 14 15 16 17 18 19 20 22 23 24 25
#>  4  8  8  8  6  4  6  8  6 10  2  2  8  2
```

(For example, there were 2 rows where `speed = 11`, and multiplied by 2 this gives 4 for item 11.)

### 4.1 Safe mode

For technical reasons it is not possible to write to the same **Andromeda** while reading from it. Writing to an **Andromeda** environment while inside a `batchApply` or `groupApply` on the that same **Andromeda** environment will therefore result in an error message. To avoid this, you can set the `safe` argument to `TRUE`. This will cause the table to first be copied to a temporary **Andromeda** before executing the function. However, this might not be very fast:

```
doSomething <- function(batch) {
  batch$speedSquared <- batch$speed^2
```

```

if (is.null(andr$cars2)) {
  andr$cars2 <- batch
} else {
  appendToTable(andr$cars2, batch)
}
}
batchApply(andr$cars, doSomething, safe = TRUE)

```

Note that this is only a restriction of `batchApply` and `groupApply`. We could have achieved the same task as the example above much faster using only `dplyr`:

```

andr$cars2 <-
  andr$cars %>%
  mutate(speedSquared = speed^2)

```

## 5 Saving and loading Andromeda objects

To reuse an `Andromeda` at a later point in time, we can save it to a permanent location:

```
saveAndromeda(andr, "c:/temp/andromeda.zip")
```

#> Disconnected `Andromeda`. This data object can no longer be used

For technical reasons, saving an `Andromeda` object closes it. If we want to continue using the `Andromeda` object, we can set `maintainConnection` to `TRUE` when calling `saveAndromeda`. This causes a temporary copy to be created first, which is then saved and closed. Obviously this will take additional time, so if you know you will no longer need the object in R after saving, it is best not to use this option.

We can load the object again using:

```
andr <- loadAndromeda("c:/temp/andromeda.zip")
```